

Contents

Computer Vision API Documentation

Overview

[What is Computer Vision?](#)

[What's new](#)

Quickstarts

[Using the client library](#)

[Using the REST API](#)

[Analyze a remote image](#)

[cURL](#)

[Go](#)

[Java](#)

[JavaScript](#)

[Node.js](#)

[Python](#)

[Analyze a local image](#)

[C#](#)

[Python](#)

[Generate a thumbnail](#)

[C#](#)

[cURL](#)

[Go](#)

[Java](#)

[JavaScript](#)

[Node.js](#)

[Python](#)

[Extract text \(Read API\)](#)

[C#](#)

[Java](#)

[JavaScript](#)

Python

Extract text (OCR API)

C#

cURL

Go

Java

JavaScript

Node.js

Python

Use a domain model

Python

Tutorials

Generate metadata for images

Concepts

Content tags

Object detection

Brand detection

Image categorization

Image descriptions

Face detection

Image type detection

Domain-specific content

Color scheme detection

Smart-cropped thumbnails

Optical Character Recognition (OCR)

Adult content detection

Containers

Read API container

Install and run containers

Configure containers

Use with Kubernetes and Helm

Use container instances

How-to guides

[Call the Computer Vision API](#)

[Analyze videos in real time](#)

Reference

[Azure CLI](#)

[Azure PowerShell](#)

[Computer Vision API v3.1 preview](#)

[Computer Vision API v3.0](#)

[Computer Vision API v2.1](#)

[Computer Vision API v2.0](#)

[Computer Vision API v1.0](#)

SDKs

[.NET](#)

[Node.js](#)

[Python](#)

[Go](#)

[Java](#)

Resources

Samples

[Explore an image processing app](#)

[Other Computer Vision samples](#)

FAQ

[Category taxonomy](#)

[Language support](#)

[Pricing and limits](#)

[UserVoice](#)

[Stack Overflow](#)

[Azure roadmap](#)

[Regional availability](#)

[Compliance](#)

[Upgrade to v3.0 from 2.0 or 2.1](#)

What is Computer Vision?

9/1/2020 • 5 minutes to read • [Edit Online](#)

IMPORTANT

TLS 1.2 is now enforced for all HTTP requests to this service. For more information, see [Azure Cognitive Services security](#).

Azure's Computer Vision service provides developers with access to advanced algorithms that process images and return information based on the visual features you're interested in. For example, Computer Vision can determine whether an image contains adult content, find specific brands or objects, or find human faces.

You can use Computer Vision in your application through a client library SDK or by calling the REST API directly. This page broadly covers what you can do with Computer Vision.

Computer Vision for digital asset management

Computer Vision can power many digital asset management (DAM) scenarios. DAM is the business process of organizing, storing, and retrieving rich media assets and managing digital rights and permissions. For example, a company may want to group and identify images based on visible logos, faces, objects, colors, and so on. Or, you might want to automatically [generate captions for images](#) and attach keywords so they're searchable. For an all-in-one DAM solution using Cognitive Services, Azure Cognitive Search, and intelligent reporting, see the [Knowledge Mining Solution Accelerator Guide](#) on GitHub. For other DAM examples, see the [Computer Vision Solution Templates](#) repository.

Analyze images for insight

You can analyze images to provide insights about their visual features and characteristics. All of the features in the table below are provided by the [Analyze Image](#) API.

ACTION	DESCRIPTION
Tag visual features	Identify and tag visual features in an image, from a set of thousands of recognizable objects, living things, scenery, and actions. When the tags are ambiguous or not common knowledge, the API response provides hints to clarify the context of the tag. Tagging isn't limited to the main subject, such as a person in the foreground, but also includes the setting (indoor or outdoor), furniture, tools, plants, animals, accessories, gadgets, and so on.
Detect objects	Object detection is similar to tagging, but the API returns the bounding box coordinates for each tag applied. For example, if an image contains a dog, cat and person, the Detect operation will list those objects together with their coordinates in the image. You can use this functionality to process further relationships between the objects in an image. It also lets you know when there are multiple instances of the same tag in an image.

ACTION	DESCRIPTION
Detect brands	Identify commercial brands in images or videos from a database of thousands of global logos. You can use this feature, for example, to discover which brands are most popular on social media or most prevalent in media product placement.
Categorize an image	Identify and categorize an entire image, using a category taxonomy with parent/child hereditary hierarchies. Categories can be used alone, or with our new tagging models. Currently, English is the only supported language for tagging and categorizing images.
Describe an image	Generate a description of an entire image in human-readable language, using complete sentences. Computer Vision's algorithms generate various descriptions based on the objects identified in the image. The descriptions are each evaluated and a confidence score generated. A list is then returned ordered from highest confidence score to lowest.
Detect faces	Detect faces in an image and provide information about each detected face. Computer Vision returns the coordinates, rectangle, gender, and age for each detected face. Computer Vision provides a subset of the Face service functionality. You can use the Face service for more detailed analysis, such as facial identification and pose detection.
Detect image types	Detect characteristics about an image, such as whether an image is a line drawing or the likelihood of whether an image is clip art.
Detect domain-specific content	Use domain models to detect and identify domain-specific content in an image, such as celebrities and landmarks. For example, if an image contains people, Computer Vision can use a domain model for celebrities to determine if the people detected in the image are known celebrities.
Detect the color scheme	Analyze color usage within an image. Computer Vision can determine whether an image is black & white or color and, for color images, identify the dominant and accent colors.
Generate a thumbnail	Analyze the contents of an image to generate an appropriate thumbnail for that image. Computer Vision first generates a high-quality thumbnail and then analyzes the objects within the image to determine the <i>area of interest</i> . Computer Vision then crops the image to fit the requirements of the area of interest. The generated thumbnail can be presented using an aspect ratio that is different from the aspect ratio of the original image, depending on your needs.
Get the area of interest	Analyze the contents of an image to return the coordinates of the <i>area of interest</i> . Instead of cropping the image and generating a thumbnail, Computer Vision returns the bounding box coordinates of the region, so the calling application can modify the original image as desired.

Optical Character Recognition (OCR)

Computer Vision includes [Optical Character Recognition \(OCR\)](#) capabilities. You can use the new Read API to extract printed and handwritten text from images and documents. It uses the latest models and works with text on a variety of surfaces and backgrounds. These include receipts, posters, business cards, letters, and whiteboards. The two OCR APIs support extracting printed text in [several languages](#).

Moderate content in images

You can use Computer Vision to [detect adult content](#) in an image and return confidence scores for different classifications. The threshold for flagging content can be set on a sliding scale to accommodate your preferences.

Use containers

[Use Computer Vision containers](#) to recognize printed and handwritten text locally by installing a standardized Docker container closer to your data.

Image requirements

Computer Vision can analyze images that meet the following requirements:

- The image must be presented in JPEG, PNG, GIF, or BMP format
- The file size of the image must be less than 4 megabytes (MB)
- The dimensions of the image must be greater than 50 x 50 pixels
 - For the Read API, the dimensions of the image must be between 50 x 50 and 10000 x 10000 pixels.

Data privacy and security

As with all of the Cognitive Services, developers using the Computer Vision service should be aware of Microsoft's policies on customer data. See the [Cognitive Services page](#) on the Microsoft Trust Center to learn more.

Next steps

Get started with Computer Vision by following a quickstart guide:

- [Quickstart: Computer Vision .NET client library](#)
- [Quickstart: Computer Vision Python client library](#)
- [Quickstart: Computer Vision Java client library](#)

What's new in Computer Vision

9/1/2020 • 2 minutes to read • [Edit Online](#)

Learn what's new in the service. These items may be release notes, videos, blog posts, and other types of information. Bookmark this page to stay up to date with the service.

July 2020

Read API v3.1 Public Preview adds Simplified Chinese support

Computer Vision's Read API v3.1 public preview adds support for Simplified Chinese.

- This preview version of the Read API supports English, Dutch, French, German, Italian, Portuguese, Simplified Chinese, and Spanish languages.

See the [Read API overview](#) to learn more.

[Learn more about Read API v3.1 Public Preview](#)

May 2020

Computer Vision API v3.0 entered General Availability, with updates to [Read API](#):

- Support for English, Dutch, French, German, Italian, Portuguese, and Spanish
- Improved accuracy
- Confidence score for each extracted word
- New output format

March 2020

- TLS 1.2 is now enforced for all HTTP requests to this service. For more information, see [Azure Cognitive Services security](#).

January 2020

Read API 3.0 Public Preview

You now have the option to use version 3.0 of the Read API to extract printed or handwritten text from images. Compared to earlier versions, 3.0 provides:

- Improved accuracy
- New output format
- Confidence score for each extracted word
- Support for both Spanish and English languages with the additional language parameter

Follow an [Extract text quickstart](#) to get starting using the 3.0 API.

Cognitive Service updates

[Azure update announcements for Cognitive Services](#)

Quickstart: Use the Computer Vision client library

9/1/2020 • 56 minutes to read • [Edit Online](#)

Get started with the Computer Vision client library. Follow these steps to install the package and try out the example code for basic tasks. Computer Vision provides you with access to advanced algorithms for processing images and returning information.

Use the Computer Vision client library to:

- Analyze an image for tags, text description, faces, adult content, and more.
- Read printed and handwritten text with the Read API.

[Reference documentation](#) | [Library source code](#) | [Package \(NuGet\)](#) | [Samples](#)

Prerequisites

- An Azure subscription - [Create one for free](#)
- The latest version of the [.NET Core SDK](#).
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (**F0**) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Setting up

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 `computer-vision-quickstart`. This command creates a simple "Hello World" C# project with a single source file: *ComputerVisionQuickstart.cs*.

```
dotnet new console -n computer-vision-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 *ComputerVisionQuickstart.cs* file in your preferred editor or IDE. Add the following `using` directives:

```
using System;
using System.Collections.Generic;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
using System.Threading.Tasks;
using System.IO;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System.Threading;
using System.Linq;
```

In the application's **Program** class, create variables for your resource's Azure endpoint and key.

```
// Add your Computer Vision subscription key and endpoint to your environment variables.
// Close/reopen your project for them to take effect.
static string subscriptionKey = "COMPUTER_VISION_SUBSCRIPTION_KEY";
static string endpoint = "COMPUTER_VISION_ENDPOINT";
```

Install the client library

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

```
dotnet add package Microsoft.Azure.CognitiveServices.Vision.ComputerVision --version 6.0.0-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 Computer Vision .NET SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to do most image operations.
ComputerVisionClientExtensions	This class contains additional methods for the ComputerVisionClient .
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of VisualFeatureTypes values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Computer Vision client library for .NET:

- [Authenticate the client](#)
- [Analyze an image](#)
- [Read printed and handwritten text](#)

Authenticate the client

NOTE

This quickstart assumes you've [created environment variables](#) for your Computer Vision key and endpoint, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT` respectively.

In a new method, instantiate a client with your endpoint and key. Create a [ApiKeyServiceClientCredentials](#) object with your key, and use it with your endpoint to create a [ComputerVisionClient](#) object.

```
/*
 * AUTHENTICATE
 * Creates a Computer Vision client used by each example.
 */
public static ComputerVisionClient Authenticate(string endpoint, string key)
{
    ComputerVisionClient client =
        new ComputerVisionClient(new ApiKeyServiceClientCredentials(key))
        { Endpoint = endpoint };
    return client;
}
```

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

```
// Create a client
ComputerVisionClient client = Authenticate(endpoint, subscriptionKey);
```

Analyze an image

The following code defines a method, `AnalyzeImageUrl`, which uses the client object to analyze a remote image and print the results. The method returns a text description, categorization, list of tags, detected faces, adult content flags, main colors, and image type.

Add the method call in your `Main` method.

```
// Analyze an image to get features and other properties.
AnalyzeImageUrl(client, ANALYZE_URL_IMAGE).Wait();
```

Set up test image

In your `Program` class, save a reference to the URL of the image you want to analyze.

```
// URL image used for analyzing an image (image of puppy)
private const string ANALYZE_URL_IMAGE =
    "https://moderatorsampleimages.blob.core.windows.net/samples/sample16.png";
```

NOTE

You can also analyze a local image. See the sample code on [GitHub](#) for scenarios involving local images.

Specify visual features

Define your new method for image analysis. Add the code below, which specifies visual features you'd like to extract in your analysis. See the [VisualFeatureTypes](#) enum for a complete list.

```

/*
 * ANALYZE IMAGE - URL IMAGE
 * Analyze URL image. Extracts captions, categories, tags, objects, faces, racy/adult content,
 * brands, celebrities, landmarks, color scheme, and image types.
 */
public static async Task AnalyzeImageUrl(ComputerVisionClient client, string imageUrl)
{
    Console.WriteLine("-----");
    Console.WriteLine("ANALYZE IMAGE - URL");
    Console.WriteLine();

    // Creating a list that defines the features to be extracted from the image.

    List<VisualFeatureTypes?> features = new List<VisualFeatureTypes?>()
    {
        VisualFeatureTypes.Categories, VisualFeatureTypes.Description,
        VisualFeatureTypes.Faces, VisualFeatureTypes.ImageType,
        VisualFeatureTypes.Tags, VisualFeatureTypes.Adult,
        VisualFeatureTypes.Color, VisualFeatureTypes.Brands,
        VisualFeatureTypes.Objects
    };
}

```

Insert any of the following code blocks into your **AnalyzeImageUrl** method to implement their features. Remember to add a closing bracket at the end.

```

}

```

Analyze

The **AnalyzeImageAsync** method returns an **ImageAnalysis** object that contains all of extracted information.

```

Console.WriteLine($"Analyzing the image {Path.GetFileName(imageUrl)}...");
Console.WriteLine();
// Analyze the URL image
ImageAnalysis results = await client.AnalyzeImageAsync(imageUrl, features);

```

The following sections show how to parse this information in detail.

Get image description

The following code gets the list of generated captions for the image. See [Describe images](#) for more details.

```

// Summarizes the image content.
Console.WriteLine("Summary:");
foreach (var caption in results.Description.Captions)
{
    Console.WriteLine($"{caption.Text} with confidence {caption.Confidence}");
}
Console.WriteLine();

```

Get image category

The following code gets the detected category of the image. See [Categorize images](#) for more details.

```
// Display categories the image is divided into.
Console.WriteLine("Categories:");
foreach (var category in results.Categories)
{
    Console.WriteLine($"{category.Name} with confidence {category.Score}");
}
Console.WriteLine();
```

Get image tags

The following code gets the set of detected tags in the image. See [Content tags](#) for more details.

```
// Image tags and their confidence score
Console.WriteLine("Tags:");
foreach (var tag in results.Tags)
{
    Console.WriteLine($"{tag.Name} {tag.Confidence}");
}
Console.WriteLine();
```

Detect objects

The following code detects common objects in the image and prints them to the console. See [Object detection](#) for more details.

```
// Objects
Console.WriteLine("Objects:");
foreach (var obj in results.Objects)
{
    Console.WriteLine($"{obj.ObjectProperty} with confidence {obj.Confidence} at location {obj.Rectangle.X}, " +
        $"{obj.Rectangle.X + obj.Rectangle.W}, {obj.Rectangle.Y}, {obj.Rectangle.Y + obj.Rectangle.H}");
}
Console.WriteLine();
```

Detect brands

The following code detects corporate brands and logos in the image and prints them to the console. See [Brand detection](#) for more details.

```
// Well-known (or custom, if set) brands.
Console.WriteLine("Brands:");
foreach (var brand in results.Brands)
{
    Console.WriteLine($"Logo of {brand.Name} with confidence {brand.Confidence} at location {brand.Rectangle.X}, " +
        $"{brand.Rectangle.X + brand.Rectangle.W}, {brand.Rectangle.Y}, {brand.Rectangle.Y + brand.Rectangle.H}");
}
Console.WriteLine();
```

Detect faces

The following code returns the detected faces in the image with their rectangle coordinates and select face attributes. See [Face detection](#) for more details.

```
// Faces
Console.WriteLine("Faces:");
foreach (var face in results.Faces)
{
    Console.WriteLine($"A {face.Gender} of age {face.Age} at location {face.FaceRectangle.Left}, " +
        $"{face.FaceRectangle.Left}, {face.FaceRectangle.Top + face.FaceRectangle.Width}, " +
        $"{face.FaceRectangle.Top + face.FaceRectangle.Height}");
}
Console.WriteLine();
```

Detect adult, racy, or gory content

The following code prints the detected presence of adult content in the image. See [Adult, racy, gory content](#) for more details.

```
// Adult or racy content, if any.
Console.WriteLine("Adult:");
Console.WriteLine($"Has adult content: {results.Adult.IsAdultContent} with confidence {results.Adult.AdultScore}");
Console.WriteLine($"Has racy content: {results.Adult.IsRacyContent} with confidence {results.Adult.RacyScore}");
Console.WriteLine();
```

Get image color scheme

The following code prints the detected color attributes in the image, like the dominant colors and accent color. See [Color schemes](#) for more details.

```
// Identifies the color scheme.
Console.WriteLine("Color Scheme:");
Console.WriteLine("Is black and white?: " + results.Color.IsBWImg);
Console.WriteLine("Accent color: " + results.Color.AccentColor);
Console.WriteLine("Dominant background color: " + results.Color.DominantColorBackground);
Console.WriteLine("Dominant foreground color: " + results.Color.DominantColorForeground);
Console.WriteLine("Dominant colors: " + string.Join(", ", results.Color.DominantColors));
Console.WriteLine();
```

Get domain-specific content

Computer Vision can use specialized models to do further analysis on images. See [Domain-specific content](#) for more details.

The following code parses data about detected celebrities in the image.

```
// Celebrities in image, if any.
Console.WriteLine("Celebrities:");
foreach (var category in results.Categories)
{
    if (category.Detail?.Celebrities != null)
    {
        foreach (var celeb in category.Detail.Celebrities)
        {
            Console.WriteLine($"{celeb.Name} with confidence {celeb.Confidence} at location {celeb.FaceRectangle.Left}, " +
                $"{celeb.FaceRectangle.Top}, {celeb.FaceRectangle.Height}, {celeb.FaceRectangle.Width}");
        }
    }
}
Console.WriteLine();
```

The following code parses data about detected landmarks in the image.

```
// Popular landmarks in image, if any.
Console.WriteLine("Landmarks:");
foreach (var category in results.Categories)
{
    if (category.Detail?.Landmarks != null)
    {
        foreach (var landmark in category.Detail.Landmarks)
        {
            Console.WriteLine($"{landmark.Name} with confidence {landmark.Confidence}");
        }
    }
}
Console.WriteLine();
```

Get the image type

The following code prints information about the type of image—whether it is clip art or a line drawing.

```
// Detects the image types.
Console.WriteLine("Image Type:");
Console.WriteLine("Clip Art Type: " + results.ImageType.ClipArtType);
Console.WriteLine("Line Drawing Type: " + results.ImageType.LineDrawingType);
Console.WriteLine();
```

Read printed and handwritten text

Computer Vision can read visible text in an image and convert it to a character stream. For more information on text recognition, see the [Optical character recognition \(OCR\)](#) conceptual doc. The code in this section uses the latest [Computer Vision SDK release for Read 3.0](#) and defines a method, `BatchReadFileUr1`, which uses the client object to detect and extract text in the image.

Add the method call in your `Main` method.

```
// Extract text (OCR) from a URL image using the Read API
ReadFileUrl(client, READ_TEXT_URL_IMAGE).Wait();
// Extract text (OCR) from a local image using the Read API
ReadFileLocal(client, READ_TEXT_LOCAL_IMAGE).Wait();
```

Set up test image

In your **Program** class, save a reference to the URL of the image you want to extract text from. This snippet includes sample images for both printed and handwritten text.

```
private const string READ_TEXT_URL_IMAGE =
"https://intelligentkioskstore.blob.core.windows.net/visionapi/suggestedphotos/3.png";
```

NOTE

You can also extract text from a local image. See the sample code on [GitHub](#) for scenarios involving local images.

Call the Read API

Define the new method for reading text. Add the code below, which calls the `ReadAsync` method for the given image. This returns an operation ID and starts an asynchronous process to read the content of the image.

```

/*
 * READ FILE - URL
 * Extracts text.
 */
public static async Task ReadFileUrl(ComputerVisionClient client, string urlFile)
{
    Console.WriteLine("-----");
    Console.WriteLine("READ FILE FROM URL");
    Console.WriteLine();

    // Read text from URL
    var textHeaders = await client.ReadAsync(urlFile, language: "en");
    // After the request, get the operation location (operation ID)
    string operationLocation = textHeaders.OperationLocation;
    Thread.Sleep(2000);

    // Retrieve the URI where the extracted text will be stored from the Operation-Location header.
    // We only need the ID and not the full URL
    const int numberOfCharsInOperationId = 36;
    string operationId = operationLocation.Substring(operationLocation.Length - numberOfCharsInOperationId);

    // Extract the text
    ReadOperationResult results;
    Console.WriteLine($"Extracting text from URL file {Path.GetFileName(urlFile)}...");
    Console.WriteLine();
    do
    {
        {
            results = await client.GetReadResultAsync(Guid.Parse(operationId));
        }
        while ((results.Status == OperationStatusCodes.Running ||
            results.Status == OperationStatusCodes.NotStarted));

    // Display the found text.
    Console.WriteLine();
    var textUrlFileResults = results.AnalyzeResult.ReadResults;
    foreach (ReadResult page in textUrlFileResults)
    {
        foreach (Line line in page.Lines)
        {
            Console.WriteLine(line.Text);
        }
    }
    Console.WriteLine();
}
}

```

Get Read results

Next, get the operation ID returned from the **ReadAsync** call, and use it to query the service for operation results. The following code checks the operation until the results are returned. It then prints the extracted text data to the console.

```
// Retrieve the URI where the extracted text will be stored from the Operation-Location header.
// We only need the ID and not the full URL
const int numberOfCharsInOperationId = 36;
string operationId = operationLocation.Substring(operationLocation.Length - numberOfCharsInOperationId);

// Extract the text
ReadOperationResult results;
Console.WriteLine($"Extracting text from URL file {Path.GetFileName(urlFile)}...");
Console.WriteLine();
do
{
    results = await client.GetReadResultAsync(Guid.Parse(operationId));
}
while ((results.Status == OperationStatusCodes.Running ||
    results.Status == OperationStatusCodes.NotStarted));
```

Display Read results

Add the following code to parse and display the retrieved text data, and finish the method definition.

```
// Display the found text.
Console.WriteLine();
var textUrlFileResults = results.AnalyzeResult.ReadResults;
foreach (ReadResult page in textUrlFileResults)
{
    foreach (Line line in page.Lines)
    {
        Console.WriteLine(line.Text);
    }
}
```

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](#)

Next steps

[Computer Vision API reference \(.NET\)](#)

- [What is Computer Vision?](#)
- The source code for this sample can be found on [GitHub](#).

[Reference documentation](#) | [Artifact \(Maven\)](#) | [Samples](#)

Prerequisites

- An Azure subscription - [Create one for free](#)

- The current version of the [Java Development Kit\(JDK\)](#)
- The [Gradle build tool](#), or another dependency manager.
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Setting up

Create a new Gradle project

In a console window (such as cmd, PowerShell, or Bash), create a new directory for your app, and navigate to it.

```
mkdir myapp && cd myapp
```

Run the `gradle init` command from your working directory. This command will create essential build files for Gradle, including *build.gradle.kts*, which is used at runtime to create and configure your application.

```
gradle init --type basic
```

When prompted to choose a DSL, select **Kotlin**.

Locate *build.gradle.kts* and open it with your preferred IDE or text editor. Then copy in the following build configuration. This configuration defines the project as a Java application whose entry point is the class **ComputerVisionQuickstarts**. It imports the Computer Vision library.

```
plugins {  
    java  
    application  
}  
application {  
    mainClassName = "ComputerVisionQuickstarts"  
}  
repositories {  
    mavenCentral()  
}
```

From your working directory, run the following command to create a project source folder:

```
mkdir -p src/main/java
```

Navigate to the new folder and create a file called *ComputerVisionQuickstarts.java*. Open it in your preferred editor or IDE and add the following `import` statements:

```
import com.microsoft.azure.cognitiveservices.vision.computervision.*;
import com.microsoft.azure.cognitiveservices.vision.computervision.implementation.ComputerVisionImpl;
import com.microsoft.azure.cognitiveservices.vision.computervision.models.*;

import java.io.File;
import java.nio.file.Files;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
```

Then add a class definition for **ComputerVisionQuickstarts**.

Install the client library

This quickstart uses the Gradle dependency manager. You can find the client library and information for other dependency managers on the [Maven Central Repository](#).

In your project's *build.gradle.kts* file, include the Computer Vision client library as a dependency.

```
dependencies {
    compile(group = "com.microsoft.azure.cognitiveservices", name = "azure-cognitiveservices-computervision",
        version = "1.0.4-beta")
}
```

Object model

The following classes and interfaces handle some of the major features of the Computer Vision Java SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to produce instances of other classes.
ComputerVision	This class comes from the client object and directly handles all of the image operations, such as image analysis, text detection, and thumbnail generation.
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of VisualFeatureTypes values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Computer Vision client library for Java:

- [Authenticate the client](#)
- [Analyze an image](#)
- [Read printed and handwritten text](#)

Authenticate the client

NOTE

This quickstart assumes you've [created an environment variable](#) for your Computer Vision key, named

```
COMPUTER_VISION_SUBSCRIPTION_KEY
```

The following code adds a `main` method to your class and creates variables for your resource's Azure endpoint and key. You'll need to enter your own endpoint string, which you can find by checking the **Overview** section of the Azure portal.

```
public static void main(String[] args) {  
    // Add your Computer Vision subscription key and endpoint to your environment  
    // variables.  
    // After setting, close and then re-open your command shell or project for the  
    // changes to take effect.  
    String subscriptionKey = System.getenv("COMPUTER_VISION_SUBSCRIPTION_KEY");  
    String endpoint = System.getenv("COMPUTER_VISION_ENDPOINT");
```

Next, add the following code to create a [ComputerVisionClient](#) object and passes it into other method(s), which you'll define later.

```
ComputerVisionClient compVisClient =  
ComputerVisionManager.authenticate(subscriptionKey).withEndpoint(endpoint);  
// END - Create an authenticated Computer Vision client.  
  
System.out.println("\nAzure Cognitive Services Computer Vision - Java Quickstart Sample");  
  
// Analyze local and remote images  
AnalyzeLocalImage(compVisClient);  
  
// Read from local file  
ReadFromFile(compVisClient, READ_SAMPLE_FILE_RELATIVE_PATH);
```

NOTE

If you created the environment variable after you launched the application, you'll need to close and reopen the editor, IDE, or shell running it to access the variable.

Analyze an image

The following code defines a method, `AnalyzeLocalImage`, which uses the client object to analyze a local image and print the results. The method returns a text description, categorization, list of tags, detected faces, adult content flags, main colors, and image type.

Set up test image

First, create a `resources/` folder in the `src/main/` folder of your project, and add an image you'd like to analyze. Then add the following method definition to your `ComputerVisionQuickstarts` class. If necessary, change the value of the `pathToLocalImage` to match your image file.

```
public static void AnalyzeLocalImage(ComputerVisionClient compVisClient) {
    /*
     * Analyze a local image:
     *
     * Set a string variable equal to the path of a local image. The image path
     * below is a relative path.
     */
    String pathToLocalImage = "src\\main\\resources\\myImage.png";
}
```

NOTE

You can also analyze a remote image using its URL. See the sample code on [GitHub](#) for scenarios involving remote images.

Specify visual features

Next, specify which visual features you'd like to extract in your analysis. See the [VisualFeatureTypes](#) enum for a complete list.

```
// This list defines the features to be extracted from the image.
List<VisualFeatureTypes> featuresToExtractFromLocalImage = new ArrayList<>();
featuresToExtractFromLocalImage.add(VisualFeatureTypes.DESRIPTION);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.CATEGORIES);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.TAGS);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.FACES);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.ADULT);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.COLOR);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.IMAGE_TYPE);
```

Analyze

This method prints detailed results to the console for each scope of image analysis. We recommend you surround this method call in a Try/Catch block. The **analyzeImageInStream** method returns an **ImageAnalysis** object that contains all of extracted information.

```
// Need a byte array for analyzing a local image.
File rawImage = new File(pathToLocalImage);
byte[] imageByteArray = Files.readAllBytes(rawImage.toPath());

// Call the Computer Vision service and tell it to analyze the loaded image.
ImageAnalysis analysis = compVisClient.computerVision().analyzeImageInStream().withImage(imageByteArray)
    .withVisualFeatures(featuresToExtractFromLocalImage).execute();
```

The following sections show how to parse this information in detail.

Get image description

The following code gets the list of generated captions for the image. For more information, see [Describe images](#).

```
// Display image captions and confidence values.
System.out.println("\nCaptions: ");
for (ImageCaption caption : analysis.description().captions()) {
    System.out.printf("\'%s\' with confidence %f\n", caption.text(), caption.confidence());
}
```

Get image category

The following code gets the detected category of the image. For more information, see [Categorize images](#).

```
// Display image category names and confidence values.
System.out.println("\nCategories: ");
for (Category category : analysis.categories()) {
    System.out.printf("\'%s\' with confidence %f\n", category.name(), category.score());
}
```

Get image tags

The following code gets the set of detected tags in the image. For more information, see [Content tags](#).

```
// Display image tags and confidence values.
System.out.println("\nTags: ");
for (ImageTag tag : analysis.tags()) {
    System.out.printf("\'%s\' with confidence %f\n", tag.name(), tag.confidence());
}
```

Detect faces

The following code returns the detected faces in the image with their rectangle coordinates and selects face attributes. For more information, see [Face detection](#).

```
// Display any faces found in the image and their location.
System.out.println("\nFaces: ");
for (FaceDescription face : analysis.faces()) {
    System.out.printf("\'%s\' of age %d at location (%d, %d), (%d, %d)\n", face.gender(), face.age(),
        face.faceRectangle().left(), face.faceRectangle().top(),
        face.faceRectangle().left() + face.faceRectangle().width(),
        face.faceRectangle().top() + face.faceRectangle().height());
}
```

Detect adult, racy, or gory content

The following code prints the detected presence of adult content in the image. For more information, see [Adult, racy, gory content](#).

```
// Display whether any adult or racy content was detected and the confidence
// values.
System.out.println("\nAdult: ");
System.out.printf("Is adult content: %b with confidence %f\n", analysis.adult().isAdultContent(),
    analysis.adult().adultScore());
System.out.printf("Has racy content: %b with confidence %f\n", analysis.adult().isRacyContent(),
    analysis.adult().racyScore());
```

Get image color scheme

The following code prints the detected color attributes in the image, like the dominant colors and accent color. For more information, see [Color schemes](#).

```
// Display the image color scheme.
System.out.println("\nColor scheme: ");
System.out.println("Is black and white: " + analysis.color().isBWImg());
System.out.println("Accent color: " + analysis.color().accentColor());
System.out.println("Dominant background color: " + analysis.color().dominantColorBackground());
System.out.println("Dominant foreground color: " + analysis.color().dominantColorForeground());
System.out.println("Dominant colors: " + String.join(", ", analysis.color().dominantColors()));
```

Get domain-specific content

Computer Vision can use specialized model to do further analysis on images. For more information, see [Domain-specific content](#).

The following code parses data about detected celebrities in the image.

```
// Display any celebrities detected in the image and their locations.
System.out.println("\nCelebrities: ");
for (Category category : analysis.categories()) {
    if (category.detail() != null && category.detail().celebrities() != null) {
        for (CelebritiesModel celeb : category.detail().celebrities()) {
            System.out.printf("\'%s\' with confidence %f at location (%d, %d), (%d, %d)\n", celeb.name(),
                celeb.confidence(), celeb.faceRectangle().left(), celeb.faceRectangle().top(),
                celeb.faceRectangle().left() + celeb.faceRectangle().width(),
                celeb.faceRectangle().top() + celeb.faceRectangle().height());
        }
    }
}
```

The following code parses data about detected landmarks in the image.

```
// Display any landmarks detected in the image and their locations.
System.out.println("\nLandmarks: ");
for (Category category : analysis.categories()) {
    if (category.detail() != null && category.detail().landmarks() != null) {
        for (LandmarksModel landmark : category.detail().landmarks()) {
            System.out.printf("\'%s\' with confidence %f\n", landmark.name(), landmark.confidence());
        }
    }
}
```

Get the image type

The following code prints information about the type of image—whether it is clip art or line drawing.

```
// Display what type of clip art or line drawing the image is.
System.out.println("\nImage type:");
System.out.println("Clip art type: " + analysis.imageType().clipArtType());
System.out.println("Line drawing type: " + analysis.imageType().lineDrawingType());
```

Read printed and handwritten text

Computer Vision can read visible text in an image and convert it to a character stream. This section defines a method, `ReadFromFile`, that takes a local file path and prints the image's text to the console.

NOTE

You can also read text in a remote image using its URL. See the sample code on [GitHub](#) for scenarios involving remote images.

Set up test image

Create a `resources/` folder in the `src/main/` folder of your project, and add an image you'd like to read text from. You can download a [sample image](#) to use here.

Then add the following method definition to your `ComputerVisionQuickstarts` class. If necessary, change the value of the `localFilePath` to match your image file.

```

/**
 * READ : Performs a Read Operation on a local image
 * @param client instantiated vision client
 * @param localFilePath local file path from which to perform the read operation against
 */
private static void ReadFromFile(ComputerVisionClient client) {
    System.out.println("-----");

    String localFilePath = "src\\main\\resources\\myImage.png";
    System.out.println("Read with local file: " + localFilePath);
}

```

Call the Read API

Then, add the following code to call the `readInStreamWithServiceResponseAsync` method for the given image.

```

try {
    File rawImage = new File(localFilePath);
    byte[] localImageBytes = Files.readAllBytes(rawImage.toPath());

    // Cast Computer Vision to its implementation to expose the required methods
    ComputerVisionImpl vision = (ComputerVisionImpl) client.computerVision();

    // Read in remote image and response header
    ReadInStreamHeaders responseHeader =
        vision.readInStreamWithServiceResponseAsync(localImageBytes, OcrDetectionLanguage.FR)
            .toBlocking()
            .single()
            .headers();
}

```

The following block of code extracts the operation ID from the response of the Read call. It uses this ID with a helper method to print the text read results to the console.

```

// Extract the operationLocation from the response header
String operationLocation = responseHeader.operationLocation();
System.out.println("Operation Location:" + operationLocation);

getAndPrintReadResult(vision, operationLocation);

```

Close out the try/catch block and the method definition.

```

    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}

```

Get Read results

Then, add a definition for the helper method. This method uses the operation ID from the previous step to query the read operation and get OCR results when they're available.

```

/**
 * Polls for Read result and prints results to console
 * @param vision Computer Vision instance
 * @return operationLocation returned in the POST Read response header
 */
private static void getAndPrintReadResult(ComputerVision vision, String operationLocation) throws
InterruptedException {
    System.out.println("Polling for Read results ...");

    // Extract OperationId from Operation Location
    String operationId = extractOperationIdFromOpLocation(operationLocation);

    boolean pollForResult = true;
    ReadOperationResult readResults = null;

    while (pollForResult) {
        // Poll for result every second
        Thread.sleep(1000);
        readResults = vision.getReadResult(UUID.fromString(operationId));

        // The results will no longer be null when the service has finished processing the request.
        if (readResults != null) {
            // Get request status
            OperationStatusCodes status = readResults.status();

            if (status == OperationStatusCodes.FAILED || status == OperationStatusCodes.SUCCEEDED) {
                pollForResult = false;
            }
        }
    }
}

```

The rest of the method parses the OCR results and prints them to the console.

```

// Print read results, page per page
for (ReadResult pageResult : readResults.analyzeResult().readResults()) {
    System.out.println("");
    System.out.println("Printing Read results for page " + pageResult.page());
    StringBuilder builder = new StringBuilder();

    for (Line line : pageResult.lines()) {
        builder.append(line.text());
        builder.append("\n");
    }

    System.out.println(builder.toString());
}
}

```

Finally, add the other helper method used above, which extracts the operation ID from the initial response.


```

/**
 * Extracts the OperationId from a Operation-Location returned by the POST Read operation
 * @param operationLocation
 * @return operationId
 */
private static String extractOperationIdFromOpLocation(String operationLocation) {
    if (operationLocation != null && !operationLocation.isEmpty()) {
        String[] splits = operationLocation.split("/");

        if (splits != null && splits.length > 0) {
            return splits[splits.length - 1];
        }
        throw new IllegalStateException("Something went wrong: Couldn't extract the operation id from the operation location");
    }
}

```

Run the application

You can build the app with:

```
gradle build
```

Run the application with the `gradle run` command:

```
gradle 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](#)

Next steps

In this quickstart, you learned how to use the Computer Vision Java library to do basis tasks. Next, explore the reference documentation to learn more about the library.

[Computer Vision reference \(Java\)](#)

- [What is Computer Vision?](#)
- The source code for this sample can be found on [GitHub](#).

[Reference documentation](#) | [Library source code](#) | [Package \(npm\)](#) | [Samples](#)

Prerequisites

- An Azure subscription - [Create one for free](#)
- The current version of [Node.js](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.

- You can use the free pricing tier (F0) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Setting up

Create a new Node.js application

In a console window (such as cmd, PowerShell, or Bash), create a new directory for your app, and navigate to it.

```
mkdir myapp && cd myapp
```

Run the `npm init` command to create a node application with a `package.json` file.

```
npm init
```

Install the client library

Install the `ms-rest-azure` and `@azure/cognitiveservices-computervision` NPM packages:

```
npm install @azure/cognitiveservices-computervision
```

Your app's `package.json` file will be updated with the dependencies.

Prepare the Node.js script

Create a new file, `index.js`, and open it in a text editor. Add the following import statements.

```
'use strict';

const async = require('async');
const fs = require('fs');
const https = require('https');
const path = require("path");
const createReadStream = require('fs').createReadStream
const sleep = require('util').promisify(setTimeout);
const ComputerVisionClient = require('@azure/cognitiveservices-computervision').ComputerVisionClient;
const ApiKeyCredentials = require('@azure/ms-rest-js').ApiKeyCredentials;
```

Then, define a function `computerVision` and declare an async series with primary function and callback function. You will add your quickstart code into the primary function, and call `computerVision` at the bottom of the script.

```
function computerVision() {
  async.series([
    async function () {
```

```

    },
    function () {
        return new Promise((resolve) => {
            resolve();
        })
    }
], (err) => {
    throw (err);
});
}

computerVision();

```

Object model

The following classes and interfaces handle some of the major features of the Computer Vision Node.js SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to do most image operations.
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of VisualFeatureTypes values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Computer Vision client library for Node.js:

- [Authenticate the client](#)
- [Analyze an image](#)
- [Read printed and handwritten text](#)

Authenticate the client

Create variables for your resource's Azure endpoint and key. 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.

```

/**
 * AUTHENTICATE
 * This single client is used for all examples.
 */
const key = process.env['COMPUTER_VISION_SUBSCRIPTION_KEY'];
const endpoint = process.env['COMPUTER_VISION_ENDPOINT']
if (!key) { throw new Error('Set your environment variables for your subscription key in
COMPUTER_VISION_SUBSCRIPTION_KEY and endpoint in COMPUTER_VISION_ENDPOINT.')} }

```

Instantiate a client with your endpoint and key. Create a [ApiKeyCredentials](#) object with your key and endpoint, and use it to create a [ComputerVisionClient](#) object.

```
const computerVisionClient = new ComputerVisionClient(  
  new ApiKeyCredentials({ inHeader: { 'Ocp-Apim-Subscription-Key': key } }), endpoint);
```

Analyze an image

The code in this section analyzes remote images to extract various visual features. You can do these operations as part of the **analyzeImage** method of the client object, or you can call them using individual methods. See the [reference documentation](#) for details.

NOTE

You can also analyze a local image. See the sample code on [GitHub](#) for scenarios involving local images.

Get image description

The following code gets the list of generated captions for the image. See [Describe images](#) for more details.

First, define the URL of an image to analyze:

```
const describeURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/celebrities.jpg';
```

Then add the following code to get the image description and print it to the console.

```
// Analyze URL image  
console.log('Analyzing URL image to describe...', describeURL.split('/').pop());  
const caption = (await computerVisionClient.describeImage(describeURL)).captions[0];  
console.log(`This may be ${caption.text} (${caption.confidence.toFixed(2)} confidence)`);
```

Get image category

The following code gets the detected category of the image. See [Categorize images](#) for more details.

```
const categoryURLImage = 'https://moderatorsampleimages.blob.core.windows.net/samples/sample16.png';  
  
// Analyze URL image  
console.log('Analyzing category in image...', categoryURLImage.split('/').pop());  
const categories = (await computerVisionClient.analyzeImage(categoryURLImage)).categories;  
console.log(`Categories: ${formatCategories(categories)}`);
```

Define the helper function `formatCategories`:

```
// Formats the image categories  
function formatCategories(categories) {  
  categories.sort((a, b) => b.score - a.score);  
  return categories.map(cat => `${cat.name} (${cat.score.toFixed(2)})`).join(', ');  
}
```

Get image tags

The following code gets the set of detected tags in the image. See [Content tags](#) for more details.

```

console.log('-----');
console.log('DETECT TAGS');
console.log();

// Image of different kind of dog.
const tagsURL = 'https://moderatorsampleimages.blob.core.windows.net/samples/sample16.png';

// Analyze URL image
console.log('Analyzing tags in image...', tagsURL.split('/').pop());
const tags = (await computerVisionClient.analyzeImage(tagsURL, { visualFeatures: ['Tags'] })).tags;
console.log(`Tags: ${formatTags(tags)}`);

```

Define the helper function `formatTags`:

```

// Format tags for display
function formatTags(tags) {
  return tags.map(tag => (`${tag.name} (${tag.confidence.toFixed(2)})`).join(', '));
}

```

Detect objects

The following code detects common objects in the image and prints them to the console. See [Object detection](#) for more details.

```

// Image of a dog
const objectURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-node-sdk-samples/master/Data/image.jpg';

// Analyze a URL image
console.log('Analyzing objects in image...', objectURL.split('/').pop());
const objects = (await computerVisionClient.analyzeImage(objectURL, { visualFeatures: ['Objects'] })).objects;
console.log();

// Print objects bounding box and confidence
if (objects.length) {
  console.log(`${objects.length} object${objects.length == 1 ? '' : 's'} found:`);
  for (const obj of objects) { console.log(`    ${obj.object} (${obj.confidence.toFixed(2)}) at ${formatRectObjects(obj.rectangle)}`); }
} else { console.log('No objects found.')}

```

Define the helper function `formatRectObjects` to return the top, left, bottom, and right coordinates, along with the width and height.

```

// Formats the bounding box
function formatRectObjects(rect) {
  return `top=${rect.y}`.padEnd(10) + `left=${rect.x}`.padEnd(10) + `bottom=${rect.y + rect.h}`.padEnd(12)
    + `right=${rect.x + rect.w}`.padEnd(10) + `(${rect.w}x${rect.h})`;
}

```

Detect brands

The following code detects corporate brands and logos in the image and prints them to the console. See [Brand detection](#) for more details.

```
const brandURLImage = 'https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/images/red-shirt-logo.jpg';

// Analyze URL image
console.log('Analyzing brands in image...', brandURLImage.split('/').pop());
const brands = (await computerVisionClient.analyzeImage(brandURLImage, { visualFeatures: ['Brands'] })).brands;

// Print the brands found
if (brands.length) {
  console.log(`${brands.length} brand${brands.length !== 1 ? 's' : ''} found:`);
  for (const brand of brands) {
    console.log(`    ${brand.name} (${brand.confidence.toFixed(2)} confidence)`);
  }
} else { console.log(`No brands found.`); }
```

Detect faces

The following code returns the detected faces in the image with their rectangle coordinates and select face attributes. See [Face detection](#) for more details.

```
const facesImageURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/faces.jpg';

// Analyze URL image.
console.log('Analyzing faces in image...', facesImageURL.split('/').pop());
// Get the visual feature for 'Faces' only.
const faces = (await computerVisionClient.analyzeImage(facesImageURL, { visualFeatures: ['Faces'] })).faces;

// Print the bounding box, gender, and age from the faces.
if (faces.length) {
  console.log(`${faces.length} face${faces.length === 1 ? '' : 's'} found:`);
  for (const face of faces) {
    console.log(`    Gender: ${face.gender}`.padEnd(20)
      + `    Age: ${face.age}`.padEnd(10) + `at ${formatRectFaces(face.faceRectangle)}`);
  }
} else { console.log('No faces found.');
```

Define the helper function `formatRectFaces`:

```
// Formats the bounding box
function formatRectFaces(rect) {
  return `top=${rect.top}`.padEnd(10) + `left=${rect.left}`.padEnd(10) + `bottom=${rect.top +
rect.height}`.padEnd(12)
  + `right=${rect.left + rect.width}`.padEnd(10) + `(${rect.width}x${rect.height})`;
}
```

Detect adult, racy, or gory content

The following code prints the detected presence of adult content in the image. See [Adult, racy, gory content](#) for more details.

Define the URL of the image to use:

```
// The URL image and local images are not racy/adult.
// Try your own racy/adult images for a more effective result.
const adultURLImage = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/celebrities.jpg';
```

Then add the following code to detect adult content and print the results to the console.

```
// Function to confirm racy or not
const isIt = flag => flag ? 'is' : "isn't";

// Analyze URL image
console.log('Analyzing image for racy/adult content...', adultURLImage.split('/').pop());
const adult = (await computerVisionClient.analyzeImage(adultURLImage, {
  visualFeatures: ['Adult']
})).adult;
console.log(`This probably ${isIt(adult.isAdultContent)} adult content (${adult.adultScore.toFixed(4)} score)`);
console.log(`This probably ${isIt(adult.isRacyContent)} racy content (${adult.racyScore.toFixed(4)} score)`);
```

Get image color scheme

The following code prints the detected color attributes in the image, like the dominant colors and accent color. See [Color schemes](#) for more details.

```
const colorURLImage = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/celebrities.jpg';

// Analyze URL image
console.log('Analyzing image for color scheme...', colorURLImage.split('/').pop());
console.log();
const color = (await computerVisionClient.analyzeImage(colorURLImage, { visualFeatures: ['Color'] })).color;
printColorScheme(color);
```

Define the helper function `printColorScheme` to print the details of the color scheme to the console.

```
// Print a detected color scheme
function printColorScheme(colors) {
  console.log(`Image is in ${colors.isBwImg ? 'black and white' : 'color'}`);
  console.log(`Dominant colors: ${colors.dominantColors.join(', ')}`);
  console.log(`Dominant foreground color: ${colors.dominantColorForeground}`);
  console.log(`Dominant background color: ${colors.dominantColorBackground}`);
  console.log(`Suggested accent color: #${colors.accentColor}`);
}
```

Get domain-specific content

Computer Vision can use specialized model to do further analysis on images. See [Domain-specific content](#) for more details.

First, define the URL of an image to analyze:

```
const domainURLImage = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/landmark.jpg';
```

The following code parses data about detected landmarks in the image.

```
// Analyze URL image
console.log('Analyzing image for landmarks...', domainURLImage.split('/').pop());
const domain = (await computerVisionClient.analyzeImageByDomain('landmarks',
domainURLImage)).result.landmarks;

// Prints domain-specific, recognized objects
if (domain.length) {
  console.log(`${domain.length} ${domain.length == 1 ? 'landmark' : 'landmarks'} found:`);
  for (const obj of domain) {
    console.log(`    ${obj.name}`.padEnd(20) + `(${obj.confidence.toFixed(2)} confidence)`
    .padEnd(20) + `${formatRectDomain(obj.faceRectangle)}`);
  }
} else {
  console.log('No landmarks found.');
```

Define the helper function `formatRectDomain` to parse the location data about detected landmarks.

```
// Formats bounding box
function formatRectDomain(rect) {
  if (!rect) return '';
  return `top=${rect.top}`.padEnd(10) + `left=${rect.left}`.padEnd(10) + `bottom=${rect.top +
rect.height}`.padEnd(12) +
  `right=${rect.left + rect.width}`.padEnd(10) + `(${rect.width}x${rect.height})`;
}
```

Get the image type

The following code prints information about the type of image—whether it is clip art or line drawing.

```
const typeURLImage = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-python-sdk-
samples/master/samples/vision/images/make_things_happen.jpg';

// Analyze URL image
console.log('Analyzing type in image...', typeURLImage.split('/').pop());
const types = (await computerVisionClient.analyzeImage(typeURLImage, { visualFeatures: ['ImageType']
})).imageType;
console.log('Image appears to be ${describeType(types)}');
```

Define the helper function `describeType`:

```
function describeType(imageType) {
  if (imageType.clipArtType && imageType.clipArtType > imageType.lineDrawingType) return 'clip art';
  if (imageType.lineDrawingType && imageType.clipArtType < imageType.lineDrawingType) return 'a line
drawing';
  return 'a photograph';
}
```

Extract text (OCR) with Read

Computer Vision can extract the visible text in an image and convert it to a character stream. This sample uses the Read operations.

NOTE

You can also read text from a local image. See the sample code on [GitHub](#) for scenarios involving local images.

Set up test images

Save a reference of the URL of the images you want to extract text from.

```
// URL images containing printed and/or handwritten text.  
// The URL can point to image files (.jpg/.png/.bmp) or multi-page files (.pdf, .tiff).  
const printedTextSampleURL = 'https://moderatorsampleimages.blob.core.windows.net/samples/sample2.jpg';  
const multiLingualTextURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/MultiLingual.png';  
const mixedMultiPagePDFURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/MultiPageHandwrittenForm.pdf';
```

Call the Read API

Add the code below, which calls the `readTextFromURL` and `readTextFromFile` functions for the given images.

```
// Recognize text in printed image from a URL  
console.log('Read printed text from URL...', printedTextSampleURL.split('/').pop());  
const printedResult = await readTextFromURL(computerVisionClient, printedTextSampleURL);  
printRecText(printedResult);  
  
// Recognize text in handwritten image from a local file  
  
const handwrittenImageLocalPath = __dirname + '\\handwritten_text.jpg';  
console.log('\nRead handwritten text from local file...', handwrittenImageLocalPath);  
const handwritingResult = await readTextFromFile(computerVisionClient, handwrittenImageLocalPath);  
printRecText(handwritingResult);  
  
// Recognize multi-lingual text in a PNG from a URL  
console.log('\nRead printed multi-lingual text in a PNG from URL...', multiLingualTextURL.split('/').pop());  
const multiLingualResult = await readTextFromURL(computerVisionClient, multiLingualTextURL);  
printRecText(multiLingualResult);  
  
// Recognize printed text and handwritten text in a PDF from a URL  
console.log('\nRead printed and handwritten text from a PDF from URL...',  
mixedMultiPagePDFURL.split('/').pop());  
const mixedPdfResult = await readTextFromURL(computerVisionClient, mixedMultiPagePDFURL);  
printRecText(mixedPdfResult);
```

Define the `readTextFromURL` and `readTextFromFile` functions. These call the `read` and `readInStream` methods on the client object, which return an operation ID and start an asynchronous process to read the content of the image. Then they use the operation ID to check the operation status until the results are returned. They then return the extracted results.

```
// Perform read and await the result from URL
async function readTextFromURL(client, url) {
  // To recognize text in a local image, replace client.read() with readTextInStream() as shown:
  let result = await client.read(url);
  // Operation ID is last path segment of operationLocation (a URL)
  let operation = result.operationLocation.split('/').slice(-1)[0];

  // Wait for read recognition to complete
  // result.status is initially undefined, since it's the result of read
  while (result.status !== STATUS_SUCCEEDED) { await sleep(1000); result = await
client.getReadResult(operation); }
  return result.analyzeResult.readResults; // Return the first page of result. Replace [0] with the desired
page if this is a multi-page file such as .pdf or .tiff.
}

// Perform read and await the result from local file
async function readTextFromFile(client, localImagePath) {
  // To recognize text in a local image, replace client.read() with readTextInStream() as shown:
  let result = await client.readInStream(() => createReadStream(localImagePath));
  // Operation ID is last path segment of operationLocation (a URL)
  let operation = result.operationLocation.split('/').slice(-1)[0];

  // Wait for read recognition to complete
  // result.status is initially undefined, since it's the result of read
  while (result.status !== STATUS_SUCCEEDED) { await sleep(1000); result = await
client.getReadResult(operation); }
  return result.analyzeResult.readResults; // Return the first page of result. Replace [0] with the desired
page if this is a multi-page file such as .pdf or .tiff.
}
```

Then, define the helper function `printRecText`, which prints the results of the Read operations to the console.

```
// Prints all text from Read result
function printRecText(readResults) {
  console.log('Recognized text:');
  for (const page in readResults) {
    if (readResults.length > 1) {
      console.log(`=== Page: ${page}`);
    }
    const result = readResults[page];
    if (result.lines.length) {
      for (const line of result.lines) {
        console.log(line.words.map(w => w.text).join(' '));
      }
    }
    else { console.log('No recognized text.')}
  }
}
```

Run the application

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

```
node index.js
```

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](#)

Next steps

[Computer Vision API reference \(Node.js\)](#)

- [What is Computer Vision?](#)
- The source code for this sample can be found on [GitHub](#).

[Reference documentation](#) | [Library source code](#) | [Package \(PiPy\)](#) | [Samples](#)

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Python 3.x](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (F0) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

NOTE

You can download the [full source code for the samples](#) presented below, and examples of every function available from `ComputerVisionClient`.

Setting up

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.

```
from azure.cognitiveservices.vision.computervision import ComputerVisionClient
from azure.cognitiveservices.vision.computervision.models import OperationStatusCodes
from azure.cognitiveservices.vision.computervision.models import VisualFeatureTypes
from msrest.authentication import CognitiveServicesCredentials

from array import array
import os
from PIL import Image
import sys
import time
```

Then, create variables for your resource's Azure endpoint and key.

```
# Add your Computer Vision subscription key to your environment variables.
if 'COMPUTER_VISION_SUBSCRIPTION_KEY' in os.environ:
    subscription_key = os.environ['COMPUTER_VISION_SUBSCRIPTION_KEY']
else:
    print("\nSet the COMPUTER_VISION_SUBSCRIPTION_KEY environment variable.\n**Restart your shell or IDE for changes to take effect.**")
    sys.exit()
# Add your Computer Vision endpoint to your environment variables.
if 'COMPUTER_VISION_ENDPOINT' in os.environ:
    endpoint = os.environ['COMPUTER_VISION_ENDPOINT']
else:
    print("\nSet the COMPUTER_VISION_ENDPOINT environment variable.\n**Restart your shell or IDE for changes to take effect.**")
    sys.exit()
```

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-computervision
```

Object model

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

NAME	DESCRIPTION
ComputerVisionClientOperationsMixin	This class directly handles all of the image operations, such as image analysis, text detection, and thumbnail generation.
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to produce instances of other classes. It implements ComputerVisionClientOperationsMixin .
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of VisualFeatureTypes values depending on your needs.

Code examples

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

- [Authenticate the client](#)
- [Analyze an image](#)
- [Read printed and handwritten text](#)

Authenticate the client

NOTE

This quickstart assumes you've [created an environment variable](#) for your Computer Vision key, named

```
COMPUTER_VISION_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 [ComputerVisionClient](#) object.

```
computervision_client = ComputerVisionClient(endpoint, CognitiveServicesCredentials(subscription_key))
```

Analyze an image

Save a reference to the URL of an image you want to analyze.

```
remote_image_url = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/landmark.jpg"
```

Get image description

The following code gets the list of generated captions for the image. See [Describe images](#) for more details.

```
'''
Describe an Image - remote
This example describes the contents of an image with the confidence score.
'''

print("==== Describe an image - remote =====")
# Call API
description_results = computervision_client.describe_image(remote_image_url )

# Get the captions (descriptions) from the response, with confidence level
print("Description of remote image: ")
if (len(description_results.captions) == 0):
    print("No description detected.")
else:
    for caption in description_results.captions:
        print('{}' with confidence {:.2f}%".format(caption.text, caption.confidence * 100))
```

Get image category

The following code gets the detected category of the image. See [Categorize images](#) for more details.

```
'''
Categorize an Image - remote
This example extracts (general) categories from a remote image with a confidence score.
'''

print("==== Categorize an image - remote =====")
# Select the visual feature(s) you want.
remote_image_features = ["categories"]
# Call API with URL and features
categorize_results_remote = computervision_client.analyze_image(remote_image_url , remote_image_features)

# Print results with confidence score
print("Categories from remote image: ")
if (len(categorize_results_remote.categories) == 0):
    print("No categories detected.")
else:
    for category in categorize_results_remote.categories:
        print('{}' with confidence {:.2f}%".format(category.name, category.score * 100))
```

Get image tags

The following code gets the set of detected tags in the image. See [Content tags](#) for more details.

```
'''
Tag an Image - remote
This example returns a tag (key word) for each thing in the image.
'''

print("==== Tag an image - remote ====")
# Call API with remote image
tags_result_remote = computervision_client.tag_image(remote_image_url )

# Print results with confidence score
print("Tags in the remote image: ")
if (len(tags_result_remote.tags) == 0):
    print("No tags detected.")
else:
    for tag in tags_result_remote.tags:
        print("{}' with confidence {:.2f}%".format(tag.name, tag.confidence * 100))
```

Detect objects

The following code detects common objects in the image and prints them to the console. See [Object detection](#) for more details.

```
'''
Detect Objects - remote
This example detects different kinds of objects with bounding boxes in a remote image.
'''

print("==== Detect Objects - remote ====")
# Get URL image with different objects
remote_image_url_objects = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/objects.jpg"
# Call API with URL
detect_objects_results_remote = computervision_client.detect_objects(remote_image_url_objects)

# Print detected objects results with bounding boxes
print("Detecting objects in remote image:")
if len(detect_objects_results_remote.objects) == 0:
    print("No objects detected.")
else:
    for object in detect_objects_results_remote.objects:
        print("object at location {}, {}, {}, {}".format( \
            object.rectangle.x, object.rectangle.x + object.rectangle.w, \
            object.rectangle.y, object.rectangle.y + object.rectangle.h))
```

Detect brands

The following code detects corporate brands and logos in the image and prints them to the console. See [Brand detection](#) for more details.

```

'''
Detect Brands - remote
This example detects common brands like logos and puts a bounding box around them.
'''

print("==== Detect Brands - remote =====")
# Get a URL with a brand logo
remote_image_url = "https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/images/gray-shirt-logo.jpg"
# Select the visual feature(s) you want
remote_image_features = ["brands"]
# Call API with URL and features
detect_brands_results_remote = computervision_client.analyze_image(remote_image_url, remote_image_features)

print("Detecting brands in remote image: ")
if len(detect_brands_results_remote.brands) == 0:
    print("No brands detected.")
else:
    for brand in detect_brands_results_remote.brands:
        print("{}' brand detected with confidence {:.1f}% at location {}, {}, {}, {}".format( \
            brand.name, brand.confidence * 100, brand.rectangle.x, brand.rectangle.x + brand.rectangle.w, \
            brand.rectangle.y, brand.rectangle.y + brand.rectangle.h))

```

Detect faces

The following code returns the detected faces in the image with their rectangle coordinates and select face attributes. See [Face detection](#) for more details.

```

'''
Detect Faces - remote
This example detects faces in a remote image, gets their gender and age,
and marks them with a bounding box.
'''

print("==== Detect Faces - remote =====")
# Get an image with faces
remote_image_url_faces = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/faces.jpg"
# Select the visual feature(s) you want.
remote_image_features = ["faces"]
# Call the API with remote URL and features
detect_faces_results_remote = computervision_client.analyze_image(remote_image_url_faces,
remote_image_features)

# Print the results with gender, age, and bounding box
print("Faces in the remote image: ")
if (len(detect_faces_results_remote.faces) == 0):
    print("No faces detected.")
else:
    for face in detect_faces_results_remote.faces:
        print("{}' of age {} at location {}, {}, {}, {}".format(face.gender, face.age, \
            face.face_rectangle.left, face.face_rectangle.top, \
            face.face_rectangle.left + face.face_rectangle.width, \
            face.face_rectangle.top + face.face_rectangle.height))

```

Detect adult, racy, or gory content

The following code prints the detected presence of adult content in the image. See [Adult, racy, gory content](#) for more details.

```
'''
Detect Adult or Racy Content - remote
This example detects adult or racy content in a remote image, then prints the adult/racy score.
The score is ranged 0.0 - 1.0 with smaller numbers indicating negative results.
'''

print("==== Detect Adult or Racy Content - remote ====")
# Select the visual feature(s) you want
remote_image_features = ["adult"]
# Call API with URL and features
detect_adult_results_remote = computervision_client.analyze_image(remote_image_url, remote_image_features)

# Print results with adult/racy score
print("Analyzing remote image for adult or racy content ... ")
print("Is adult content: {} with confidence {:.2f}".format(detect_adult_results_remote.adult.is_adult_content,
detect_adult_results_remote.adult.adult_score * 100))
print("Has racy content: {} with confidence {:.2f}".format(detect_adult_results_remote.adult.is_racy_content,
detect_adult_results_remote.adult.racy_score * 100))
```

Get image color scheme

The following code prints the detected color attributes in the image, like the dominant colors and accent color. See [Color schemes](#) for more details.

```
'''
Detect Color - remote
This example detects the different aspects of its color scheme in a remote image.
'''

print("==== Detect Color - remote ====")
# Select the feature(s) you want
remote_image_features = ["color"]
# Call API with URL and features
detect_color_results_remote = computervision_client.analyze_image(remote_image_url, remote_image_features)

# Print results of color scheme
print("Getting color scheme of the remote image: ")
print("Is black and white: {}".format(detect_color_results_remote.color.is_bw_img))
print("Accent color: {}".format(detect_color_results_remote.color.accent_color))
print("Dominant background color: {}".format(detect_color_results_remote.color.dominant_color_background))
print("Dominant foreground color: {}".format(detect_color_results_remote.color.dominant_color_foreground))
print("Dominant colors: {}".format(detect_color_results_remote.color.dominant_colors))
```

Get domain-specific content

Computer Vision can use specialized model to do further analysis on images. See [Domain-specific content](#) for more details.

The following code parses data about detected celebrities in the image.


```
'''
Detect Domain-specific Content - remote
This example detects celebrities and landmarks in remote images.
'''

print("==== Detect Domain-specific Content - remote ====")
# URL of one or more celebrities
remote_image_url_celebs = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/faces.jpg"
# Call API with content type (celebrities) and URL
detect_domain_results_celebs_remote = computervision_client.analyze_image_by_domain("celebrities",
remote_image_url_celebs)

# Print detection results with name
print("Celebrities in the remote image:")
if len(detect_domain_results_celebs_remote.result["celebrities"]) == 0:
    print("No celebrities detected.")
else:
    for celeb in detect_domain_results_celebs_remote.result["celebrities"]:
        print(celeb["name"])
```

The following code parses data about detected landmarks in the image.

```
# Call API with content type (landmarks) and URL
detect_domain_results_landmarks = computervision_client.analyze_image_by_domain("landmarks",
remote_image_url)
print()

print("Landmarks in the remote image:")
if len(detect_domain_results_landmarks.result["landmarks"]) == 0:
    print("No landmarks detected.")
else:
    for landmark in detect_domain_results_landmarks.result["landmarks"]:
        print(landmark["name"])
```

Get the image type

The following code prints information about the type of image—whether it is clip art or line drawing.

```

'''
Detect Image Types - remote
This example detects an image's type (clip art/line drawing).
'''

print("==== Detect Image Types - remote ====")
# Get URL of an image with a type
remote_image_url_type = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/type-image.jpg"
# Select visual feature(s) you want
remote_image_features = VisualFeatureTypes.image_type
# Call API with URL and features
detect_type_results_remote = computervision_client.analyze_image(remote_image_url_type,
remote_image_features)

# Prints type results with degree of accuracy
print("Type of remote image:")
if detect_type_results_remote.image_type.clip_art_type == 0:
    print("Image is not clip art.")
elif detect_type_results_remote.image_type.line_drawing_type == 1:
    print("Image is ambiguously clip art.")
elif detect_type_results_remote.image_type.line_drawing_type == 2:
    print("Image is normal clip art.")
else:
    print("Image is good clip art.")

if detect_type_results_remote.image_type.line_drawing_type == 0:
    print("Image is not a line drawing.")
else:
    print("Image is a line drawing")

```

Read printed and handwritten text

Computer Vision can read visible text in an image and convert it to a character stream. You do this in two parts.

Call the Read API

First, use the following code to call the **read** method for the given image. This returns an operation ID and starts an asynchronous process to read the content of the image.

```

'''
Batch Read File, recognize handwritten text - remote
This example will extract handwritten text in an image, then print results, line by line.
This API call can also recognize handwriting (not shown).
'''

print("==== Batch Read File - remote ====")
# Get an image with handwritten text
remote_image_handw_text_url = "https://raw.githubusercontent.com/MicrosoftDocs/azure-docs/master/articles/cognitive-services/Computer-vision/Images/readsample.jpg"

# Call API with URL and raw response (allows you to get the operation location)
recognize_handw_results = computervision_client.read(remote_image_handw_text_url, raw=True)

```

Get Read results

Next, get the operation ID returned from the **read** call, and use it to query the service for operation results. The following code checks the operation at one-second intervals until the results are returned. It then prints the extracted text data to the console.

```
# Get the operation location (URL with an ID at the end) from the response
operation_location_remote = recognize_handw_results.headers["Operation-Location"]
# Grab the ID from the URL
operation_id = operation_location_remote.split("/")[-1]

# Call the "GET" API and wait for it to retrieve the results
while True:
    get_handw_text_results = computervision_client.get_read_result(operation_id)
    if get_handw_text_results.status not in ['notStarted', 'running']:
        break
    time.sleep(1)

# Print the detected text, line by line
if get_handw_text_results.status == OperationStatusCodes.succeeded:
    for text_result in get_handw_text_results.analyze_result.read_results:
        for line in text_result.lines:
            print(line.text)
            print(line.bounding_box)

print()
```

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](#)

Next steps

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

[Computer Vision API reference \(Python\)](#)

- [What is Computer Vision?](#)
- The source code for this sample can be found on [GitHub](#).

[Reference documentation](#) | [Library source code](#) | [Package](#)

Prerequisites

- An Azure subscription - [Create one for free](#)
- The latest version of [Go](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Setting up

Create a Go project directory

In a console window (cmd, PowerShell, Terminal, Bash), create a new workspace for your Go project, named `my-app`, and navigate to it.

```
mkdir -p my-app/{src, bin, pkg}
cd my-app
```

Your workspace will contain three folders:

- **src** - This directory will contain source code and packages. Any packages installed with the `go get` command will go in this directory.
- **pkg** - This directory will contain the compiled Go package objects. These files all have an `.a` extension.
- **bin** - This directory will contain the binary executable files that are created when you run `go install`.

TIP

To learn more about the structure of a Go workspace, see the [Go language documentation](#). This guide includes information for setting `$GOPATH` and `$GOROOT`.

Install the client library for Go

Next, install the client library for Go:

```
go get -u https://github.com/Azure/azure-sdk-for-go/tree/master/services/cognitiveservices/v2.1/computervision
```

or if you use dep, within your repo run:

```
dep ensure -add https://github.com/Azure/azure-sdk-for-go/tree/master/services/cognitiveservices/v2.1/computervision
```

Create a Go application

Next, create a file in the `src` directory named `sample-app.go`:

```
cd src
touch sample-app.go
```

Open `sample-app.go` in your preferred IDE or text editor. Then add the package name and import the following libraries:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/Azure/azure-sdk-for-go/services/cognitiveservices/v2.0/computervision"
    "github.com/Azure/go-autorest/autorest"
    "io"
    "log"
    "os"
    "strings"
    "time"
)
```

Also, declare a context at the root of your script. You'll need this object to execute most Computer Vision function calls:

```
// Declare global so don't have to pass it to all of the tasks.
var computerVisionContext context.Context
```

Next, you'll begin adding code to carry out different Computer Vision operations.

Object model

The following classes and interfaces handle some of the major features of the Computer Vision Go SDK.

NAME	DESCRIPTION
BaseClient	This class is needed for all Computer Vision functionality, such as image analysis and text reading. You instantiate it with your subscription information, and you use it to do most image operations.
ImageAnalysis	This type contains the results of an AnalyzeImage function call. There are similar types for each of the category-specific functions.
ReadOperationResult	This type contains the results of a Batch Read operation.
VisualFeatureTypes	This type defines the different kinds of image analysis that can be done in a standard Analyze operation. You specify a set of VisualFeatureTypes values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Computer Vision client library for Go:

- [Authenticate the client](#)
- [Analyze an image](#)
- [Read printed and handwritten text](#)

Authenticate the client

NOTE

This step assumes you've [created environment variables](#) for your Computer Vision key and endpoint, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT` respectively.

Create a `main` function and add the following code to it to instantiate a client with your endpoint and key.

```
/*
 * Configure the Computer Vision client
 * Set environment variables for COMPUTER_VISION_SUBSCRIPTION_KEY and COMPUTER_VISION_ENDPOINT,
 * then restart your command shell or your IDE for changes to take effect.
 */
computerVisionKey := os.Getenv("COMPUTER_VISION_SUBSCRIPTION_KEY")

if (computerVisionKey == "") {
    log.Fatal("\n\nPlease set a COMPUTER_VISION_SUBSCRIPTION_KEY environment variable.\n" +
        "**You may need to restart your shell or IDE after it's set.**\n")
}

endpointURL := os.Getenv("COMPUTER_VISION_ENDPOINT")
if (endpointURL == "") {
    log.Fatal("\n\nPlease set a COMPUTER_VISION_ENDPOINT environment variable.\n" +
        "**You may need to restart your shell or IDE after it's set.**\n")
}

computerVisionClient := computervision.New(endpointURL);
computerVisionClient.Authorizer = autorest.NewCognitiveServicesAuthorizer(computerVisionKey)

computerVisionContext = context.Background()
/*
 * END - Configure the Computer Vision client
 */
```

Analyze an image

The following code uses the client object to analyze a remote image and print the results to the console. You can get a text description, categorization, list of tags, detected objects, detected brands, detected faces, adult content flags, main colors, and image type.

Set up test image

First save a reference to the URL of the image you want to analyze. Put this inside your `main` function.

```
landmarkImageURL := "https://github.com/Azure-Samples/cognitive-services-sample-data-
files/raw/master/ComputerVision/Images/landmark.jpg"
```

NOTE

You can also analyze a local image. See the sample code on [GitHub](#) for scenarios involving local images.

Specify visual features

The following function calls extract different visual features from the sample image. You'll define these functions in the following sections.

```
// Analyze features of an image, remote
DescribeRemoteImage(computerVisionClient, landmarkImageURL)
CategorizeRemoteImage(computerVisionClient, landmarkImageURL)
TagRemoteImage(computerVisionClient, landmarkImageURL)
DetectFacesRemoteImage(computerVisionClient, facesImageURL)
DetectObjectsRemoteImage(computerVisionClient, objectsImageURL)
DetectBrandsRemoteImage(computerVisionClient, brandsImageURL)
DetectAdultOrRacyContentRemoteImage(computerVisionClient, adultRacyImageURL)
DetectColorSchemeRemoteImage(computerVisionClient, brandsImageURL)
DetectDomainSpecificContentRemoteImage(computerVisionClient, landmarkImageURL)
DetectImageTypesRemoteImage(computerVisionClient, detectTypeImageURL)
GenerateThumbnailRemoteImage(computerVisionClient, adultRacyImageURL)
```

Get image description

The following function gets the list of generated captions for the image. For more information about image description, see [Describe images](#).

```
func DescribeRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("DESCRIBE IMAGE - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    maxNumberDescriptionCandidates := new(int32)
    *maxNumberDescriptionCandidates = 1

    remoteImageDescription, err := client.DescribeImage(
        computerVisionContext,
        remoteImage,
        maxNumberDescriptionCandidates,
        "") // language
    if err != nil { log.Fatal(err) }

    fmt.Println("Captions from remote image: ")
    if len(*remoteImageDescription.Captions) == 0 {
        fmt.Println("No captions detected.")
    } else {
        for _, caption := range *remoteImageDescription.Captions {
            fmt.Printf("%v' with confidence %.2f%%\n", *caption.Text, *caption.Confidence * 100)
        }
    }
    fmt.Println()
}
```

Get image category

The following function gets the detected category of the image. For more information, see [Categorize images](#).

```

func CategorizeRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("CATEGORIZE IMAGE - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesCategories}
    imageAnalysis, err := client.AnalyzeImage(
        computerVisionContext,
        remoteImage,
        features,
        []computervision.Details{},
        "")
    if err != nil { log.Fatal(err) }

    fmt.Println("Categories from remote image: ")
    if len(*imageAnalysis.Categories) == 0 {
        fmt.Println("No categories detected.")
    } else {
        for _, category := range *imageAnalysis.Categories {
            fmt.Printf("%v' with confidence %.2f%%\n", *category.Name, *category.Score * 100)
        }
    }
    fmt.Println()
}

```

Get image tags

The following function gets the set of detected tags in the image. For more information, see [Content tags](#).

```

func TagRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("TAG IMAGE - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    remoteImageTags, err := client.TagImage(
        computerVisionContext,
        remoteImage,
        "")
    if err != nil { log.Fatal(err) }

    fmt.Println("Tags in the remote image: ")
    if len(*remoteImageTags.Tags) == 0 {
        fmt.Println("No tags detected.")
    } else {
        for _, tag := range *remoteImageTags.Tags {
            fmt.Printf("%v' with confidence %.2f%%\n", *tag.Name, *tag.Confidence * 100)
        }
    }
    fmt.Println()
}

```

Detect objects

The following function detects common objects in the image and prints them to the console. For more information, see [Object detection](#).


```

func DetectObjectsRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("DETECT OBJECTS - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    imageAnalysis, err := client.DetectObjects(
        computerVisionContext,
        remoteImage,
    )
    if err != nil { log.Fatal(err) }

    fmt.Println("Detecting objects in remote image: ")
    if len(*imageAnalysis.Objects) == 0 {
        fmt.Println("No objects detected.")
    } else {
        // Print the objects found with confidence level and bounding box locations.
        for _, object := range *imageAnalysis.Objects {
            fmt.Printf("'%' with confidence %.2f%% at location (%v, %v), (%v, %v)\n",
                *object.Object, *object.Confidence * 100,
                *object.Rectangle.X, *object.Rectangle.X + *object.Rectangle.W,
                *object.Rectangle.Y, *object.Rectangle.Y + *object.Rectangle.H)
        }
    }
    fmt.Println()
}

```

Detect brands

The following code detects corporate brands and logos in the image and prints them to the console. For more information, [Brand detection](#).

First, declare a reference to a new image within your `main` function.

```

brandsImageURL := "https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/images/gray-shirt-logo.jpg"

```

The following code defines the brand detection function.

```

func DetectBrandsRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("DETECT BRANDS - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // Define the kinds of features you want returned.
    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesBrands}

    imageAnalysis, err := client.AnalyzeImage(
        computerVisionContext,
        remoteImage,
        features,
        []computervision.Details{},
        "en")
    if err != nil { log.Fatal(err) }

    fmt.Println("Detecting brands in remote image: ")
    if len(*imageAnalysis.Brands) == 0 {
        fmt.Println("No brands detected.")
    } else {
        // Get bounding box around the brand and confidence level it's correctly identified.
        for _, brand := range *imageAnalysis.Brands {
            fmt.Printf("'%' with confidence %.2f%% at location (%v, %v), (%v, %v)\n",
                *brand.Name, *brand.Confidence * 100,
                *brand.Rectangle.X, *brand.Rectangle.X + *brand.Rectangle.W,
                *brand.Rectangle.Y, *brand.Rectangle.Y + *brand.Rectangle.H)
        }
    }
    fmt.Println()
}

```

Detect faces

The following function returns the detected faces in the image with their rectangle coordinates and certain face attributes. For more information, see [Face detection](#).

```

func DetectFacesRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("DETECT FACES - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // Define the features you want returned with the API call.
    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesFaces}
    imageAnalysis, err := client.AnalyzeImage(
        computerVisionContext,
        remoteImage,
        features,
        []computervision.Details{},
        "")
    if err != nil { log.Fatal(err) }

    fmt.Println("Detecting faces in a remote image ...")
    if len(*imageAnalysis.Faces) == 0 {
        fmt.Println("No faces detected.")
    } else {
        // Print the bounding box locations of the found faces.
        for _, face := range *imageAnalysis.Faces {
            fmt.Printf("%v' of age %v at location (%v, %v), (%v, %v)\n",
                face.Gender, *face.Age,
                *face.FaceRectangle.Left, *face.FaceRectangle.Top,
                *face.FaceRectangle.Left + *face.FaceRectangle.Width,
                *face.FaceRectangle.Top + *face.FaceRectangle.Height)
        }
    }
    fmt.Println()
}

```

Detect adult, racy, or gory content

The following function prints the detected presence of adult content in the image. For more information, see [Adult, racy, gory content](#).

```

func DetectAdultOrRacyContentRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("DETECT ADULT OR RACY CONTENT - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // Define the features you want returned from the API call.
    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesAdult}
    imageAnalysis, err := client.AnalyzeImage(
        computerVisionContext,
        remoteImage,
        features,
        []computervision.Details{},
        "") // language, English is default
    if err != nil { log.Fatal(err) }

    // Print whether or not there is questionable content.
    // Confidence levels: low means content is OK, high means it's not.
    fmt.Println("Analyzing remote image for adult or racy content: ");
    fmt.Printf("Is adult content: %v with confidence %.2f%%\n", *imageAnalysis.Adult.IsAdultContent,
        *imageAnalysis.Adult.AdultScore * 100)
    fmt.Printf("Has racy content: %v with confidence %.2f%%\n", *imageAnalysis.Adult.IsRacyContent,
        *imageAnalysis.Adult.RacyScore * 100)
    fmt.Println()
}

```

Get image color scheme

The following function prints the detected color attributes in the image, like the dominant colors and accent color. For more information, see [Color schemes](#).

```
func DetectColorSchemeRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("DETECT COLOR SCHEME - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // Define the features you'd like returned with the result.
    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesColor}
    imageAnalysis, err := client.AnalyzeImage(
        computervisionContext,
        remoteImage,
        features,
        []computervision.Details{},
        "") // language, English is default
    if err != nil { log.Fatal(err) }

    fmt.Println("Color scheme of the remote image: ");
    fmt.Printf("Is black and white: %v\n", *imageAnalysis.Color.IsBWImg)
    fmt.Printf("Accent color: 0x%x\n", *imageAnalysis.Color.AccentColor)
    fmt.Printf("Dominant background color: %v\n", *imageAnalysis.Color.DominantColorBackground)
    fmt.Printf("Dominant foreground color: %v\n", *imageAnalysis.Color.DominantColorForeground)
    fmt.Printf("Dominant colors: %v\n", strings.Join(*imageAnalysis.Color.DominantColors, ", "))
    fmt.Println()
}
```

Get domain-specific content

Computer Vision can use specialized models to do further analysis on images. For more information, see [Domain-specific content](#).

The following code parses data about detected celebrities in the image.

```

func DetectDomainSpecificContentRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("DETECT DOMAIN-SPECIFIC CONTENT - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    fmt.Println("Detecting domain-specific content in the local image ...")

    // Check if there are any celebrities in the image.
    celebrities, err := client.AnalyzeImageByDomain(
        computerVisionContext,
        "celebrities",
        remoteImage,
        "") // language, English is default
    if err != nil { log.Fatal(err) }

    fmt.Println("\nCelebrities: ")

    // Marshal the output from AnalyzeImageByDomain into JSON.
    data, err := json.MarshalIndent(celebrities.Result, "", "\t")

    // Define structs for which to unmarshal the JSON.
    type Celebrities struct {
        Name string `json:"name"`
    }

    type CelebrityResult struct {
        Celebrities []Celebrities `json:"celebrities"`
    }

    var celebrityResult CelebrityResult

    // Unmarshal the data.
    err = json.Unmarshal(data, &celebrityResult)
    if err != nil { log.Fatal(err) }

    // Check if any celebrities detected.
    if len(celebrityResult.Celebrities) == 0 {
        fmt.Println("No celebrities detected.")
    } else {
        for _, celebrity := range celebrityResult.Celebrities {
            fmt.Printf("name: %v\n", celebrity.Name)
        }
    }
}

```

The following code parses data about detected landmarks in the image.

```

fmt.Println("\nLandmarks: ")

// Check if there are any landmarks in the image.
landmarks, err := client.AnalyzeImageByDomain(
    computerVisionContext,
    "landmarks",
    remoteImage,
    "")
if err != nil { log.Fatal(err) }

// Marshal the output from AnalyzeImageByDomain into JSON.
data, err = json.MarshalIndent(landmarks.Result, "", "\t")

// Define structs for which to unmarshal the JSON.
type Landmarks struct {
    Name string `json:"name"`
}

type LandmarkResult struct {
    Landmarks []Landmarks `json:"landmarks"`
}

var landmarkResult LandmarkResult

// Unmarshal the data.
err = json.Unmarshal(data, &landmarkResult)
if err != nil { log.Fatal(err) }

// Check if any celebrities detected.
if len(landmarkResult.Landmarks) == 0 {
    fmt.Println("No landmarks detected.")
} else {
    for _, landmark := range landmarkResult.Landmarks {
        fmt.Printf("name: %v\n", landmark.Name)
    }
}
fmt.Println()
}

```

Get the image type

The following function prints information about the type of image—whether it's clip art or a line drawing.

```

func DetectImageTypesRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("DETECT IMAGE TYPES - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesImageType}

    imageAnalysis, err := client.AnalyzeImage(
        computerVisionContext,
        remoteImage,
        features,
        []computervision.Details{},
        "")
    if err != nil { log.Fatal(err) }

    fmt.Println("Image type of remote image:")

    fmt.Println("\nClip art type: ")
    switch *imageAnalysis.ImageType.ClipArtType {
    case 0:
        fmt.Println("Image is not clip art.")
    case 1:
        fmt.Println("Image is ambiguously clip art.")
    case 2:
        fmt.Println("Image is normal clip art.")
    case 3:
        fmt.Println("Image is good clip art.")
    }

    fmt.Println("\nLine drawing type: ")
    if *imageAnalysis.ImageType.LineDrawingType == 1 {
        fmt.Println("Image is a line drawing.")
    } else {
        fmt.Println("Image is not a line drawing.")
    }
    fmt.Println()
}

```

Read printed and handwritten text

Computer Vision can read visible text in an image and convert it to a character stream. The code in this section defines a function, `RecognizeTextReadAPIRemoteImage`, which uses the client object to detect and extract printed or handwritten text in the image.

Add the sample image reference and function call in your `main` function.

```

// Analyze text in an image, remote
BatchReadFileRemoteImage(computerVisionClient, printedImageURL)

```

NOTE

You can also extract text from a local image. See the sample code on [GitHub](#) for scenarios involving local images.

Call the Read API

Define the new function for reading text, `RecognizeTextReadAPIRemoteImage`. Add the code below, which calls the `BatchReadFile` method for the given image. This method returns an operation ID and starts an asynchronous process to read the content of the image.

```

func BatchReadFileRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----")
    fmt.Println("BATCH READ FILE - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // The response contains a field called "Operation-Location",
    // which is a URL with an ID that you'll use for GetReadOperationResult to access OCR results.
    textHeaders, err := client.BatchReadFile(computerVisionContext, remoteImage)
    if err != nil { log.Fatal(err) }

    // Use ExtractHeader from the autorest library to get the Operation-Location URL
    operationLocation := autorest.ExtractHeaderValue("Operation-Location", textHeaders.Response)

    numberOfCharsInOperationId := 36
    operationId := string(operationLocation[len(operationLocation)-numberOfCharsInOperationId :
len(operationLocation)])

```

Get Read results

Next, get the operation ID returned from the **BatchReadFile** call, and use it with the **GetReadOperationResult** method to query the service for operation results. The following code checks the operation at one-second intervals until the results are returned. It then prints the extracted text data to the console.

```

readOperationResult, err := client.GetReadOperationResult(computerVisionContext, operationId)
if err != nil { log.Fatal(err) }

// Wait for the operation to complete.
i := 0
maxRetries := 10

fmt.Println("Recognizing text in a remote image with the batch Read API ...")
for readOperationResult.Status != computervision.Failed &&
    readOperationResult.Status != computervision.Succeeded {
    if i >= maxRetries {
        break
    }
    i++

    fmt.Printf("Server status: %v, waiting %v seconds...\n", readOperationResult.Status, i)
    time.Sleep(1 * time.Second)

    readOperationResult, err = client.GetReadOperationResult(computerVisionContext, operationId)
    if err != nil { log.Fatal(err) }
}

```

Display Read results

Add the following code to parse and display the retrieved text data, and finish the function definition.

```

// Display the results.
fmt.Println()
for _, recResult := range *(readOperationResult.RecognitionResults) {
    for _, line := range *recResult.Lines {
        fmt.Println(*line.Text)
    }
}

```

Run the application

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


```
go run sample-app.go
```

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](#)

Next steps

[Computer Vision API reference \(Go\)](#)

- [What is Computer Vision?](#)
- The source code for this sample can be found on [GitHub](#).

Quickstart: Analyze a remote image using the Computer Vision REST API and cURL

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you'll analyze a remotely stored image to extract visual features using the Computer Vision REST API. With the [Analyze Image](#) method, you can extract visual features based on image content.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [cURL](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Create and run the sample command

To create and run the sample, do the following steps:

1. Copy the following command into a text editor.
2. Make the following changes in the command where needed:
 - a. Replace the value of `<subscriptionKey>` with your subscription key.
 - b. Replace the first part of the request URL (`westcentralus`) with the text in your own endpoint URL.

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](#).

- c. Optionally, change the image URL in the request body (`http://upload.wikimedia.org/wikipedia/commons/3/3c/Shaki_waterfall.jpg`) to the URL of a different image to be analyzed.
3. Open a command prompt window.
 4. Paste the command from the text editor into the command prompt window, and then run the command.

```
curl -H "Ocp-Apim-Subscription-Key: <subscriptionKey>" -H "Content-Type: application/json"
"https://westcentralus.api.cognitive.microsoft.com/vision/v3.0/analyze?
visualFeatures=Categories,Description&details=Landmarks" -d "
{"url": "http://upload.wikimedia.org/wikipedia/commons/3/3c/Shaki_waterfall.jpg"}
```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the command prompt window, similar to the following example:

```

{
  "categories": [
    {
      "name": "outdoor_water",
      "score": 0.9921875,
      "detail": {
        "landmarks": []
      }
    }
  ],
  "description": {
    "tags": [
      "nature",
      "water",
      "waterfall",
      "outdoor",
      "rock",
      "mountain",
      "rocky",
      "grass",
      "hill",
      "covered",
      "hillside",
      "standing",
      "side",
      "group",
      "walking",
      "white",
      "man",
      "large",
      "snow",
      "grazing",
      "forest",
      "slope",
      "herd",
      "river",
      "giraffe",
      "field"
    ],
    "captions": [
      {
        "text": "a large waterfall over a rocky cliff",
        "confidence": 0.916458423253597
      }
    ]
  },
  "requestId": "b6e33879-abb2-43a0-a96e-02cb5ae0b795",
  "metadata": {
    "height": 959,
    "width": 1280,
    "format": "Jpeg"
  }
}

```

Next steps

Explore the Computer Vision API used to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

[Explore the Computer Vision API](#)

Quickstart: Analyze a remote image using the Computer Vision REST API with Go

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you'll analyze a remotely stored image to extract visual features using the Computer Vision REST API. With the [Analyze Image](#) method, you can extract visual features based on image content.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Go](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Copy the below code into a text editor.
2. Optionally, replace the value of `imageUrl` with the URL of a different image that you want to analyze.
3. Save the code as a file with a `.go` extension. For example, `analyze-image.go`.
4. Open a command prompt window.
5. At the prompt, run the `go build` command to compile the package from the file. For example,
`go build analyze-image.go`.
6. At the prompt, run the compiled package. For example, `analyze-image`.

```

package main

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

func main() {
    // Add your Computer Vision subscription key and endpoint to your environment variables.
    subscriptionKey := os.Getenv("COMPUTER_VISION_SUBSCRIPTION_KEY")
    endpoint := os.Getenv("COMPUTER_VISION_ENDPOINT")

    uriBase := endpoint + "vision/v3.0/analyze"
    const imageUrl =
        "https://upload.wikimedia.org/wikipedia/commons/3/3c/Shaki_waterfall.jpg"

    const params = "?visualFeatures=Description&details=Landmarks"
    uri := uriBase + params
    const imageUrlEnc = "{\"url\":\"" + imageUrl + "\"}"

    reader := strings.NewReader(imageUrlEnc)

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

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

    // Add request 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 from the byte array
    var f interface{}
    json.Unmarshal(data, &f)

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

```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the command prompt window, similar to the following example:

```
{
  "categories": [
    {
      "detail": {
        "landmarks": []
      },
      "name": "outdoor_water",
      "score": 0.9921875
    }
  ],
  "description": {
    "captions": [
      {
        "confidence": 0.916458423253597,
        "text": "a large waterfall over a rocky cliff"
      }
    ]
  },
  "tags": [
    "nature",
    "water",
    "waterfall",
    "outdoor",
    "rock",
    "mountain",
    "rocky",
    "grass",
    "hill",
    "covered",
    "hillside",
    "standing",
    "side",
    "group",
    "walking",
    "white",
    "man",
    "large",
    "snow",
    "grazing",
    "forest",
    "slope",
    "herd",
    "river",
    "giraffe",
    "field"
  ]
},
  "metadata": {
    "format": "Jpeg",
    "height": 959,
    "width": 1280
  },
  "requestId": "a92f89ab-51f8-4735-a58d-507da2213fc2"
}
```

Next steps

Explore the Computer Vision API used to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Analyze a remote image using the Computer Vision REST API and Java

9/1/2020 • 3 minutes to read • [Edit Online](#)

In this quickstart, you'll analyze a remotely stored image to extract visual features by using Java and the Computer Vision REST API. With the [Analyze Image](#) method, you can extract visual features based on image content.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Java™ Platform, Standard Edition Development Kit 7 or 8](#) (JDK 7 or 8)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (F0) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample application

To create and run the sample, do the following steps:

1. Create a new Java project in your favorite IDE or editor. If the option is available, create the Java project from a command line application template.
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:4.5.5)
 - [Apache HTTP core](#) (org.apache.httpcomponents:httpcore:4.4.9)
 - [JSON library](#) (org.json:json:20180130)
3. Add the following `import` statements to the file that contains the `Main` public class for your project.

```
import java.net.URI;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
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.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import org.json.JSONObject;
```

4. Replace the `AnalyzeImage` public class with the following code.
5. Optionally, replace the value of `imageToAnalyze` with the URL of a different image that you want to analyze.


```

public class AnalyzeImage {
    // *****
    // *** Update or verify the following values. ***
    // *****

    // Add your Computer Vision subscription key and endpoint to your environment variables.
    // After setting, close and then re-open your command shell or project for the changes to take effect.
    private static String subscriptionKey = System.getenv("COMPUTER_VISION_SUBSCRIPTION_KEY");
    private static String endpoint = System.getenv("COMPUTER_VISION_ENDPOINT");

    private static final String uriBase = endpoint + "vision/v3.0/analyze";
    private static final String imageToAnalyze =
        "https://upload.wikimedia.org/wikipedia/commons/" +
        "1/12/Broadway_and_Times_Square_by_night.jpg";

    public static void main(String[] args) {
        CloseableHttpClient httpClient = HttpClientBuilder.create().build();

        try {
            URIBuilder builder = new URIBuilder(uriBase);

            // Request parameters. All of them are optional.
            builder.setParameter("visualFeatures", "Categories,Description,Color");

            // Prepare the URI for the REST API method.
            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 requestEntity =
                new StringEntity("{\"url\":\"" + imageToAnalyze + "\"}");
            request.setEntity(requestEntity);

            // Call the REST API method and get the response entity.
            HttpResponse response = httpClient.execute(request);
            HttpEntity entity = response.getEntity();

            if (entity != null) {
                // Format and display the JSON response.
                String jsonString = EntityUtils.toString(entity);
                JSONObject json = new JSONObject(jsonString);
                System.out.println("REST Response:\n");
                System.out.println(json.toString(2));
            }
        } catch (Exception e) {
            // Display error message.
            System.out.println(e.getMessage());
        }
    }
}

```

Compile and run the program

1. Save, then build the Java project.
2. If you're using an IDE, run `Main`.

Alternately, if you're running the program from a command line window, run the following commands. These commands presume your libraries are in a folder named `libs` that is in the same folder as `Main.java`; if not, you will need to replace `libs` with the path to your libraries.

1. Compile the file `Main.java`.

```
javac -cp ".;libs/*" Main.java
```

2. Run the program. It will send the request to the QnA Maker API to create the KB, then it will poll for the results every 30 seconds. Each response is printed to the command line window.

```
java -cp ".;libs/*" Main
```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the console window, similar to the following example:

REST Response:

```
{
  "metadata": {
    "width": 1826,
    "format": "Jpeg",
    "height": 2436
  },
  "color": {
    "dominantColorForeground": "Brown",
    "isBwImg": false,
    "accentColor": "B74314",
    "dominantColorBackground": "Brown",
    "dominantColors": ["Brown"]
  },
  "requestId": "bbffe1a1-4fa3-4a6b-a4d5-a4964c58a811",
  "description": {
    "captions": [{
      "confidence": 0.8241405091548035,
      "text": "a group of people on a city street filled with traffic at night"
    }],
    "tags": [
      "outdoor",
      "building",
      "street",
      "city",
      "busy",
      "people",
      "filled",
      "traffic",
      "many",
      "table",
      "car",
      "group",
      "walking",
      "bunch",
      "crowded",
      "large",
      "night",
      "light",
      "standing",
      "man",
      "tall",
      "umbrella",
      "riding",
      "sign",
      "crowd"
    ]
  },
  "categories": [{
    "score": 0.625,
    "name": "outdoor_street"
  }]
}
```

Next steps

Explore a Java Swing application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; plus detect, categorize, tag, and describe visual features, including faces, in an image. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

[Computer Vision API Java Tutorial](#)

Quickstart: Analyze a remote image using the REST API and JavaScript in Computer Vision

9/1/2020 • 3 minutes to read • [Edit Online](#)

In this quickstart, you will analyze a remotely stored image to extract visual features using the Computer Vision REST API. With the [Analyze Image](#) method, you can extract visual features based on image content.

Prerequisites

- An Azure subscription - [Create one for free](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Create and run the sample

To create and run the sample, do the following steps:

1. Create a file called *analyze-image.html*, open it in a text editor, and copy the following code into it.
2. Optionally, replace the value of the `value` attribute for the `inputImage` control with the URL of a different image that you want to analyze.
3. Open a browser window.
4. In the browser, drag and drop the file into the browser window.
5. When the webpage is displayed in the browser, paste your subscription key and endpoint URL into the appropriate input boxes.
6. Select the **Analyze Image** button.

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

<script type="text/javascript">
  function processImage() {
    // *****
    // *** Update or verify the following values. ***
    // *****

    var subscriptionKey = document.getElementById("subscriptionKey").value;
    var endpoint = document.getElementById("endpointUrl").value;

    var uriBase = endpoint + "vision/v3.0/analyze";

    // Request parameters.
    var params = {
      "visualFeatures": "Categories,Description,Color",
      "details": "",
      "language": "en",
```

```

    });

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

    // Make 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;
        alert(errorString);
    });
    });
</script>

<h1>Analyze image:</h1>
Enter the URL to an image, then click the <strong>Analyze image</strong> button.
<br><br>
Subscription key:
<input type="text" name="subscriptionKey" id="subscriptionKey"
    value="" />
Endpoint URL:
<input type="text" name="endpointUrl" id="endpointUrl"
    value="" />
<br><br>
Image to analyze:
<input type="text" name="inputImage" id="inputImage"
    value="https://upload.wikimedia.org/wikipedia/commons/3/3c/Shaki_waterfall.jpg" />
<button onclick="processImage()">Analyze image</button>
<br><br>
<div id="wrapper" style="width:1020px; display:table;">
    <div id="jsonOutput" style="width:600px; display:table-cell;">
        Response:
        <br><br>
        <textarea id="responseTextArea" class="UIInput"
            style="width:580px; height:400px;"></textarea>
    </div>
    <div id="imageDiv" style="width:420px; display:table-cell;">
        Source image:
        <br><br>
        <img id="sourceImage" width="400" />
    </div>
</div>
</body>
</html>

```

Examine the response

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the browser window, similar to the following example:

```

{
  "categories": [
    {
      "name": "outdoor_water",
      "score": 0.9921875,
      "detail": {
        "landmarks": []
      }
    }
  ],
  "description": {
    "tags": [
      "nature",
      "water",
      "waterfall",
      "outdoor",
      "rock",
      "mountain",
      "rocky",
      "grass",
      "hill",
      "covered",
      "hillside",
      "standing",
      "side",
      "group",
      "walking",
      "white",
      "man",
      "large",
      "snow",
      "grazing",
      "forest",
      "slope",
      "herd",
      "river",
      "giraffe",
      "field"
    ],
    "captions": [
      {
        "text": "a large waterfall over a rocky cliff",
        "confidence": 0.916458423253597
      }
    ]
  },
  "color": {
    "dominantColorForeground": "Grey",
    "dominantColorBackground": "Green",
    "dominantColors": [
      "Grey",
      "Green"
    ],
    "accentColor": "4D5E2F",
    "isBwImg": false
  },
  "requestId": "73ef10ce-a4ea-43c6-aae7-70325777e4b3",
  "metadata": {
    "height": 959,
    "width": 1280,
    "format": "Jpeg"
  }
}

```

Next steps

Explore a JavaScript application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; plus detect, categorize, tag, and describe visual features, including faces, in an image. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

[Computer Vision API JavaScript Tutorial](#)

Quickstart: Analyze a remote image using the Computer Vision REST API with Node.js

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you'll analyze a remotely stored image to extract visual features using the Computer Vision REST API with Node.js. With the [Analyze Image](#) method, you can extract visual features based on image content.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Node.js](#) 4.x or later
- [npm](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Install the npm `request` package.
 - a. Open a command prompt window as an administrator.
 - b. Run the following command:

```
npm install request
```

- c. After the package is successfully installed, close the command prompt window.
2. Copy the following code into a text editor.
 3. Optionally, replace the value of `imageUri` with the URL of a different image that you want to analyze.
 4. Optionally, replace the value of the `language` request parameter with a different language.
 5. Save the code as a file with a `.js` extension. For example, `analyze-image.js`.
 6. Open a command prompt window.
 7. At the prompt, use the `node` command to run the file. For example, `node analyze-image.js`.

```

'use strict';

const request = require('request');

let subscriptionKey = process.env['COMPUTER_VISION_SUBSCRIPTION_KEY'];
let endpoint = process.env['COMPUTER_VISION_ENDPOINT']
if (!subscriptionKey) { throw new Error('Set your environment variables for your subscription key and endpoint.')}

var uriBase = endpoint + 'vision/v3.0/analyze';

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

// Request parameters.
const params = {
  'visualFeatures': 'Categories,Description,Color',
  'details': '',
  'language': 'en'
};

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);
});

```

Examine the response

A successful response is returned in JSON. The sample parses and displays a successful response in the command prompt window, similar to the following example:

```

{
  "categories": [
    {
      "name": "outdoor_water",
      "score": 0.9921875,
      "detail": {
        "landmarks": []
      }
    }
  ],
  "description": {
    "tags": [
      "nature",
      "water",
      "waterfall",
      "outdoor",
      "rock",
      "mountain",
      "rocky",
      "grass",
      "hill",
      "covered",
      "hillside",
      "standing",
      "side",
      "group",
      "walking",
      "white",
      "man",
      "large",
      "snow",
      "grazing",
      "forest",
      "slope",
      "herd",
      "river",
      "giraffe",
      "field"
    ],
    "captions": [
      {
        "text": "a large waterfall over a rocky cliff",
        "confidence": 0.916458423253597
      }
    ]
  },
  "color": {
    "dominantColorForeground": "Grey",
    "dominantColorBackground": "Green",
    "dominantColors": [
      "Grey",
      "Green"
    ],
    "accentColor": "4D5E2F",
    "isBwImg": false
  },
  "requestId": "81b4e400-e3c1-41f1-9020-e6871ad9f0ed",
  "metadata": {
    "height": 959,
    "width": 1280,
    "format": "Jpeg"
  }
}

```

Clean up resources

When no longer needed, delete the file, and then uninstall the npm `request` package. To uninstall the package, do the following steps:

1. Open a command prompt window as an administrator.
2. Run the following command:

```
npm uninstall request
```

3. After the package is successfully uninstalled, close the command prompt window.

Next steps

Next, explore the Computer Vision APIs used to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text.

[Explore the Computer Vision API](#)

Quickstart: Analyze a remote image using the Computer Vision REST API and Python

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you'll analyze a remotely stored image to extract visual features using the Computer Vision REST API. With the [Analyze Image](#) method, you can extract visual features based on image content.

You can run this quickstart in a step-by-step fashion using a Jupyter notebook on [MyBinder](#). To launch Binder, select the following button:

launch binder

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Python](#) and the following packages:
 - `requests`
 - `matplotlib`
 - `pillow`
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Copy the following code into a text editor.
2. 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, `analyze-image.py`.
4. Open a command prompt window.
5. At the prompt, use the `python` command to run the sample. For example, `python analyze-image.py`.

```

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

# Add your Computer Vision subscription key and endpoint to your environment variables.
if 'COMPUTER_VISION_SUBSCRIPTION_KEY' in os.environ:
    subscription_key = os.environ['COMPUTER_VISION_SUBSCRIPTION_KEY']
else:
    print("\nSet the COMPUTER_VISION_SUBSCRIPTION_KEY environment variable.\n**Restart your shell or IDE for changes to take effect.**")
    sys.exit()

if 'COMPUTER_VISION_ENDPOINT' in os.environ:
    endpoint = os.environ['COMPUTER_VISION_ENDPOINT']

analyze_url = endpoint + "vision/v3.0/analyze"

# Set image_url to the URL of an image that you want to analyze.
image_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/1/12/" + \
    "Broadway_and_Times_Square_by_night.jpg/450px-Broadway_and_Times_Square_by_night.jpg"

headers = {'Ocp-Apim-Subscription-Key': subscription_key}
params = {'visualFeatures': 'Categories,Description,Color'}
data = {'url': image_url}
response = requests.post(analyze_url, headers=headers,
                        params=params, json=data)
response.raise_for_status()

# The 'analysis' object contains various fields that describe the image. The most
# relevant caption for the image is obtained from the 'description' property.
analysis = response.json()
print(json.dumps(response.json()))
image_caption = analysis["description"][0]["captions"][0]["text"].capitalize()

# Display the image and overlay it with the caption.
image = Image.open(BytesIO(requests.get(image_url).content))
plt.imshow(image)
plt.axis("off")
_ = plt.title(image_caption, size="x-large", y=-0.1)
plt.show()

```

Examine the response

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the command prompt window, similar to the following example:

```

{
  "categories": [
    {
      "name": "outdoor_",
      "score": 0.00390625,
      "detail": {
        "landmarks": []
      }
    },
    {
      "name": "outdoor_street",
      "score": 0.33984375,
      "detail": {
        "landmarks": []
      }
    }
  ]
}

```

```

    },
    ],
    "description": {
      "tags": [
        "building",
        "outdoor",
        "street",
        "city",
        "people",
        "busy",
        "table",
        "walking",
        "traffic",
        "filled",
        "large",
        "many",
        "group",
        "night",
        "light",
        "crowded",
        "bunch",
        "standing",
        "man",
        "sign",
        "crowd",
        "umbrella",
        "riding",
        "tall",
        "woman",
        "bus"
      ],
    },
    "captions": [
      {
        "text": "a group of people on a city street at night",
        "confidence": 0.9122243847383961
      }
    ]
  },
  "color": {
    "dominantColorForeground": "Brown",
    "dominantColorBackground": "Brown",
    "dominantColors": [
      "Brown"
    ],
    "accentColor": "B54316",
    "isBwImg": false
  },
  "requestId": "c11894eb-de3e-451b-9257-7c8b168073d1",
  "metadata": {
    "height": 600,
    "width": 450,
    "format": "Jpeg"
  }
}

```

Next steps

Next, explore a Python application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; plus detect, categorize, tag, and describe visual features in images.

[Computer Vision API Python Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Analyze a local image using the Computer Vision REST API and C#

9/1/2020 • 4 minutes to read • [Edit Online](#)

In this quickstart, you will analyze a locally stored image to extract visual features using the Computer Vision REST API. With the [Analyze Image](#) method, you can extract visual feature information from image content.

Prerequisites

- An Azure subscription - [Create one for free](#)
- You must have [Visual Studio 2015](#) or later
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (F0) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample application

To create the sample in Visual Studio, do the following steps:

1. Create a new Visual Studio solution/project in Visual Studio, using the Visual C# Console App (.NET Core Framework) template.
2. Install the Newtonsoft.Json NuGet package.
 - a. On the menu, click **Tools**, select **NuGet Package Manager**, then **Manage NuGet Packages for Solution**.
 - b. Click the **Browse** tab, and in the **Search** box type "Newtonsoft.Json" (if it is not already displayed).
 - c. Select **Newtonsoft.Json**, then click the checkbox next to your project name, and **Install**.
3. Copy/paste the sample code snippet below, into your Program.cs file. Adjust the namespace name if it's different from the one you created.
4. Add an image of your choosing to your bin/debug/netcoreappX.X folder, then add the image name (with extension) to the 'imageFilePath' variable.
5. Run the program.

```
using Newtonsoft.Json.Linq;
using System;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;

namespace CSHttpClientSample
{
    static class Program
    {
        // Add your Computer Vision subscription key and endpoint to your environment variables.
        static string subscriptionKey =
            Environment.GetEnvironmentVariable("COMPUTER_VISION_SUBSCRIPTION_KEY");
```



```

static string endpoint = Environment.GetEnvironmentVariable("COMPUTER_VISION_ENDPOINT");

// the Analyze method endpoint
static string uriBase = endpoint + "vision/v3.0/analyze";

// Image you want analyzed (add to your bin/debug/netcoreappX.X folder)
// For sample images, download one from here (png or jpg):
// https://github.com/Azure-Samples/cognitive-services-sample-data-
files/tree/master/ComputerVision/Images
static string imageFilePath = @"my-sample-image";

public static void Main()
{
    // Call the API
    MakeAnalysisRequest(imageFilePath).Wait();

    Console.WriteLine("\nPress Enter to exit...");
    Console.ReadLine();
}

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

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

        // Request parameters. A third optional parameter is "details".
        // The Analyze Image method returns information about the following
        // visual features:
        // Categories: categorizes image content according to a
        // taxonomy defined in documentation.
        // Description: describes the image content with a complete
        // sentence in supported languages.
        // Color: determines the accent color, dominant color,
        // and whether an image is black & white.
        string requestParameters =
            "visualFeatures=Categories,Description,Color";

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

        HttpResponseMessage response;

        // Read the contents of the specified local image
        // into a byte array.
        byte[] byteData = GetImageAsByteArray(imageFilePath);

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

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

```

```

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

        // Display the JSON response.
        Console.WriteLine("\nResponse:\n\n{0}\n",
            JToken.Parse(contentString).ToString());
    }
    catch (Exception e)
    {
        Console.WriteLine("\n" + e.Message);
    }
}

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

```

Examine the response

A successful response is returned in JSON (based on your own image used) in the console window, similar to the following example:

```
{
  "categories": [
    {
      "name": "abstract_",
      "score": 0.00390625
    },
    {
      "name": "others_",
      "score": 0.0234375
    },
    {
      "name": "outdoor_",
      "score": 0.00390625
    }
  ],
  "description": {
    "tags": [
      "road",
      "building",
      "outdoor",
      "street",
      "night",
      "black",
      "city",
      "white",
      "light",
      "sitting",
      "riding",
      "man",
      "side",
      "empty",
      "rain",
      "corner",
      "traffic",
      "lit",
      "hydrant",
      "stop",
      "board",
      "parked",
      "bus",
      "tall"
    ],
    "captions": [
      {
        "text": "a close up of an empty city street at night",
        "confidence": 0.7965622853462756
      }
    ]
  },
  "requestId": "ddd1ac9-7e66-4c47-bdef-222f3fe5aa23",
  "metadata": {
    "width": 3733,
    "height": 1986,
    "format": "Jpeg"
  },
  "color": {
    "dominantColorForeground": "Black",
    "dominantColorBackground": "Black",
    "dominantColors": [
      "Black",
      "Grey"
    ],
    "accentColor": "666666",
    "isBWImg": true
  }
}
```

Next steps

Explore a basic Windows application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; plus detect, categorize, tag, and describe visual features, including faces, in an image.

[Computer Vision API C# Tutorial](#)

Quickstart: Analyze a local image using the Computer Vision REST API and Python

9/1/2020 • 3 minutes to read • [Edit Online](#)

In this quickstart, you'll analyze a locally stored image to extract visual features using the Computer Vision REST API. With the [Analyze Image](#) method, you can extract visual features based on image content.

You can run this quickstart in a step-by-step fashion using a Jupyter notebook on [MyBinder](#). To launch Binder, select the following button:

launch binder

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Python](#) and the following packages:
 - `requests`
 - `matplotlib`
 - `pillow`
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (F0) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Copy the following code into a text editor.
2. Replace the value of `image_path` with the path and file name of a different image that you want to analyze.
3. Save the code as a file with an `.py` extension. For example, `analyze-local-image.py`.
4. Open a command prompt window.
5. At the prompt, use the `python` command to run the sample. For example, `python analyze-local-image.py`.

```

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

# Add your Computer Vision subscription key and endpoint to your environment variables.
if 'COMPUTER_VISION_SUBSCRIPTION_KEY' in os.environ:
    subscription_key = os.environ['COMPUTER_VISION_SUBSCRIPTION_KEY']
else:
    print("\nSet the COMPUTER_VISION_SUBSCRIPTION_KEY environment variable.\n**Restart your shell or IDE for changes to take effect.**")
    sys.exit()

if 'COMPUTER_VISION_ENDPOINT' in os.environ:
    endpoint = os.environ['COMPUTER_VISION_ENDPOINT']

analyze_url = endpoint + "vision/v3.0/analyze"

# Set image_path to the local path of an image that you want to analyze.
# Sample images are here, if needed:
# https://github.com/Azure-Samples/cognitive-services-sample-data-files/tree/master/ComputerVision/Images
image_path = "C:/Documents/ImageToAnalyze.jpg"

# Read the image into a byte array
image_data = open(image_path, "rb").read()
headers = {'Ocp-Apim-Subscription-Key': subscription_key,
           'Content-Type': 'application/octet-stream'}
params = {'visualFeatures': 'Categories,Description,Color'}
response = requests.post(
    analyze_url, headers=headers, params=params, data=image_data)
response.raise_for_status()

# The 'analysis' object contains various fields that describe the image. The most
# relevant caption for the image is obtained from the 'description' property.
analysis = response.json()
print(analysis)
image_caption = analysis["description"]["captions"][0]["text"].capitalize()

# Display the image and overlay it with the caption.
image = Image.open(BytesIO(image_data))
plt.imshow(image)
plt.axis("off")
_ = plt.title(image_caption, size="x-large", y=-0.1)
plt.show()

```

Examine the response

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the command prompt window, similar to the following example:

```

{
  "categories": [
    {
      "name": "outdoor_",
      "score": 0.00390625,
      "detail": {
        "landmarks": []
      }
    },
    {
      "name": "outdoor street".

```

```

    name : "category_street",
    "score": 0.33984375,
    "detail": {
      "landmarks": []
    }
  }
],
"description": {
  "tags": [
    "building",
    "outdoor",
    "street",
    "city",
    "people",
    "busy",
    "table",
    "walking",
    "traffic",
    "filled",
    "large",
    "many",
    "group",
    "night",
    "light",
    "crowded",
    "bunch",
    "standing",
    "man",
    "sign",
    "crowd",
    "umbrella",
    "riding",
    "tall",
    "woman",
    "bus"
  ],
  "captions": [
    {
      "text": "a group of people on a city street at night",
      "confidence": 0.9122243847383961
    }
  ]
},
"color": {
  "dominantColorForeground": "Brown",
  "dominantColorBackground": "Brown",
  "dominantColors": [
    "Brown"
  ],
  "accentColor": "B54316",
  "isBwImg": false
},
"requestId": "c11894eb-de3e-451b-9257-7c8b168073d1",
"metadata": {
  "height": 600,
  "width": 450,
  "format": "Jpeg"
}
}

```

Next steps

Next, explore a Python application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

[Computer Vision API Python Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Generate a thumbnail using the Computer Vision REST API and C#

9/1/2020 • 4 minutes to read • [Edit Online](#)

In this quickstart, you generate a thumbnail from an image by using the Computer Vision REST API. With the [Get Thumbnail](#) method, you can generate a thumbnail of an image. You specify the height and width, which can differ from the aspect ratio of the input image. Computer Vision uses smart cropping to intelligently identify the area of interest and generate cropping coordinates based on that region.

Prerequisites

- An Azure subscription - [Create one for free](#)
- You must have [Visual Studio 2015](#) or later
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample application

To create the sample in Visual Studio, do the following steps:

1. Create a new Visual Studio solution in Visual Studio, using the Visual C# Console App template.
2. Install the Newtonsoft.Json NuGet package.
 - a. On the menu, click **Tools**, select **NuGet Package Manager**, then **Manage NuGet Packages for Solution**.
 - b. Click the **Browse** tab, and in the **Search** box type "Newtonsoft.Json".
 - c. Select **Newtonsoft.Json** when it displays, then click the checkbox next to your project name, and **Install**.
3. Run the program.
4. At the prompt, enter the path to a local image.

```
using Newtonsoft.Json.Linq;
using System;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;

namespace CSHttpClientSample
{
    static class Program
    {
        // Add your Computer Vision subscription key and base endpoint to your environment variables.
        static string subscriptionKey = Environment.GetEnvironmentVariable("COMPUTER_VISION_SUBSCRIPTION_KEY");
        static string endpoint = Environment.GetEnvironmentVariable("COMPUTER_VISION_ENDPOINT");

        // The GenerateThumbnail method endpoint
        static string uriBase = endpoint + "vision/v3.0/generateThumbnail";
```

```

// Add an image to your bin/debug/netcoreappX.X folder, then add the image name (with extension), here
static string imagePath = @"my-image-name";

public static void Main()
{

    MakeThumbNailRequest(imageFilePath).Wait();

    Console.WriteLine("\nPress Enter to exit...");
    Console.ReadLine();
}

/// <summary>
/// Gets a thumbnail image from the specified image file by using
/// the Computer Vision REST API.
/// </summary>
/// <param name="imageFilePath">The image file to use to create the thumbnail image.</param>
static async Task MakeThumbNailRequest(string imagePath)
{
    try
    {
        HttpClient client = new HttpClient();

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

        // Request parameters.
        // The width and height parameters specify a thumbnail that's
        // 200 pixels wide and 150 pixels high.
        // The smartCropping parameter is set to true, to enable smart cropping.
        string requestParameters = "width=200&height=150&smartCropping=true";

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

        HttpResponseMessage response;

        // Read the contents of the specified local image
        // into a byte array.
        byte[] byteData = GetImageAsByteArray(imageFilePath);

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

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

        // Check the HTTP status code of the response. If successful, display
        // display the response and save the thumbnail.
        if (response.IsSuccessStatusCode)
        {
            // Display the response data.
            Console.WriteLine("\nResponse:\n{0}", response);

            // Get the image data for the thumbnail from the response.
            byte[] thumbnailImageData =
                await response.Content.ReadAsByteArrayAsync();

            // Save the thumbnail to the same folder as the original image,
            // using the original name with the suffix "_thumb".
            // Note: This will overwrite an existing file of the same name.

```

```

        string thumbnailFilePath =
            imagePath.Insert(imageFilePath.Length - 4, "_thumb");
        File.WriteAllBytes(thumbnailFilePath, thumbnailImageData);
        Console.WriteLine("\nThumbnail written to: {0}", thumbnailFilePath);
    }
    else
    {
        // Display the JSON error data.
        string errorString = await response.Content.ReadAsStringAsync();
        Console.WriteLine("\n\nResponse:\n{0}\n",
            JToken.Parse(errorString).ToString());
    }
}
catch (Exception e)
{
    Console.WriteLine("\n" + e.Message);
}
}

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

```

Examine the response

A successful response is returned as binary data, which represents the image data for the thumbnail. If the request succeeds, the thumbnail is saved to the same folder as the local image, using the original name with the suffix "_thumb". If the request fails, the response contains an error code and a message to help determine what went wrong.

The sample application displays a successful response in the console window, similar to the following example:

Response:

```

StatusCode: 200, ReasonPhrase: 'OK', Version: 1.1, Content: System.Net.Http.StreamContent, Headers:
{
    Pragma: no-cache
    apim-request-id: 131eb5b4-5807-466d-9656-4c1ef0a64c9b
    Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
    x-content-type-options: nosniff
    Cache-Control: no-cache
    Date: Tue, 06 Jun 2017 20:54:07 GMT
    X-AspNet-Version: 4.0.30319
    X-Powered-By: ASP.NET
    Content-Length: 5800
    Content-Type: image/jpeg
    Expires: -1
}

```

Next steps

Explore a basic Windows application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; plus detect, categorize, tag, and describe visual features, including faces, in an image. To rapidly experiment with the Computer Vision APIs, try the [Open API testing console](#).

[Computer Vision API C# Tutorial](#)

Quickstart: Generate a thumbnail using the Computer Vision REST API and cURL

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you generate a thumbnail from an image using the Computer Vision REST API. You specify the desired height and width, which can differ in aspect ratio from the input image. Computer Vision uses smart cropping to intelligently identify the area of interest and generate cropping coordinates around that region.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [cURL](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Create and run the sample command

To create and run the sample, do the following steps:

1. Copy the following command into a text editor.
2. Make the following changes in the command where needed:
 - a. Replace the value of `<subscriptionKey>` with your subscription key.
 - b. Replace the value of `<thumbnailFile>` with the path and name of the file in which to save the thumbnail.
 - c. Replace the first part of the request URL (`westcentralus`) with the text in your own endpoint URL.

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](#).

- d. Optionally, change the image URL in the request body (

```
https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Shorkie_Poo_Puppy.jpg/1280px-Shorkie_Poo_Puppy.jpg\
```

) to the URL of a different image from which to generate a thumbnail.

3. Open a command prompt window.
4. Paste the command from the text editor into the command prompt window.
5. Press enter to run the program.

```
curl -H "Ocp-Apim-Subscription-Key: <subscriptionKey>" -o <thumbnailFile> -H "Content-Type: application/json" "https://westus.api.cognitive.microsoft.com/vision/v3.0/generateThumbnail?width=100&height=100&smartCropping=true" -d "{\n  \"url\": \"https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Shorkie_Poo_Puppy.jpg/1280px-Shorkie_Poo_Puppy.jpg\"\n}"
```

Examine the response

A successful response writes the thumbnail image to the file specified in `<thumbnailFile>`. If the request fails, the response contains an error code and a message to help determine what went wrong. If the request seems to succeed but the created thumbnail is not a valid image file, it might be that your subscription key is not valid.

Next steps

Explore the Computer Vision API to how to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

[Explore the Computer Vision API](#)

Quickstart: Generate a thumbnail using the Computer Vision REST API with Go

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you'll generate a thumbnail from an image using the Computer Vision REST API. You specify the height and width, which can differ in aspect ratio from the input image. Computer Vision uses smart cropping to intelligently identify the area of interest and generate cropping coordinates based on that region.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Go](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Copy the following code into a text editor.
2. Optionally, replace the value of `imageUrl` with the URL of a different image from which you want to generate a thumbnail.
3. Save the code as a file with a `.go` extension. For example, `get-thumbnail.go`.
4. Open a command prompt window.
5. At the prompt, run the `go build` command to compile the package from the file. For example, `go build get-thumbnail.go`.
6. At the prompt, run the compiled package. For example, `get-thumbnail`.

```
package main

import (
    "bytes"
    "fmt"
    "io/ioutil"
    "io"
    "log"
    "net/http"
    "os"
    "strings"
    "time"
)

func main() {
    // Add your Computer Vision subscription key and endpoint to your environment variables.
    subscriptionKey := os.Getenv("COMPUTER_VISION_SUBSCRIPTION_KEY")
    endpoint := os.Getenv("COMPUTER_VISION_ENDPOINT")
```

```

uriBase := endpoint + "vision/v3.0/generateThumbnail"
const imageUrl = "https://upload.wikimedia.org/wikipedia/commons/9/94/Bloodhound_Puppy.jpg"

const params = "?width=100&height=100&smartCropping=true"
uri := uriBase + params
const imageUrlEnc = "{\"url\":\"" + imageUrl + "\"}"

reader := strings.NewReader(imageUrlEnc)

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

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

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

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

defer resp.Body.Close()

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

// Convert byte[] to io.Reader type
readerThumb := bytes.NewReader(data)

// Write the image binary to file
file, err := os.Create("thumb_local.png")
if err != nil { log.Fatal(err) }
defer file.Close()
_, err = io.Copy(file, readerThumb)
if err != nil { log.Fatal(err) }

fmt.Println("The thumbnail from local has been saved to file.")
fmt.Println()
}

```

Examine the response

A successful response contains the thumbnail image binary data. If the request fails, the response contains an error code and a message to help determine what went wrong.

Next steps

Explore the Computer Vision API to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Generate a thumbnail using the Computer Vision REST API and Java

9/1/2020 • 3 minutes to read • [Edit Online](#)

In this quickstart, you'll generate a thumbnail from an image using the Computer Vision REST API. You specify the height and width, which can differ from the aspect ratio of the input image. Computer Vision uses smart cropping to intelligently identify the area of interest and generate cropping coordinates based on that region.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Java™ Platform, Standard Edition Development Kit 7 or 8](#) (JDK 7 or 8)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample application

To create and run the sample, do the following steps:

1. Create a new Java project in your favorite IDE or editor. If the option is available, create the Java project from a command line application template.
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.X)
 - [Apache HTTP core](#) (org.apache.httpcomponents:httpcore:4.4.X)
 - [JSON library](#) (org.json:json:20180130)
3. Add the following `import` statements to your main class:

```
import java.awt.*;
import javax.swing.*;
import java.net.URI;
import java.io.InputStream;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
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.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import org.json.JSONObject;
```

4. Add the rest of the sample code below, beneath the imports (change to your class name if needed).
5. Add your Computer Vision subscription key and endpoint to your environment variables.
6. Optionally, replace the value of `imageToAnalyze` with the URL of your own image.
7. Save, then build the Java project.
8. If you're using an IDE, run `GenerateThumbnail`. Otherwise, run from the command line (commands below).

```
/**
 * This sample uses the following libraries (create a "lib" folder to place them in):
 * Apache HTTP client:
 * org.apache.httpcomponents:httpclient:4.5.X
 * Apache HTTP core:
 * org.apache.httpcomponents:httpcore:4.4.X
 * JSON library:
 * org.json:json:20180130
 *
 * To build/run from the command line:
 *   javac GenerateThumbnail.java -cp .;lib\*
 *   java -cp .;lib\* GenerateThumbnail
 */

public class GenerateThumbnail {

    // Add your Computer Vision subscription key and endpoint to your environment
    // variables. Then, close and then re-open your command shell or project for the
    // changes to take effect.
    private static String subscriptionKey = System.getenv("COMPUTER_VISION_SUBSCRIPTION_KEY");
    private static String endpoint = System.getenv("COMPUTER_VISION_ENDPOINT");
    // The endpoint path
    private static final String uriBase = endpoint + "vision/v3.0/generateThumbnail";
    // It's optional if you'd like to use your own image instead of this one.
    private static final String imageToAnalyze =
"https://upload.wikimedia.org/wikipedia/commons/9/94/Bloodhound_Puppy.jpg";

    public static void main(String[] args) {
        CloseableHttpClient httpClient = HttpClientBuilder.create().build();

        try {
            URIBuilder uriBuilder = new URIBuilder(uriBase);

            // Request parameters.
            uriBuilder.setParameter("width", "100");
            uriBuilder.setParameter("height", "150");
            uriBuilder.setParameter("smartCropping", "true");

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

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

            // Request body.
            StringEntity requestEntity = new StringEntity("{\"url\":\"" + imageToAnalyze + "\"}");
            request.setEntity(requestEntity);

            // Call the REST API method and get the response entity.
            HttpResponse response = httpClient.execute(request);
            HttpEntity entity = response.getEntity();

            System.out.println("status" + response.getStatusLine().getStatusCode());

            // Check for success.
```

```

        if (response.getStatusLine().getStatusCode() == 200) {
            // Display the thumbnail.
            System.out.println("\nDisplaying thumbnail.\n");
            displayImage(entity.getContent());
        } else {
            // Format and display the JSON error message.
            String jsonString = EntityUtils.toString(entity);
            JSONObject json = new JSONObject(jsonString);
            System.out.println("Error:\n");
            System.out.println(json.toString(2));
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}

// Displays the given input stream as an image.
public static void displayImage(InputStream inputStream) {
    try {
        BufferedImage bufferedImage = ImageIO.read(inputStream);

        ImageIcon imageIcon = new ImageIcon(bufferedImage);

        JLabel jLabel = new JLabel();
        jLabel.setIcon(imageIcon);

        JFrame jFrame = new JFrame();
        jFrame.setLayout(new FlowLayout());
        jFrame.setSize(100, 150);

        jFrame.add(jLabel);
        jFrame.setVisible(true);
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
}

```

Examine the response

A successful response is returned as binary data, which represents the image data for the thumbnail. If the request succeeds, the thumbnail is generated from the binary data in the response and displayed in a separate window created by the sample application. If the request fails, the response is displayed in the console window. The response for the failed request contains an error code and a message to help determine what went wrong.

Next steps

Explore a Java Swing application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

Computer Vision API Java Tutorial

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Generate a thumbnail using the Computer Vision REST API and JavaScript

9/1/2020 • 3 minutes to read • [Edit Online](#)

In this quickstart, you will generate a thumbnail from an image using the Computer Vision REST API. You specify the height and width, which can differ in aspect ratio from the input image. Computer Vision uses smart cropping to intelligently identify the area of interest and generate cropping coordinates based on that region.

Prerequisites

- An Azure subscription - [Create one for free](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (**F0**) to try the service, and upgrade later to a paid tier for production.

Create and run the sample

To create and run the sample, do the following steps:

1. Create a file called *get-thumbnail.html*, open it in a text editor, and copy the following code into it.
2. Optionally, replace the value of the `value` attribute of the `inputImage` control with the URL of a different image that you want to analyze.
3. Open a browser window.
4. In the browser, drag and drop the file into the browser window.
5. When the webpage is displayed in the browser, paste your subscription key and endpoint URL into the appropriate input boxes.
6. Finally, select the **Generate thumbnail** button.

```
<!DOCTYPE html>
<html>
<head>
  <title>Thumbnail Sample</title>
</head>
<body>

<script type="text/javascript">
  function processImage() {
    // *****
    // *** Update or verify the following values. ***
    // *****

    var subscriptionKey = document.getElementById("subscriptionKey").value;
    var endpoint = document.getElementById("endpointUrl").value;

    var uriBase = endpoint + "vision/v3.0/generateThumbnail";

    // Request parameters.
    var params = "?width=100&height=150&smartCropping=true";

    // Display the source image.
    var sourceImageUrl = document.getElementById("inputImage").value;
```

```

document.querySelector("#sourceImage").src = sourceImageUrl;

// Prepare the REST API call:

// Create the HTTP Request object.
var xhr = new XMLHttpRequest();

// Identify the request as a POST, with the URL and parameters.
xhr.open("POST", uriBase + params);

// Add the request headers.
xhr.setRequestHeader("Content-Type", "application/json");
xhr.setRequestHeader("Ocp-Apim-Subscription-Key", subscriptionKey);

// Set the response type to "blob" for the thumbnail image data.
xhr.responseType = "blob";

// Process the result of the REST API call.
xhr.onreadystatechange = function(e) {
    if(xhr.readyState === XMLHttpRequest.DONE) {

        // Thumbnail successfully created.
        if (xhr.status === 200) {
            // Show response headers.
            var s = JSON.stringify(xhr.getAllResponseHeaders(), null, 2);
            document.getElementById("responseTextArea").value =
                JSON.stringify(xhr.getAllResponseHeaders(), null, 2);

            // Show thumbnail image.
            var urlCreator = window.URL || window.webkitURL;
            var imageUrl = urlCreator.createObjectURL(this.response);
            document.querySelector("#thumbnailImage").src = imageUrl;
        } else {
            // Display the error message. The error message is the response
            // body as a JSON string. The code in this code block extracts
            // the JSON string from the blob response.
            var reader = new FileReader();

            // This event fires after the blob has been read.
            reader.addEventListener('loadend', (e) => {
                document.getElementById("responseTextArea").value =
                    JSON.stringify(JSON.parse(e.srcElement.result), null, 2);
            });

            // Start reading the blob as text.
            reader.readAsText(xhr.response);
        }
    }
};

// Make the REST API call.
xhr.send({'url': ' ' + ' ' + sourceImageUrl + ' '});
};
</script>

```

<h1>Generate thumbnail image:</h1>

Enter the URL to an image to use in creating a thumbnail image,
then click the Generate thumbnail button.

Subscription key:

<input type="text" name="subscriptionKey" id="subscriptionKey"
value="" />

Endpoint URL:

<input type="text" name="endpointUrl" id="endpointUrl"
value="" />

Image for thumbnail:

<input type="text" name="inputImage" id="inputImage"

value="https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Shorkie Poo Puppyv.jpg/1280px-

```

Shorkie_Poo_Puppy.jpg" />
<button onclick="processImage()">Generate thumbnail</button>
<br><br>
<div id="wrapper" style="width:1160px; 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 id="thumbnailDiv" style="width:140px; display:table-cell;">
    Thumbnail:
    <br><br>
    <img id="thumbnailImage" />
  </div>
</div>
</body>
</html>

```

Examine the response

A successful response is returned as binary data, which represents the image data for the thumbnail. If the request succeeds, the thumbnail is generated from the binary data in the response and displayed in the browser window. If the request fails, the response is displayed in the console window. The response for the failed request contains an error code and a message to help determine what went wrong.

Next steps

Explore a JavaScript application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; plus detect, categorize, tag, and describe visual features, including faces, in an image. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

[Computer Vision API JavaScript Tutorial](#)

Quickstart: Generate a thumbnail using the Computer Vision REST API and Node.js

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you'll generate a thumbnail from an image using the Computer Vision REST API. With the [Get Thumbnail](#) method, you can generate a thumbnail of an image. You specify the height and width, which can differ from the aspect ratio of the input image. Computer Vision uses smart cropping to intelligently identify the area of interest and generate cropping coordinates based on that region.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Node.js](#) 4.x or later
- [npm](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (**F0**) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Install the npm `request` package.

- a. Open a command prompt window as an administrator.
- b. Run the following command:

```
npm install request
```

- c. After the package is successfully installed, close the command prompt window.
2. Copy the following code into a text editor.
 3. Optionally, replace the value of `imageUri` with the URL of a different image that you want to analyze.
 4. Save the code as a file with a `.js` extension. For example, `get-thumbnail.js`.
 5. Open a command prompt window.
 6. At the prompt, use the `node` command to run the file. For example, `node get-thumbnail.js`.


```

'use strict';

const fs = require('fs');
const request = require('request').defaults({ encoding: null });

let subscriptionKey = process.env['COMPUTER_VISION_SUBSCRIPTION_KEY'];
let endpoint = process.env['COMPUTER_VISION_ENDPOINT']

var uriBase = endpoint + 'vision/v3.0/generateThumbnail';

const imageUrl = 'https://upload.wikimedia.org/wikipedia/commons/9/94/Bloodhound_Puppy.jpg';

// Request parameters.
const params = {
  'width': '100',
  'height': '100',
  'smartCropping': 'true'
};

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

// Post the request and get the response (an image stream)
request.post(options, (error, response, body) => {
  // Write the stream to file
  var buf = Buffer.from(body, 'base64');
  fs.writeFile('thumbnail.png', buf, function (err) {
    if (err) throw err;
  });

  console.log('Image saved')
});

```

Examine the response

A popup of the thumbnail image will display. A successful response is returned as binary data, which represents the image data for the thumbnail. If the request fails, the response is displayed in the console window. The response for the failed request contains an error code and a message to help determine what went wrong.

Next steps

Next, explore the Computer Vision API used to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text.

[Explore the Computer Vision API](#)

Quickstart: Generate a thumbnail using the Computer Vision REST API and Python

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you'll generate a thumbnail from an image using the Computer Vision REST API. With the [Get Thumbnail](#) method, you can specify the desired height and width, and Computer Vision uses smart cropping to intelligently identify the area of interest and generate cropping coordinates based on that region.

Prerequisites

- An Azure subscription - [Create one for free](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.
- A code editor such as [Visual Studio Code](#).

Create and run the sample

To create and run the sample, copy the following code into the code editor.

```

import os
import sys
import requests
# If you are using a Jupyter notebook, uncomment the following lines.
# %matplotlib inline
# import matplotlib.pyplot as plt
from PIL import Image
from io import BytesIO

# Add your Computer Vision subscription key and endpoint to your environment variables.
subscription_key = os.environ['COMPUTER_VISION_SUBSCRIPTION_KEY']
endpoint = os.environ['COMPUTER_VISION_ENDPOINT']

thumbnail_url = endpoint + "vision/v3.0/generateThumbnail"

# Set image_url to the URL of an image that you want to analyze.
image_url = "https://upload.wikimedia.org/wikipedia/commons/9/94/Bloodhound_Puppy.jpg"

# Construct URL
headers = {'Ocp-Apim-Subscription-Key': subscription_key}
params = {'width': '50', 'height': '50', 'smartCropping': 'true'}
data = {'url': image_url}
# Call API
response = requests.post(thumbnail_url, headers=headers, params=params, json=data)
response.raise_for_status()

# Open the image from bytes
thumbnail = Image.open(BytesIO(response.content))

# Verify the thumbnail size.
print("Thumbnail is {0}-by-{1}".format(*thumbnail.size))

# Save thumbnail to file
thumbnail.save('thumbnail.png')

# Display image
thumbnail.show()

# Optional. Display the thumbnail from Jupyter.
# plt.imshow(thumbnail)
# plt.axis("off")

```

Next, do the following:

1. (Optional) Replace the value of `image_url` with the URL of your own image.
2. Save the code as a file with an `.py` extension. For example, `get-thumbnail.py`.
3. Open a command prompt window.
4. At the prompt, use the `python` command to run the sample. For example, `python get-thumbnail.py`.

Examine the response

A successful response is returned as binary data which represents the image data for the thumbnail. The sample should display this image. If the request fails, the response is displayed in the command prompt window and should contain an error code.

Run in Jupyter (optional)

You can optionally run this quickstart in a step-by-step fashion using a Jupyter notebook on [MyBinder](#). To launch Binder, select the following button:

launch binder

Next steps

Next, explore a Python application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

[Computer Vision API Python Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Extract text using the Computer Vision 3.0 REST API Read operation and C#

9/1/2020 • 6 minutes to read • [Edit Online](#)

In this quickstart, you'll extract printed and handwritten text from an image using the new OCR technology available as part of the Computer Vision 3.0 REST API. With the new [Read](#) and [Get Read Result](#) methods, you can detect text in an image and extract recognized characters into a machine-readable character stream.

IMPORTANT

The [Read](#) method runs asynchronously. This method does not return any information in the body of a successful response. Instead, the Batch Read method returns a URI in the value of the `Operation-Location` response header field. You can then call this URI, which represents the [Get Read Result](#) API, to both check the status and return the results of the Read method call.

Prerequisites

- An Azure subscription - [Create one for free](#)
- You must have [Visual Studio 2015](#) or later
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample application

To create the sample in Visual Studio:

1. Create a new Visual Studio solution in Visual Studio, using the Visual C# Console App template.
2. Install the Newtonsoft.Json NuGet package.
 - a. On the menu, click **Tools**, select **NuGet Package Manager**, then **Manage NuGet Packages for Solution**.
 - b. Click the **Browse** tab, and in the **Search** box type "Newtonsoft.Json".
 - c. Select **Newtonsoft.Json** when it displays, then click the checkbox next to your project name, and **Install**.
3. Copy and paste the code below into the Program.cs file in your solution.
4. Set `imageFilePath` to the path of your own image. You can download a [sample image](#) to use.
5. Run the program.

```
using Newtonsoft.Json.Linq;  
using System;  
using System.IO;  
using System.Linq;  
using System.Net.Http;  
using System.Net.Http.Headers;
```

```

using System.Threading.Tasks;
using System.Web;

namespace CSHttpClientSample
{
    static class Program
    {
        // Add your Computer Vision subscription key and endpoint to your environment variables.
        static string subscriptionKey =
Environment.GetEnvironmentVariable("COMPUTER_VISION_SUBSCRIPTION_KEY");

        // An endpoint should have a format like "https://westus.api.cognitive.microsoft.com"
        static string endpoint = Environment.GetEnvironmentVariable("COMPUTER_VISION_ENDPOINT");

        // the Batch Read method endpoint
        static string uriBase = endpoint + "/vision/v3.0/read/analyze";

        // Add a local image with text here (png or jpg is OK)
        static string imageFilePath = @"my-image.png";

        static void Main(string[] args)
        {
            // Call the REST API method.
            Console.WriteLine("\nExtracting text...\n");
            ReadText(imageFilePath).Wait();

            Console.WriteLine("\nPress Enter to exit...");
            Console.ReadLine();
        }

        /// <summary>
        /// Gets the text from the specified image file by using
        /// the Computer Vision REST API.
        /// </summary>
        /// <param name="imageFilePath">The image file with text.</param>
        static async Task ReadText(string imageFilePath)
        {
            try
            {
                HttpClient client = new HttpClient();

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

                string url = uriBase;

                HttpResponseMessage response;

                // Two REST API methods are required to extract text.
                // One method to submit the image for processing, the other method
                // to retrieve the text found in the image.

                // operationLocation stores the URI of the second REST API method,
                // returned by the first REST API method.
                string operationLocation;

                // Reads the contents of the specified local image
                // into a byte array.
                byte[] byteData = GetImageAsByteArray(imageFilePath);

                // Adds the byte array as an octet stream to the request body.
                using (ByteArrayContent content = new ByteArrayContent(byteData))
                {
                    // This example uses the "application/octet-stream" content type.
                    // The other content types you can use are "application/json"
                    // and "multipart/form-data".
                    content.Headers.ContentType =

```

```

        new MediaTypeHeaderValue("application/octet-stream");

        // The first REST API method, Batch Read, starts
        // the async process to analyze the written text in the image.
        response = await client.PostAsync(url, content);
    }

    // The response header for the Batch Read method contains the URI
    // of the second method, Read Operation Result, which
    // returns the results of the process in the response body.
    // The Batch Read operation does not return anything in the response body.
    if (response.IsSuccessStatusCode)
    {
        operationLocation =
            response.Headers.GetValues("Operation-Location").FirstOrDefault();
    }
    else
    {
        // Display the JSON error data.
        string errorString = await response.Content.ReadAsStringAsync();
        Console.WriteLine("\n\nResponse:\n{0}\n",
            JToken.Parse(errorString).ToString());
        return;
    }

    // If the first REST API method completes successfully, the second
    // REST API method retrieves the text written in the image.
    //
    // Note: The response may not be immediately available. Text
    // recognition is an asynchronous operation that can take a variable
    // amount of time depending on the length of the text.
    // You may need to wait or retry this operation.
    //
    // This example checks once per second for ten seconds.
    string contentString;
    int i = 0;
    do
    {
        System.Threading.Thread.Sleep(1000);
        response = await client.GetAsync(operationLocation);
        contentString = await response.Content.ReadAsStringAsync();
        ++i;
    }
    while (i < 60 && contentString.IndexOf("\"status\":\"succeeded\"") == -1);

    if (i == 60 && contentString.IndexOf("\"status\":\"succeeded\"") == -1)
    {
        Console.WriteLine("\nTimeout error.\n");
        return;
    }

    // Display the JSON response.
    Console.WriteLine("\nResponse:\n\n{0}\n",
        JToken.Parse(contentString).ToString());
}
catch (Exception e)
{
    Console.WriteLine("\n" + e.Message);
}
}

/// <summary>
/// Returns the contents of the specified file as a byte array.
/// </summary>
/// <param name="imageFilePath">The image file to read.</param>
/// <returns>The byte array of the image data.</returns>
static byte[] GetImageAsByteArray(string imageFilePath)
{
    // Open a read-only file stream for the specified file.
    using (FileStream fileStream =
        new FileStream(imageFilePath, FileMode.Open, FileAccess.Read))

```

```

        new FileStream(fileName, FileMode.Open, FileAccess.Read),
        {
            // Read the file's contents into a byte array.
            BinaryReader binaryReader = new BinaryReader(fileStream);
            return binaryReader.ReadBytes((int)fileStream.Length);
        }
    }
}

```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the console window, similar to the following example:

```

{
  "status": "succeeded",
  "createdDateTime": "2020-05-28T05:13:21Z",
  "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
  "analyzeResult": {
    "version": "3.0.0",
    "readResults": [
      {
        "page": 1,
        "language": "en",
        "angle": 0.8551,
        "width": 2661,
        "height": 1901,
        "unit": "pixel",
        "lines": [
          {
            "boundingBox": [
              67,
              646,
              2582,
              713,
              2580,
              876,
              67,
              821
            ],
            "text": "The quick brown fox jumps",
            "words": [
              {
                "boundingBox": [
                  143,
                  650,
                  435,
                  661,
                  436,
                  823,
                  144,
                  824
                ],
                "text": "The",
                "confidence": 0.958
              },
              {
                "boundingBox": [
                  540,
                  665,
                  926,
                  679,
                  926,
                  825,
                  541,

```



```
      823
    ],
    "text": "quick",
    "confidence": 0.57
  },
  {
    "boundingBox": [
      1125,
      686,
      1569,
      700,
      1569,
      838,
      1125,
      828
    ],
    "text": "brown",
    "confidence": 0.799
  },
  {
    "boundingBox": [
      1674,
      703,
      1966,
      711,
      1966,
      851,
      1674,
      841
    ],
    "text": "fox",
    "confidence": 0.442
  },
  {
    "boundingBox": [
      2083,
      714,
      2580,
      725,
      2579,
      876,
      2083,
      855
    ],
    "text": "jumps",
    "confidence": 0.878
  }
]
},
{
  "boundingBox": [
    187,
    1062,
    485,
    1056,
    486,
    1120,
    189,
    1126
  ],
  "text": "over",
  "words": [
    {
      "boundingBox": [
        190,
        1064,
        439,
        1059,
        441,
```

```

        1122,
        192,
        1126
    ],
    "text": "over",
    "confidence": 0.37
}
]
},
{
    "boundingBox": [
        664,
        1008,
        1973,
        1023,
        1969,
        1178,
        664,
        1154
    ],
    "text": "the lazy dog!",
    "words": [
        {
            "boundingBox": [
                668,
                1008,
                923,
                1015,
                923,
                1146,
                669,
                1117
            ],
            "text": "the",
            "confidence": 0.909
        },
        {
            "boundingBox": [
                1107,
                1018,
                1447,
                1023,
                1445,
                1178,
                1107,
                1162
            ],
            "text": "lazy",
            "confidence": 0.853
        },
        {
            "boundingBox": [
                1639,
                1024,
                1974,
                1023,
                1971,
                1170,
                1636,
                1178
            ],
            "text": "dog!",
            "confidence": 0.41
        }
    ]
}
]
}
]
}
]

```

```
}  
}
```

Clean up resources

When no longer needed, delete the Visual Studio solution. To do so, open File Explorer, navigate to the folder in which you created the Visual Studio solution, and delete the folder.

Next steps

Explore a basic Windows application that uses Computer Vision to perform optical character recognition (OCR). Create smart-cropped thumbnails; plus detect, categorize, tag, and describe visual features, including faces, in an image.

[Computer Vision API C# Tutorial](#)

Quickstart: Extract printed and handwritten text using the Computer Vision REST API and Java

9/1/2020 • 5 minutes to read • [Edit Online](#)

In this quickstart, you'll extract printed and handwritten text from an image using the Computer Vision REST API. With the [Read](#) and [Get Read Result](#) methods, you can detect text in an image and extract recognized characters into a machine-readable character stream.

IMPORTANT

The [Read](#) method runs asynchronously. This method does not return any information in the body of a successful response. Instead, the Batch Read method returns a URI in the value of the `Operation-Location` response header field. You can then call this URI, which represents the [Get Read Result](#) API, to both check the status and return the results of the Read method call.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Java™ Platform, Standard Edition Development Kit 7 or 8](#) (JDK 7 or 8)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample application

To create and run the sample, do the following steps:

1. Create a new Java project in your favorite IDE or editor. If the option is available, create the Java project from a command line application template.
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:4.5.5)
 - [Apache HTTP core](#) (org.apache.httpcomponents:httpcore:4.4.9)
 - [JSON library](#) (org.json:json:20180130)
3. Add the following `import` statements to the file that contains the `Main` public class for your project.

```
import java.net.URI;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.utils.URIBuilder;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import org.apache.http.Header;
import org.json.JSONObject;
```

4. Replace the `Main` public class with the following code.
5. Optionally, replace the value of `imageToAnalyze` with the URL of a different image from which you want to extract text.
6. Save, then build the Java project.
7. If you're using an IDE, run `Main`. Otherwise, open a command prompt window and then use the `java` command to run the compiled class. For example, `java Main`.

```
public class Main {

    // Add your Computer Vision subscription key and endpoint to your environment variables.
    // After setting, close and then re-open your command shell or project for the changes to take effect.
    private static String subscriptionKey = System.getenv("COMPUTER_VISION_SUBSCRIPTION_KEY");
    private static String endpoint = System.getenv("COMPUTER_VISION_ENDPOINT");

    private static String uriBase = endpoint + "/vision/v3.0/read/analyze";

    private static String imageToAnalyze =
        "https://raw.githubusercontent.com/MicrosoftDocs/azure-docs/master/articles/cognitive-services/Computer-vision/Images/readsample.jpg";

    public static void main(String[] args) {
        CloseableHttpClient httpTextClient = HttpClientBuilder.create().build();
        CloseableHttpClient httpResultClient = HttpClientBuilder.create().build();

        System.out.println("Endpoint: " + endpoint);
        System.out.println("Subscription key: " + subscriptionKey);

        try {
            // This operation requires two REST API calls. One to submit the image
            // for processing, the other to retrieve the text found in the image.

            URIBuilder builder = new URIBuilder(uriBase);

            // Prepare the URI for the REST API method.
            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 requestEntity =
                new StringEntity("{\"url\":\"" + imageToAnalyze + "\"}");
            request.setEntity(requestEntity);

            // Two REST API methods are required to extract text.
            // One method to submit the image for processing, the other method
```

```

// to retrieve the text found in the image.

// Call the first REST API method to detect the text.
HttpResponse response = httpTextClient.execute(request);

// Check for success.
if (response.getStatusLine().getStatusCode() != 202) {
    // Format and display the JSON error message.
    HttpEntity entity = response.getEntity();
    String jsonString = EntityUtils.toString(entity);
    JSONObject json = new JSONObject(jsonString);
    System.out.println("Error:\n");
    System.out.println(json.toString(2));
    return;
}

// Store the URI of the second REST API method.
// This URI is where you can get the results of the first REST API method.
String operationLocation = null;

// The 'Operation-Location' response header value contains the URI for
// the second REST API method.
Header[] responseHeaders = response.getAllHeaders();
for (Header header : responseHeaders) {
    if (header.getName().equals("Operation-Location")) {
        operationLocation = header.getValue();
        break;
    }
}

if (operationLocation == null) {
    System.out.println("\nError retrieving Operation-Location.\nExiting.");
    System.exit(1);
}

// If the first REST API method completes successfully, the second
// REST API method retrieves the text written in the image.
//
// Note: The response may not be immediately available. Text
// recognition is an asynchronous operation that can take a variable
// amount of time depending on the length of the text.
// You may need to wait or retry this operation.

System.out.println("\nText submitted.\n" +
    "Waiting 10 seconds to retrieve the recognized text.\n");
Thread.sleep(10000);

// Call the second REST API method and get the response.
HttpGet resultRequest = new HttpGet(operationLocation);
resultRequest.setHeader("Ocp-Apim-Subscription-Key", subscriptionKey);

HttpResponse resultResponse = httpResultClient.execute(resultRequest);
HttpEntity responseEntity = resultResponse.getEntity();

if (responseEntity != null) {
    // Format and display the JSON response.
    String jsonString = EntityUtils.toString(responseEntity);
    JSONObject json = new JSONObject(jsonString);
    System.out.println("Text recognition result response: \n");
    System.out.println(json.toString(2));
}
} catch (Exception e) {
    System.out.println(e.getMessage());
}
}
}

```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the console window, similar to the following example:

```
{
  "status": "succeeded",
  "createdDateTime": "2020-05-28T05:13:21Z",
  "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
  "analyzeResult": {
    "version": "3.0.0",
    "readResults": [
      {
        "page": 1,
        "language": "en",
        "angle": 0.8551,
        "width": 2661,
        "height": 1901,
        "unit": "pixel",
        "lines": [
          {
            "boundingBox": [
              67,
              646,
              2582,
              713,
              2580,
              876,
              67,
              821
            ],
            "text": "The quick brown fox jumps",
            "words": [
              {
                "boundingBox": [
                  143,
                  650,
                  435,
                  661,
                  436,
                  823,
                  144,
                  824
                ],
                "text": "The",
                "confidence": 0.958
              },
              {
                "boundingBox": [
                  540,
                  665,
                  926,
                  679,
                  926,
                  825,
                  541,
                  823
                ],
                "text": "quick",
                "confidence": 0.57
              },
              {
                "boundingBox": [
                  1125,
                  686,
                  1569,
                  700,
```

```

        1569,
        838,
        1125,
        828
    ],
    "text": "brown",
    "confidence": 0.799
},
{
    "boundingBox": [
        1674,
        703,
        1966,
        711,
        1966,
        851,
        1674,
        841
    ],
    "text": "fox",
    "confidence": 0.442
},
{
    "boundingBox": [
        2083,
        714,
        2580,
        725,
        2579,
        876,
        2083,
        855
    ],
    "text": "jumps",
    "confidence": 0.878
}
]
},
{
    "boundingBox": [
        187,
        1062,
        485,
        1056,
        486,
        1120,
        189,
        1126
    ],
    "text": "over",
    "words": [
        {
            "boundingBox": [
                190,
                1064,
                439,
                1059,
                441,
                1122,
                192,
                1126
            ],
            "text": "over",
            "confidence": 0.37
        }
    ]
}
],
{
    "boundingBox": [

```



```

        664,
        1008,
        1973,
        1023,
        1969,
        1178,
        664,
        1154
    ],
    "text": "the lazy dog!",
    "words": [
        {
            "boundingBox": [
                668,
                1008,
                923,
                1015,
                923,
                1146,
                669,
                1117
            ],
            "text": "the",
            "confidence": 0.909
        },
        {
            "boundingBox": [
                1107,
                1018,
                1447,
                1023,
                1445,
                1178,
                1107,
                1162
            ],
            "text": "lazy",
            "confidence": 0.853
        },
        {
            "boundingBox": [
                1639,
                1024,
                1974,
                1023,
                1971,
                1170,
                1636,
                1178
            ],
            "text": "dog!",
            "confidence": 0.41
        }
    ]
}

```

Next steps

Next, explore a Java Swing application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

Computer Vision API Java Tutorial

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Extract printed and handwritten text using the Computer Vision REST API and JavaScript

9/1/2020 • 5 minutes to read • [Edit Online](#)

In this quickstart, you'll extract printed and handwritten text from an image using the Computer Vision REST API. With the [Read](#) and [Get Read Result](#) methods, you can detect text in an image and extract recognized characters into a machine-readable character stream.

IMPORTANT

The [Read](#) method runs asynchronously. This method does not return any information in the body of a successful response. Instead, the Batch Read method returns a URI in the value of the `Operation-Location` response header field. You can then call this URI, which represents the [Get Read Result](#) API, to both check the status and return the results of the Read method call.

Prerequisites

- An Azure subscription - [Create one for free](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Create and run the sample

To create and run the sample, do the following steps:

1. Copy the following code into a text editor.
2. Optionally, replace the value of the `value` attribute for the `inputImage` control with the URL of a different image from which you want to extract text.
3. Save the code as a file with an `.html` extension. For example, `get-text.html`.
4. Open a browser window.
5. When the webpage is displayed in the browser, fill the required parameters, and choose the **Read image** button.

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

<script type="text/javascript">
  function processImage() {
    // *****
    // *** Update or verify the following values. ***
    // *****

    let subscriptionKey = document.getElementById("key").value;
```

```

let endpoint = document.getElementById("endpoint").value;
if (!subscriptionKey) { throw new Error('Please enter your subscription key and endpoint.')} }

var uriBase = endpoint + "/vision/v3.0/read/analyze";

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

// This operation requires two REST API calls. One to submit the image
// for processing, the other to retrieve the text found in the image.
//
// Make the first REST API call to submit the image for processing.
$.ajax({
    url: uriBase,

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

    type: "POST",

    // Request body.
    data: JSON.stringify({url: sourceImageUrl}),
})

.done(function(data, textStatus, jqXHR) {
    // Show progress.
    $("#responseTextArea").val("Text submitted. " +
        "Waiting 10 seconds to retrieve the recognized text.");

    // Note: The response may not be immediately available. Text
    // recognition is an asynchronous operation that can take a variable
    // amount of time depending on the length of the text you want to
    // recognize. You may need to wait or retry the GET operation.
    //
    // Wait ten seconds before making the second REST API call.
    setTimeout(function () {
        // "Operation-Location" in the response contains the URI
        // to retrieve the recognized text.
        var operationLocation = jqXHR.getResponseHeader("Operation-Location");

        // Make the second REST API call and get the response.
        $.ajax({
            url: operationLocation,

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

            type: "GET",
        })

        .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 === "") ? "" :

```

```

        errorString += (jQuery.parseJSON(jqXHR.responseText) === null) ?
            (jQuery.parseJSON(jqXHR.responseText).message) ?
                jQuery.parseJSON(jqXHR.responseText).message :
                jQuery.parseJSON(jqXHR.responseText).error.message;
        alert(errorString);
    });
}, 10000);
})

.fail(function(jqXHR, textStatus, errorThrown) {
    // Put the JSON description into the text area.
    $("#responseTextArea").val(JSON.stringify(jqXHR, null, 2));

    // Display error message.
    var errorString = (errorThrown === "") ? "Error. " :
        errorThrown + " (" + jqXHR.status + "): ";
    errorString += (jqXHR.responseText === "") ? "" :
        (jQuery.parseJSON(jqXHR.responseText).message) ?
            jQuery.parseJSON(jqXHR.responseText).message :
            jQuery.parseJSON(jqXHR.responseText).error.message;
    alert(errorString);
});
});
</script>
<h1>Read text from image:</h1>
Enter the URL to an image of text, then click
the <strong>Read image</strong> button.
<br><br>
Endpoint:
<input type="text" name="endpoint" id="endpoint" value="" style="width: 300px;"/>
<div style="margin: 20px;">Example: https://westus2.api.cognitive.microsoft.com</div>
Subscription Key:
<input type="text" name="key" id="key" value="" style="width: 300px;"/>
<br><br>

Image to read:
<input type="text" name="inputImage" id="inputImage"
    value="https://raw.githubusercontent.com/MicrosoftDocs/azure-docs/master/articles/cognitive-
services/Computer-vision/Images/readsample.jpg" />
<button onclick="processImage()">Read image</button>
<br><br>
<div id="wrapper" style="width:1020px; display:table;">
    <div id="jsonOutput" style="width:600px; display:table-cell;">
        Response:
        <br><br>
        <textarea id="responseTextArea" class="UIInput"
            style="width:580px; height:400px;"></textarea>
    </div>
    <div id="imageDiv" style="width:420px; display:table-cell;">
        Source image:
        <br><br>
        <img id="sourceImage" width="400" />
    </div>
</div>
</body>

```

Examine the response

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the browser window, similar to the following example:

```

{
  "status": "succeeded",
  "createdDateTime": "2020-05-28T05:13:21Z",
  "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
  "analyzeResult": {

```

```
"version": "3.0.0",
"readResults": [
  {
    "page": 1,
    "language": "en",
    "angle": 0.8551,
    "width": 2661,
    "height": 1901,
    "unit": "pixel",
    "lines": [
      {
        "boundingBox": [
          67,
          646,
          2582,
          713,
          2580,
          876,
          67,
          821
        ],
        "text": "The quick brown fox jumps",
        "words": [
          {
            "boundingBox": [
              143,
              650,
              435,
              661,
              436,
              823,
              144,
              824
            ],
            "text": "The",
            "confidence": 0.958
          },
          {
            "boundingBox": [
              540,
              665,
              926,
              679,
              926,
              825,
              541,
              823
            ],
            "text": "quick",
            "confidence": 0.57
          },
          {
            "boundingBox": [
              1125,
              686,
              1569,
              700,
              1569,
              838,
              1125,
              828
            ],
            "text": "brown",
            "confidence": 0.799
          },
          {
            "boundingBox": [
              1674,
              703,
```

```

        1966,
        711,
        1966,
        851,
        1674,
        841
    ],
    "text": "fox",
    "confidence": 0.442
},
{
    "boundingBox": [
        2083,
        714,
        2580,
        725,
        2579,
        876,
        2083,
        855
    ],
    "text": "jumps",
    "confidence": 0.878
}
]
},
{
    "boundingBox": [
        187,
        1062,
        485,
        1056,
        486,
        1120,
        189,
        1126
    ],
    "text": "over",
    "words": [
        {
            "boundingBox": [
                190,
                1064,
                439,
                1059,
                441,
                1122,
                192,
                1126
            ],
            "text": "over",
            "confidence": 0.37
        }
    ]
},
{
    "boundingBox": [
        664,
        1008,
        1973,
        1023,
        1969,
        1178,
        664,
        1154
    ],
    "text": "the lazy dog!",
    "words": [
        {

```

```

        "boundingBox": [
            668,
            1008,
            923,
            1015,
            923,
            1146,
            669,
            1117
        ],
        "text": "the",
        "confidence": 0.909
    },
    {
        "boundingBox": [
            1107,
            1018,
            1447,
            1023,
            1445,
            1178,
            1107,
            1162
        ],
        "text": "lazy",
        "confidence": 0.853
    },
    {
        "boundingBox": [
            1639,
            1024,
            1974,
            1023,
            1971,
            1170,
            1636,
            1178
        ],
        "text": "dog!",
        "confidence": 0.41
    }
}
]
}
]
}
]
}
}
}

```

Next steps

Explore a JavaScript application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

[Computer Vision API JavaScript Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Extract printed and handwritten text using the Computer Vision REST API and Python

9/1/2020 • 4 minutes to read • [Edit Online](#)

In this quickstart, you'll extract printed and handwritten text from an image using the Computer Vision REST API. With the [Read](#) and [Get Read Result](#) methods, you can detect text in an image and extract recognized characters into a machine-readable character stream.

IMPORTANT

The [Read](#) method runs asynchronously. This method does not return any information in the body of a successful response. Instead, the Batch Read method returns a URI in the value of the `Operation-Location` response header field. You can then call this URI, which represents the [Get Read Result](#) API, to both check the status and return the results of the Read method call.

You can run this quickstart in a step-by-step fashion using a Jupyter notebook on [MyBinder](#). To launch Binder, select the following button:

[launch binder](#)

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Python](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Copy the following code into a text editor.
2. Optionally, replace the value of `image_url` with the URL of a different image from which you want to extract text.
3. Save the code as a file with an `.py` extension. For example, `get-text.py`.
4. Open a command prompt window.
5. At the prompt, use the `python` command to run the sample. For example, `python get-text.py`.

```
import json
import os
import sys
import requests
import time

# If you are using a Jupyter notebook, uncomment the following line
```

```

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

missing_env = False
# Add your Computer Vision subscription key and endpoint to your environment variables.
if 'COMPUTER_VISION_ENDPOINT' in os.environ:
    endpoint = os.environ['COMPUTER_VISION_ENDPOINT']
else:
    print("From Azure Cognitive Service, retrieve your endpoint and subscription key.")
    print("\nSet the COMPUTER_VISION_ENDPOINT environment variable, such as\n"
          "\"https://westus2.api.cognitive.microsoft.com\".\n")
    missing_env = True

if 'COMPUTER_VISION_SUBSCRIPTION_KEY' in os.environ:
    subscription_key = os.environ['COMPUTER_VISION_SUBSCRIPTION_KEY']
else:
    print("From Azure Cognitive Service, retrieve your endpoint and subscription key.")
    print("\nSet the COMPUTER_VISION_SUBSCRIPTION_KEY environment variable, such as\n"
          "\"1234567890abcdef1234567890abcdef\".\n")
    missing_env = True

if missing_env:
    print("**Restart your shell or IDE for changes to take effect.**")
    sys.exit()

text_recognition_url = endpoint + "/vision/v3.0/read/analyze"

# Set image_url to the URL of an image that you want to recognize.
image_url = "https://raw.githubusercontent.com/MicrosoftDocs/azure-docs/master/articles/cognitive-services/Computer-vision/Images/readsample.jpg"

headers = {'Ocp-Apim-Subscription-Key': subscription_key}
data = {'url': image_url}
response = requests.post(
    text_recognition_url, headers=headers, json=data)
response.raise_for_status()

# Extracting text requires two API calls: One call to submit the
# image for processing, the other to retrieve the text found in the image.

# Holds the URI used to retrieve the recognized text.
operation_url = response.headers["Operation-Location"]

# The recognized text isn't immediately available, so poll to wait for completion.
analysis = {}
poll = True
while (poll):
    response_final = requests.get(
        response.headers["Operation-Location"], headers=headers)
    analysis = response_final.json()

    print(json.dumps(analysis, indent=4))

    time.sleep(1)
    if ("analyzeResult" in analysis):
        poll = False
    if ("status" in analysis and analysis['status'] == 'failed'):
        poll = False

polygons = []
if ("analyzeResult" in analysis):
    # Extract the recognized text, with bounding boxes.
    polygons = [(line["boundingBox"], line["text"])
                for line in analysis["analyzeResult"]["readResults"][0]["lines"]]

```

```
# Display the image and overlay it with the extracted text.
image = Image.open(BytesIO(requests.get(image_url).content))
ax = plt.imshow(image)
for polygon in polygons:
    vertices = [(polygon[0][i], polygon[0][i+1])
                for i in range(0, len(polygon[0]), 2)]
    text = polygon[1]
    patch = Polygon(vertices, closed=True, fill=False, linewidth=2, color='y')
    ax.axes.add_patch(patch)
    plt.text(vertices[0][0], vertices[0][1], text, fontsize=20, va="top")
plt.show()
```

Examine the response

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the command prompt window, similar to the following example:

```
{
  "status": "succeeded",
  "createdDateTime": "2020-05-28T05:13:21Z",
  "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
  "analyzeResult": {
    "version": "3.0.0",
    "readResults": [
      {
        "page": 1,
        "language": "en",
        "angle": 0.8551,
        "width": 2661,
        "height": 1901,
        "unit": "pixel",
        "lines": [
          {
            "boundingBox": [
              67,
              646,
              2582,
              713,
              2580,
              876,
              67,
              821
            ],
            "text": "The quick brown fox jumps",
            "words": [
              {
                "boundingBox": [
                  143,
                  650,
                  435,
                  661,
                  436,
                  823,
                  144,
                  824
                ],
                "text": "The",
                "confidence": 0.958
              },
              {
                "boundingBox": [
                  540,
                  665,
                  926,
                  679,
                  926,
```

```
        825,  
        541,  
        823  
    ],  
    "text": "quick",  
    "confidence": 0.57  
},  
{  
    "boundingBox": [  
        1125,  
        686,  
        1569,  
        700,  
        1569,  
        838,  
        1125,  
        828  
    ],  
    "text": "brown",  
    "confidence": 0.799  
},  
{  
    "boundingBox": [  
        1674,  
        703,  
        1966,  
        711,  
        1966,  
        851,  
        1674,  
        841  
    ],  
    "text": "fox",  
    "confidence": 0.442  
},  
{  
    "boundingBox": [  
        2083,  
        714,  
        2580,  
        725,  
        2579,  
        876,  
        2083,  
        855  
    ],  
    "text": "jumps",  
    "confidence": 0.878  
}  
]  
},  
{  
    "boundingBox": [  
        187,  
        1062,  
        485,  
        1056,  
        486,  
        1120,  
        189,  
        1126  
    ],  
    "text": "over",  
    "words": [  
        {  
            "boundingBox": [  
                190,  
                1064,  
                439,
```

```

        1059,
        441,
        1122,
        192,
        1126
    ],
    "text": "over",
    "confidence": 0.37
}
]
},
{
    "boundingBox": [
        664,
        1008,
        1973,
        1023,
        1969,
        1178,
        664,
        1154
    ],
    "text": "the lazy dog!",
    "words": [
        {
            "boundingBox": [
                668,
                1008,
                923,
                1015,
                923,
                1146,
                669,
                1117
            ],
            "text": "the",
            "confidence": 0.909
        },
        {
            "boundingBox": [
                1107,
                1018,
                1447,
                1023,
                1445,
                1178,
                1107,
                1162
            ],
            "text": "lazy",
            "confidence": 0.853
        },
        {
            "boundingBox": [
                1639,
                1024,
                1974,
                1023,
                1971,
                1170,
                1636,
                1178
            ],
            "text": "dog!",
            "confidence": 0.41
        }
    ]
}
]
}
1

```

```
}  
}  
}  
}
```

Next steps

Next, explore a Python application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

[Computer Vision API Python Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Extract text using the Computer Vision 2.0 REST API OCR operation and C#

9/1/2020 • 4 minutes to read • [Edit Online](#)

IMPORTANT

If you're extracting text in English, Dutch, French, German, Italian, Portuguese, Spanish, or Simplified Chinese (preview), we recommend you use the newer [Read operation](#). A [C# quickstart](#) is available.

In this quickstart, you'll extract printed text from an image using the Computer Vision REST API [OCR operation](#) feature. With this operation, you can detect printed text in an image and extract recognized characters into a machine-usable character stream.

Prerequisites

- An Azure subscription - [Create one for free](#)
- You must have [Visual Studio 2015](#) or later
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample application

To create the sample in Visual Studio, do the following steps:

1. Create a new Visual Studio solution in Visual Studio, using the Visual C# Console App template.
2. Install the Newtonsoft.Json NuGet package.
 - a. On the menu, click **Tools**, select **NuGet Package Manager**, then **Manage NuGet Packages for Solution**.
 - b. Click the **Browse** tab, and in the **Search** box type "Newtonsoft.Json".
 - c. Select **Newtonsoft.Json** when it displays, then click the checkbox next to your project name, and **Install**.
3. Run the program.
4. At the prompt, enter the path to a local image.

```
using Newtonsoft.Json.Linq;
using System;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;

namespace CSHttpClientSample
{
    static class Program
    {
```

```

// Add your Computer Vision subscription key and endpoint to your environment variables.
static string subscriptionKey = Environment.GetEnvironmentVariable("COMPUTER_VISION_SUBSCRIPTION_KEY");

static string endpoint = Environment.GetEnvironmentVariable("COMPUTER_VISION_ENDPOINT");

// the OCR method endpoint
static string uriBase = endpoint + "vision/v2.1/ocr";

static async Task Main()
{
    // Get the path and filename to process from the user.
    Console.WriteLine("Optical Character Recognition:");
    Console.Write("Enter the path to an image with text you wish to read: ");
    string imageFilePath = Console.ReadLine();

    if (File.Exists(imageFilePath))
    {
        // Call the REST API method.
        Console.WriteLine("\nWait a moment for the results to appear.\n");
        await MakeOCRRequest(imageFilePath);
    }
    else
    {
        Console.WriteLine("\nInvalid file path");
    }
    Console.WriteLine("\nPress Enter to exit...");
    Console.ReadLine();
}

/// <summary>
/// Gets the text visible in the specified image file by using
/// the Computer Vision REST API.
/// </summary>
/// <param name="imageFilePath">The image file with printed text.</param>
static async Task MakeOCRRequest(string imageFilePath)
{
    try
    {
        HttpClient client = new HttpClient();

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

        // Request parameters.
        // The language parameter doesn't specify a language, so the
        // method detects it automatically.
        // The detectOrientation parameter is set to true, so the method detects and
        // and corrects text orientation before detecting text.
        string requestParameters = "language=unk&detectOrientation=true";

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

        HttpResponseMessage response;

        // Read the contents of the specified local image
        // into a byte array.
        byte[] byteData = GetImageAsByteArray(imageFilePath);

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

```



```

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

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

    // Display the JSON response.
    Console.WriteLine("\nResponse:\n\n{0}\n",
        JToken.Parse(contentString).ToString());
}
catch (Exception e)
{
    Console.WriteLine("\n" + e.Message);
}
}

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

```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the console window, similar to the following example:

```

{
  "language": "en",
  "textAngle": -1.5000000000000335,
  "orientation": "Up",
  "regions": [
    {
      "boundingBox": "154,49,351,575",
      "lines": [
        {
          "boundingBox": "165,49,340,117",
          "words": [
            {
              "boundingBox": "165,49,63,109",
              "text": "A"
            },
            {
              "boundingBox": "261,50,244,116",
              "text": "GOAL"
            }
          ]
        }
      ],
    },
    {
      "boundingBox": "165,169,339,93",
      "words": [
        {

```

```

        "boundingBox": "165,169,339,93",
        "text": "WITHOUT"
    }
]
},
{
    "boundingBox": "159,264,342,117",
    "words": [
        {
            "boundingBox": "159,264,64,110",
            "text": "A"
        },
        {
            "boundingBox": "255,266,246,115",
            "text": "PLAN"
        }
    ]
},
{
    "boundingBox": "161,384,338,119",
    "words": [
        {
            "boundingBox": "161,384,86,113",
            "text": "IS"
        },
        {
            "boundingBox": "274,387,225,116",
            "text": "JUST"
        }
    ]
},
{
    "boundingBox": "154,506,341,118",
    "words": [
        {
            "boundingBox": "154,506,62,111",
            "text": "A"
        },
        {
            "boundingBox": "248,508,247,116",
            "text": "WISH"
        }
    ]
}
]
}
]
}
}

```

Next steps

Explore a basic Windows application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; plus detect, categorize, tag, and describe visual features, including faces, in an image.

[Computer Vision API C# Tutorial](#)

Quickstart: Extract printed text (OCR) using the Computer Vision REST API and cURL

9/1/2020 • 2 minutes to read • [Edit Online](#)

In this quickstart, you'll extract printed text with optical character recognition (OCR) from an image using the Computer Vision REST API. With the [OCR](#) method, you can detect printed text in an image and extract recognized characters into a machine-usable character stream.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [cURL](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Create and run the sample command

To create and run the sample, do the following steps:

1. Copy the following command into a text editor.
2. Make the following changes in the command where needed:
 - a. Replace the value of `<subscriptionKey>` with your subscription key.
 - b. Replace the first part of the request URL (`westcentralus`) with the text in your own endpoint URL.

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](#).

- c. Optionally, change the image URL in the request body (

```
https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/Atomist_quote_from_Democritus.png/338px-Atomist_quote_from_Democritus.png\
```

) to the URL of a different image to be analyzed.

3. Open a command prompt window.
4. Paste the command from the text editor into the command prompt window, and then run the command.

```
curl -H "Ocp-Apim-Subscription-Key: <subscriptionKey>" -H "Content-Type: application/json"
"https://westcentralus.api.cognitive.microsoft.com/vision/v3.0/ocr?language=unk&detectOrientation=true" -d "
{ \"url\": \"https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/Atomist_quote_from_Democritus.png/338px-
Atomist_quote_from_Democritus.png\" }
```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the command prompt window, similar to the following example:

```

{
  "language": "en",
  "orientation": "Up",
  "textAngle": 0,
  "regions": [
    {
      "boundingBox": "21,16,304,451",
      "lines": [
        {
          "boundingBox": "28,16,288,41",
          "words": [
            {
              "boundingBox": "28,16,288,41",
              "text": "NOTHING"
            }
          ]
        },
        {
          "boundingBox": "27,66,283,52",
          "words": [
            {
              "boundingBox": "27,66,283,52",
              "text": "EXISTS"
            }
          ]
        },
        {
          "boundingBox": "27,128,292,49",
          "words": [
            {
              "boundingBox": "27,128,292,49",
              "text": "EXCEPT"
            }
          ]
        },
        {
          "boundingBox": "24,188,292,54",
          "words": [
            {
              "boundingBox": "24,188,292,54",
              "text": "ATOMS"
            }
          ]
        },
        {
          "boundingBox": "22,253,297,32",
          "words": [
            {
              "boundingBox": "22,253,105,32",
              "text": "AND"
            },
            {
              "boundingBox": "144,253,175,32",
              "text": "EMPTY"
            }
          ]
        },
        {
          "boundingBox": "21,298,304,60",
          "words": [
            {
              "boundingBox": "21,298,304,60",
              "text": "SPACE."
            }
          ]
        },
        {
          "boundingBox": "26,387,294,37",

```

```
    "words": [
      {
        "boundingBox": "26,387,210,37",
        "text": "Everything"
      },
      {
        "boundingBox": "249,389,71,27",
        "text": "else"
      }
    ]
  },
  {
    "boundingBox": "127,431,198,36",
    "words": [
      {
        "boundingBox": "127,431,31,29",
        "text": "is"
      },
      {
        "boundingBox": "172,431,153,36",
        "text": "opinion."
      }
    ]
  }
]
}
```

Next steps

Explore the Computer Vision API used to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

[Explore the Computer Vision API](#)

Quickstart: Extract printed text (OCR) using the Computer Vision REST API with Go

9/1/2020 • 3 minutes to read • [Edit Online](#)

NOTE

If you're extracting English language text, consider using the new [Read operation](#). A [Go quickstart](#) is available.

In this quickstart, you will extract printed text with optical character recognition (OCR) from an image using the Computer Vision REST API. With the [OCR](#) method, you can detect printed text in an image and extract recognized characters into a machine-usable character stream.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Go](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Copy the following code into a text editor.
2. Optionally, replace the value of `imageUrl` with the URL of a different image that you want to analyze.
3. Save the code as a file with a `.go` extension. For example, `get-printed-text.go`.
4. Open a command prompt window.
5. At the prompt, run the `go build` command to compile the package from the file. For example,
`go build get-printed-text.go`.
6. At the prompt, run the compiled package. For example, `get-printed-text`.

```

package main

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

func main() {
    // Add your Computer Vision subscription key and endpoint to your environment variables.
    // Add your Computer Vision subscription key and endpoint to your environment variables.
    subscriptionKey := os.Getenv("COMPUTER_VISION_SUBSCRIPTION_KEY")
    endpoint := os.Getenv("COMPUTER_VISION_ENDPOINT")

    uriBase := endpoint + "vision/v3.0/ocr"
    const imageUrl = "https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/" +
        "Atomist_quote_from_Democritus.png/338px-Atomist_quote_from_Democritus.png"

    params := "?language=unk&detectOrientation=true"
    uri := uriBase + params
    const imageUrlEnc = "{\"url\":\"" + imageUrl + "\"}"

    reader := strings.NewReader(imageUrlEnc)

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

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

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

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

    defer resp.Body.Close()

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

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

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

```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the command prompt window, similar to the following example:

```
{
  "language": "en",
  "orientation": "Up",
  "regions": [
    {
      "boundingBox": "21,16,304,451",
      "lines": [
        {
          "boundingBox": "28,16,288,41",
          "words": [
            {
              "boundingBox": "28,16,288,41",
              "text": "NOTHING"
            }
          ]
        },
        {
          "boundingBox": "27,66,283,52",
          "words": [
            {
              "boundingBox": "27,66,283,52",
              "text": "EXISTS"
            }
          ]
        },
        {
          "boundingBox": "27,128,292,49",
          "words": [
            {
              "boundingBox": "27,128,292,49",
              "text": "EXCEPT"
            }
          ]
        },
        {
          "boundingBox": "24,188,292,54",
          "words": [
            {
              "boundingBox": "24,188,292,54",
              "text": "ATOMS"
            }
          ]
        },
        {
          "boundingBox": "22,253,297,32",
          "words": [
            {
              "boundingBox": "22,253,105,32",
              "text": "AND"
            },
            {
              "boundingBox": "144,253,175,32",
              "text": "EMPTY"
            }
          ]
        },
        {
          "boundingBox": "21,298,304,60",
          "words": [
            {
              "boundingBox": "21,298,304,60",
              "text": "SPACE."
            }
          ]
        }
      ]
    }
  ]
}
```



```
    }
  ]
},
{
  "boundingBox": "26,387,294,37",
  "words": [
    {
      "boundingBox": "26,387,210,37",
      "text": "Everything"
    },
    {
      "boundingBox": "249,389,71,27",
      "text": "else"
    }
  ]
},
{
  "boundingBox": "127,431,198,36",
  "words": [
    {
      "boundingBox": "127,431,31,29",
      "text": "is"
    },
    {
      "boundingBox": "172,431,153,36",
      "text": "opinion."
    }
  ]
}
]
}
],
"textAngle": 0
}
```

Next steps

Explore the Computer Vision API used to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text. To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

[Explore the Computer Vision API](#)

Quickstart: Extract printed text (OCR) using the Computer Vision REST API and Java

9/1/2020 • 3 minutes to read • [Edit Online](#)

NOTE

If you're extracting English language text, consider using the new [Read operation](#). A [Java quickstart](#) is available.

In this quickstart, you'll extract printed text with optical character recognition (OCR) from an image using the Computer Vision REST API. With the [OCR](#) method, you can detect printed text in an image and extract recognized characters into a machine-usable character stream.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Java™ Platform, Standard Edition Development Kit 7 or 8](#) (JDK 7 or 8)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample application

To create and run the sample, do the following steps:

1. Create a new Java project in your favorite IDE or editor. If the option is available, create the Java project from a command line application template.
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:4.5.5)
 - [Apache HTTP core](#) (org.apache.httpcomponents:httpcore:4.4.9)
 - [JSON library](#) (org.json:json:20180130)
3. Add the following `import` statements to the file that contains the `Main` public class for your project.

```
import java.net.URI;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
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.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import org.json.JSONObject;
```

4. Replace the `Main` public class with the following code.
5. Optionally, replace the value of `imageToAnalyze` with the URL of a different image from which you want to extract printed text.
6. Save, then build the Java project.
7. If you're using an IDE, run `Main`. Otherwise, open a command prompt window and then use the `java` command to run the compiled class. For example, `java Main`.

```
public class Main {

    // Add your Computer Vision subscription key and endpoint to your environment variables.
    private static String subscriptionKey = System.getenv("COMPUTER_VISION_SUBSCRIPTION_KEY");
    private static String endpoint = System.getenv("COMPUTER_VISION_ENDPOINT");

    private static final String uriBase = endpoint + "vision/v3.0/ocr";

    private static final String imageToAnalyze =
        "https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/" +
        "Atomist_quote_from_Democritus.png/338px-Atomist_quote_from_Democritus.png";

    public static void main(String[] args) {
        CloseableHttpClient httpClient = HttpClientBuilder.create().build();

        try {
            URIBuilder uriBuilder = new URIBuilder(uriBase);

            uriBuilder.setParameter("language", "unk");
            uriBuilder.setParameter("detectOrientation", "true");

            // Request parameters.
            URI uri = uriBuilder.build();
            HttpPost request = new HttpPost(uri);

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

            // Request body.
            StringEntity requestEntity =
                new StringEntity("{\"url\":\"" + imageToAnalyze + "\"}");
            request.setEntity(requestEntity);

            // Call the REST API method and get the response entity.
            HttpResponse response = httpClient.execute(request);
            HttpEntity entity = response.getEntity();

            if (entity != null) {
                // Format and display the JSON response.
                String jsonString = EntityUtils.toString(entity);
                JSONObject json = new JSONObject(jsonString);
                System.out.println("REST Response:\n");
                System.out.println(json.toString(2));
            }
        } catch (Exception e) {
            // Display error message.
            System.out.println(e.getMessage());
        }
    }
}
```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the console window, similar to the following example:

REST Response:

```
{
  "orientation": "Up",
  "regions": [{
    "boundingBox": "21,16,304,451",
    "lines": [
      {
        "boundingBox": "28,16,288,41",
        "words": [{
          "boundingBox": "28,16,288,41",
          "text": "NOTHING"
        }]
      },
      {
        "boundingBox": "27,66,283,52",
        "words": [{
          "boundingBox": "27,66,283,52",
          "text": "EXISTS"
        }]
      },
      {
        "boundingBox": "27,128,292,49",
        "words": [{
          "boundingBox": "27,128,292,49",
          "text": "EXCEPT"
        }]
      },
      {
        "boundingBox": "24,188,292,54",
        "words": [{
          "boundingBox": "24,188,292,54",
          "text": "ATOMS"
        }]
      },
      {
        "boundingBox": "22,253,297,32",
        "words": [
          {
            "boundingBox": "22,253,105,32",
            "text": "AND"
          },
          {
            "boundingBox": "144,253,175,32",
            "text": "EMPTY"
          }
        ]
      },
      {
        "boundingBox": "21,298,304,60",
        "words": [{
          "boundingBox": "21,298,304,60",
          "text": "SPACE."
        }]
      },
      {
        "boundingBox": "26,387,294,37",
        "words": [
          {
            "boundingBox": "26,387,210,37",
            "text": "Everything"
          },
          {
            "boundingBox": "249,389,71,27",
            "text": "..."
          }
        ]
      }
    ]
  }
}
```

```

        "text": "else"
    }
]
},
{
    "boundingBox": "127,431,198,36",
    "words": [
        {
            "boundingBox": "127,431,31,29",
            "text": "is"
        },
        {
            "boundingBox": "172,431,153,36",
            "text": "opinion."
        }
    ]
}
]
}
},
"textAngle": 0,
"language": "en"
}

```

Next steps

Explore a Java Swing application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

[Computer Vision API Java Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Extract printed text (OCR) using the Computer Vision REST API and JavaScript

9/1/2020 • 3 minutes to read • [Edit Online](#)

NOTE

If you're extracting English language text, consider using the new [Read operation](#). A [JavaScript quickstart](#) is available.

In this quickstart, you'll extract printed text with optical character recognition (OCR) from an image using the Computer Vision REST API. With the [OCR](#) method, you can detect printed text in an image and extract recognized characters into a machine-usable character stream.

Prerequisites

- An Azure subscription - [Create one for free](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (**F0**) to try the service, and upgrade later to a paid tier for production.

Create and run the sample

To create and run the sample, do the following steps:

1. Create a file called *get-printed-text.html*, open it in a text editor, and copy the following code into it.
2. Optionally, replace the value of the `value` attribute for the `inputImage` control with the URL of a different image that you want to analyze.
3. Open a browser window.
4. In the browser, drag and drop the file into the browser window.
5. When the webpage is displayed in the browser, paste your subscription key and endpoint URL into the appropriate input boxes.
6. Select the **Read image** button.

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

<script type="text/javascript">
  function processImage() {
    // *****
    // *** Update or verify the following values. ***
    // *****

    var subscriptionKey = document.getElementById("subscriptionKey").value;
    var endpoint = document.getElementById("endpointUrl").value;
```

```

var uriBase = endpoint + "vision/v3.0/ocr";

// Request parameters.
var params = {
    "language": "unk",
    "detectOrientation": "true",
};

// 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(jqXHR){
        jqXHR.setRequestHeader("Content-Type","application/json");
        jqXHR.setRequestHeader("Ocp-Apim-Subscription-Key", subscriptionKey);
    },

    type: "POST",

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

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

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

<h1>Optical Character Recognition (OCR):</h1>
Enter the URL to an image of printed text, then
click the <strong>Read image</strong> button.
<br><br>
Subscription key:
<input type="text" name="subscriptionKey" id="subscriptionKey"
    value="" />
Endpoint URL:
<input type="text" name="endpointUrl" id="endpointUrl"
    value="" />
<br><br>
Image to read:
<input type="text" name="inputImage" id="inputImage"
    value="https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/Atomist_quote_from_Democritus.png/338px-
Atomist_quote_from_Democritus.png" />
<button onclick="processImage()">Read image</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>

```

```

        style="width:420px; height:100px; text-align:center;">
    </div>
    <div id="imageDiv" style="width:420px; display:table-cell;">
        Source image:
        <br><br>
        <img id="sourceImage" width="400" />
    </div>
</div>
</body>
</html>

```

Examine the response

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the browser window, similar to the following example:

```

{
  "language": "en",
  "orientation": "Up",
  "textAngle": 0,
  "regions": [
    {
      "boundingBox": "21,16,304,451",
      "lines": [
        {
          "boundingBox": "28,16,288,41",
          "words": [
            {
              "boundingBox": "28,16,288,41",
              "text": "NOTHING"
            }
          ]
        }
      ],
    },
    {
      "boundingBox": "27,66,283,52",
      "words": [
        {
          "boundingBox": "27,66,283,52",
          "text": "EXISTS"
        }
      ]
    },
    {
      "boundingBox": "27,128,292,49",
      "words": [
        {
          "boundingBox": "27,128,292,49",
          "text": "EXCEPT"
        }
      ]
    },
    {
      "boundingBox": "24,188,292,54",
      "words": [
        {
          "boundingBox": "24,188,292,54",
          "text": "ATOMS"
        }
      ]
    },
    {
      "boundingBox": "22,253,297,32",
      "words": [
        {
          "boundingBox": "22,253,105,32",
          "text": "AND"
        }
      ]
    }
  ]
}

```



```

    },
    {
      "boundingBox": "144,253,175,32",
      "text": "EMPTY"
    }
  ]
},
{
  "boundingBox": "21,298,304,60",
  "words": [
    {
      "boundingBox": "21,298,304,60",
      "text": "SPACE."
    }
  ]
},
{
  "boundingBox": "26,387,294,37",
  "words": [
    {
      "boundingBox": "26,387,210,37",
      "text": "Everything"
    },
    {
      "boundingBox": "249,389,71,27",
      "text": "else"
    }
  ]
},
{
  "boundingBox": "127,431,198,36",
  "words": [
    {
      "boundingBox": "127,431,31,29",
      "text": "is"
    },
    {
      "boundingBox": "172,431,153,36",
      "text": "opinion."
    }
  ]
}
]
}
]
}
}

```

Next steps

Next, explore a JavaScript application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

[Computer Vision API JavaScript Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Extract printed text (OCR) using the Computer Vision REST API and Node.js

9/1/2020 • 3 minutes to read • [Edit Online](#)

NOTE

If you're extracting English language text, consider using the new [Read operation](#).

In this quickstart, you'll extract printed text with optical character recognition (OCR) from an image using the Computer Vision REST API. With the [OCR](#) method, you can detect printed text in an image and extract recognized characters into a machine-usable character stream.

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Node.js](#) 4.x or later
- [npm](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Install the npm `request` package.
 - a. Open a command prompt window as an administrator.
 - b. Run the following command:

```
npm install request
```
 - c. After the package is successfully installed, close the command prompt window.
2. Copy the following code into a text editor.
3. Optionally, replace the value of `imageUri` with the URL of a different image from which you want to extract printed text.
4. Save the code as a file with a `.js` extension. For example, `get-printed-text.js`.
5. Open a command prompt window.
6. At the prompt, use the `node` command to run the file. For example, `node get-printed-text.js`.

```

'use strict';

const request = require('request');

let subscriptionKey = process.env['COMPUTER_VISION_SUBSCRIPTION_KEY'];
let endpoint = process.env['COMPUTER_VISION_ENDPOINT']
if (!subscriptionKey) { throw new Error('Set your environment variables for your subscription key and endpoint.')}

var uriBase = endpoint + 'vision/v3.0/ocr';

const imageUrl = 'https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/' +
  'Atomist_quote_from_Democritus.png/338px-Atomist_quote_from_Democritus.png';

// Request parameters.
const params = {
  'language': 'unk',
  'detectOrientation': 'true',
};

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);
});

```

Examine the response

A successful response is returned in JSON. The sample parses and displays a successful response in the command prompt window, similar to the following example:

```

{
  "language": "en",
  "orientation": "Up",
  "textAngle": 0,
  "regions": [
    {
      "boundingBox": "21,16,304,451",
      "lines": [
        {
          "boundingBox": "28,16,288,41",
          "words": [
            {
              "boundingBox": "28,16,288,41",
              "text": "NOTHING"
            }
          ]
        }
      ],
    },
    {
      "boundingBox": "27,66,283,52",
      "words": [

```

```

    {
      "boundingBox": "27,66,283,52",
      "text": "EXISTS"
    }
  ]
},
{
  "boundingBox": "27,128,292,49",
  "words": [
    {
      "boundingBox": "27,128,292,49",
      "text": "EXCEPT"
    }
  ]
},
{
  "boundingBox": "24,188,292,54",
  "words": [
    {
      "boundingBox": "24,188,292,54",
      "text": "ATOMS"
    }
  ]
},
{
  "boundingBox": "22,253,297,32",
  "words": [
    {
      "boundingBox": "22,253,105,32",
      "text": "AND"
    },
    {
      "boundingBox": "144,253,175,32",
      "text": "EMPTY"
    }
  ]
},
{
  "boundingBox": "21,298,304,60",
  "words": [
    {
      "boundingBox": "21,298,304,60",
      "text": "SPACE."
    }
  ]
},
{
  "boundingBox": "26,387,294,37",
  "words": [
    {
      "boundingBox": "26,387,210,37",
      "text": "Everything"
    },
    {
      "boundingBox": "249,389,71,27",
      "text": "else"
    }
  ]
},
{
  "boundingBox": "127,431,198,36",
  "words": [
    {
      "boundingBox": "127,431,31,29",
      "text": "is"
    },
    {
      "boundingBox": "172,431,153,36",
      "text": "opinion "
    }
  ]
}

```

```
    text : opinion.  
  }  
}  
}  
}  
}  
}  
}
```

Clean up resources

When no longer needed, delete the file, and then uninstall the npm `request` package. To uninstall the package, do the following steps:

1. Open a command prompt window as an administrator.
2. Run the following command:

```
npm uninstall request
```

3. After the package is successfully uninstalled, close the command prompt window.

Next steps

Next, explore the Computer Vision APIs used to analyze an image, detect celebrities and landmarks, create a thumbnail, and extract printed and handwritten text.

[Explore the Computer Vision API](#)

Quickstart: Extract printed text (OCR) using the Computer Vision REST API and Python

9/1/2020 • 3 minutes to read • [Edit Online](#)

NOTE

If you're extracting English language text, consider using the new [Read operation](#). A [Python quickstart](#) is available.

In this quickstart, you will extract printed text with optical character recognition (OCR) from an image using the Computer Vision REST API. With the [OCR](#) method, you can detect printed text in an image and extract recognized characters into a machine-usable character stream.

You can run this quickstart in a step-by-step fashion using a Jupyter notebook on [MyBinder](#). To launch Binder, select the following button:

launch [binder](#)

Prerequisites

- An Azure subscription - [Create one for free](#)
- You must have [Python](#) installed if you want to run the sample locally.
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the sample

To create and run the sample, do the following steps:

1. Copy the following code into a text editor.
2. Optionally, replace the value of `image_url` with the URL of a different image from which you want to extract printed text.
3. Save the code as a file with an `.py` extension. For example, `get-printed-text.py`.
4. Open a command prompt window.
5. At the prompt, use the `python` command to run the sample. For example, `python get-printed-text.py`.

```

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

# Add your Computer Vision subscription key and endpoint to your environment variables.
if 'COMPUTER_VISION_SUBSCRIPTION_KEY' in os.environ:
    subscription_key = os.environ['COMPUTER_VISION_SUBSCRIPTION_KEY']
else:
    print("\nSet the COMPUTER_VISION_SUBSCRIPTION_KEY environment variable.\n**Restart your shell or IDE for changes to take effect.**")
    sys.exit()

if 'COMPUTER_VISION_ENDPOINT' in os.environ:
    endpoint = os.environ['COMPUTER_VISION_ENDPOINT']

ocr_url = endpoint + "vision/v3.0/ocr"

# Set image_url to the URL of an image that you want to analyze.
image_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/" + \
    "Atomist_quote_from_Democritus.png/338px-Atomist_quote_from_Democritus.png"

headers = {'Ocp-Apim-Subscription-Key': subscription_key}
params = {'language': 'unk', 'detectOrientation': 'true'}
data = {'url': image_url}
response = requests.post(ocr_url, headers=headers, params=params, json=data)
response.raise_for_status()

analysis = response.json()

# Extract the word bounding boxes and text.
line_infos = [region["lines"] for region in analysis["regions"]]
word_infos = []
for line in line_infos:
    for word_metadata in line:
        for word_info in word_metadata["words"]:
            word_infos.append(word_info)
word_infos

# Display the image and overlay it with the extracted text.
plt.figure(figsize=(5, 5))
image = Image.open(BytesIO(requests.get(image_url).content))
ax = plt.imshow(image, alpha=0.5)
for word in word_infos:
    bbox = [int(num) for num in word["boundingBox"].split(",")]
    text = word["text"]
    origin = (bbox[0], bbox[1])
    patch = Rectangle(origin, bbox[2], bbox[3],
                      fill=False, linewidth=2, color='y')
    ax.axes.add_patch(patch)
    plt.text(origin[0], origin[1], text, fontsize=20, weight="bold", va="top")
plt.show()
plt.axis("off")

```

Upload image from local storage

If you want to analyze a local image, set the Content-Type header to application/octet-stream, and set the request body to a byte array instead of JSON data.

```

image_path = "<path-to-local-image-file>"
# Read the image into a byte array
image_data = open(image_path, "rb").read()
# Set Content-Type to octet-stream
headers = {'Ocp-Apim-Subscription-Key': subscription_key, 'Content-Type': 'application/octet-stream'}
# put the byte array into your post request
response = requests.post(ocr_url, headers=headers, params=params, data = image_data)

```

Examine the response

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the command prompt window, similar to the following example:

```

{
  "language": "en",
  "orientation": "Up",
  "textAngle": 0,
  "regions": [
    {
      "boundingBox": "21,16,304,451",
      "lines": [
        {
          "boundingBox": "28,16,288,41",
          "words": [
            {
              "boundingBox": "28,16,288,41",
              "text": "NOTHING"
            }
          ]
        },
        {
          "boundingBox": "27,66,283,52",
          "words": [
            {
              "boundingBox": "27,66,283,52",
              "text": "EXISTS"
            }
          ]
        },
        {
          "boundingBox": "27,128,292,49",
          "words": [
            {
              "boundingBox": "27,128,292,49",
              "text": "EXCEPT"
            }
          ]
        },
        {
          "boundingBox": "24,188,292,54",
          "words": [
            {
              "boundingBox": "24,188,292,54",
              "text": "ATOMS"
            }
          ]
        },
        {
          "boundingBox": "22,253,297,32",
          "words": [
            {
              "boundingBox": "22,253,105,32",
              "text": "AND"
            }
          ]
        }
      ]
    }
  ]
}

```



```

        "boundingBox": "144,253,175,32",
        "text": "EMPTY"
    }
]
},
{
    "boundingBox": "21,298,304,60",
    "words": [
        {
            "boundingBox": "21,298,304,60",
            "text": "SPACE."
        }
    ]
},
{
    "boundingBox": "26,387,294,37",
    "words": [
        {
            "boundingBox": "26,387,210,37",
            "text": "Everything"
        },
        {
            "boundingBox": "249,389,71,27",
            "text": "else"
        }
    ]
},
{
    "boundingBox": "127,431,198,36",
    "words": [
        {
            "boundingBox": "127,431,31,29",
            "text": "is"
        },
        {
            "boundingBox": "172,431,153,36",
            "text": "opinion."
        }
    ]
}
]
}
]
}
}

```

Next steps

Next, explore a Python application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

[Computer Vision API Python Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Quickstart: Use a domain model using the REST API and Python in Computer Vision

9/1/2020 • 4 minutes to read • [Edit Online](#)

In this quickstart, you'll use a domain model to identify landmarks or, optionally, celebrities in a remotely stored image using the Computer Vision REST API. With the [Recognize Domain Specific Content](#) method, you can apply a domain-specific model to recognize content within an image.

You can run this quickstart in a step-by-step fashion using a Jupyter notebook on [MyBinder](#). To launch Binder, select the following button:

launch binder

Prerequisites

- An Azure subscription - [Create one for free](#)
- [Python](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (F0) to try the service, and upgrade later to a paid tier for production.
- [Create environment variables](#) for the key and endpoint URL, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT`, respectively.

Create and run the landmarks sample

To create and run the landmark sample, do the following steps:

1. Copy the following code into a text editor.
2. Optionally, replace the value of `image_url` with the URL of a different image in which you want to detect landmarks.
3. Save the code as a file with an `.py` extension. For example, `get-landmarks.py`.
4. Open a command prompt window.
5. At the prompt, use the `python` command to run the sample. For example, `python get-landmarks.py`.

```

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

# Add your Computer Vision subscription key and endpoint to your environment variables.
if 'COMPUTER_VISION_SUBSCRIPTION_KEY' in os.environ:
    subscription_key = os.environ['COMPUTER_VISION_SUBSCRIPTION_KEY']
else:
    print("\nSet the COMPUTER_VISION_SUBSCRIPTION_KEY environment variable.\n**Restart your shell or IDE for changes to take effect.**")
    sys.exit()

if 'COMPUTER_VISION_ENDPOINT' in os.environ:
    endpoint = os.environ['COMPUTER_VISION_ENDPOINT']

landmark_analyze_url = endpoint + "vision/v3.0/models/landmarks/analyze"

# Set image_url to the URL of an image that you want to analyze.
image_url = "https://upload.wikimedia.org/wikipedia/commons/f/f6/" + \
    "Bunker_Hill_Monument_2005.jpg"

headers = {'Ocp-Apim-Subscription-Key': subscription_key}
params = {'model': 'landmarks'}
data = {'url': image_url}
response = requests.post(
    landmark_analyze_url, headers=headers, params=params, json=data)
response.raise_for_status()

# The 'analysis' object contains various fields that describe the image. The
# most relevant landmark for the image is obtained from the 'result' property.
analysis = response.json()
assert analysis["result"]["landmarks"] is not []
print(analysis)
landmark_name = analysis["result"]["landmarks"][0]["name"].capitalize()

# Display the image and overlay it with the landmark name.
image = Image.open(BytesIO(requests.get(image_url).content))
plt.imshow(image)
plt.axis("off")
_ = plt.title(landmark_name, size="x-large", y=-0.1)
plt.show()

```

Examine the response for the landmarks sample

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the command prompt window, similar to the following example:

```
{
  "result": {
    "landmarks": [
      {
        "name": "Bunker Hill Monument",
        "confidence": 0.9768505096435547
      }
    ]
  },
  "requestId": "659a10cd-44bb-44db-9147-a295b853b2b8",
  "metadata": {
    "height": 1600,
    "width": 1200,
    "format": "Jpeg"
  }
}
```

Create and run the celebrities sample

To create and run the landmark sample, do 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. Replace the value of `vision_base_url` with the endpoint URL for the Computer Vision resource in the Azure region where you obtained your subscription keys, if necessary.
 - c. Optionally, replace the value of `image_url` with the URL of a different image in which you want to detect celebrities.
3. Save the code as a file with an `.py` extension. For example, `get-celebrities.py`.
4. Open a command prompt window.
5. At the prompt, use the `python` command to run the sample. For example, `python get-celebrities.py`.

```

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

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

vision_base_url = "https://westcentralus.api.cognitive.microsoft.com/vision/v2.1/"

celebrity_analyze_url = vision_base_url + "models/celebrities/analyze"

# Set image_url to the URL of an image that you want to analyze.
image_url = "https://upload.wikimedia.org/wikipedia/commons/d/d9/" + \
    "Bill_gates_portrait.jpg"

headers = {'Ocp-Apim-Subscription-Key': subscription_key}
params = {'model': 'celebrities'}
data = {'url': image_url}
response = requests.post(
    celebrity_analyze_url, headers=headers, params=params, json=data)
response.raise_for_status()

# The 'analysis' object contains various fields that describe the image. The
# most relevant celebrity for the image is obtained from the 'result' property.
analysis = response.json()
assert analysis["result"]["celebrities"] is not []
print(analysis)
celebrity_name = analysis["result"]["celebrities"][0]["name"].capitalize()

# Display the image and overlay it with the celebrity name.
image = Image.open(BytesIO(requests.get(image_url).content))
plt.imshow(image)
plt.axis("off")
_ = plt.title(celebrity_name, size="x-large", y=-0.1)
plt.show()

```

Examine the response for the celebrities sample

A successful response is returned in JSON. The sample webpage parses and displays a successful response in the command prompt window, similar to the following example:

```
{
  "result": {
    "celebrities": [
      {
        "faceRectangle": {
          "top": 123,
          "left": 156,
          "width": 187,
          "height": 187
        },
        "name": "Bill Gates",
        "confidence": 0.9993845224380493
      }
    ]
  },
  "requestId": "f14ec1d0-62d4-4296-9ceb-6b5776dc2020",
  "metadata": {
    "height": 521,
    "width": 550,
    "format": "Jpeg"
  }
}
```

Clean up resources

When no longer needed, delete the files for both samples.

Next steps

Next, explore a Python application that uses Computer Vision to perform optical character recognition (OCR); create smart-cropped thumbnails; and detect, categorize, tag, and describe visual features in images.

[Computer Vision API Python Tutorial](#)

- To rapidly experiment with the Computer Vision API, try the [Open API testing console](#).

Tutorial: Use Computer Vision to generate image metadata in Azure Storage

9/1/2020 • 6 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to integrate the Azure Computer Vision service into a web app to generate metadata for uploaded images. This is useful for [digital asset management \(DAM\)](#) scenarios, such as if a company wants to quickly generate descriptive captions or searchable keywords for all of its images.

A full app guide can be found in the [Azure Storage and Cognitive Services Lab](#) on GitHub, and this tutorial essentially covers Exercise 5 of the lab. You may want to create the full application by following every step, but if you only want to learn how to integrate Computer Vision into an existing web app, read along here.

This tutorial shows you how to:

- Create a Computer Vision resource in Azure
- Perform image analysis on Azure Storage images
- Attach metadata to Azure Storage images
- Check image metadata using Azure Storage Explorer

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- [Visual Studio 2017 Community edition](#) or higher, with the "ASP.NET and web development" and "Azure development" workloads installed.
- An Azure Storage account with a blob container set up for image storage (follow [Exercises 1 of the Azure Storage Lab](#) if you need help with this step).
- The Azure Storage Explorer tool (follow [Exercise 2 of the Azure Storage Lab](#) if you need help with this step).
- An ASP.NET web application with access to Azure Storage (follow [Exercise 3 of the Azure Storage Lab](#) to create such an app quickly).

Create a Computer Vision resource

You'll need to create a Computer Vision resource for your Azure account; this resource manages your access to Azure's Computer Vision service.

1. Follow the instructions in [Create an Azure Cognitive Services resource](#) to create a Computer Vision resource.
2. Then go to the menu for your resource group and click the Computer Vision API subscription that you just created. Copy the URL under **Endpoint** to somewhere you can easily retrieve it in a moment. Then click **Show access keys**.

Delete

Essentials

Resource group (change)	API type
IntelliPixResources	Computer Vision API
Status	Pricing tier
Active	Free
Location	Endpoint
South Central US	https://southcentralus.api.cognitive.microsoft.com/
Subscription name (change)	Manage keys
Subscription ID	Show access keys ...

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](#).

3. In the next window, copy the value of **KEY 1** to the clipboard.

Manage keys

vision-api-key

Regenerate Key1

Regenerate Key2

Notice: It may take up to 10 minutes for the newly (re)generated keys to take effect.

NAME

vision-api-key

KEY 1

KEY 2

Add Computer Vision credentials

Next, you'll add the required credentials to your app so that it can access Computer Vision resources.

Open your ASP.NET web application in Visual Studio and navigate to the **Web.config** file at the root of the project. Add the following statements to the `<appSettings>` section of the file, replacing `VISION_KEY` with the key you copied in the previous step, and `VISION_ENDPOINT` with the URL you saved in the step before.

```
<add key="SubscriptionKey" value="VISION_KEY" />
<add key="VisionEndpoint" value="VISION_ENDPOINT" />
```

Then in the Solution Explorer, right-click the project and use the **Manage NuGet Packages** command to install the package **Microsoft.Azure.CognitiveServices.Vision.ComputerVision**. This package contains the types needed to call the Computer Vision API.

Add metadata generation code

Next, you'll add the code that actually leverages the Computer Vision service to create metadata for images. These steps will apply to the ASP.NET app in the lab, but you can adapt them to your own app. What's important is that at this point you have an ASP.NET web application that can upload images to an Azure Storage container, read images from it, and display them in the view. If you're unsure about this step, it's best to follow [Exercise 3 of the Azure Storage Lab](#).

1. Open the *HomeController.cs* file in the project's **Controllers** folder and add the following `using` statements at the top of the file:

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

2. Then, go to the **Upload** method; this method converts and uploads images to blob storage. Add the following code immediately after the block that begins with `// Generate a thumbnail` (or at the end of your image-blob-creation process). This code takes the blob containing the image (`photo`), and uses Computer Vision to generate a description for that image. The Computer Vision API also generates a list of keywords that apply to the image. The generated description and keywords are stored in the blob's metadata so that they can be retrieved later on.

```
// Submit the image to Azure's Computer Vision API
ComputerVisionClient vision = new ComputerVisionClient(
    new ApiKeyServiceClientCredentials(ConfigurationManager.AppSettings["SubscriptionKey"]),
    new System.Net.Http.DelegatingHandler[] { });
vision.Endpoint = ConfigurationManager.AppSettings["VisionEndpoint"];

VisualFeatureTypes[] features = new VisualFeatureTypes[] { VisualFeatureTypes.Description };
var result = await vision.AnalyzeImageAsync(photo.Uri.ToString(), features);

// Record the image description and tags in blob metadata
photo.Metadata.Add("Caption", result.Description.Captions[0].Text);

for (int i = 0; i < result.Description.Tags.Count; i++)
{
    string key = String.Format("Tag{0}", i);
    photo.Metadata.Add(key, result.Description.Tags[i]);
}

await photo.SetMetadataAsync();
```

3. Next, go to the **Index** method in the same file. This method enumerates the stored image blobs in the targeted blob container (as `IListBlobItem` instances) and passes them to the application view. Replace the `foreach` block in this method with the following code. This code calls `CloudBlockBlob.FetchAttributes` to get each blob's attached metadata. It extracts the computer-generated description (`caption`) from the metadata and adds it to the `BlobInfo` object, which gets passed to the view.

```

foreach (IListBlobItem item in container.ListBlobs())
{
    var blob = item as CloudBlockBlob;

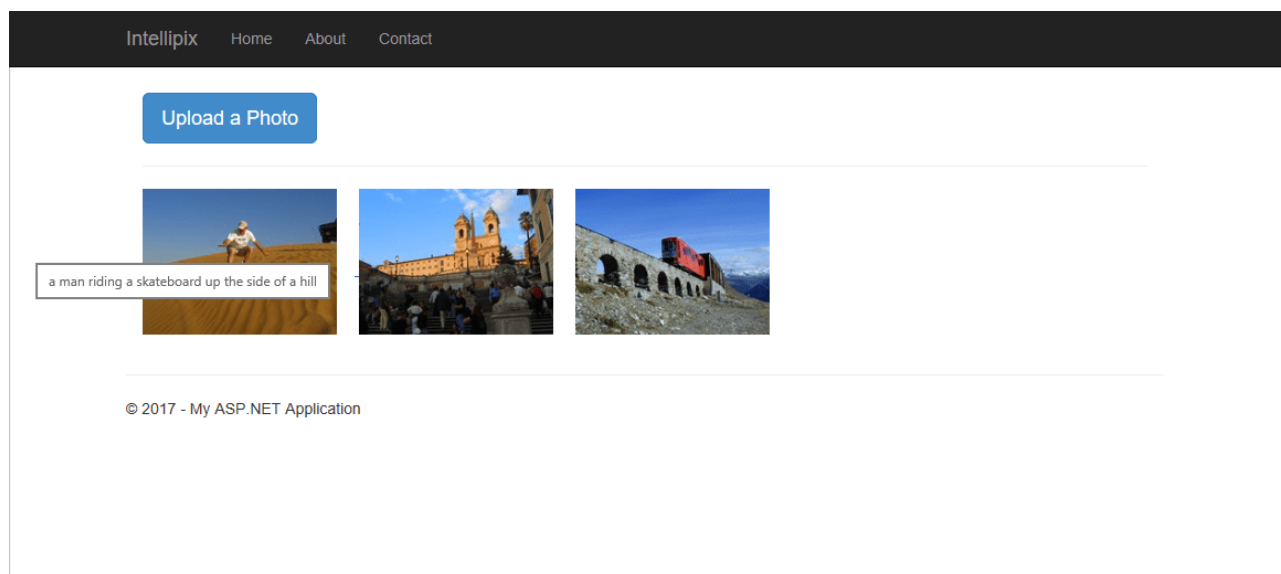
    if (blob != null)
    {
        blob.FetchAttributes(); // Get blob metadata
        var caption = blob.Metadata.ContainsKey("Caption") ? blob.Metadata["Caption"] : blob.Name;

        blobs.Add(new BlobInfo()
        {
            ImageUri = blob.Uri.ToString(),
            ThumbnailUri = blob.Uri.ToString().Replace("/photos/", "/thumbnails/"),
            Caption = caption
        });
    }
}

```

Test the app

Save your changes in Visual Studio and press **Ctrl+F5** to launch the application in your browser. Use the app to upload a few images, either from the "photos" folder in the lab's resources or from your own folder. When you hover the cursor over one of the images in the view, a tooltip window should appear and display the computer-generated caption for the image.



To view all of the attached metadata, use the Azure Storage Explorer to view the storage container you're using for images. Right-click any of the blobs in the container and select **Properties**. In the dialog, you'll see a list of key-value pairs. The computer-generated image description is stored in the item "Caption," and the search keywords are stored in "Tag0," "Tag1," and so on. When you're finished, click **Cancel** to close the dialog.

Properties

Name	Dubai.jpg
Container	photos
Etag	"0x8D515A95F092484"
LastModified	Tue, 17 Oct 2017 21:52:33 GMT
Size	1980812
BlobType	BlockBlob
LeaseState	available

Metadata

Caption	a man riding a skateboard up the side of a hill	✕
Tag0	outdoor	✕
Tag1	nature	✕
Tag2	man	✕
Tag3		✕

Add metadata

Save

Cancel

Clean up resources

If you'd like to keep working on your web app, see the [Next steps](#) section. If you don't plan to continue using this application, you should delete all app-specific resources. To do delete resources, you can delete the resource group that contains your Azure Storage subscription and Computer Vision resource. This will remove the storage account, the blobs uploaded to it, and the App Service resource needed to connect with the ASP.NET web app.

To delete the resource group, open the **Resource groups** tab in the portal, navigate to the resource group you used for this project, and click **Delete resource group** at the top of the view. You'll be asked to type the resource group's name to confirm you want to delete it, because once deleted, a resource group can't be recovered.

Next steps

In this tutorial, you set up Azure's Computer Vision service in an existing web app to automatically generate captions and keywords for blob images as they're uploaded. Next, refer to the Azure Storage Lab, Exercise 6, to learn how to add search functionality to your web app. This takes advantage of the search keywords that the Computer Vision service generates.

[Add search to your app](#)

Applying content tags to images

3/25/2020 • 2 minutes to read • [Edit Online](#)

Computer Vision returns tags based on thousands of recognizable objects, living beings, scenery, and actions. When tags are ambiguous or not common knowledge, the API response provides 'hints' to clarify the meaning of the tag in context of a known setting. Tags are not organized as a taxonomy and no inheritance hierarchies exist. A collection of content tags forms the foundation for an image 'description' displayed as human readable language formatted in complete sentences. Note, that at this point English is the only supported language for image description.

After uploading an image or specifying an image URL, Computer Vision algorithms output tags based on the objects, living beings, and actions identified in the image. Tagging is not limited to the main subject, such as a person in the foreground, but also includes the setting (indoor or outdoor), furniture, tools, plants, animals, accessories, gadgets etc.

Image tagging example

The following JSON response illustrates what Computer Vision returns when tagging visual features detected in the example image.



```
{
  "tags": [
    {
      "name": "grass",
      "confidence": 0.9999995231628418
    },
    {
      "name": "outdoor",
      "confidence": 0.99992108345031738
    },
    {
      "name": "house",
      "confidence": 0.99685388803482056
    },
    {
      "name": "sky",
      "confidence": 0.99532157182693481
    },
    {
      "name": "building",
      "confidence": 0.99436837434768677
    },
    {
      "name": "tree",
      "confidence": 0.98880356550216675
    },
    {
      "name": "lawn",
      "confidence": 0.788884699344635
    },
    {
      "name": "green",
      "confidence": 0.71250593662261963
    },
    {
      "name": "residential",
      "confidence": 0.70859086513519287
    },
    {
      "name": "grassy",
      "confidence": 0.46624681353569031
    }
  ],
  "requestId": "06f39352-e445-42dc-96fb-0a1288ad9cf1",
  "metadata": {
    "height": 200,
    "width": 300,
    "format": "Jpeg"
  }
}
```

Use the API

The tagging feature is part of the [Analyze Image](#) API. You can call this API through a native SDK or through REST calls. Include `Tags` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"tags"` section.

- [Quickstart: Computer Vision .NET SDK](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Next steps

Learn the related concepts of [categorizing images](#) and [describing images](#).

Detect common objects in images

3/22/2020 • 2 minutes to read • [Edit Online](#)

Object detection is similar to [tagging](#), but the API returns the bounding box coordinates (in pixels) for each object found. For example, if an image contains a dog, cat and person, the Detect operation will list those objects together with their coordinates in the image. You can use this functionality to process the relationships between the objects in an image. It also lets you determine whether there are multiple instances of the same tag in an image.

The Detect API applies tags based on the objects or living things identified in the image. There is currently no formal relationship between the tagging taxonomy and the object detection taxonomy. At a conceptual level, the Detect API only finds objects and living things, while the Tag API can also include contextual terms like "indoor", which can't be localized with bounding boxes.

Object detection example

The following JSON response illustrates what Computer Vision returns when detecting objects in the example image.



```

{
  "objects":[
    {
      "rectangle":{
        "x":730,
        "y":66,
        "w":135,
        "h":85
      },
      "object":"kitchen appliance",
      "confidence":0.501
    },
    {
      "rectangle":{
        "x":523,
        "y":377,
        "w":185,
        "h":46
      },
      "object":"computer keyboard",
      "confidence":0.51
    },
    {
      "rectangle":{
        "x":471,
        "y":218,
        "w":289,
        "h":226
      },
      "object":"Laptop",
      "confidence":0.85,
      "parent":{
        "object":"computer",
        "confidence":0.851
      }
    },
    {
      "rectangle":{
        "x":654,
        "y":0,
        "w":584,
        "h":473
      },
      "object":"person",
      "confidence":0.855
    }
  ],
  "requestId":"a7fde8fd-cc18-4f5f-99d3-897dcd07b308",
  "metadata":{
    "width":1260,
    "height":473,
    "format":"Jpeg"
  }
}

```

Limitations

It's important to note the limitations of object detection so you can avoid or mitigate the effects of false negatives (missed objects) and limited detail.

- Objects are generally not detected if they're small (less than 5% of the image).
- Objects are generally not detected if they're arranged closely together (a stack of plates, for example).
- Objects are not differentiated by brand or product names (different types of sodas on a store shelf, for example). However, you can get brand information from an image by using the [Brand detection](#) feature.

Use the API

The object detection feature is part of the [Analyze Image](#) API. You can call this API through a native SDK or through REST calls. Include `objects` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"objects"` section.

- [Quickstart: Computer Vision .NET SDK](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Detect popular brands in images

3/22/2020 • 2 minutes to read • [Edit Online](#)

Brand detection is a specialized mode of [object detection](#) that uses a database of thousands of global logos to identify commercial brands in images or video. You can use this feature, for example, to discover which brands are most popular on social media or most prevalent in media product placement.

The Computer Vision service detects whether there are brand logos in a given image; if so, it returns the brand name, a confidence score, and the coordinates of a bounding box around the logo.

The built-in logo database covers popular brands in consumer electronics, clothing, and more. If you find that the brand you're looking for is not detected by the Computer Vision service, you may be better served creating and training your own logo detector using the [Custom Vision](#) service.

Brand detection example

The following JSON responses illustrate what Computer Vision returns when detecting brands in the example images.



```
"brands": [
  {
    "name": "Microsoft",
    "rectangle": {
      "x": 20,
      "y": 97,
      "w": 62,
      "h": 52
    }
  }
]
```

In some cases, the brand detector will pick up both the logo image and the stylized brand name as two separate logos.



```
"brands":[
  {
    "name": "Microsoft",
    "rectangle": {
      "x": 58,
      "y": 106,
      "w": 55,
      "h": 46
    }
  },
  {
    "name": "Microsoft",
    "rectangle": {
      "x": 58,
      "y": 86,
      "w": 202,
      "h": 63
    }
  }
]
```

Use the API

The brand detection feature is part of the [Analyze Image](#) API. You can call this API through a native SDK or through REST calls. Include `Brands` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"brands"` section.

- [Quickstart: Computer Vision .NET SDK](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Categorize images by subject matter

3/25/2020 • 2 minutes to read • [Edit Online](#)

In addition to tags and a description, Computer Vision returns the taxonomy-based categories detected in an image. Unlike tags, categories are organized in a parent/child hereditary hierarchy, and there are fewer of them (86, as opposed to thousands of tags). All category names are in English. Categorization can be done by itself or alongside the newer tags model.

The 86-category concept

Computer vision can categorize an image broadly or specifically, using the list of 86 categories in the following diagram. For the full taxonomy in text format, see [Category Taxonomy](#).

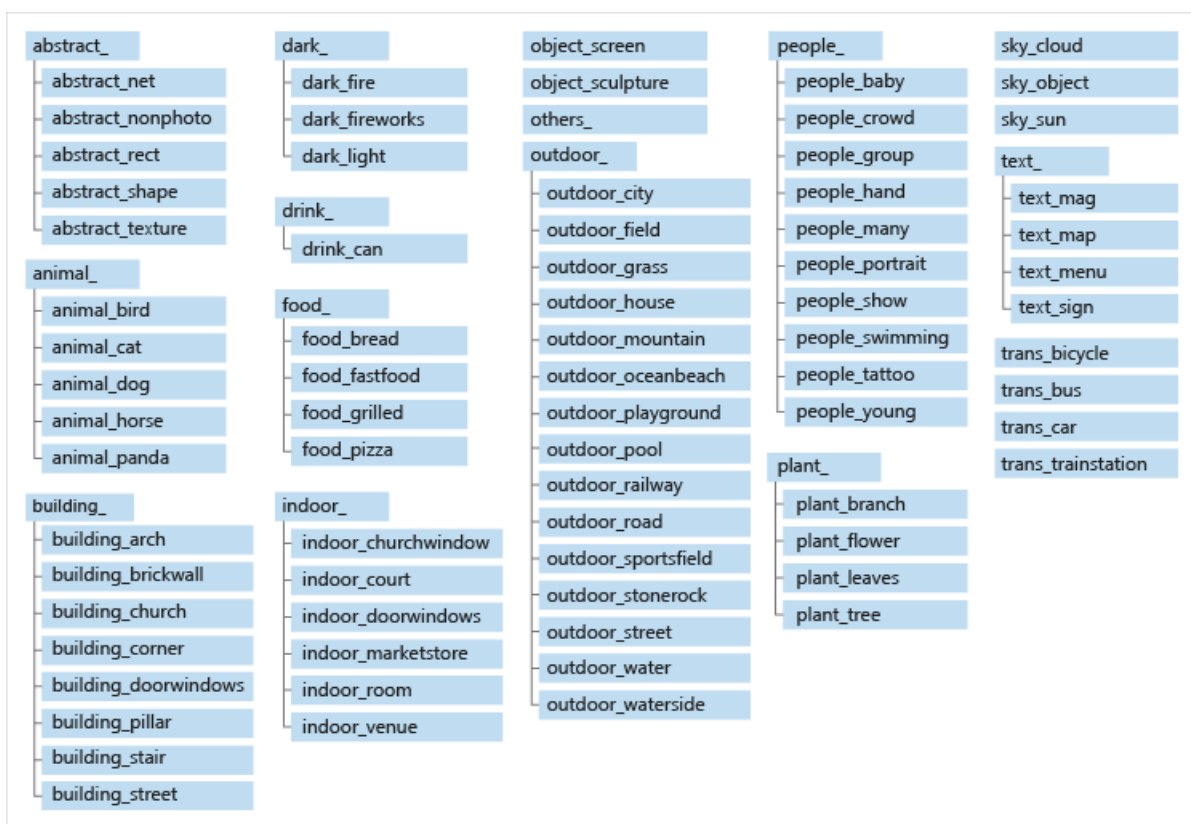
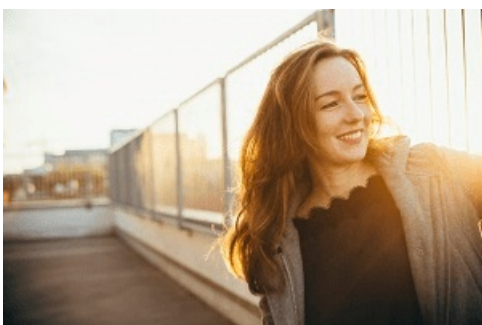


Image categorization examples

The following JSON response illustrates what Computer Vision returns when categorizing the example image based on its visual features.



```

{
  "categories": [
    {
      "name": "people_",
      "score": 0.81640625
    }
  ],
  "requestId": "bae7f76a-1cc7-4479-8d29-48a694974705",
  "metadata": {
    "height": 200,
    "width": 300,
    "format": "Jpeg"
  }
}

```

The following table illustrates a typical image set and the category returned by Computer Vision for each image.





IMAGE	CATEGORY
	people_group
	animal_dog
	outdoor_mountain

IMAGE	CATEGORY
	food_bread

Use the API

The categorization feature is part of the [Analyze Image](#) API. You can call this API through a native SDK or through REST calls. Include `Categories` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"categories"` section.

- [Quickstart: Computer Vision .NET SDK](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Next steps

Learn the related concepts of [tagging images](#) and [describing images](#).

Describe images with human-readable language

3/25/2020 • 2 minutes to read • [Edit Online](#)

Computer Vision can analyze an image and generate a human-readable sentence that describes its contents. The algorithm actually returns several descriptions based on different visual features, and each description is given a confidence score. The final output is a list of descriptions ordered from highest to lowest confidence.

Image description example

The following JSON response illustrates what Computer Vision returns when describing the example image based on its visual features.



```
{
  "description": {
    "tags": ["outdoor", "building", "photo", "city", "white", "black", "large", "sitting", "old",
"water", "skyscraper", "many", "boat", "river", "group", "street", "people", "field", "tall", "bird",
"standing"],
    "captions": [
      {
        "text": "a black and white photo of a city",
        "confidence": 0.95301952483304808
      },
      {
        "text": "a black and white photo of a large city",
        "confidence": 0.94085190563213816
      },
      {
        "text": "a large white building in a city",
        "confidence": 0.93108362931954824
      }
    ]
  },
  "requestId": "b20bfc83-fb25-4b8d-a3f8-b2a1f084b159",
  "metadata": {
    "height": 300,
    "width": 239,
    "format": "Jpeg"
  }
}
```

Use the API

The image description feature is part of the [Analyze Image](#) API. You can call this API through a native SDK or through REST calls. Include `Description` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"description"` section.

- [Quickstart: Computer Vision .NET SDK](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Next steps

Learn the related concepts of [tagging images](#) and [categorizing images](#).

Face detection with Computer Vision

3/25/2020 • 2 minutes to read • [Edit Online](#)

Computer Vision can detect human faces within an image and generate the age, gender, and rectangle for each detected face.

NOTE

This feature is also offered by the Azure [Face](#) service. See this alternative for more detailed face analysis, including face identification and pose detection.

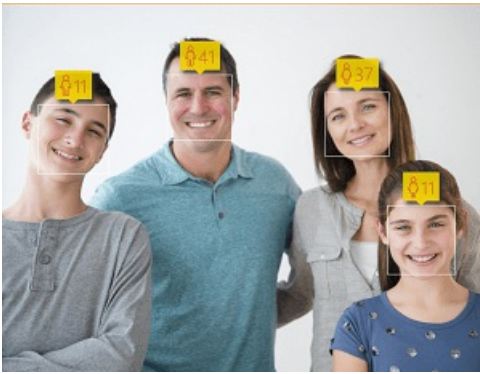
Face detection examples

The following example demonstrates the JSON response returned by Computer Vision for an image containing a single human face.



```
{
  "faces": [
    {
      "age": 23,
      "gender": "Female",
      "faceRectangle": {
        "top": 45,
        "left": 194,
        "width": 44,
        "height": 44
      }
    }
  ],
  "requestId": "8439ba87-de65-441b-a0f1-c85913157ecd",
  "metadata": {
    "height": 200,
    "width": 300,
    "format": "Png"
  }
}
```

The next example demonstrates the JSON response returned for an image containing multiple human faces.



```
{
  "faces": [
    {
      "age": 11,
      "gender": "Male",
      "faceRectangle": {
        "top": 62,
        "left": 22,
        "width": 45,
        "height": 45
      }
    },
    {
      "age": 11,
      "gender": "Female",
      "faceRectangle": {
        "top": 127,
        "left": 240,
        "width": 42,
        "height": 42
      }
    },
    {
      "age": 37,
      "gender": "Female",
      "faceRectangle": {
        "top": 55,
        "left": 200,
        "width": 41,
        "height": 41
      }
    },
    {
      "age": 41,
      "gender": "Male",
      "faceRectangle": {
        "top": 45,
        "left": 103,
        "width": 39,
        "height": 39
      }
    }
  ],
  "requestId": "3a383cbe-1a05-4104-9ce7-1b5cf352b239",
  "metadata": {
    "height": 230,
    "width": 300,
    "format": "Png"
  }
}
```

Use the API

The face detection feature is part of the [Analyze Image](#) API. You can call this API through a native SDK or through REST calls. Include `Faces` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"faces"` section.

- [Quickstart: Computer Vision .NET SDK](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Detecting image types with Computer Vision

3/25/2020 • 2 minutes to read • [Edit Online](#)

With the [Analyze Image](#) API, Computer Vision can analyze the content type of images, indicating whether an image is clip art or a line drawing.

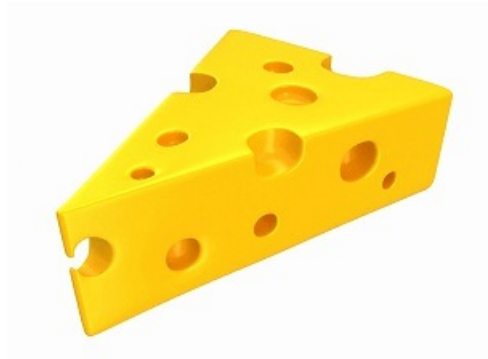
Detecting clip art

Computer Vision analyzes an image and rates the likelihood of the image being clip art on a scale of 0 to 3, as described in the following table.

VALUE	MEANING
0	Non-clip-art
1	Ambiguous
2	Normal-clip-art
3	Good-clip-art

Clip art detection examples

The following JSON responses illustrates what Computer Vision returns when rating the likelihood of the example images being clip art.



```
{
  "imageType": {
    "clipArtType": 3,
    "lineDrawingType": 0
  },
  "requestId": "88c48d8c-80f3-449f-878f-6947f3b35a27",
  "metadata": {
    "height": 225,
    "width": 300,
    "format": "Jpeg"
  }
}
```



```
{
  "imageType": {
    "clipArtType": 0,
    "lineDrawingType": 0
  },
  "requestId": "a9c8490a-2740-4e04-923b-e8f4830d0e47",
  "metadata": {
    "height": 200,
    "width": 300,
    "format": "Jpeg"
  }
}
```

Detecting line drawings

Computer Vision analyzes an image and returns a boolean value indicating whether the image is a line drawing.

Line drawing detection examples

The following JSON responses illustrates what Computer Vision returns when indicating whether the example images are line drawings.



```
{
  "imageType": {
    "clipArtType": 2,
    "lineDrawingType": 1
  },
  "requestId": "6442dc22-476a-41c4-aa3d-9ceb15172f01",
  "metadata": {
    "height": 268,
    "width": 300,
    "format": "Jpeg"
  }
}
```



```
{
  "imageType": {
    "clipArtType": 0,
    "lineDrawingType": 0
  },
  "requestId": "98437d65-1b05-4ab7-b439-7098b5dfdcbf",
  "metadata": {
    "height": 200,
    "width": 300,
    "format": "Jpeg"
  }
}
```

Use the API

The image type detection feature is part of the [Analyze Image](#) API. You can call this API through a native SDK or through REST calls. Include `ImageType` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"imageType"` section.

- [Quickstart: Computer Vision .NET SDK](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Detect domain-specific content

8/9/2019 • 2 minutes to read • [Edit Online](#)

In addition to tagging and high-level categorization, Computer Vision also supports further domain-specific analysis using models that have been trained on specialized data.

There are two ways to use the domain-specific models: by themselves (scoped analysis) or as an enhancement to the categorization feature.

Scoped analysis

You can analyze an image using only the chosen domain-specific model by calling the [Models/<model>/Analyze](#) API.

The following is a sample JSON response returned by the `models/celebrities/analyze` API for the given image:




```
{
  "result": {
    "celebrities": [{
      "faceRectangle": {
        "top": 391,
        "left": 318,
        "width": 184,
        "height": 184
      },
      "name": "Satya Nadella",
      "confidence": 0.99999856948852539
    }]
  },
  "requestId": "8217262a-1a90-4498-a242-68376a4b956b",
  "metadata": {
    "width": 800,
    "height": 1200,
    "format": "Jpeg"
  }
}
```

Enhanced categorization analysis

You can also use domain-specific models to supplement general image analysis. You do this as part of [high-level categorization](#) by specifying domain-specific models in the *details* parameter of the [Analyze](#) API call.

In this case, the 86-category taxonomy classifier is called first. If any of the detected categories have a matching domain-specific model, the image is passed through that model as well and the results are added.

The following JSON response shows how domain-specific analysis can be included as the `detail` node in a broader categorization analysis.

```
"categories":[
  {
    "name":"abstract_",
    "score":0.00390625
  },
  {
    "name":"people_",
    "score":0.83984375,
    "detail":{
      "celebrities":[
        {
          "name":"Satya Nadella",
          "faceRectangle":{
            "left":597,
            "top":162,
            "width":248,
            "height":248
          },
          "confidence":0.999028444
        }
      ],
      "landmarks":[
        {
          "name":"Forbidden City",
          "confidence":0.9978346
        }
      ]
    }
  }
]
```

List the domain-specific models

Currently, Computer Vision supports the following domain-specific models:

NAME	DESCRIPTION
celebrities	Celebrity recognition, supported for images classified in the <code>people_</code> category
landmarks	Landmark recognition, supported for images classified in the <code>outdoor_</code> or <code>building_</code> categories

Calling the [Models](#) API will return this information along with the categories to which each model can apply:

```
{
  "models": [
    {
      "name": "celebrities",
      "categories": [
        "people_",
        "人_",
        "pessoas_",
        "gente_"
      ]
    },
    {
      "name": "landmarks",
      "categories": [
        "outdoor_",
        "户外_",
        "屋外_",
        "aoarlivre_",
        "alairelibre_",
        "building_",
        "建筑_",
        "建物_",
        "edifício_"
      ]
    }
  ]
}
```

Next steps

Learn concepts about [categorizing images](#).

Detect color schemes in images

3/25/2020 • 2 minutes to read • [Edit Online](#)

Computer Vision analyzes the colors in an image to provide three different attributes: the dominant foreground color, the dominant background color, and the set of dominant colors for the image as a whole. Returned colors belong to the set: black, blue, brown, gray, green, orange, pink, purple, red, teal, white, and yellow.

Computer Vision also extracts an accent color, which represents the most vibrant color in the image, based on a combination of dominant colors and saturation. The accent color is returned as a hexadecimal HTML color code.

Computer Vision also returns a boolean value indicating whether an image is in black and white.

Color scheme detection examples



The following example illustrates the JSON response returned by Computer Vision when detecting the color scheme of the example image. In this case, the example image is not a black and white image, but the dominant foreground and background colors are black, and the dominant colors for the image as a whole are black and white.



```
{
  "color": {
    "dominantColorForeground": "Black",
    "dominantColorBackground": "Black",
    "dominantColors": ["Black", "White"],
    "accentColor": "BB6D10",
    "isBwImg": false
  },
  "requestId": "0dc394bf-db50-4871-bdcc-13707d9405ea",
  "metadata": {
    "height": 202,
    "width": 300,
    "format": "Jpeg"
  }
}
```

Dominant color examples

The following table shows the returned foreground, background, and image colors for each sample image.

IMAGE	DOMINANT COLORS
	Foreground: Black Background: White Colors: Black, White, Green
	Foreground: Black Background: Black Colors: Black

Accent color examples

The following table shows the returned accent color, as a hexadecimal HTML color value, for each example image.






IMAGE	ACCENT COLOR
	#BB6D10
	#C6A205

IMAGE	ACCENT COLOR
	#474A84

Black & white detection examples

The following table shows Computer Vision's black and white evaluation in the sample images.

IMAGE	BLACK & WHITE?
	true
	false

Use the API

The color scheme detection feature is part of the [Analyze Image](#) API. You can call this API through a native SDK or through REST calls. Include `color` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"color"` section.

- [Quickstart: Computer Vision .NET SDK](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Generating smart-cropped thumbnails with Computer Vision

8/9/2019 • 2 minutes to read • [Edit Online](#)

A thumbnail is a reduced-size representation of an image. Thumbnails are used to represent images and other data in a more economical, layout-friendly way. The Computer Vision API uses smart cropping, together with resizing the image, to create intuitive thumbnails for a given image.

The Computer Vision thumbnail generation algorithm works as follows:

1. Remove distracting elements from the image and identify the *area of interest*—the area of the image in which the main object(s) appears.
2. Crop the image based on the identified *area of interest*.
3. Change the aspect ratio to fit the target thumbnail dimensions.

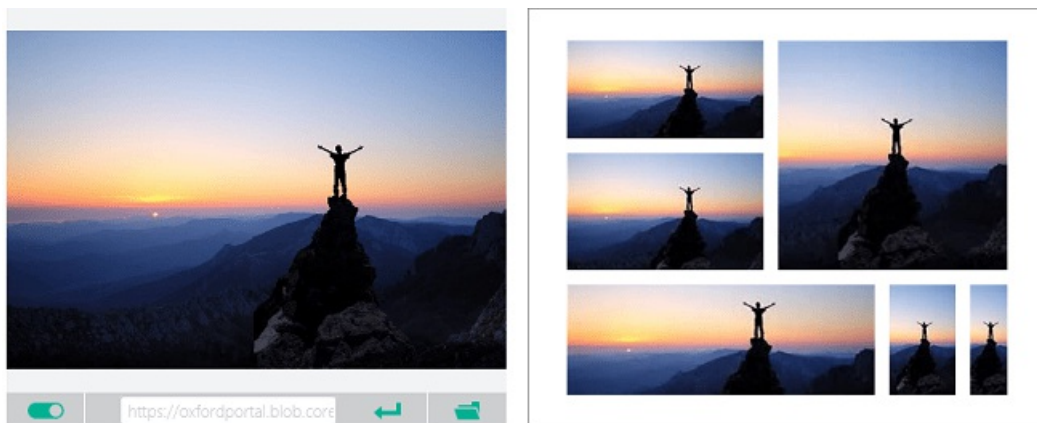
Area of interest

When you upload an image, the Computer Vision API analyzes it to determine the *area of interest*. It can then use this region to determine how to crop the image. The cropping operation, however, will always match the desired aspect ratio if one is specified.







You can also get the raw bounding box coordinates of this same *area of interest* by calling the **areaOfInterest** API instead. You can then use this information to modify the original image however you wish.

Examples

The generated thumbnail can vary widely depending on what you specify for height, width, and smart cropping, as shown in the following image.



The following table illustrates typical thumbnails generated by Computer Vision for the example images. The thumbnails were generated for a specified target height and width of 50 pixels, with smart cropping enabled.

IMAGE	THUMBNAIL
	
	
	

Next steps

Learn about [tagging images](#) and [categorizing images](#).

Optical Character Recognition (OCR)

9/1/2020 • 5 minutes to read • [Edit Online](#)

Azure's Computer Vision API includes Optical Character Recognition (OCR) capabilities that extract printed or handwritten text from images. You can extract text from images, such as photos of license plates or containers with serial numbers, as well as from documents - invoices, bills, financial reports, articles, and more.

Read API

The Computer Vision [Read API](#) is Azure's latest OCR technology that extracts printed text (in several languages), handwritten text (English only), digits, and currency symbols from images and multi-page PDF documents. It's optimized to extract text from text-heavy images and multi-page PDF documents with mixed languages. It supports detecting both printed and handwritten text in the same image or document.

Input requirements

The Read API's **Read** operation takes images and documents as its input. They have the following requirements:

- Supported file formats: JPEG, PNG, BMP, PDF, and TIFF
- For PDF AND TIFF, up to 2000 pages are processed. For free tier subscribers, only the first two pages are processed.
- The file size must be less than 50 MB and dimensions at least 50 x 50 pixels and at most 10000 x 10000 pixels.
- The PDF dimensions must be at most 17 x 17 inches, corresponding to legal or A3 paper sizes and smaller.

NOTE

Language input

The [Read operation](#) has an optional request parameter for language. This is the BCP-47 language code of the text in the document. Read supports auto language identification and multilingual documents, so only provide a language code if you would like to force the document to be processed as that specific language.

The Read operation

The [Read operation](#) takes an image or PDF document as the input and extracts text asynchronously. The call returns with a response header field called `Operation-Location`. The `Operation-Location` value is a URL that contains the Operation ID to be used in the next step.

RESPONSE HEADER	RESULT URL
Operation-Location	<code>https://cognitiveservice/vision/v3.0/read/analyzeResults/49a3632fc4b-4387-aa06-090cfbf0064f</code>

The Get Read Results operation

The second step is to call the [Get Read Results](#) operation. This operation takes as input the operation ID that was created by the Read operation. It returns a JSON response that contains a **status** field with the following possible values. You call this operation iteratively until it returns with the **succeeded** value. Use an interval of 1 to 2 seconds to avoid exceeding the requests per second (RPS) rate.

FIELD	TYPE	POSSIBLE VALUES
status	string	notStarted: The operation has not started.
		running: The operation is being processed.
		failed: The operation has failed.
		succeeded: The operation has succeeded.

NOTE

The free tier limits the request rate to 20 calls per minute. The paid tier allows 10 requests per second (RPS) that can be increased upon request. Use the Azure support channel or your account team to request a higher request per second (RPS) rate.

When the **status** field has the **succeeded** value, the JSON response contains the extracted text content from your image or document. The JSON response maintains the original line groupings of recognized words. It includes the extracted text lines and their bounding box coordinates. Each text line includes all extracted words with their coordinates and confidence scores.

Sample JSON output

See the following example of a successful JSON response:

```

{
  "status": "succeeded",
  "createdDateTime": "2020-05-28T05:13:21Z",
  "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
  "analyzeResult": {
    "version": "3.0.0",
    "readResults": [
      {
        "page": 1,
        "language": "en",
        "angle": 0.8551,
        "width": 2661,
        "height": 1901,
        "unit": "pixel",
        "lines": [
          {
            "boundingBox": [
              67,
              646,
              2582,
              713,
              2580,
              876,
              67,
              821
            ],
            "text": "The quick brown fox jumps",
            "words": [
              {
                "boundingBox": [
                  143,
                  650,
                  435,
                  661,
                  436,
                  823,
                  144,
                  824
                ],
                "text": "The",
                "confidence": 0.958
              }
            ]
          }
        ]
      }
    ]
  }
}

```

Follow the [Extract printed and handwritten text](#) quickstart to implement OCR using C# and the REST API.

Language support

Printed text

The [Read 3.0 API](#) supports extracting printed text in English, Spanish, German, French, Italian, Portuguese, and Dutch languages.

The [Read 3.1 API public preview](#) adds support for Simplified Chinese. If your scenario requires supporting more languages, see the [OCR API](#) section.

See the [Supported languages](#) for the full list of OCR-supported languages.

Handwritten text

The Read operation currently supports extracting handwritten text exclusively in English.

Integration options

Use the REST API or client SDK

The [Read 3.x REST API](#) is the preferred option for most customers because of ease of integration and fast productivity out of the box. Azure and the Computer Vision service handle scale, performance, data security, and compliance needs while you focus on meeting your customers' needs.

Use containers for on-premise deployment

The [Read 2.0 Docker container \(preview\)](#) enables you to deploy the new OCR capabilities in your own local environment. Containers are great for specific security and data governance requirements.

Read OCR examples

Text from images

The following Read API output shows the extracted text from an image with different text angles, colors, and fonts.



Text from documents

Read API can also take PDF documents as input.

Contoso

Address:
1 Redmond way Suite
6000 Redmond, WA
99243

Invoice For: **Microsoft**
1020 Enterprise Way
Sunnyvale, CA 87659

34278587	Invoice Number	Invoice Date	Invoice Due Date	Charges	VAT ID
34278587					
34278587		6/18/2017	6/24/2017	\$56,651.49	PT

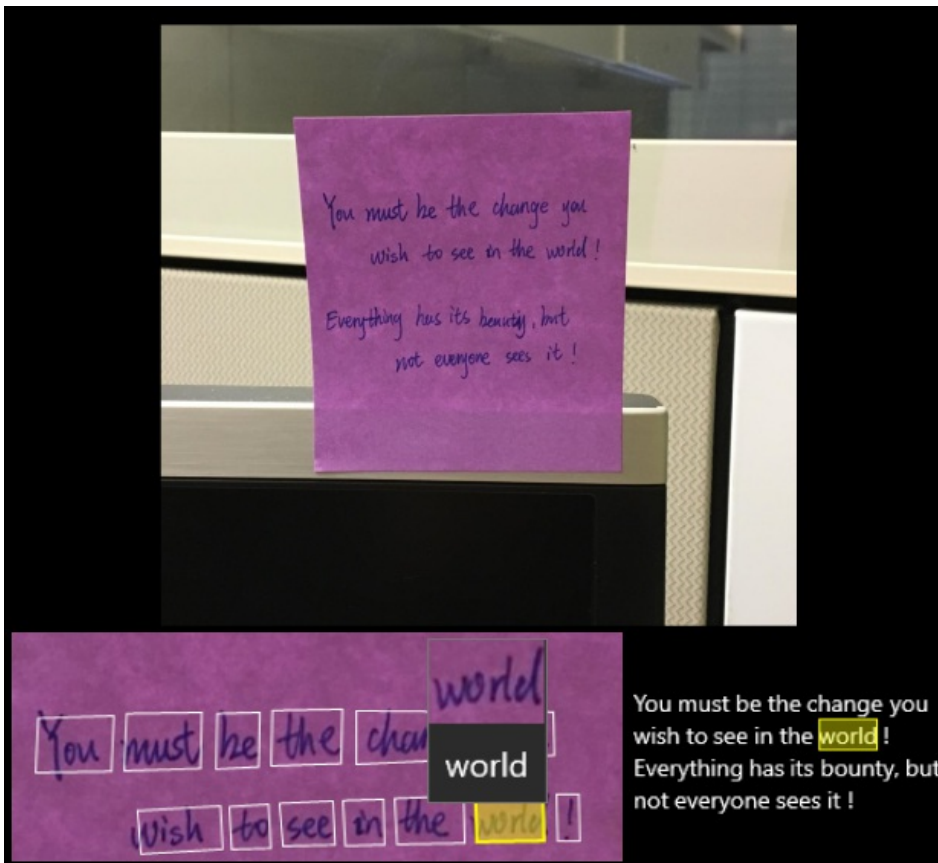
Text

Text extracted

Contoso
Address:
Invoice For: Microsoft
1 Redmond way Suite
1020 Enterprise Way
6000 Redmond, WA
Sunnyvale, CA 87659
99243
Invoice Number
Invoice Date
Invoice Due Date
Charges
VAT ID
34278587
6/18/2017
6/24/2017
\$56,651.49
PT

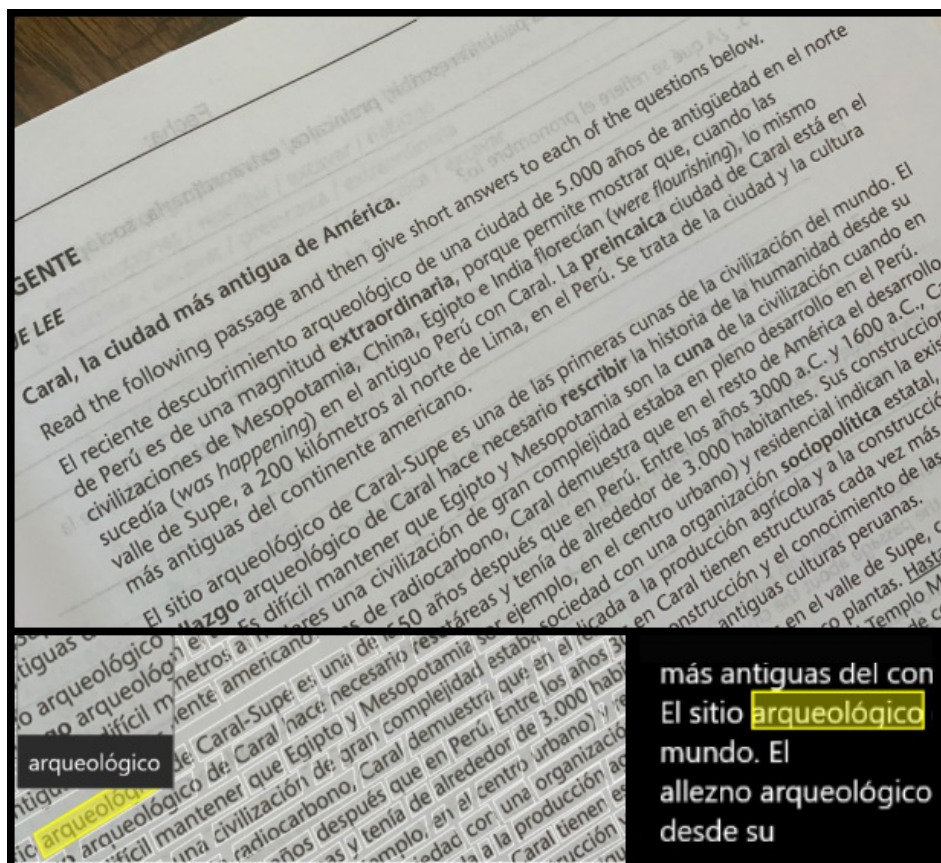
Handwritten text

The Read operation extracts handwritten text from images (currently only in English).



Printed text

The Read operation can extract printed text in several different languages.



Mixed language documents

The Read API supports images and documents that contain multiple different languages, commonly known as mixed language documents. It works by classifying each text line in the document into the detected language before extracting its text contents.

English	Welcome to the United States.
French	Bienvenue en France.
Italian	Benvenuti in Italia.
German	Willkommen in Deutschland.
Spanish	Bienvenido a España.
Portuguese	Bem-vindo a Portugal.
Dutch	Welkom in Nederland.

OCR API

The [OCR API](#) uses an older recognition model, supports only images, and executes synchronously, returning

immediately with the detected text. See the [OCR supported languages](#) then Read API.

Data privacy and security

As with all the cognitive services, developers using the Read/OCR services should be aware of Microsoft policies on customer data. See the Cognitive Services page on the [Microsoft Trust Center](#) to learn more.

NOTE

The Computer Vision 2.0 RecognizeText operations are in the process of getting deprecated in favor of the new Read API covered in this article. Existing customers should [transition to using Read operations](#).

Next steps

- Learn about the [Read 3.0 REST API](#).
- Learn about the [Read 3.1 public preview REST API](#) with added support for Simplified Chinese.
- Follow the [Extract text](#) quickstart to implement OCR using C#, Java, JavaScript, or Python along with REST API.

Detect adult content

10/1/2019 • 2 minutes to read • [Edit Online](#)

Computer Vision can detect adult material in images so that developers can restrict the display of these images in their software. Content flags are applied with a score between zero and one so that developers can interpret the results according to their own preferences.

NOTE

Much of this functionality is offered by the [Azure Content Moderator](#) service. See this alternative for solutions to more rigorous content moderation scenarios, such as text moderation and human review workflows.

Content flag definitions

Within the "adult" classification are several different categories:

- **Adult** images are defined as those which are explicitly sexual in nature and often depict nudity and sexual acts.
- **Racy** images are defined as images that are sexually suggestive in nature and often contain less sexually explicit content than images tagged as **Adult**.
- **Gory** images are defined as those which depict gore.

Use the API

You can detect adult content with the [Analyze Image](#) API. When you add the value of `Adult` to the `visualFeatures` query parameter, the API returns three boolean properties—`isAdultContent`, `isRacyContent`, and `isGoryContent`—in its JSON response. The method also returns corresponding properties—`adultScore`, `racyScore`, and `goreScore`—which represent confidence scores between zero and one for each respective category.

- [Quickstart: Analyze an image \(.NET SDK\)](#)
- [Quickstart: Analyze an image \(REST API\)](#)

Install and run Read containers (Preview)

9/1/2020 • 13 minutes to read • [Edit Online](#)

Containers enable you to run the Computer Vision APIs in your own environment. Containers are great for specific security and data governance requirements. In this article you'll learn how to download, install, and run a Computer Vision container.

A single Docker container, *Read*, is available for Computer Vision. The *Read* container allows you to detect and extract *printed text* from images of various objects with different surfaces and backgrounds, such as receipts, posters, and business cards. Additionally, the *Read* container detects *handwritten text* in images and provides PDF, TIFF, and multi-page file support. For more information, see the [Read API documentation](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

You must meet the following prerequisites before using the containers:

REQUIRED	PURPOSE
Docker Engine	<p>You need the Docker Engine 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.</p> <p>Docker must be configured to allow the containers to connect with and send billing data to Azure.</p> <p>On Windows, Docker must also be configured to support Linux containers.</p>
Familiarity with Docker	<p>You should have a basic understanding of Docker concepts, like registries, repositories, containers, and container images, as well as knowledge of basic <code>docker</code> commands.</p>
Computer Vision resource	<p>In order to use the container, you must have:</p> <p>An Azure Computer Vision resource and the associated API key the endpoint URI. Both values are available on the Overview and Keys pages for the resource and are required to start the container.</p> <p>{API_KEY}: One of the two available resource keys on the Keys page</p> <p>{ENDPOINT_URI}: The endpoint as provided on the Overview page</p>

Request access to the private container registry

Fill out and submit the [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.

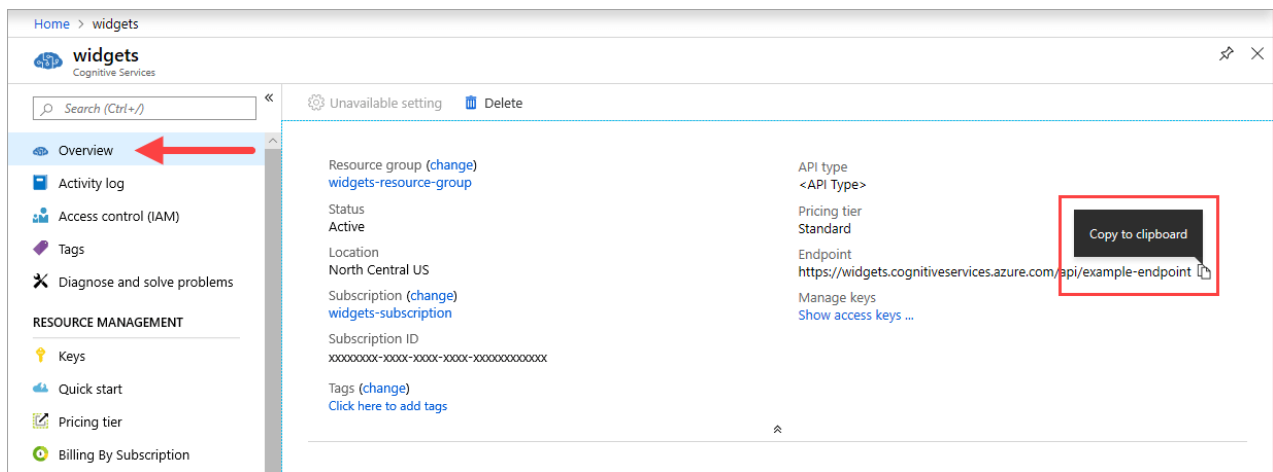
```
cat <passwordFile> | docker login containerpreview.azurecr.io -u <username> --password-stdin
```

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.

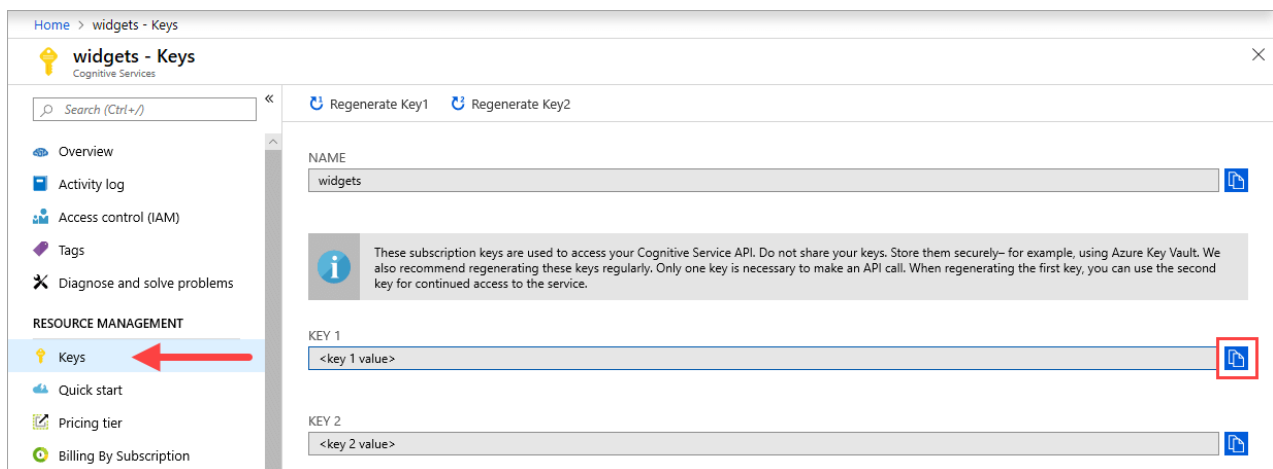
Endpoint URI `{ENDPOINT_URI}`

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.

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](#).

Advanced Vector Extension support

The **host** computer is the computer that runs the docker container. The host *must support* [Advanced Vector Extensions](#) (AVX2). You can check for AVX2 support on Linux hosts with the following command:

```
grep -q avx2 /proc/cpuinfo && echo AVX2 supported || echo No AVX2 support detected
```

WARNING

The host computer is *required* to support AVX2. The container *will not* function correctly without AVX2 support.

Container requirements and recommendations

NOTE

The requirements and recommendations are based on benchmarks with a single request per second, using an 8-MB image of a scanned business letter that contains 29 lines and a total of 803 characters.

The following table describes the minimum and recommended allocation of resources for each Read container.

CONTAINER	MINIMUM	RECOMMENDED	TPS (MINIMUM, MAXIMUM)
Read	1 cores, 8-GB memory, 0.24 TPS	8 cores, 16-GB memory, 1.17 TPS	0.24, 1.17

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

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 Read are available.

CONTAINER	CONTAINER REGISTRY / REPOSITORY / IMAGE NAME
Read	<code>containerpreview.azurecr.io/microsoft/cognitive-services-read:latest</code>

Use the `docker pull` command to download a container image.

Docker pull for the Read container

```
docker pull containerpreview.azurecr.io/microsoft/cognitive-services-read: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>

How to use the container

Once 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 16g --cpus 8 \
  containerpreview.azurecr.io/microsoft/cognitive-services-read \
  Eula=accept \
  Billing={ENDPOINT_URI} \
  ApiKey={API_KEY}
```

This command:

- Runs the Read container from the container image.
- Allocates 8 CPU core and 16 gigabytes (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; otherwise, 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.

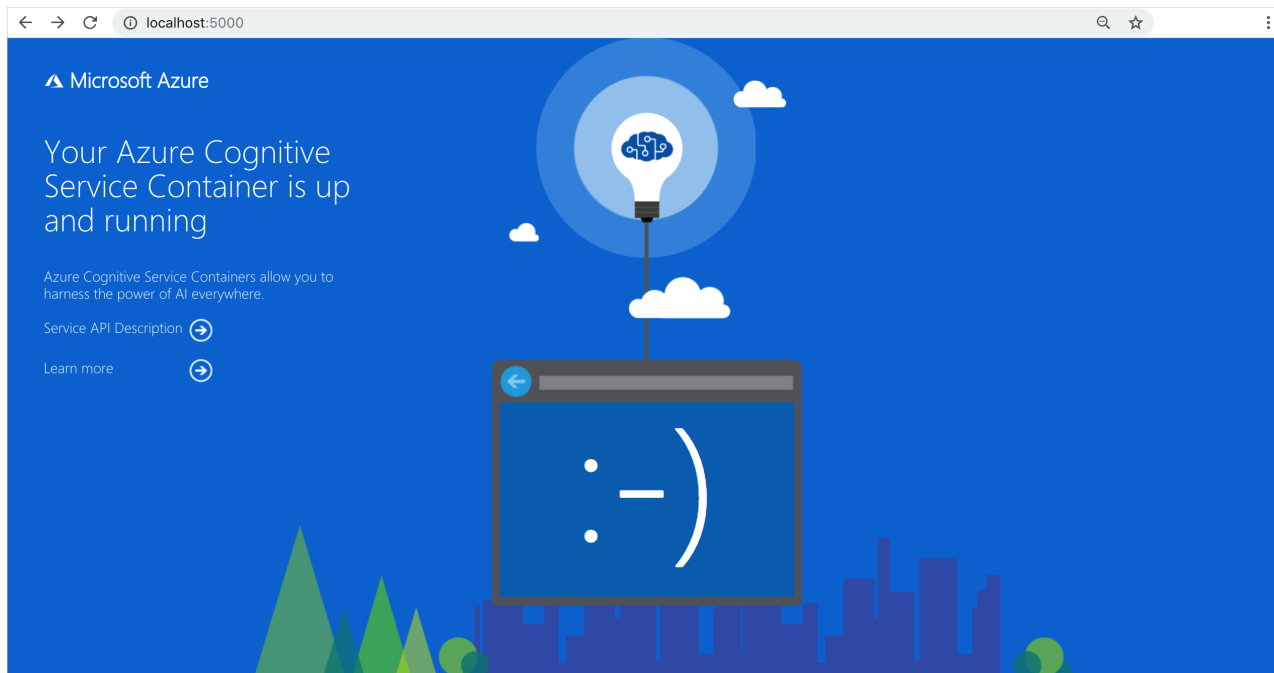
You can have this container and a different Azure Cognitive Services container running on the HOST together. You also can have multiple containers of the same Cognitive Services container running.

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
<code>http://localhost:5000/</code>	The container provides a home page.
<code>http://localhost:5000/ready</code>	Requested with GET, this provides a verification that the container is ready to accept a query against the model. This request can be used for Kubernetes liveness and readiness probes .

REQUEST URL	PURPOSE
<code>http://localhost:5000/status</code>	Also requested with GET, this verifies if the api-key used to start the container is valid without causing an endpoint query. This request can be used for Kubernetes liveness and readiness probes .
<code>http://localhost:5000/swagger</code>	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.



Query the container's prediction endpoint

The container provides REST-based query prediction endpoint APIs.

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

Asynchronous read

You can use the `POST /vision/v2.0/read/core/asyncBatchAnalyze` and

`GET /vision/v2.0/read/operations/{operationId}` operations in concert to asynchronously read an image, similar to how the Computer Vision service uses those corresponding REST operations. The asynchronous POST method will return an `operationId` that is used as the identifier to the HTTP GET request.

From the swagger UI, select the `asyncBatchAnalyze` to expand it in the browser. Then select **Try it out** > **Choose file**. In this example, we'll use the following image:

Tabs vs Spaces

When the asynchronous POST has run successfully, it returns an **HTTP 202** status code. As part of the response, there is an `operation-location` header that holds the result endpoint for the request.

```
content-length: 0
date: Fri, 13 Sep 2019 16:23:01 GMT
operation-location: http://localhost:5000/vision/v2.0/read/operations/a527d445-8a74-4482-8cb3-c98a65ec7ef9
server: Kestrel
```

The `operation-location` is the fully qualified URL and is accessed via an HTTP GET. Here is the JSON response from executing the `operation-location` URL from the preceding image:

```

{
  "status": "Succeeded",
  "recognitionResults": [
    {
      "page": 1,
      "clockwiseOrientation": 2.42,
      "width": 502,
      "height": 252,
      "unit": "pixel",
      "lines": [
        {
          "boundingBox": [ 56, 39, 317, 50, 313, 134, 53, 123 ],
          "text": "Tabs VS",
          "words": [
            {
              "boundingBox": [ 90, 43, 243, 53, 243, 123, 94, 125 ],
              "text": "Tabs",
              "confidence": "Low"
            },
            {
              "boundingBox": [ 259, 55, 313, 62, 313, 122, 259, 123 ],
              "text": "VS"
            }
          ]
        }
      ],
    },
    {
      "boundingBox": [ 221, 148, 417, 146, 417, 206, 227, 218 ],
      "text": "Spaces",
      "words": [
        {
          "boundingBox": [ 230, 148, 416, 141, 419, 211, 232, 218 ],
          "text": "Spaces"
        }
      ]
    }
  ]
}

```

Synchronous read

You can use the `POST /vision/v2.0/read/core/Analyze` operation to synchronously read an image. When the image is read in its entirety, then and only then does the API return a JSON response. The only exception to this is if an error occurs. When an error occurs the following JSON is returned:

```

{
  status: "Failed"
}

```

The JSON response object has the same object graph as the asynchronous version. If you're a JavaScript user and want type safety, the following types could be used to cast the JSON response as an `AnalyzeResult` object.

```

export interface AnalyzeResult {
  status: Status;
  recognitionResults?: RecognitionResult[] | null;
}

export enum Status {
  NotStarted = 0,
  Running = 1,
  Failed = 2,
  Succeeded = 3
}

export enum Unit {
  Pixel = 0,
  Inch = 1
}

export interface RecognitionResult {
  page?: number | null;
  clockwiseOrientation?: number | null;
  width?: number | null;
  height?: number | null;
  unit?: Unit | null;
  lines?: Line[] | null;
}

export interface Line {
  boundingBox?: number[] | null;
  text: string;
  words?: Word[] | null;
}

export enum Confidence {
  High = 0,
  Low = 1
}

export interface Word {
  boundingBox?: number[] | null;
  text: string;
  confidence?: Confidence | null;
}

```

For an example use-case, see the [TypeScript sandbox here](#) and select **Run** to visualize its ease-of-use.

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 enabled, the container generates log files that are helpful to troubleshoot issues that happen while starting or running the container.

TIP

For more troubleshooting information and guidance, see [Cognitive Services containers frequently asked questions \(FAQ\)](#).

Billing

The Cognitive Services containers send billing information to Azure, using the corresponding 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 metering / billing endpoint. 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 serving requests.

Billing arguments

The `docker run` command will start the container when all three of the following options are provided with valid values:

OPTION	DESCRIPTION
<code>ApiKey</code>	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 <code>Billing</code> .
<code>Billing</code>	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.
<code>Eula</code>	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](#)
- [Azure Cognitive Services](#)

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 downloading, installing, and running Computer Vision

containers. In summary:

- Computer Vision provides a Linux container for Docker, encapsulating Read.
- Container images are downloaded from the "Container Preview" container registry in Azure.
- Container images run in Docker.
- You can use either the REST API or SDK to call operations in Read containers by specifying the host URI of the container.
- You must specify billing information when instantiating a container.

IMPORTANT

Cognitive Services containers are not licensed to run without being connected to Azure for metering. Customers need to enable the containers to communicate billing information with the metering service at all times. Cognitive Services containers do not send customer data (for example, the image or text that is being analyzed) to Microsoft.

Next steps

- Review [Configure containers](#) for configuration settings
- Review [Computer Vision overview](#) to learn more about recognizing printed and handwritten text
- Refer to the [Computer Vision API](#) for details about the methods supported by the container.
- Refer to [Frequently asked questions \(FAQ\)](#) to resolve issues related to Computer Vision functionality.
- Use more [Cognitive Services Containers](#)

Configure Computer Vision Docker containers

4/7/2020 • 7 minutes to read • [Edit Online](#)

You configure the Computer Vision container's runtime environment by 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 ASPNET 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 `Cognitive Services` 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	<code>InstrumentationKey</code>	String	<p>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.</p> <p>Example:</p> <pre>InstrumentationKey=123456789</pre>

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 `vision/v1.0` routing to the endpoint URI as shown in the following table.

REQUIRED	NAME	DATA TYPE	DESCRIPTION
Yes	<code>Billing</code>	String	<p>Billing endpoint URI</p> <p>Example:</p> <pre>Billing=https://westcentralus.api.cognitiv</pre>

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	<code>Eula</code>	String	<p>License acceptance</p> <p>Example:</p> <pre>Eula=accept</pre>

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
<code>Host</code>	String	The IP address or DNS host name of the Fluentd server.
<code>Port</code>	Integer	The port of the Fluentd server. The default value is 24224.
<code>HeartbeatMs</code>	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).
<code>SendBufferSize</code>	Integer	The network buffer space, in bytes, allocated for send operations. The default value is 32768 bytes (32 kilobytes).

NAME	DATA TYPE	DESCRIPTION
<code>TlsConnectionEstablishmentTimeoutMs</code>	Integer	The timeout, in milliseconds, to establish a SSL/TLS connection with the Fluentd server. The default value is 10000 milliseconds (10 seconds). If <code>UseTLS</code> is set to false, this value is ignored.
<code>UseTLS</code>	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
<code>HTTPS_PROXY</code>	string	The proxy to use, for example, <code>https://proxy:8888</code> <code><proxy-url></code>
<code>HTTPS_PROXY_CREDS</code>	string	Any credentials needed to authenticate against the proxy, for example, <code>username:password</code> .
<code><proxy-user></code>	string	The user for the proxy.
<code><proxy-password></code>	string	The password associated with <code><proxy-user></code> 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> \
HTTPS_PROXY=<proxy-url> \
HTTPS_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 <code>Console</code> logging provider. All of the ASP.NET Core configuration settings and default values for this logging provider are supported.
Debug	The ASP.NET Core <code>Debug</code> logging provider. All of the ASP.NET Core configuration settings and default values for this logging provider are supported.
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 `debug`, 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
<code>Format</code>	String	The output format for log files. Note: This value must be set to <code>json</code> to enable the logging provider. If this value is specified without also specifying an output mount while instantiating a container, an error occurs.
<code>MaxFileSize</code>	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 ASPNET 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 Computer Vision 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	<code>Input</code>	String	Computer Vision containers do not use this.
Optional	<code>Output</code>	String	The target of the output mount. The default value is <code>/output</code> . This is the location of the logs. This includes container logs. Example: <code>--mount type=bind,src=c:\output,target=/output</code>

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
<code>{API_KEY}</code>	The endpoint key of the <code>Computer Vision</code> resource on the Azure <code>Computer Vision</code> Keys page.	<code>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx</code>
<code>{ENDPOINT_URI}</code>	The billing endpoint value is available on the Azure <code>Computer Vision</code> 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.

Container Docker examples

The following Docker examples are for the Read container.

Basic example

```
docker run --rm -it -p 5000:5000 --memory 16g --cpus 8 \  
containerpreview.azurecr.io/microsoft/cognitive-services-read \  
Eula=accept \  
Billing={ENDPOINT_URI} \  
ApiKey={API_KEY}
```

Logging example

```
docker run --rm -it -p 5000:5000 --memory 16g --cpus 8 \  
containerpreview.azurecr.io/microsoft/cognitive-services-read \  
Eula=accept \  
Billing={ENDPOINT_URI} \  
ApiKey={API_KEY} \  
Logging:Console:LogLevel:Default=Information
```

Next steps

- Review [How to install and run containers](#).

Use Computer Vision container with Kubernetes and Helm

4/7/2020 • 7 minutes to read • [Edit Online](#)

One option to manage your Computer Vision containers on-premises is to use Kubernetes and Helm. Using Kubernetes and Helm to define a Computer Vision container image, we'll create a Kubernetes package. This package will be deployed to a Kubernetes cluster on-premises. Finally, we'll explore how to test the deployed services. For more information about running Docker containers without Kubernetes orchestration, see [install and run Computer Vision containers](#).

Prerequisites

The following prerequisites before using Computer Vision containers on-premises:

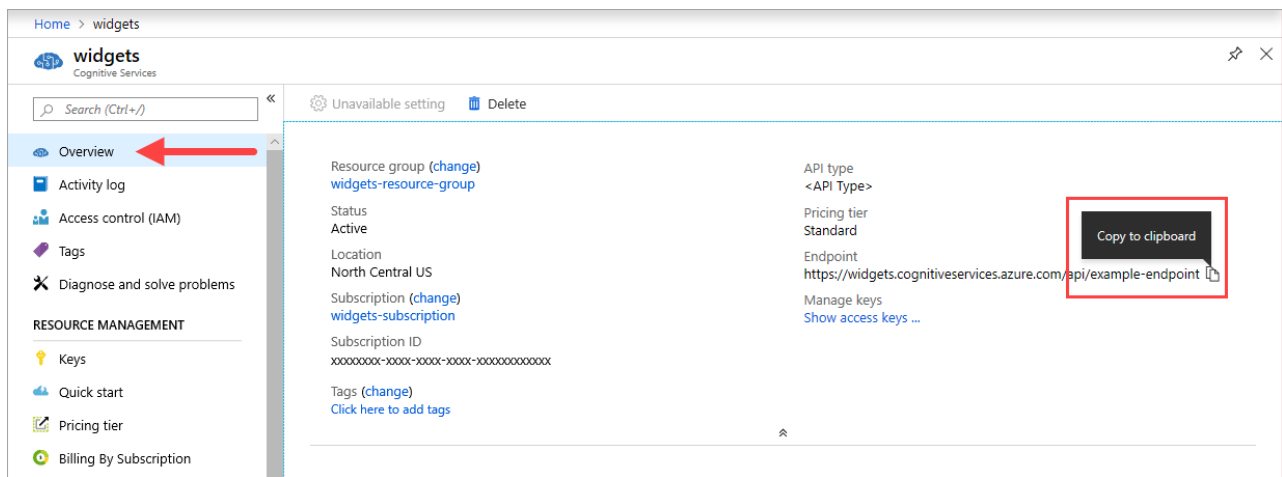
REQUIRED	PURPOSE
Azure Account	If you don't have an Azure subscription, create a free account before you begin.
Kubernetes CLI	The Kubernetes CLI is required for managing the shared credentials from the container registry. Kubernetes is also needed before Helm, which is the Kubernetes package manager.
Helm CLI	Install the Helm CLI , which is used to to install a helm chart (container package definition).
Computer Vision resource	<p>In order to use the container, you must have:</p> <p>An Azure Computer Vision resource and the associated API key the endpoint URI. Both values are available on the Overview and Keys pages for the resource and are required to start the container.</p> <p>{API_KEY}: One of the two available resource keys on the Keys page</p> <p>{ENDPOINT_URI}: The endpoint as provided on the Overview page</p>

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.

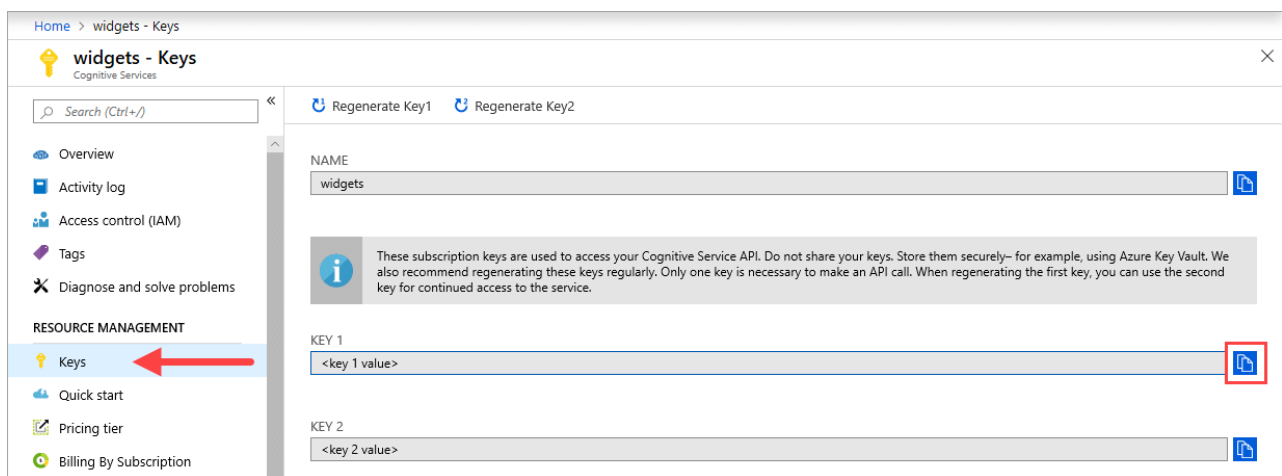
Endpoint URI `{ENDPOINT_URI}`

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.

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

NOTE

The requirements and recommendations are based on benchmarks with a single request per second, using an 8-MB image of a scanned business letter that contains 29 lines and a total of 803 characters.

The following table describes the minimum and recommended allocation of resources for each Read container.

CONTAINER	MINIMUM	RECOMMENDED	TPS (MINIMUM, MAXIMUM)
Read	1 cores, 8-GB memory, 0.24 TPS	8 cores, 16-GB memory, 1.17 TPS	0.24, 1.17

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

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

Connect to the Kubernetes cluster

The host computer is expected to have an available Kubernetes cluster. See this tutorial on [deploying a Kubernetes cluster](#) for a conceptual understanding of how to deploy a Kubernetes cluster to a host computer.

Sharing Docker credentials with the Kubernetes cluster

To allow the Kubernetes cluster to `docker pull` the configured image(s) from the `containerpreview.azurecr.io` container registry, you need to transfer the docker credentials into the cluster. Execute the `kubectl create` command below to create a *docker-registry secret* based on the credentials provided from the container registry access prerequisite.

From your command-line interface of choice, run the following command. Be sure to replace the `<username>`, `<password>`, and `<email-address>` with the container registry credentials.

```
kubectl create secret docker-registry containerpreview \
  --docker-server=containerpreview.azurecr.io \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-email=<email-address>
```

NOTE

If you already have access to the `containerpreview.azurecr.io` container registry, you could create a Kubernetes secret using the generic flag instead. Consider the following command that executes against your Docker configuration JSON.

```
kubectl create secret generic containerpreview \
  --from-file=.dockerconfigjson=~/.docker/config.json \
  --type=kubernetes.io/dockerconfigjson
```

The following output is printed to the console when the secret has been successfully created.

```
secret "containerpreview" created
```

To verify that the secret has been created, execute the `kubectl get` with the `secrets` flag.

```
kubectl get secrets
```

Executing the `kubectl get secrets` prints all the configured secrets.

NAME	TYPE	DATA	AGE
containerpreview	kubernetes.io/dockerconfigjson	1	30s

Configure Helm chart values for deployment

Start by creating a folder named *read*, then paste the following YAML content into a new file named *Chart.yaml*.

```
apiVersion: v1
name: read
version: 1.0.0
description: A Helm chart to deploy the microsoft/cognitive-services-read to a Kubernetes cluster
```

To configure the Helm chart default values, copy and paste the following YAML into a file named `values.yaml`. Replace the `# {ENDPOINT_URI}` and `# {API_KEY}` comments with your own values.

```
# These settings are deployment specific and users can provide customizations

read:
  enabled: true
  image:
    name: cognitive-services-read
    registry: containerpreview.azurecr.io/
    repository: microsoft/cognitive-services-read
    tag: latest
  pullSecret: containerpreview # Or an existing secret
  args:
    eula: accept
    billing: # {ENDPOINT_URI}
    apikey: # {API_KEY}
```

IMPORTANT

If the `billing` and `apikey` values are not provided, the services will expire after 15 min. Likewise, verification will fail as the services will not be available.

Create a *templates* folder under the *read* directory. Copy and paste the following YAML into a file named `deployment.yaml`. The `deployment.yaml` file will serve as a Helm template.

Templates generate manifest files, which are YAML-formatted resource descriptions that Kubernetes can understand. - [Helm Chart Template Guide](#)

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: read
spec:
  template:
    metadata:
      labels:
        app: read-app
    spec:
      containers:
        - name: {{.Values.read.image.name}}
          image: {{.Values.read.image.registry}}{{.Values.read.image.repository}}
          ports:
            - containerPort: 5000
          env:
            - name: EULA
              value: {{.Values.read.image.args.eula}}
            - name: billing
              value: {{.Values.read.image.args.billing}}
            - name: apikey
              value: {{.Values.read.image.args.apikey}}
          imagePullSecrets:
            - name: {{.Values.read.image.pullSecret}}

---
apiVersion: v1
kind: Service
metadata:
  name: read
spec:
  type: LoadBalancer
  ports:
    - port: 5000
  selector:
    app: read-app

```

The template specifies a load balancer service and the deployment of your container/image for Read.

The Kubernetes package (Helm chart)

The *Helm chart* contains the configuration of which docker image(s) to pull from the `containerpreview.azurecr.io` container registry.

A [Helm chart](#) is a collection of files that describe a related set of Kubernetes resources. A single chart might be used to deploy something simple, like a memcached pod, or something complex, like a full web app stack with HTTP servers, databases, caches, and so on.

The provided *Helm charts* pull the docker images of the Computer Vision Service, and the corresponding service from the `containerpreview.azurecr.io` container registry.

Install the Helm chart on the Kubernetes cluster

To install the *helm chart*, we'll need to execute the `helm install` command. Ensure to execute the install command from the directory above the `read` folder.

```
helm install read ./read
```

Here is an example output you might expect to see from a successful install execution:

```

NAME: read
LAST DEPLOYED: Thu Sep 04 13:24:06 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Pod(related)
NAME                READY  STATUS             RESTARTS  AGE
read-57cb76bcf7-45sdh  0/1    ContainerCreating   0          0s

==> v1/Service
NAME      TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
read     LoadBalancer  10.110.44.86  localhost    5000:31301/TCP   0s

==> v1beta1/Deployment
NAME      READY  UP-TO-DATE  AVAILABLE  AGE
read     0/1    1           0          0s

```

The Kubernetes deployment can take over several minutes to complete. To confirm that both pods and services are properly deployed and available, execute the following command:

```
kubectl get all
```

You should expect to see something similar to the following output:

```

kubectl get all
NAME                READY  STATUS             RESTARTS  AGE
pod/read-57cb76bcf7-45sdh  1/1    Running            0          17s

NAME                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
service/kubernetes  ClusterIP   10.96.0.1     <none>       443/TCP          45h
service/read        LoadBalancer 10.110.44.86  localhost    5000:31301/TCP   17s

NAME                READY  UP-TO-DATE  AVAILABLE  AGE
deployment.apps/read  1/1    1           1          17s

NAME                DESIRED  CURRENT  READY  AGE
replicaset.apps/read-57cb76bcf7  1        1        1      17s

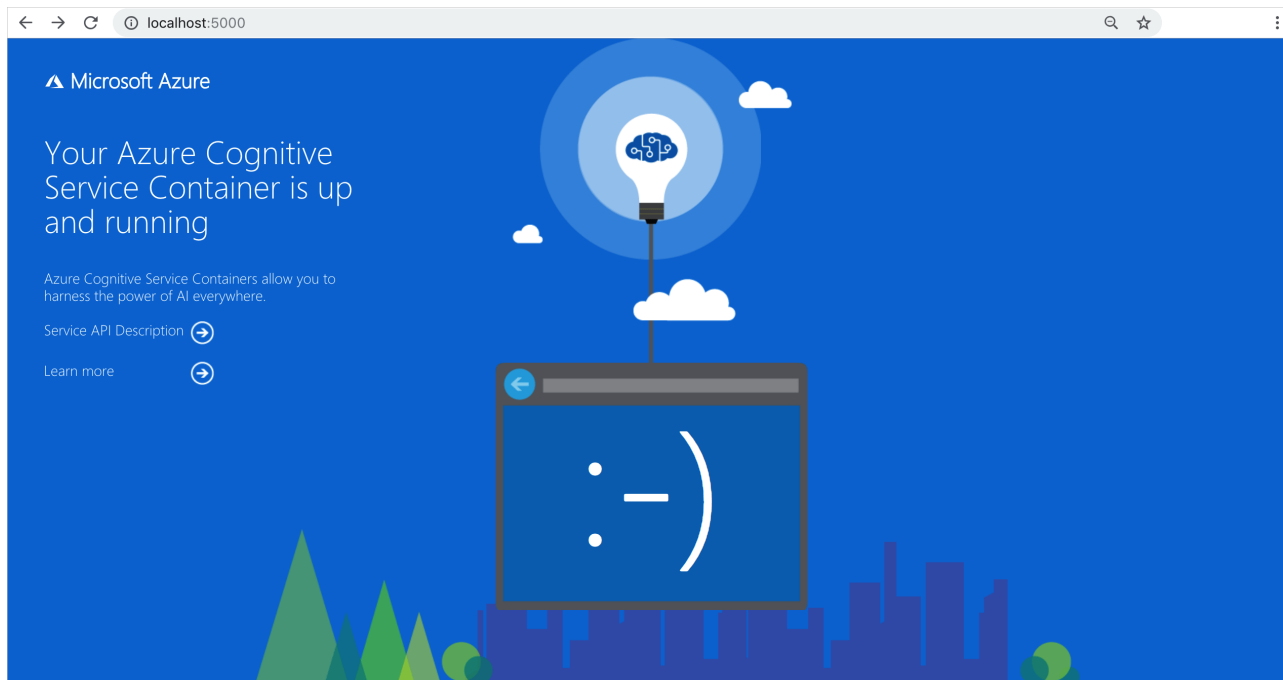
```

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
<code>http://localhost:5000/</code>	The container provides a home page.
<code>http://localhost:5000/ready</code>	Requested with GET, this provides a verification that the container is ready to accept a query against the model. This request can be used for Kubernetes liveness and readiness probes .

REQUEST URL	PURPOSE
<code>http://localhost:5000/status</code>	Also requested with GET, this verifies if the api-key used to start the container is valid without causing an endpoint query. This request can be used for Kubernetes liveness and readiness probes .
<code>http://localhost:5000/swagger</code>	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

For more details on installing applications with Helm in Azure Kubernetes Service (AKS), [visit here](#).

[Cognitive Services Containers](#)

Deploy the Computer Vision container to Azure Container Instances

4/7/2020 • 5 minutes to read • [Edit Online](#)

Learn how to deploy the Cognitive Services [Computer Vision](#) container to Azure [Container Instances](#). This procedure demonstrates the creation of the Computer Vision 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

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 Computer Vision resource

1. Sign into the [Azure portal](#).
2. Click [Create Computer Vision](#) 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	<input type="text" value="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 and , see [gathering required parameters](#) for details.

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 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. For more information of the YAML reference for Container instances, see [YAML reference: Azure Container Instances](#).


```

apiVersion: 2018-10-01
location: # < Valid location >
name: # < Container Group name >
properties:
  imageRegistryCredentials: # This is only required if you are pulling a non-public image that requires authentication to access.
  - server: containerpreview.azurecr.io
    username: # < The username for the preview container registry >
    password: # < The password for the preview container registry >
  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
  volumes: # This node, is only required for container instances that pull their model in at runtime, such as LUIS.
  - name: aci-file-share
    azureFile:
      shareName: # < File share name >
      storageAccountName: # < Storage account name>
      storageAccountKey: # < Storage account key >
  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.

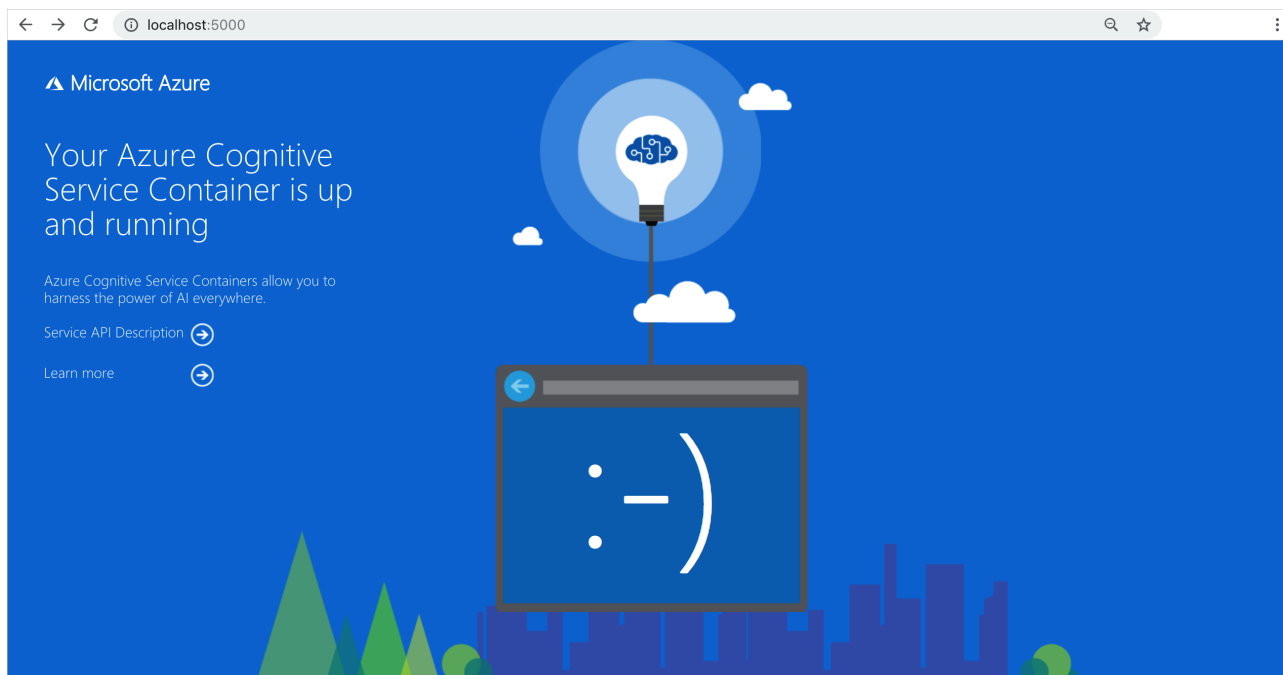
TIP

Pay close attention to the locations of public preview Azure Cognitive Service offerings, as the YAML will need 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
<code>http://localhost:5000/</code>	The container provides a home page.
<code>http://localhost:5000/ready</code>	Requested with GET, this provides a verification that the container is ready to accept a query against the model. This request can be used for Kubernetes liveness and readiness probes .
<code>http://localhost:5000/status</code>	Also requested with GET, this verifies if the api-key used to start the container is valid without causing an endpoint query. This request can be used for Kubernetes liveness and readiness probes .
<code>http://localhost:5000/swagger</code>	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](#)

Call the Computer Vision API

9/1/2020 • 5 minutes to read • [Edit Online](#)

This article demonstrates how to call the Computer Vision API by using the REST API. The samples are written both in C# by using the Computer Vision API client library and as HTTP POST or GET calls. The article focuses on:

- Getting tags, a description, and categories
- Getting domain-specific information, or "celebrities"

The examples in this article demonstrate the following features:

- Analyzing an image to return an array of tags and a description
- Analyzing an image with a domain-specific model (specifically, the "celebrities" model) to return the corresponding result in JSON

The features offer the following options:

- **Option 1:** Scoped Analysis - Analyze only a specified model
- **Option 2:** Enhanced Analysis - Analyze to provide additional details by using [86-categories taxonomy](#)

Prerequisites

- An Azure subscription - [Create one for free](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (**F0**) to try the service, and upgrade later to a paid tier for production.
- An image URL or a path to a locally stored image
- Supported input methods: a raw image binary in the form of an application/octet-stream, or an image URL
- Supported image file formats: JPEG, PNG, GIF, and BMP
- Image file size: 4 MB or less
- Image dimensions: 50 × 50 pixels or greater

Authorize the API call

Every call to the Computer Vision API requires a subscription key. This key must be either passed through a query string parameter or specified in the request header.

You can pass the subscription key by doing any of the following:

- Pass it through a query string, as in this Computer Vision API example:

```
https://westus.api.cognitive.microsoft.com/vision/v2.1/analyze?
visualFeatures=Description,Tags&subscription-key=<Your subscription key>
```

- Specify it in the HTTP request header:

```
ocp-apim-subscription-key: <Your subscription key>
```

- When you use the client library, pass the key through the constructor of `ComputerVisionClient`, and specify the region in a property of the client:

```
var visionClient = new ComputerVisionClient(new ApiKeyServiceClientCredentials("Your subscriptionKey"))
{
    Endpoint = "https://westus.api.cognitive.microsoft.com"
}
```

Upload an image to the Computer Vision API service

The basic way to perform the Computer Vision API call is by uploading an image directly to return tags, a description, and celebrities. You do this by sending a "POST" request with the binary image in the HTTP body together with the data read from the image. The upload method is the same for all Computer Vision API calls. The only difference is the query parameters that you specify.

For a specified image, get tags and a description by using either of the following options:

Option 1: Get a list of tags and a description

```
POST https://westus.api.cognitive.microsoft.com/vision/v2.1/analyze?
visualFeatures=Description,Tags&subscription-key=<Your subscription key>
```

```
using System.IO;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;

ImageAnalysis imageAnalysis;
var features = new VisualFeatureTypes[] { VisualFeatureTypes.Tags, VisualFeatureTypes.Description };

using (var fs = new FileStream(@"C:\Vision\Sample.jpg", FileMode.Open))
{
    imageAnalysis = await visionClient.AnalyzeImageInStreamAsync(fs, features);
}
```

Option 2: Get a list of tags only or a description only

For tags only, run:

```
POST https://westus.api.cognitive.microsoft.com/vision/v2.1/tag?subscription-key=<Your subscription key>
var tagResults = await visionClient.TagImageAsync("http://contoso.com/example.jpg");
```

For a description only, run:

```
POST https://westus.api.cognitive.microsoft.com/vision/v2.1/describe?subscription-key=<Your subscription key>
using (var fs = new FileStream(@"C:\Vision\Sample.jpg", FileMode.Open))
{
    imageDescription = await visionClient.DescribeImageInStreamAsync(fs);
}
```

Get domain-specific analysis (celebrities)

Option 1: Scoped analysis - Analyze only a specified model

```
POST https://westus.api.cognitive.microsoft.com/vision/v2.1/models/celebrities/analyze
var celebritiesResult = await visionClient.AnalyzeImageInDomainAsync(url, "celebrities");
```

For this option, all other query parameters {visualFeatures, details} are not valid. If you want to see all supported models, use:

```
GET https://westus.api.cognitive.microsoft.com/vision/v2.1/models
var models = await visionClient.ListModelsAsync();
```

Option 2: Enhanced analysis - Analyze to provide additional details by using 86-categories taxonomy

For applications where you want to get a generic image analysis in addition to details from one or more domain-specific models, extend the v1 API by using the models query parameter.

```
POST https://westus.api.cognitive.microsoft.com/vision/v2.1/analyze?details=celebrities
```

When you invoke this method, you first call the [86-category](#) classifier. If any of the categories matches that of a known or matching model, a second pass of classifier invocations occurs. For example, if "details=all" or "details" includes "celebrities," you call the celebrities model after you call the 86-category classifier. The result includes the category person. In contrast with Option 1, this method increases latency for users who are interested in celebrities.

In this case, all v1 query parameters behave in the same way. If you don't specify visualFeatures=categories, it's implicitly enabled.

Retrieve and understand the JSON output for analysis

Here's an example:

```
{
  "tags": [
    {
      "name": "outdoor",
      "score": 0.976
    },
    {
      "name": "bird",
      "score": 0.95
    }
  ],
  "description": {
    "tags": [
      "outdoor",
      "bird"
    ],
    "captions": [
      {
        "text": "partridge in a pear tree",
        "confidence": 0.96
      }
    ]
  }
}
```

FIELD	TYPE	CONTENT
Tags	object	The top-level object for an array of tags.
tags[].Name	string	The keyword from the tags classifier.
tags[].Score	number	The confidence score, between 0 and 1.

FIELD	TYPE	CONTENT
description	object	The top-level object for a description.
description.tags[]	string	The list of tags. If there is insufficient confidence in the ability to produce a caption, the tags might be the only information available to the caller.
description.captions[].text	string	A phrase describing the image.
description.captions[].confidence	number	The confidence score for the phrase.

Retrieve and understand the JSON output of domain-specific models

Option 1: Scoped analysis - Analyze only a specified model

The output is an array of tags, as shown in the following example:

```
{
  "result": [
    {
      "name": "golden retriever",
      "score": 0.98
    },
    {
      "name": "Labrador retriever",
      "score": 0.78
    }
  ]
}
```

Option 2: Enhanced analysis - Analyze to provide additional details by using the "86-categories" taxonomy

For domain-specific models using Option 2 (enhanced analysis), the categories return type is extended, as shown in the following example:

```
{
  "requestId": "87e44580-925a-49c8-b661-d1c54d1b83b5",
  "metadata": {
    "width": 640,
    "height": 430,
    "format": "Jpeg"
  },
  "result": {
    "celebrities": [
      {
        "name": "Richard Nixon",
        "faceRectangle": {
          "left": 107,
          "top": 98,
          "width": 165,
          "height": 165
        },
        "confidence": 0.9999827
      }
    ]
  }
}
```

The categories field is a list of one or more of the [86 categories](#) in the original taxonomy. Categories that end in an underscore match that category and its children (for example, "people_" or "people_group," for the celebrities model).

FIELD	TYPE	CONTENT
categories	<code>object</code>	The top-level object.
categories[].name	<code>string</code>	The name from the 86-category taxonomy list.
categories[].score	<code>number</code>	The confidence score, between 0 and 1.
categories[].detail	<code>object?</code>	(Optional) The detail object.

If multiple categories match (for example, the 86-category classifier returns a score for both "people_" and "people_young," when model=celebrities), the details are attached to the most general level match ("people_" in that example).

Error responses

These errors are identical to those in vision.analyze, with the additional NotSupportedModel error (HTTP 400), which might be returned in both the Option 1 and Option 2 scenarios. For Option 2 (enhanced analysis), if any of the models that are specified in the details isn't recognized, the API returns a NotSupportedModel, even if one or more of them are valid. To find out what models are supported, you can call listModels.

Next steps

To use the REST API, go to [Computer Vision API Reference](#).

Analyze videos in near real time

9/1/2020 • 7 minutes to read • [Edit Online](#)

This article demonstrates how to perform near real-time analysis on frames that are taken from a live video stream by using the Computer Vision API. The basic elements of such an analysis are:

- Acquiring frames from a video source.
- Selecting which frames to analyze.
- Submitting these frames to the API.
- Consuming each analysis result that's returned from the API call.

The samples in this article are written in C#. To access the code, go to the [Video frame analysis sample](#) page on GitHub.

Approaches to running near real-time analysis

You can solve the problem of running near real-time analysis on video streams by using a variety of approaches. This article outlines three of them, in increasing levels of sophistication.

Design an infinite loop

The simplest design for near real-time analysis is an infinite loop. In each iteration of this loop, you 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 your analysis were to consist of a lightweight, client-side algorithm, this approach would be suitable. However, when the analysis occurs in the cloud, the resulting latency means that an API call might take several seconds. During this time, you're not capturing images, and your thread is essentially doing nothing. Your maximum frame rate is limited by the latency of the API calls.

Allow the API calls to run in parallel

Although a simple, single-threaded loop makes sense for a lightweight, client-side algorithm, it doesn't fit well with the latency of a cloud API call. The solution to this problem is to allow the long-running API call to run in parallel with the frame-grabbing. In C#, you could do this by using task-based parallelism. For example, you can run the following code:

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

With this approach, you launch each analysis in a separate task. The task can run in the background while you continue grabbing new frames. The approach avoids blocking the main thread as you wait for an API call to return. However, the approach can present certain disadvantages:

- It costs you some of the guarantees that the simple version provided. That is, multiple API calls might occur in parallel, and the results might get returned in the wrong order.
- It could also cause multiple threads to enter the `ConsumeResult()` function simultaneously, which might be dangerous if the function isn't thread-safe.
- Finally, this simple code doesn't keep track of the tasks that get created, so exceptions silently disappear. Thus, you need to add a "consumer" thread that tracks the analysis tasks, raises exceptions, kills long-running tasks, and ensures that the results get consumed in the correct order, one at a time.

Design a producer-consumer system

For your final approach, designing a "producer-consumer" system, you build a producer thread that looks similar to your previously mentioned infinite loop. However, instead of consuming the analysis results as soon as they're available, the producer simply places the tasks in 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);
    }
}

```

You also create a consumer thread, which 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, you can guarantee that the 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();

    // Wait 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);
    }
}

```

Implement the solution

Get started quickly

To help get your app up and running as quickly as possible, we've implemented the system that's described in the preceding section. It's intended to be flexible enough to accommodate many scenarios, while being easy to use. To

access the code, go to the [Video frame analysis sample](#) page on GitHub.

The library contains the `FrameGrabber` class, which implements the previously discussed producer-consumer system to process video frames from a webcam. Users 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 when a new analysis result is available.

To illustrate some of the possibilities, we've provided two sample apps that use the library.

The first sample app is a simple console app that grabs frames from the default webcam and then submits them to the Face service for face detection. A simplified version of the app is reproduced in the following code:

```
using System;
using System.Linq;
using Microsoft.Azure.CognitiveServices.Vision.Face;
using Microsoft.Azure.CognitiveServices.Vision.Face.Models;
using VideoFrameAnalyzer;

namespace BasicConsoleSample
{
    internal class Program
    {
        const string ApiKey = "<your API key>";
        const string Endpoint = "https://<your API region>.api.cognitive.microsoft.com";

        private static async Task Main(string[] args)
        {
            // Create grabber.
            FrameGrabber<DetectedFace[]> grabber = new FrameGrabber<DetectedFace[]>();

            // Create Face Client.
            FaceClient faceClient = new FaceClient(new ApiKeyServiceClientCredentials(ApiKey))
            {
                Endpoint = Endpoint
            };

            // Set up a listener for when we acquire a new frame.
            grabber.NewFrameProvided += (s, e) =>
            {
                Console.WriteLine($"New frame acquired at {e.Frame.Metadata.Timestamp}");
            };

            // Set up a Face API call.
            grabber.AnalysisFunction = async frame =>
            {
                Console.WriteLine($"Submitting frame acquired at {frame.Metadata.Timestamp}");
                // Encode image and submit to Face service.
                return (await
                    faceClient.Face.DetectWithStreamAsync(frame.Image.ToMemoryStream(".jpg")).ToArray());
            };

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

            // Tell grabber when to call the API.
            // See also TriggerAnalysisOnPredicate
            grabber.TriggerAnalysisOnInterval(TimeSpan.FromMilliseconds(3000));

            // Start running in the background.
        }
    }
}
```

```

// Start running in the background.
await grabber.StartProcessingCameraAsync();

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

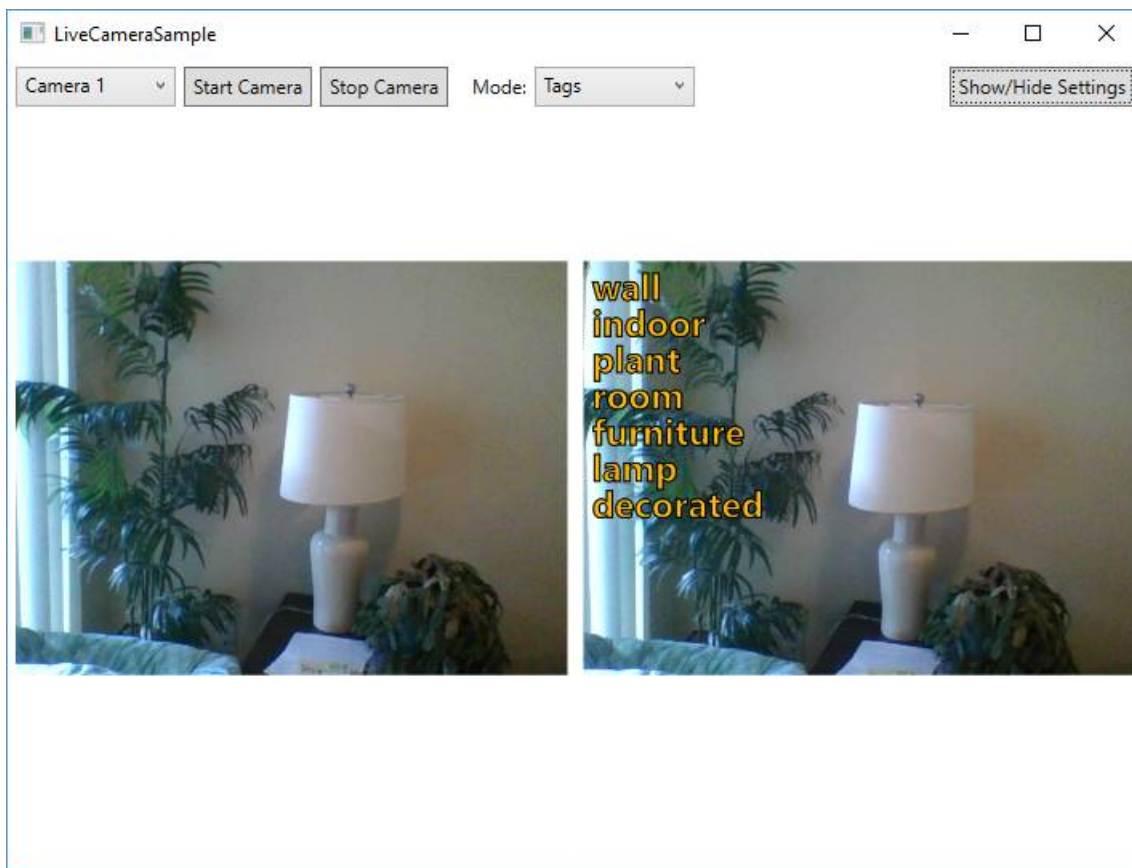
// Stop, blocking until done.
await grabber.StopProcessingAsync();
}
}
}

```

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

In most modes, there's a visible delay between the live video on the left and the visualized analysis on the right. This delay is the time that it takes to make the API call. An exception is in the "EmotionsWithClientFaceDetect" mode, which performs face detection locally on the client computer by using OpenCV before it submits any images to Azure Cognitive Services.

By using this approach, you can visualize the detected face immediately. You can then update the emotions later, after the API call returns. This demonstrates the possibility of a "hybrid" approach. That is, some simple processing can be performed on the client, and then Cognitive Services APIs can be used to augment this processing with more advanced analysis when necessary.



Integrate the samples into your codebase

To get started with this sample, do the following:

1. Create an [Azure account](#). If you already have one, you can skip to the next step.
2. Create resources for Computer Vision and Face in the Azure portal to get your key and endpoint. Make sure to select the free tier (F0) during setup.
 - [Computer Vision](#)

- [Face](#) After the resources are deployed, click **Go to resource** to collect your key and endpoint for each resource.
3. Clone the [Cognitive-Samples-VideoFrameAnalysis](#) GitHub repo.
 4. Open the sample in Visual Studio 2015 or later, and then build and run the sample applications:
 - For BasicConsoleSample, the Face key is hard-coded directly in [BasicConsoleSample/Program.cs](#).
 - For LiveCameraSample, enter the keys in the **Settings** pane of the app. The keys are persisted across sessions as user data.

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

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

Summary

In this article, you learned how to run near real-time analysis on live video streams by using the Face and Computer Vision services. You also learned how you can use our sample code to get started.

Feel free to provide feedback and suggestions in the [GitHub repository](#). To provide broader API feedback, go to our [UserVoice site](#).

Sample: Explore an image processing app with C#

9/1/2020 • 19 minutes to read • [Edit Online](#)

Explore a basic Windows application that uses Computer Vision to perform optical character recognition (OCR), create smart-cropped thumbnails, plus detect, categorize, tag and describe visual features, including faces, in an image. The below example lets you submit an image URL or a locally stored file. You can use this open source example as a template for building your own app for Windows using the Computer Vision API and Windows Presentation Foundation (WPF), a part of .NET Framework.

- Get the sample app from GitHub
- Open and build the sample app in Visual Studio
- Run the sample app and interact with it to perform various scenarios
- Explore the various scenarios included with the sample app

Prerequisites

Before exploring the sample app, ensure that you've met the following prerequisites:

- You must have [Visual Studio 2015](#) or later.
- An Azure subscription - [Create one for free](#)
- Once you have your Azure subscription, [create a Computer Vision resource](#) in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
 - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
 - You can use the free pricing tier (F0) to try the service, and upgrade later to a paid tier for production.

Get the sample app

The Computer Vision sample app is available on GitHub from the `Microsoft/Cognitive-Vision-Windows` repository. This repository also includes the `Microsoft/Cognitive-Common-Windows` repository as a Git submodule. You can recursively clone this repository, including the submodule, either by using the `git clone --recurse-submodules` command from the command line, or by using GitHub Desktop.

For example, to recursively clone the repository for the Computer Vision sample app from a command prompt, run the following command:

```
git clone --recurse-submodules https://github.com/Microsoft/Cognitive-Vision-Windows.git
```

IMPORTANT

Do not download this repository as a ZIP. Git doesn't include submodules when downloading a repository as a ZIP.

Get optional sample images

You can optionally use the sample images included with the [Face](#) sample app, available on GitHub from the `Microsoft/Cognitive-Face-Windows` repository. That sample app includes a folder, `/Data`, which contains multiple images of people. You can recursively clone this repository, as well, by the methods described for the Computer Vision sample app.

For example, to recursively clone the repository for the Face sample app from a command prompt, run the following command:

```
git clone --recurse-submodules https://github.com/Microsoft/Cognitive-Face-Windows.git
```

Open and build the sample app in Visual Studio

You must build the sample app first, so that Visual Studio can resolve dependencies, before you can run or explore the sample app. To open and build the sample app, do the following steps:

1. Open the Visual Studio solution file, `/Sample-WPF/VisionAPI-WPF-Samples.sln`, in Visual Studio.
2. Ensure that the Visual Studio solution contains two projects:
 - SampleUserControlLibrary
 - VisionAPI-WPF-Samples

If the SampleUserControlLibrary project is unavailable, confirm that you've recursively cloned the `Microsoft/Cognitive-Vision-Windows` repository.

3. In Visual Studio, either press Ctrl+Shift+B or choose **Build** from the ribbon menu and then choose **Build Solution** to build the solution.

Run and interact with the sample app

You can run the sample app, to see how it interacts with you and with the Computer Vision client library when performing various tasks, such as generating thumbnails or tagging images. To run and interact with the sample app, do the following steps:

1. After the build is complete, either press F5 or choose **Debug** from the ribbon menu and then choose **Start debugging** to run the sample app.
2. When the sample app is displayed, choose **Subscription Key Management** from the navigation pane to display the Subscription Key Management page.

The screenshot shows the 'Subscription Key Management' page in the Vision API application. The page has a teal header with the text 'Vision API'. On the left side, there is a navigation pane with the following items: 'Subscription Key Management' (selected), 'Select a scenario:', 'Analyze Image', 'Analyze Image with Domain Model', 'Describe Image', 'Generate Tags', 'Recognize Text (OCR)', 'Recognize Text V2 (English)', and 'Get Thumbnail'. The main content area on the right contains the following text: 'To use the service, you need to ensure that you have right subscription key. Please note that each service (Face, Emotion, Speech, etc) has its own subscription key. If you do not have key yet, please use the link to get a key first, then paste the key into the textbox below.' Below this text is a link 'Get Key'. There are two input fields: 'Subscription Key:' with the placeholder text 'Paste your subscription key here firstly' and 'Endpoint:' with the placeholder text 'Paste your endpoint here to start'. At the bottom of the input fields are two buttons: 'Save Setting' and 'Delete Setting'.

3. Enter your subscription key in **Subscription Key**.
4. Enter the endpoint URL in **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](#).

5. If you don't want to enter your subscription key and endpoint URL the next time you run the sample app, choose **Save Setting** to save the subscription key and endpoint URL to your computer. If you want to

delete your previously-saved subscription key and endpoint URL, choose **Delete Setting**.

NOTE

The sample app uses isolated storage, and `System.IO.IsolatedStorage`, to store your subscription key and endpoint URL.

6. Under **Select a scenario** in the navigation pane, select one of the scenarios currently included with the sample app:

SCENARIO	DESCRIPTION
Analyze Image	Uses the Analyze Image operation to analyze a local or remote image. You can choose the visual features and language for the analysis, and see both the image and the results.
Analyze Image with Domain Model	Uses the List Domain Specific Models operation to list the domain models from which you can select, and the Recognize Domain Specific Content operation to analyze a local or remote image using the selected domain model. You can also choose the language for the analysis.
Describe Image	Uses the Describe Image operation to create a human-readable description of a local or remote image. You can also choose the language for the description.
Generate Tags	Uses the Tag Image operation to tag the visual features of a local or remote image. You can also choose the language used for the tags.
Recognize Text (OCR)	Uses the OCR operation to recognize and extract printed text from an image. You can either choose the language to use, or let Computer Vision auto-detect the language.
Recognize Text V2 (English)	Uses the Recognize Text and Get Recognize Text Operation Result operations to asynchronously recognize and extract printed or handwritten text from an image.
Get Thumbnail	Uses the Get Thumbnail operation to generate a thumbnail for a local or remote image.

The following screenshot illustrates the page provided for the Analyze Image scenario, after analyzing a sample image.

Vision API

Subscription Key Management

Select a scenario:

Analyze Image

Analyze Image with Domain Model

Describe Image

Generate Tags

Recognize Text (OCR)

Get Thumbnail

Analyze an Image

Please click either [Load Image] or paste in an image url and click [Analyze]

Load Image

<https://oxfordportal.blob.core.windows.net/vision/Analysis/1-1.jpg>

Analyze

Analyzing Done



[21:52:43.370111]: Description :
[21:52:43.386402]: Caption : a man swimming in a pool of water; Confidence : 0.752564820236237
[21:52:43.386402]: Tags : water, person, sport, swimming, pool,
[21:52:43.402029]: Tags :
[21:52:43.402029]: Name : water; Confidence : 0.999414682388306; Hint :
[21:52:43.402029]: Name : person; Confidence : 0.936775147914886; Hint :
[21:52:43.417652]: Name : sport; Confidence : 0.848687767982483; Hint :
[21:52:43.417652]: Name : swimming; Confidence : 0.845447421073914; Hint : sport
[21:52:43.433278]: Name : water sport; Confidence : 0.827535569667816; Hint : sport
[21:52:43.433278]: Name : pool; Confidence : 0.805495202541351; Hint :

Explore the sample app

The Visual Studio solution for the Computer Vision sample app contains two projects:

- SampleUserControlLibrary

The SampleUserControlLibrary project provides functionality shared by multiple Cognitive Services samples.

The project contains the following:

- SampleScenarios

A UserControl that provides a standardized presentation, such as the title bar, navigation pane, and content pane, for samples. The Computer Vision sample app uses this control in the MainWindow.xaml window to display scenario pages and access information shared across scenarios, such as the subscription key and endpoint URL.

- SubscriptionKeyPage

A Page that provides a standardized layout for entering a subscription key and endpoint URL for the sample app. The Computer Vision sample app uses this page to manage the subscription key and endpoint URL used by the scenario pages.

- VideoResultControl

A UserControl that provides a standardized presentation for video information. The Computer Vision sample app doesn't use this control.

- VisionAPI-WPF-Samples

The main project for the Computer Vision sample app, this project contains all of the interesting functionality for Computer Vision. The project contains the following:

- AnalyzeInDomainPage.xaml

The scenario page for the Analyze Image with Domain Model scenario.

- AnalyzeImage.xaml

The scenario page for the Analyze Image scenario.

- DescribePage.xaml
The scenario page for the Describe Image scenario.
- ImageScenarioPage.cs
The ImageScenarioPage class, from which all of the scenario pages in the sample app are derived. This class manages functionality, such as providing credentials and formatting output, shared by all of the scenario pages.
- MainWindow.xaml
The main window for the sample app, it uses the SampleScenarios control to present the SubscriptionKeyPage and scenario pages.
- OCRPage.xaml
The scenario page for the Recognize Text (OCR) scenario.
- RecognizeLanguage.cs
The RecognizeLanguage class, which provides information about the languages supported by the various methods in the sample app.
- TagsPage.xaml
The scenario page for the Generate Tags scenario.
- TextRecognitionPage.xaml
The scenario page for the Recognize Text V2 (English) scenario.
- ThumbnailPage.xaml
The scenario page for the Get Thumbnail scenario.

Explore the sample code

Key portions of sample code are framed with comment blocks that start with `KEY SAMPLE CODE STARTS HERE` and end with `KEY SAMPLE CODE ENDS HERE`, to make it easier for you to explore the sample app. These key portions of sample code contain the code most relevant to learning how to use the Computer Vision API client library to do various tasks. You can search for `KEY SAMPLE CODE STARTS HERE` in Visual Studio to move between the most relevant sections of code in the Computer Vision sample app.

For example, the `UploadAndAnalyzeImageAsync` method, shown following and included in AnalyzePage.xaml, demonstrates how to use the client library to analyze a local image by invoking the `ComputerVisionClient.AnalyzeImageInStreamAsync` method.

```

private async Task<ImageAnalysis> UploadAndAnalyzeImageAsync(string imagePath)
{
    // -----
    // KEY SAMPLE CODE STARTS HERE
    // -----

    //
    // Create Cognitive Services Vision API Service client.
    //
    using (var client = new ComputerVisionClient(Credentials) { Endpoint = Endpoint })
    {
        Log("ComputerVisionClient is created");

        using (Stream imageFileStream = File.OpenRead(imageFilePath))
        {
            //
            // Analyze the image for all visual features.
            //
            Log("Calling ComputerVisionClient.AnalyzeImageInStreamAsync()...");
            VisualFeatureTypes[] visualFeatures = GetSelectedVisualFeatures();
            string language = (_language.SelectedItem as RecognizeLanguage).ShortCode;
            ImageAnalysis analysisResult = await client.AnalyzeImageInStreamAsync(imageFileStream,
visualFeatures, null, language);
            return analysisResult;
        }
    }

    // -----
    // KEY SAMPLE CODE ENDS HERE
    // -----
}

```

Explore the client library

This sample app uses the Computer Vision API client library, a thin C# client wrapper for the Computer Vision API in Azure Cognitive Services. The client library is available from NuGet in the [Microsoft.Azure.CognitiveServices.Vision.ComputerVision](#) package. When you built the Visual Studio application, you retrieved the client library from its corresponding NuGet package. You can also view the source code for the client library in the `/ClientLibrary` folder of the `Microsoft/Cognitive-Vision-Windows` repository.

The client library's functionality centers around the `ComputerVisionClient` class, in the `Microsoft.Azure.CognitiveServices.Vision.ComputerVision` namespace, while the models used by the `ComputerVisionClient` class when interacting with Computer Vision are found in the `Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models` namespace. In the various XAML scenario pages included with the sample app, you'll find the following `using` directives for those namespaces:

```

// -----
// KEY SAMPLE CODE STARTS HERE
// Use the following namespace for ComputerVisionClient.
// -----
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
// -----
// KEY SAMPLE CODE ENDS HERE
// -----

```

You'll learn more about the various methods included with the `ComputerVisionClient` class as you explore the scenarios included with the Computer Vision sample app.

Explore the Analyze Image scenario

This scenario is managed by the `AnalyzePage.xaml` page. You can choose the visual features and language for the analysis, and see both the image and the results. The scenario page does this by using one of the following methods, depending on the source of the image:

- `UploadAndAnalyzeImageAsync`

This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.AnalyzeImageInStreamAsync` method.

- `AnalyzeUrlAsync`

This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.AnalyzeImageAsync` method.

The `UploadAndAnalyzeImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the visual features and language selected in the scenario page. It calls the `ComputerVisionClient.AnalyzeImageInStreamAsync` method, passing the `Stream` for the file, the visual features, and the language, then returns the result as an `ImageAnalysis` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `AnalyzeUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the visual features and language selected in the scenario page. It calls the `ComputerVisionClient.AnalyzeImageInStreamAsync` method, passing the image URL, the visual features, and the language, then returns the result as an `ImageAnalysis` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Analyze Image with Domain Model scenario

This scenario is managed by the `AnalyzeInDomainPage.xaml` page. You can choose a domain model, such as `celebrities` or `landmarks`, and language to perform a domain-specific analysis of the image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- `GetAvailableDomainModelsAsync`

This method gets the list of available domain models from Computer Vision and populates the `_domainModelComboBox` ComboBox control on the page, using the `ComputerVisionClient.ListModelsAsync` method.

- `UploadAndAnalyzeInDomainImageAsync`

This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.AnalyzeImageByDomainInStreamAsync` method.

- `AnalyzeInDomainUrlAsync`

This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.AnalyzeImageByDomainAsync` method.

The `UploadAndAnalyzeInDomainImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the language selected in the scenario page. It calls the `ComputerVisionClient.AnalyzeImageByDomainInStreamAsync` method, passing the `Stream` for the file, the name of the domain model, and the language, then returns the result as an `DomainModelResults` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `AnalyzeInDomainUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the language selected in the scenario page. It calls the `ComputerVisionClient.AnalyzeImageByDomainAsync` method, passing the image URL, the visual features, and the

language, then returns the result as an `DomainModelResults` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Describe Image scenario

This scenario is managed by the `DescribePage.xaml` page. You can choose a language to create a human-readable description of the image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- `UploadAndDescribeImageAsync`
This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.DescribeImageInStreamAsync` method.
- `DescribeUrlAsync`
This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.DescribeImageAsync` method.

The `UploadAndDescribeImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the language selected in the scenario page. It calls the `ComputerVisionClient.DescribeImageInStreamAsync` method, passing the `Stream` for the file, the maximum number of candidates (in this case, 3), and the language, then returns the result as an `ImageDescription` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `DescribeUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the language selected in the scenario page. It calls the `ComputerVisionClient.DescribeImageAsync` method, passing the image URL, the maximum number of candidates (in this case, 3), and the language, then returns the result as an `ImageDescription` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Generate Tags scenario

This scenario is managed by the `TagsPage.xaml` page. You can choose a language to tag the visual features of an image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- `UploadAndGetTagsForImageAsync`
This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.TagImageInStreamAsync` method.
- `GenerateTagsForUrlAsync`
This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.TagImageAsync` method.

The `UploadAndGetTagsForImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the language selected in the scenario page. It calls the `ComputerVisionClient.TagImageInStreamAsync` method, passing the `Stream` for the file and the language, then returns the result as a `TagResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `GenerateTagsForUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the language selected in the scenario page. It calls the `ComputerVisionClient.TagImageAsync` method, passing the image URL and the language, then returns the result as a `TagResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the

scenario page.

Explore the Recognize Text (OCR) scenario

This scenario is managed by the `OCRPage.xaml` page. You can choose a language to recognize and extract printed text from an image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- `UploadAndRecognizeImageAsync`
This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.RecognizePrintedTextInStreamAsync` method.
- `RecognizeUrlAsync`
This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.RecognizePrintedTextAsync` method.

The `UploadAndRecognizeImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the language selected in the scenario page. It calls the `ComputerVisionClient.RecognizePrintedTextInStreamAsync` method, indicating that orientation is not detected and passing the `Stream` for the file and the language, then returns the result as an `OcrResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `RecognizeUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the language selected in the scenario page. It calls the `ComputerVisionClient.RecognizePrintedTextAsync` method, indicating that orientation is not detected and passing the image URL and the language, then returns the result as an `OcrResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Recognize Text V2 (English) scenario

This scenario is managed by the `TextRecognitionPage.xaml` page. You can choose the recognition mode and a language to asynchronously recognize and extract either printed or handwritten text from an image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- `UploadAndRecognizeImageAsync`
This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `RecognizeAsync` method and passing a parameterized delegate for the `ComputerVisionClient.RecognizeTextInStreamAsync` method.
- `RecognizeUrlAsync`
This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `RecognizeAsync` method and passing a parameterized delegate for the `ComputerVisionClient.RecognizeTextAsync` method.
- `RecognizeAsync` This method handles the asynchronous calling for both the `UploadAndRecognizeImageAsync` and `RecognizeUrlAsync` methods, as well as polling for results by calling the `ComputerVisionClient.GetTextOperationResultAsync` method.

Unlike the other scenarios included in the Computer Vision sample app, this scenario is asynchronous, in that one method is called to start the process, but a different method is called to check on the status and return the results of that process. The logical flow in this scenario is somewhat different from that in the other scenarios.

The `UploadAndRecognizeImageAsync` method opens the local file specified in `imageFilePath` for reading as a `Stream`, then calls the `RecognizeAsync` method, passing:

- A lambda expression for a parameterized asynchronous delegate of the `ComputerVisionClient.RecognizeTextInStreamAsync` method, with the `Stream` for the file and the recognition mode as parameters, in `GetHeadersAsyncFunc`.
- A lambda expression for a delegate to get the `Operation-Location` response header value, in `GetOperationUrlFunc`.

The `RecognizeUrlAsync` method calls the `RecognizeAsync` method, passing:

- A lambda expression for a parameterized asynchronous delegate of the `ComputerVisionClient.RecognizeTextAsync` method, with the URL of the remote image and the recognition mode as parameters, in `GetHeadersAsyncFunc`.
- A lambda expression for a delegate to get the `Operation-Location` response header value, in `GetOperationUrlFunc`.

When the `RecognizeAsync` method is completed, both `UploadAndRecognizeImageAsync` and `RecognizeUrlAsync` methods return the result as a `TextOperationResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `RecognizeAsync` method calls the parameterized delegate for either the `ComputerVisionClient.RecognizeTextInStreamAsync` OR `ComputerVisionClient.RecognizeTextAsync` method passed in `GetHeadersAsyncFunc` and waits for the response. The method then calls the delegate passed in `GetOperationUrlFunc` to get the `Operation-Location` response header value from the response. This value is the URL used to retrieve the results of the method passed in `GetHeadersAsyncFunc` from Computer Vision.

The `RecognizeAsync` method then calls the `ComputerVisionClient.GetTextOperationResultAsync` method, passing the URL retrieved from the `Operation-Location` response header, to get the status and result of the method passed in `GetHeadersAsyncFunc`. If the status doesn't indicate that the method completed, successfully or unsuccessfully, the `RecognizeAsync` method calls `ComputerVisionClient.GetTextOperationResultAsync` 3 more times, waiting 3 seconds between calls. The `RecognizeAsync` method returns the results to the method that called it.

Explore the Get Thumbnail scenario

This scenario is managed by the `ThumbnailPage.xaml` page. You can indicate whether to use smart cropping, and specify desired height and width, to generate a thumbnail from an image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- `UploadAndThumbnailImageAsync`
This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.GenerateThumbnailInStreamAsync` method.
- `ThumbnailUrlAsync`
This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.GenerateThumbnailAsync` method.

The `UploadAndThumbnailImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`. It calls the `ComputerVisionClient.GenerateThumbnailInStreamAsync` method, passing the width, height, the `Stream` for the file, and whether to use smart cropping, then returns the result as a `Stream`. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `RecognizeUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It calls the `ComputerVisionClient.GenerateThumbnailAsync` method, passing the width, height, the URL for the image, and whether to use smart cropping, then returns the result as a `Stream`. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Clean up resources

When no longer needed, delete the folder into which you cloned the `Microsoft/Cognitive-Vision-Windows` repository. If you opted to use the sample images, also delete the folder into which you cloned the `Microsoft/Cognitive-Face-Windows` repository.

Next steps

[Get started with Face service](#)

Computer Vision API Frequently Asked Questions

7/26/2019 • 2 minutes to read • [Edit Online](#)

TIP

If you can't find answers to your questions in this FAQ, try asking the Computer Vision API community on [StackOverflow](#) or contact [Help and Support on UserVoice](#)

Question: *Can I train Computer Vision API to use custom tags? For example, I would like to feed in pictures of cat breeds to 'train' the AI, then receive the breed value on an AI request.*

Answer: This function is currently not available. However, our engineers are working to bring this functionality to Computer Vision.

Question: *Can Computer Vision be used locally without an internet connection?*

Answer: We currently do not offer an on-premises or local solution.

Question: *Can Computer Vision be used to read license plates?*

Answer: The Vision API offers good text-detection with OCR, but it is not currently optimized for license plates. We are constantly trying to improve our services and have added OCR for auto license plate recognition to our list of feature requests.

Question: *What types of writing surfaces are supported for handwriting recognition?*

Answer: The technology works with different kinds of surfaces, including whiteboards, white paper, and yellow sticky notes.

Question: *How long does the handwriting recognition operation take?*

Answer: The amount of time that it takes depends on the length of the text. For longer texts, it can take up to several seconds. Therefore, after the Recognize Handwritten Text operation completes, you may need to wait before you can retrieve the results using the Get Handwritten Text Operation Result operation.

Question: *How does the handwriting recognition technology handle text that was inserted using a caret in the middle of a line?*

Answer: Such text is returned as a separate line by the handwriting recognition operation.

Question: *How does the handwriting recognition technology handle crossed-out words or lines?*

Answer: If the words are crossed out with multiple lines to render them unrecognizable, the handwriting recognition operation doesn't pick them up. However, if the words are crossed out using a single line, that crossing is treated as noise, and the words still get picked up by the handwriting recognition operation.

Question: *What text orientations are supported for the handwriting recognition technology?*

Answer: Text oriented at angles of up to around 30 degrees to 40 degrees may get picked up by the handwriting recognition operation.

Computer Vision 86-category taxonomy

7/26/2019 • 2 minutes to read • [Edit Online](#)

abstract_

abstract_net

abstract_nonphoto

abstract_rect

abstract_shape

abstract_texture

animal_

animal_bird

animal_cat

animal_dog

animal_horse

animal_panda

building_

building_arch

building_brickwall

building_church

building_corner

building_doorwindows

building_pillar

building_stair

building_street

dark_

drink_

drink_can

dark_fire

dark_fireworks

sky_object

food_

food_bread

food_fastfood

food_grilled

food_pizza

indoor_

indoor_churchwindow

indoor_court

indoor_doorwindows

indoor_marketstore

indoor_room

indoor_venue

dark_light

others_

outdoor_

outdoor_city

outdoor_field

outdoor_grass

outdoor_house

outdoor_mountain

outdoor_oceanbeach

outdoor_playground

outdoor_railway

outdoor_road

outdoor_sportsfield

outdoor_stonerock

outdoor_street

outdoor_water

outdoor_waterside

people_

people_baby

people_crowd

people_group

people_hand

people_many

people_portrait

people_show

people_tattoo

people_young

plant_

plant_branch

plant_flower

plant_leaves

plant_tree

object_screen

object_sculpture

sky_cloud

sky_sun

people_swimming

outdoor_pool

text_

text_mag

text_map

text_menu

text_sign

trans_bicycle

trans_bus

trans_car

trans_trainstation

Language support for Computer Vision

9/1/2020 • 2 minutes to read • [Edit Online](#)

Some features of Computer Vision support multiple languages; any features not mentioned here only support English.

Optical Character Recognition (OCR)

Computer Vision's OCR APIs support several languages. They do not require you to specify a language code. See [Optical Character Recognition \(OCR\)](#) for more information.

LANGUAGE	LANGUAGE CODE	OCR API	READ V3.0	READ V3.1 PUBLIC PREVIEW
Arabic	ar	✓		
Chinese (Simplified)	zh-Hans	✓		✓
Chinese (Traditional)	zh-Hant	✓		
Czech	cs	✓		
Danish	da	✓		
Dutch	nl	✓	✓	✓
English	en	✓	✓	✓
Finnish	fi	✓		
French	fr	✓	✓	✓
German	de	✓	✓	✓
Greek	el	✓		
Hungarian	hu	✓		
Italian	it	✓	✓	✓
Japanese	ja	✓		
Korean	ko	✓		
Norwegian	nb	✓		
Polish	pl	✓		

LANGUAGE	LANGUAGE CODE	OCR API	READ V3.0	READ V3.1 PUBLIC PREVIEW
Portuguese	pt	✓	✓	✓
Romanian	ro	✓		
Russian	ru	✓		
Serbian (Cyrillic)	sr-Cyrl	✓		
Serbian (Latin)	sr-Latn	✓		
Slovak	sk	✓		
Spanish	es	✓	✓	✓
Swedish	sw	✓		
Turkish	tr	✓		

Image analysis

Some actions of the [Analyze - Image](#) API can return results in other languages, specified with the `language` query parameter. Other actions return results in English regardless of what language is specified, and others throw an exception for unsupported languages. Actions are specified with the `visualFeatures` and `details` query parameters; see the [Overview](#) for a list of all the actions you can do with image analysis.

LANG UAGE	LANG UAGE CODE	CATE GORI ES	TAGS	DESC RIPTI ON	ADUL T	BRAN DS	COLO R	FACE S	IMAG ETYP E	OBJE CTS	CELE BRITI ES	LAND MAR KS
Chine se	zh	✓	✓	✓	-	-	-	-	-	✗	✓	✓
Englis h	en	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Japan ese	ja	✓	✓	✓	-	-	-	-	-	✗	✓	✓
Portu gues e	pt	✓	✓	✓	-	-	-	-	-	✗	✓	✓
Spani sh	es	✓	✓	✓	-	-	-	-	-	✗	✓	✓

Next steps

Get started using the Computer Vision features mentioned in this guide.

- [Analyze a local image \(REST\)](#)
- [Extract printed text \(REST\)](#)

Upgrade to Computer Vision v3.0 Read API from v2.0/v2.1

9/1/2020 • 5 minutes to read • [Edit Online](#)

This guide shows how to upgrade your existing Computer Vision v2.0 or v2.1 REST API code to v3.0 Read operations.

Upgrade `Batch Read File` to `Read`

1. Change the API path for `Batch Read File` 2.x as follows:

READ 2.X	READ 3.0
<code>https://{endpoint}/vision/v2.0/read/core/asyncBatchAnalyze</code>	<code>https://{endpoint}/vision/v3.0/read/analyze[?language]</code>

A new optional *language* parameter is available. If you do not know the language of your document, or it may be multilingual, don't include it.

2. Change the API path for `Get Read Results` in 2.x as follows:

READ 2.X	READ 3.0
<code>https://{endpoint}/vision/v2.0/read/operations/{operationId}</code>	<code>https://{endpoint}/vision/v3.0/read/analyzeResults/{operationId}</code>

3. Change the code for checking the json results from `Get Read Operation Result`. When the call to `Get Read Operation Result` is successful, it returns a status string field in the JSON body. The following values from v2.0 have been changed to better align with the other Cognitive Service REST APIs.

READ 2.X	READ 3.0
<code>"NotStarted"</code>	<code>"notStarted"</code>
<code>"Running"</code>	<code>"running"</code>
<code>"Failed"</code>	<code>"failed"</code>
<code>"Succeeded"</code>	<code>"succeeded"</code>

4. Change your code to interpret the final recognition result JSON from `Get Read Operation Result`.

Note the following changes to the json:

- In v2.x, `"Get Read Operation Result"` will return the OCR recognition json when the status is `"Succeeded"`. In v3.0, this field is `"succeeded"`.
- To get the root for page array, change the json hierarchy from `"recognitionResults"` to `"analyzeResult"` / `"readResults"`. The per-page line and words json hierarchy remains unchanged, so no code changes are required.

- The page angle `"clockwiseOrientation"` has been renamed to `"angle"` and the range has been changed from 0 - 360 degrees to -180 to 180 degrees. Depending on your code, you may or may not have to make changes as most math functions can handle either range.
- The v3.0 API also introduces the following improvements you can optionally leverage: -
 - `"createdDateTime"` and `"lastUpdatedDateTime"` are added so you can track the duration of processing. See documentation for more details.
 - `"version"` tells you the version of the API used to generate results
 - A per-word `"confidence"` has been added. This value is calibrated so that a value 0.95 means that there is a 95% chance the recognition is correct. The confidence score can be used to select which text to send to human review.

In 2.X, the output format is as follows:

```
{
  {
    "status": "Succeeded",
    "recognitionResults": [
      {
        "page": 1,
        "language": "en",
        "clockwiseOrientation": 349.59,
        "width": 2661,
        "height": 1901,
        "unit": "pixel",
        "lines": [
          {
            "boundingBox": [
              67,
              646,
              2582,
              713,
              2580,
              876,
              67,
              821
            ],
            "text": "The quick brown fox jumps",
            "words": [
              {
                "boundingBox": [
                  143,
                  650,
                  435,
                  661,
                  436,
                  823,
                  144,
                  824
                ],
                "text": "The",
              },
              // The rest of result is omitted for brevity
            ]
          }
        ]
      }
    ]
  }
}
```

In v3.0, it has been adjusted:

```

{
  {
    "status": "succeeded",
    "createdDateTime": "2020-05-28T05:13:21Z",
    "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
    "analyzeResult": {
      "version": "3.0.0",
      "readResults": [
        {
          "page": 1,
          "language": "en",
          "angle": 0.8551,
          "width": 2661,
          "height": 1901,
          "unit": "pixel",
          "lines": [
            {
              "boundingBox": [
                67,
                646,
                2582,
                713,
                2580,
                876,
                67,
                821
              ],
              "text": "The quick brown fox jumps",
              "words": [
                {
                  "boundingBox": [
                    143,
                    650,
                    435,
                    661,
                    436,
                    823,
                    144,
                    824
                  ],
                  "text": "The",
                  "confidence": 0.958
                },
                // The rest of result is omitted for brevity
              ]
            }
          ]
        }
      ]
    }
  }
}

```

Upgrade from Recognize Text to Read

Recognize Text is a *preview* operation which is being *deprecated in all versions of Computer Vision API*. You must migrate from **Recognize Text** to **Read** (v3.0) or **Batch Read File** (v2.0, v2.1). v3.0 of **Read** includes newer, better models for text recognition and additional features, so it is recommended. To upgrade from **Recognize Text** to **Read**:

1. Change the API path for **Recognize Text** v2.x as follows:

RECOGNIZE TEXT 2.X	READ 3.0
<code>https://{endpoint}/vision/v2.0/recognizeText[?mode]</code>	<code>https://{endpoint}/vision/v3.0/read/analyze[?language]</code>

The *mode* parameter is not supported in **Read**. Both handwritten and printed text will automatically be supported.

A new optional *language* parameter is available in v3.0. If you do not know the language of your document, or it may be multilingual, don't include it.

2. Change the API path for `Get Recognize Text Operation Result` v2.x as follows:

RECOGNIZE TEXT 2.X	READ 3.0
<code>https://{endpoint}/vision/v2.0/textOperations/{operationId}</code>	<code>https://{endpoint}/vision/v3.0/read/analyzeResults/{operationId}</code>

3. Change the code for checking the json results from `Get Recognize Text Operation Result` . When the call to `Get Read Operation Result` is successful, it returns a status string field in the JSON body.

RECOGNIZE TEXT 2.X	READ 3.0
<code>"NotStarted"</code>	<code>"notStarted"</code>
<code>"Running"</code>	<code>"running"</code>
<code>"Failed"</code>	<code>"failed"</code>
<code>"Succeeded"</code>	<code>"succeeded"</code>

4. Change your code to interpret the final recognition result JSON from `Get Recognize Text Operation Result` to support `Get Read Operation Result` .

Note the following changes to the json:

- In v2.x, `"Get Read Operation Result"` will return the OCR recognition json when the status is `"Succeeded"` . In v3.0, this field is `"succeeded"` .
- To get the root for page array, change the json hierarchy from `"recognitionResult"` to `"analyzeResult" / "readResults"` . The per-page line and words json hierarchy remains unchanged, so no code changes are required.
- The v3.0 API also introduces the following improvements you can optionally leverage. See the API reference for more details: `"createdDateTime"` and `"lastUpdatedDateTime"` are added so you can track the duration of processing. See documentation for more details.
 - `"version"` tells you the version of the API used to generate results
 - A per-word `"confidence"` has been added. This value is calibrated so that a value 0.95 means that there is a 95% chance the recognition is correct. The confidence score can be used to select which text to send to human review.
 - `"angle"` general orientation of the text in clockwise direction, measured in degrees between (-180, 180].
 - `"width"` and `"height"` give you the dimensions of your document, and `"unit"` provides the unit of those dimensions (pixels or inches, depending on document type.)
 - `"page"` multipage documents are supported
 - `"language"` the input language of the document (from the optional *language* parameter.)

In 2.X, the output format is as follows:

```

{
  {
    "status": "Succeeded",
    "recognitionResult": [
      {
        "lines": [
          {
            "boundingBox": [
              67,
              646,
              2582,
              713,
              2580,
              876,
              67,
              821
            ],
            "text": "The quick brown fox jumps",
            "words": [
              {
                "boundingBox": [
                  143,
                  650,
                  435,
                  661,
                  436,
                  823,
                  144,
                  824
                ],
                "text": "The",
              },
              // The rest of result is omitted for brevity
            ]
          }
        ]
      }
    ]
  }
}

```

In v3.0, it has been adjusted:

```

{
  {
    "status": "succeeded",
    "createdDateTime": "2020-05-28T05:13:21Z",
    "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
    "analyzeResult": {
      "version": "3.0.0",
      "readResults": [
        {
          "page": 1,
          "angle": 0.8551,
          "width": 2661,
          "height": 1901,
          "unit": "pixel",
          "lines": [
            {
              "boundingBox": [
                67,
                646,
                2582,
                713,
                2580,
                876,
                67,
                821
              ],
              "text": "The quick brown fox jumps",
              "words": [
                {
                  "boundingBox": [
                    143,
                    650,
                    435,
                    661,
                    436,
                    823,
                    144,
                    824
                  ],
                  "text": "The",
                  "confidence": 0.958
                },
                // The rest of result is omitted for brevity
              ]
            }
          ]
        }
      ]
    }
  }
}

```

All other operations

There are no other breaking changes between v2.X and v3.0 of Computer Vision API. You may simply modify the API path to replace `v2.0` with `v3.0`.