Azure Cognitive Services Succinctly

By **Ed Freitas**

Succinctly eBook Series

Syncfusion®

Azure Cognitive Services Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ed Freitas is a consultant on software development related to financial process automation, accounts payable processing, and data extraction.

He loves technology and enjoys playing soccer, running, traveling, life-hacking, learning, and spending time with his family.

You can reach him at https://edfreitas.me.

Acknowledgments

Many thanks to all the people who contributed to this book, including the amazing Syncfusion team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Graham High from Syncfusion, and James McCaffrey from Microsoft Research. Thank you.

This book is dedicated to my father—for everything you did, for everyone you loved—thank you.

Introduction

We are all well aware of today's incredible advances in software engineering, and likely have heard the phrase "software is eating the world."

Most of this can be attributed to the rise of artificial intelligence (AI), which has been traditionally an area of computer engineering reserved for researchers and computer scientists with a PhD in machine learning (ML) or related fields.

Microsoft Azure is a cloud computing service designed and created by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers.

Azure provides over 100 services, categorized as software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS), and supports many different programming languages, tools, and frameworks, including both Microsoft-specific and third-party applications, allowing developers to manage applications in the cloud, on-premise, and at the edge.

Azure Cognitive Services is a comprehensive family of AI services and cognitive APIs that help software developers who are not necessarily AI experts build intelligent applications.

Essentially, Cognitive Services is lowering the barrier for developers to add AI capabilities to their apps—without needing AI or ML expertise.

By using state-of-the-art APIs, Cognitive Services provides developers the ability to see, hear, speak, search, understand, and infuse decision-making capabilities into their apps—which is both compelling and powerful.

Therefore, developers that include Cognitive Services APIs into their apps are able to benefit from a broad and comprehensive portfolio of domain-specific AI capabilities that are available on the market, with a fraction of the cost and effort to implement such features, compared to traditional AI or ML implementations.

This gives developers the ability to confidently build applications with first-class AI services that, to a certain extent, achieve human parity in fields such as computer vision, speech, and language. As these services are part of Azure, they can be easily deployed from anywhere to the cloud or edge with containers.

To any software developer out there, this is an eye-catching and compelling argument. With Cognitive Services, you now have the ability to make your applications smarter.

In this book, we'll explore some of the major services that Azure offers in this space, and write some code that uses them, which should give you a solid foundation to add some AI and ML capabilities to your own apps.

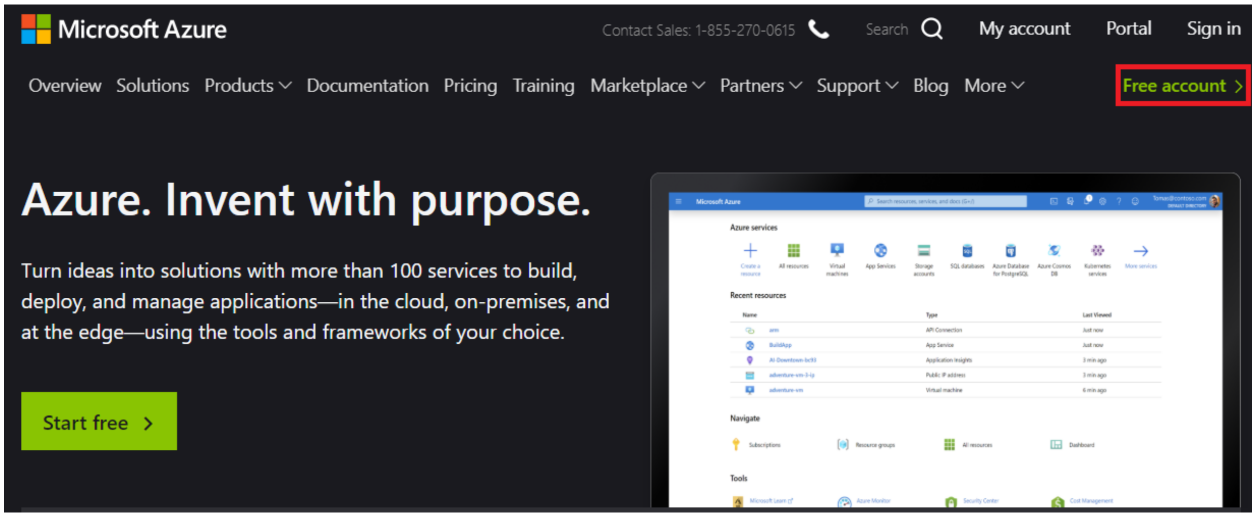
You can download the full source code for each of the Visual Studio projects that will be built throughout this book from this link. Let's dive right in!

Chapter 1 Getting Started

Signing up

Getting started with Azure is quite easy—all you need to do is sign in with your Microsoft account (Outlook.com, Office 365, Live.com, MSN, or Hotmail), or, if you don't have one, sign up for an account.

If you don't have an account, click the **Free account** link on the Azure page to sign up. Please note that Azure is an evolving service, and as such, the website might change. In any case, you should be able to follow along.

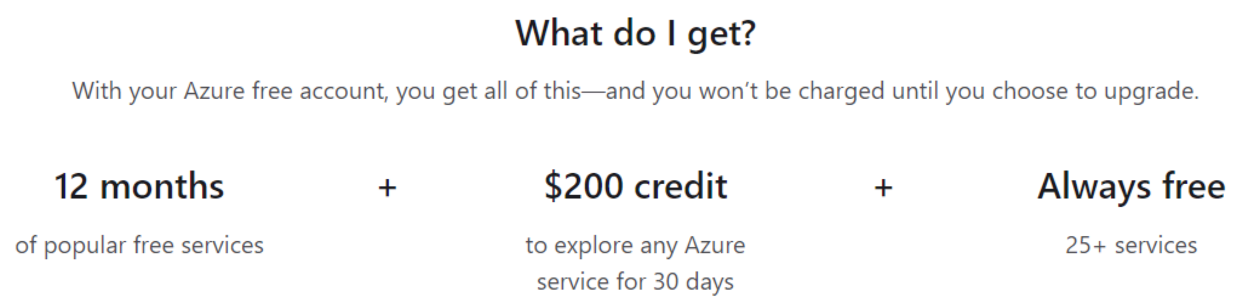


Azure. Invent with purpose.

Turn ideas into solutions with more than 100 services to build, deploy, and manage applications—in the cloud, on-premises, and at the edge—using the tools and frameworks of your choice.

Figure 1-a: Azure Website (Main Page)

Microsoft sometimes gives away free Azure credits to encourage new users to sign up. Even if you are a die-hard Amazon Web Services fan, I suggest you give Azure a try. At the time of writing this book, the following Azure benefits are being offered for new sign-ups.



What do I get?

With your Azure free account, you get all of this—and you won't be charged until you choose to upgrade.

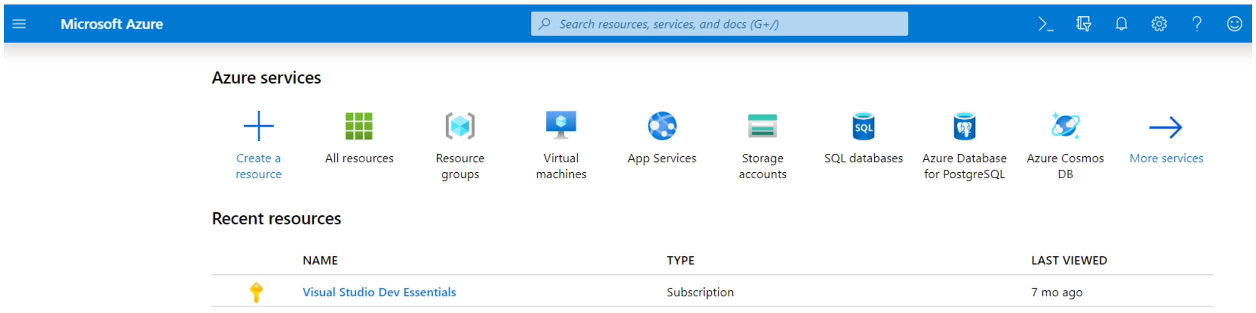
12 months		$200 credit		Always free
of popular free services	+	to explore any Azure service for 30 days	+	25+ services

Figure 1-b: Azure Offers

In my experience, being able to benefit from the services of more than one cloud provider not only broadens your knowledge and exposure to different technologies, but also allows you to check which service works best for your particular use case or application.

Navigating the Azure portal

Once you have signed up for an Azure account, you'll be directed to the main Azure portal page.



NAME	TYPE	LAST VIEWED
Visual Studio Dev Essentials	Subscription	7 mo ago

Figure 1-c: Azure Portal (Main Page)

The most popular and widely used Azure services are listed on the top of the screen. The site is intuitive and easy to follow, and you can access and search for any Azure service by clicking **Create a resource**, by opening the menu sidebar, or by simply using the search toolbar.

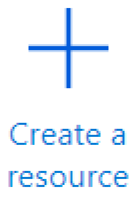


Figure 1-d: Create a Resource Button—Azure Portal (Main Page)

When you click **Create a resource**, you should see a screen that looks similar to the following one.

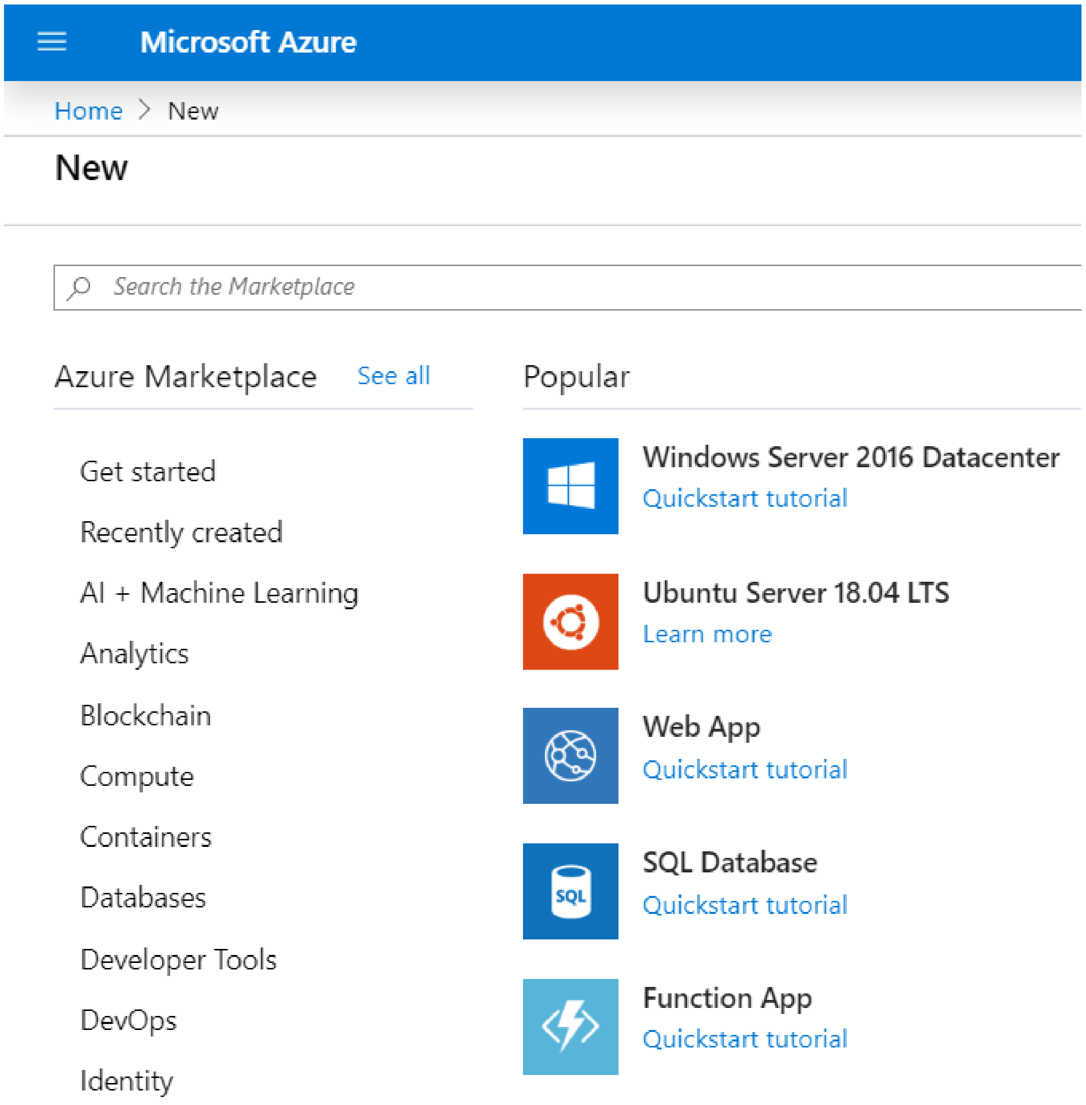


Figure 1-e: Azure Resources Options (via the Create a Resource Button)

When you click the sidebar button, you should see a screen that looks similar to the following one.

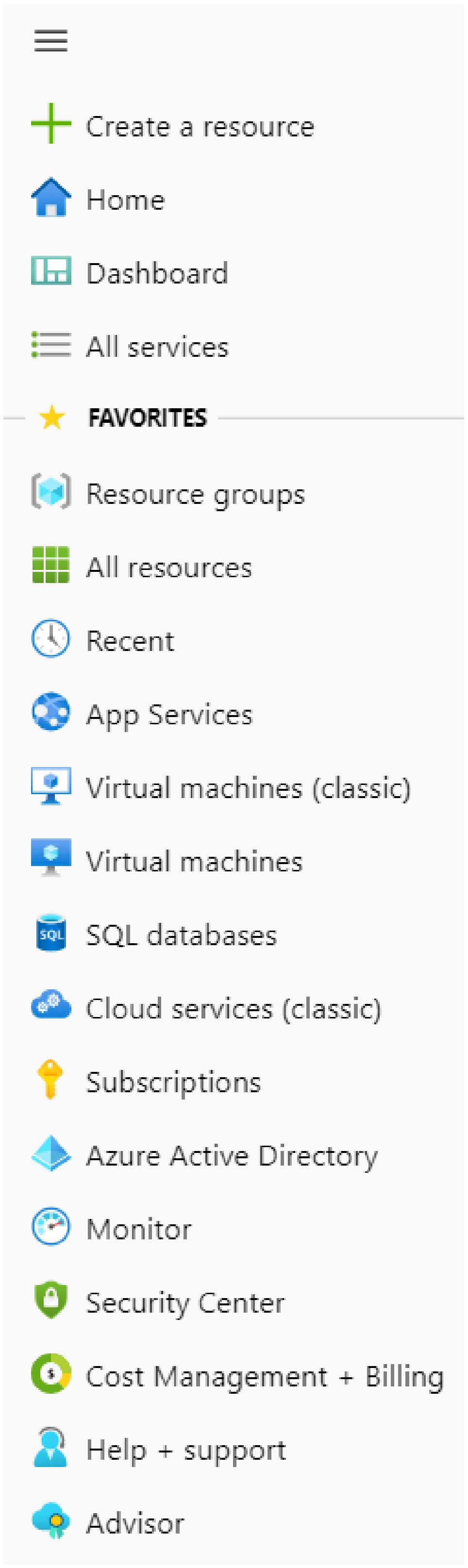


Figure 1-f: Azure Resources Options (via the Sidebar Menu)

My preferred way to access and create a Cognitive Services resource is to enter a specific keyword into the search bar, and then select the correct option, which we can see as follows.

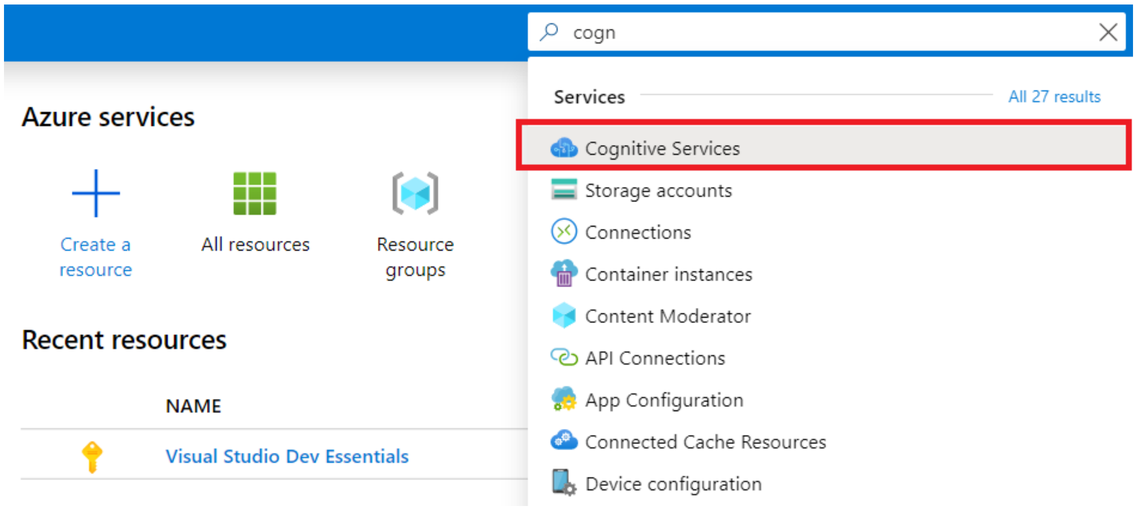


Figure 1-g: Azure Resources Options (via the Search Bar)

Next, click the **Cognitive Services** option—this will display the following screen (or one very similar).

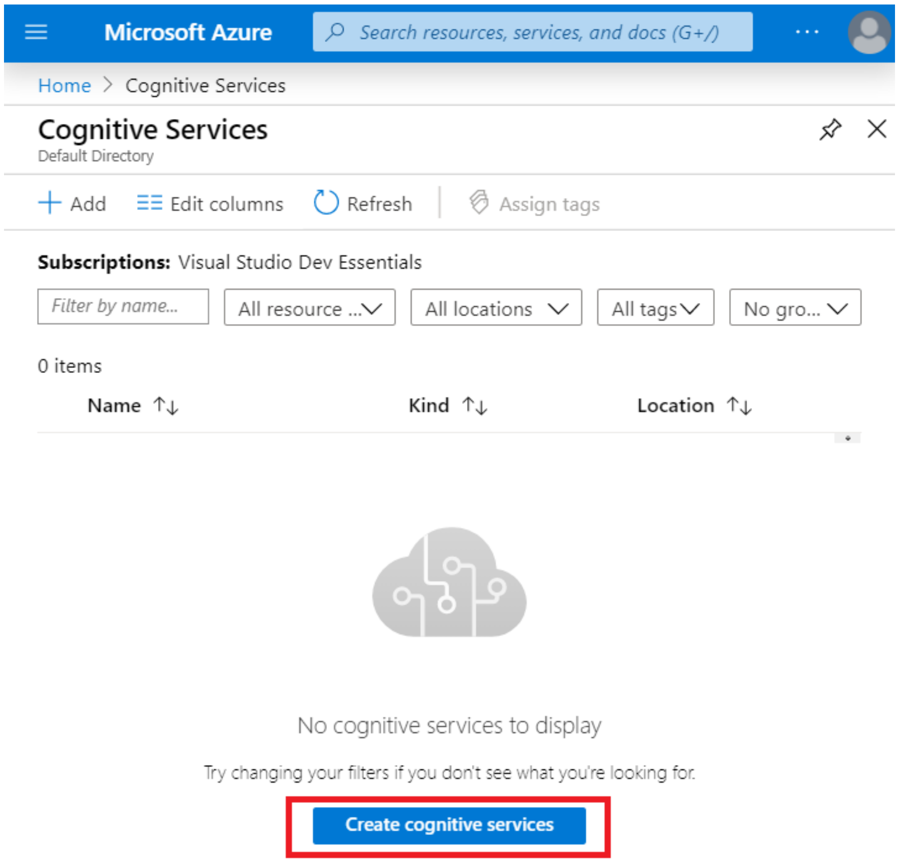
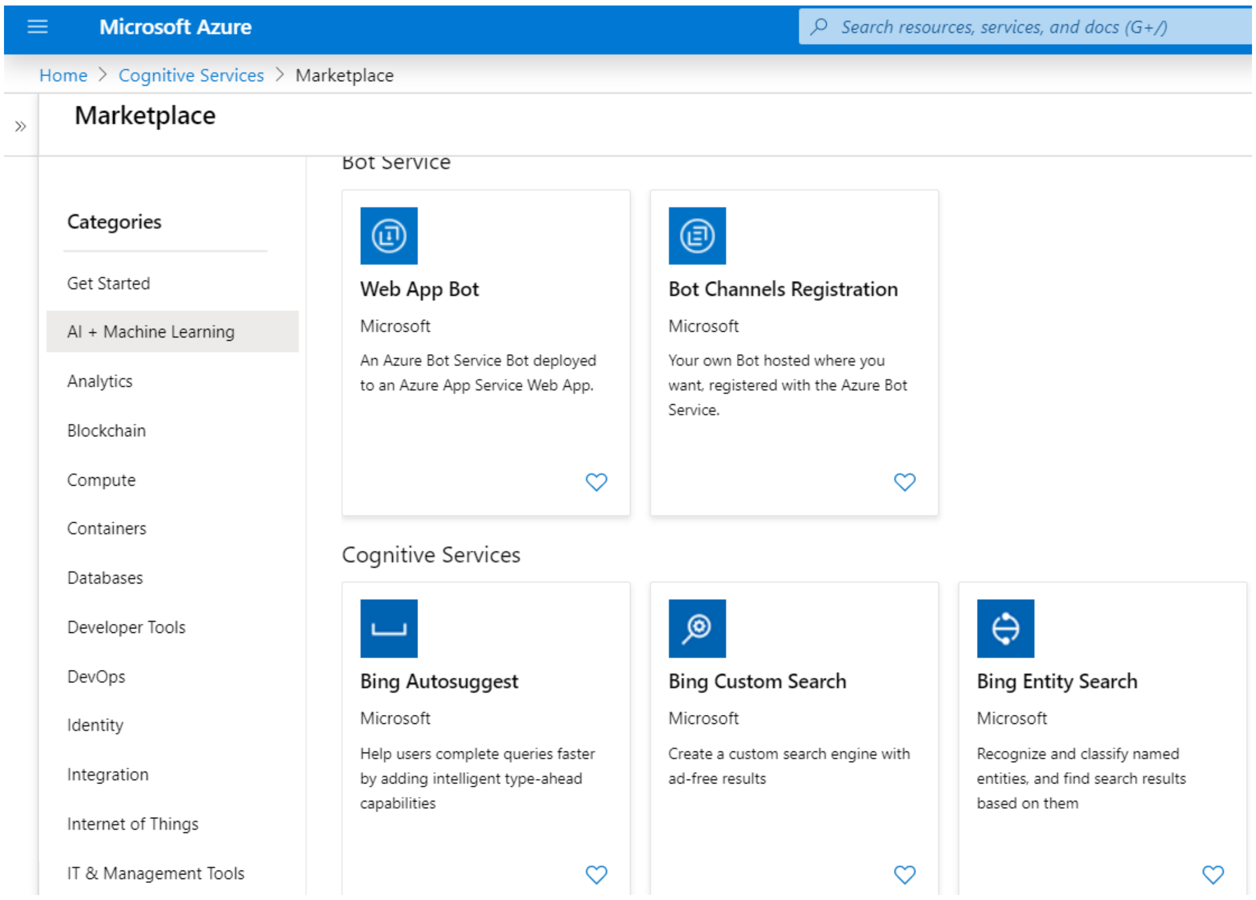

Microsoft Azure
Search resources, services, and docs (G+/)
Home > Cognitive Services
Cognitive Services
Default Directory
Add
Edit columns
Refresh
Assign tags
Subscriptions: Visual Studio Dev Essentials
Filter by name...
All resource ...
All locations
All tags
No gro...
0 items
Name
Kind
Location
No cognitive services to display
Try changing your filters if you don't see what you're looking for.
Create cognitive services

Figure 1-h: Cognitive Services (Main Screen)

Next, click **Create cognitive services**. Once you do that, you'll see the following screen (or one quite similar).



Microsoft Azure

Search resources, services, and docs (G+/)

Home > Cognitive Services > Marketplace

Marketplace

Categories

- Get Started
- AI + Machine Learning
- Analytics
- Blockchain
- Compute
- Containers
- Databases
- Developer Tools
- DevOps
- Identity
- Integration
- Internet of Things
- IT & Management Tools

Bot Service

Web App Bot
Microsoft
An Azure Bot Service Bot deployed to an Azure App Service Web App.

Bot Channels Registration
Microsoft
Your own Bot hosted where you want, registered with the Azure Bot Service.

Cognitive Services

Bing Autosuggest
Microsoft
Help users complete queries faster by adding intelligent type-ahead capabilities

Bing Custom Search
Microsoft
Create a custom search engine with ad-free results

Bing Entity Search
Microsoft
Recognize and classify named entities, and find search results based on them

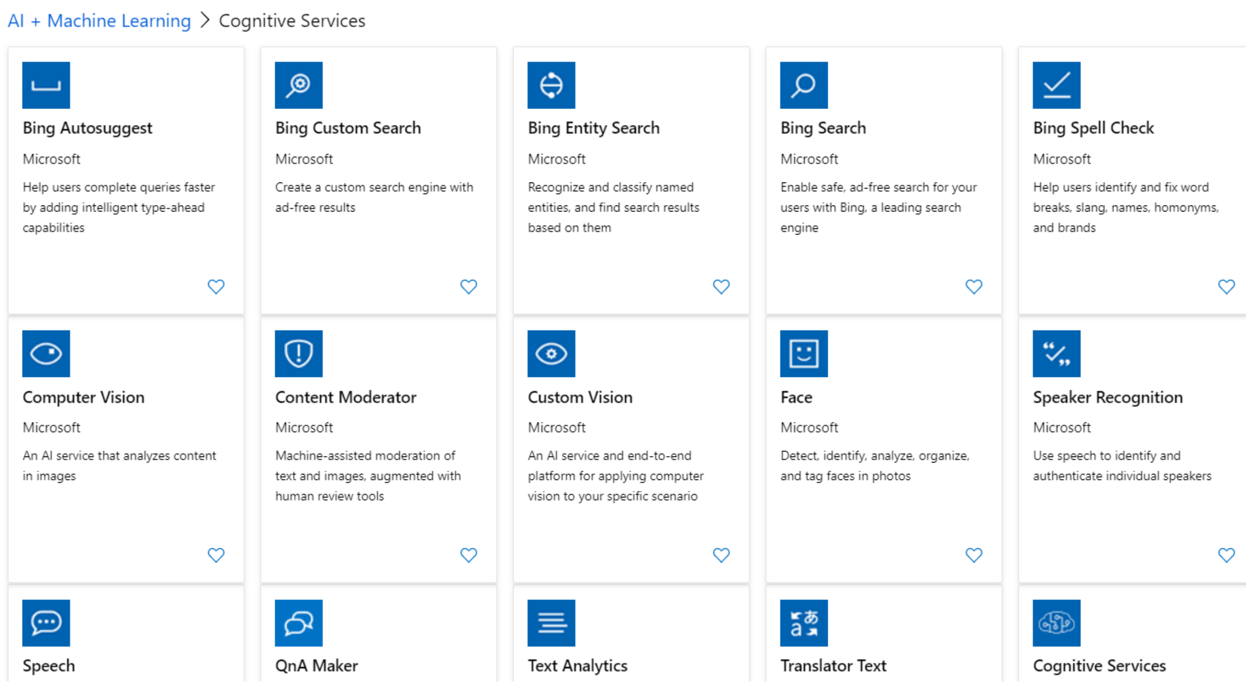
Figure 1-i: Cognitive Services (Various Options)

If you scroll to the right-hand side of this screen, you should find a **See More** link, which looks as follows.

See More

Figure 1-j: See More Link

If you click **See More**, all the available Cognitive Services will be displayed on the screen. At the time of writing this book, the central part of that screen looks as follows.



AI + Machine Learning > Cognitive Services

Bing Autosuggest
Microsoft
Help users complete queries faster by adding intelligent type-ahead capabilities

Bing Custom Search
Microsoft
Create a custom search engine with ad-free results

Bing Entity Search
Microsoft
Recognize and classify named entities, and find search results based on them

Bing Search
Microsoft
Enable safe, ad-free search for your users with Bing, a leading search engine

Bing Spell Check
Microsoft
Help users identify and fix word breaks, slang, names, homonyms, and brands

Computer Vision
Microsoft
An AI service that analyzes content in images

Content Moderator
Microsoft
Machine-assisted moderation of text and images, augmented with human review tools

Custom Vision
Microsoft
An AI service and end-to-end platform for applying computer vision to your specific scenario

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

Speaker Recognition
Microsoft
Use speech to identify and authenticate individual speakers

Speech

QnA Maker

Text Analytics

Translator Text

Cognitive Services

Figure 1-k: List of Cognitive Services

From this list, we can select the service we are interested in using. Now you know how to search and find Cognitive Services within the Azure portal.

Summary

At this stage, you should have signed up for an Azure portal account and explored a bit of the resources that it has to offer.

You probably also looked at the different types of Cognitive Services available, which we will explore in detail in the next chapters.

For the different services, which we will create and use in the next chapters, instead of following an extensive Azure portal navigation process, we will search directly for the service we are interested in using the top search bar.

In the next chapter, we’ll kick off by exploring the decision APIs, which will help us integrate smarter and faster decision-making capabilities into our applications.

Chapter 2 Decision

Quick intro

Cognitive Services APIs can be divided into five distinct categories based on the area of AI they touch. They are decision, language, speech, vision, and web search.

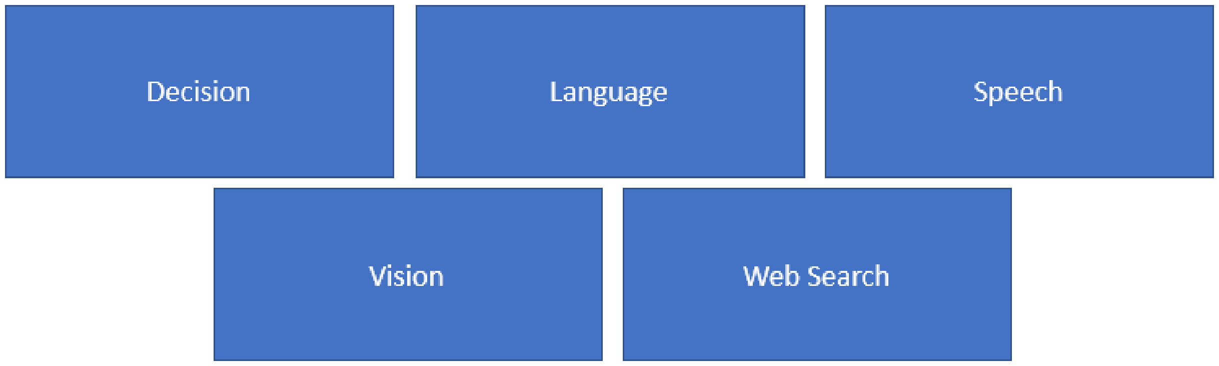


Figure 2-a: Cognitive Services Categories

In this chapter, we will explore the decision category of APIs, particularly the Content Moderator. As the name implies, decision APIs allow developers to infuse logic into their apps that give these applications the ability to make faster and smarter decisions.

At the time of writing this book, the following Cognitive Services decision APIs are available.

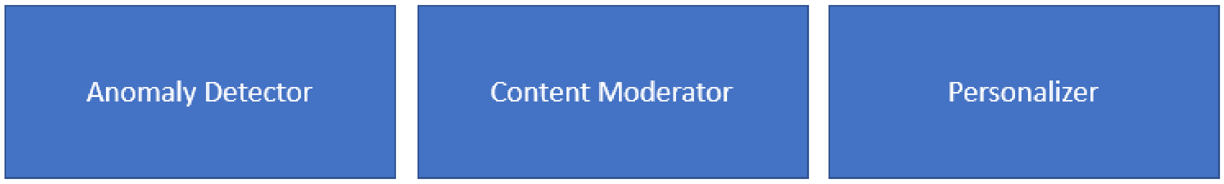


Figure 2-b: Cognitive Services Decision APIs

The Anomaly Detector API scans data for patterns that can be used to detect anomalies within data. It uses an inference engine that analyzes data from a time-series perspective, and is able to use the right algorithm to help highlight potential incidents, fraud, and significant data changes before they actually occur.

We'll be focusing on the Content Moderator set of APIs in this chapter. These give developers the ability to detect potential offensive and unwanted text and images, including profanity and undesirable text, as well as adult images and videos.

The Personalizer API gives developers the ability to deliver a personalized and relevant experience for each user, which is achieved by using an automatic optimization model based on reinforcement learning.

Take a step back and think about this for a moment: personalization is what made Amazon the retail powerhouse and company it is today, by giving each user a tailored buying and checkout experience, based on the user's tastes. This gave Amazon the edge in retail.

That same level of fine-grained personalization is also available to you as a developer via the Personalizer API, which you can use to infuse personalization into your apps. There is an equivalent API from Amazon Web Services called Personalize.

Content Moderator APIs

Content Moderator is a set of APIs that is responsible for inspecting text, image, and video content for material that is potentially offensive, risky, or deemed undesirable.

To better understand how these Content Moderator APIs and their respective features work together, let’s look at the following diagram, which provides an overview.

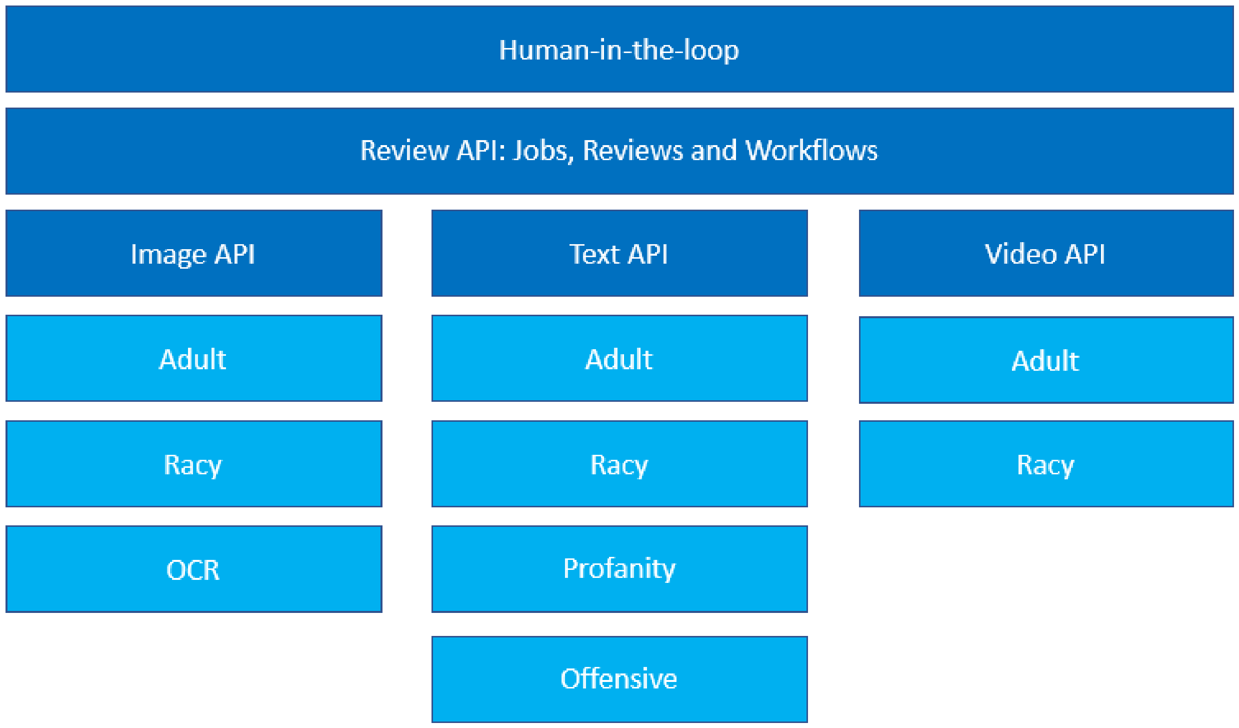


Figure 2-c: Content Moderator APIs and Features

When any of these APIs find such material, it applies appropriate labels to the content, and your app can decide what to do with it.

The application can then handle the flagged content in order to comply with specific regulations or maintain the desired environment for users.

Once flagged content has been identified, it's also possible to include human verification in what is known as *human-in-the-loop*, which is done with a review tool.

The main three Content Moderator APIs are Image, Text, and Video:

- The **Image API** is able to detect adult and racy content within images and to perform optical character recognition.
- The **Text API** is able to detect profanity and adult, racy, and offensive text content.
- The **Video API** is able to detect adult and racy content within videos.

These APIs constitute the core of what the Content Moderator is able to provide as a service. The output of what these three APIs return can be combined with content workflow, review, and approval processes that might involve a human-in-the-loop, or more.

Now that we've explored a bit of the theory behind Content Moderator APIs, let's create an instance of the service to start working with it.

Creating a Content Moderator instance

To create a Content Moderator instance, let's go to the Azure portal, and in the search bar, type **moderator**.

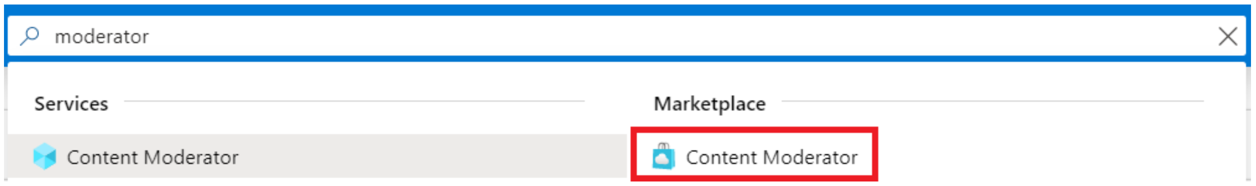
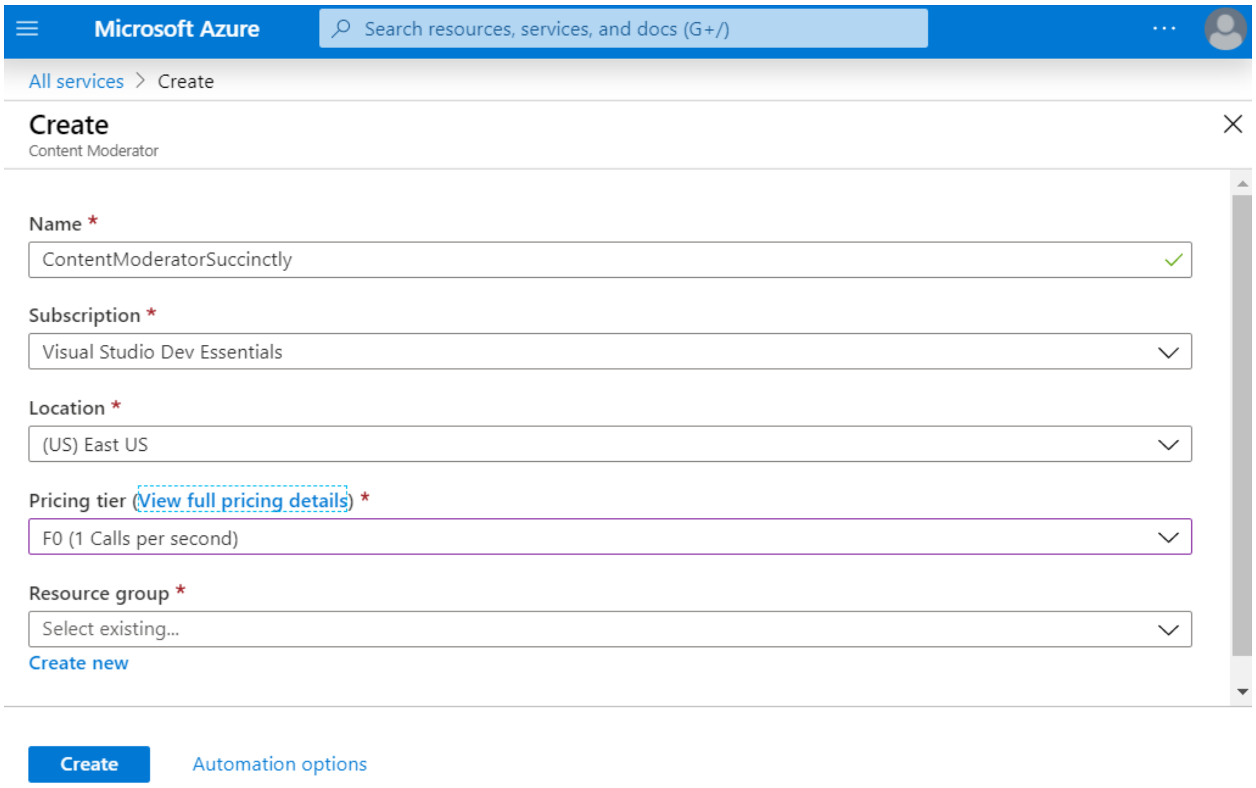


Figure 2-d: Searching for Content Moderator (Azure Portal)

Then, select the **Content Moderator** option from the list of **Marketplace** results, which will display the following screen.



Name *
ContentModeratorSuccinctly

Subscription *
Visual Studio Dev Essentials

Location *
(US) East US

Pricing tier (View full pricing details) *
F0 (1 Calls per second)

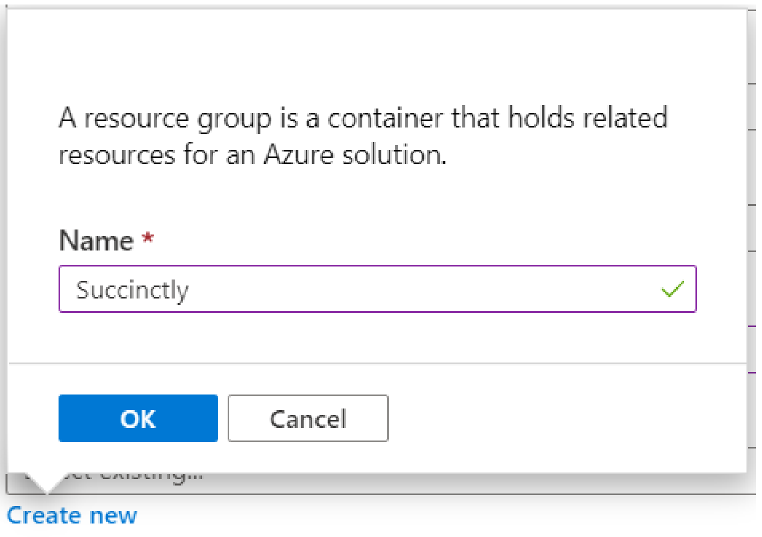
Resource group *
Select existing...

Figure 2-e: Create Content Moderator (Azure Portal)

You'll need to specify a **Name** and **Subscription**, select a **Location**, and indicate the **Pricing tier**—in my case, I've chosen the **F0** tier.

The **F0** pricing tier includes one call per second to the service. This is the free tier option, which is more than enough for experimenting with the service, so I would recommend you choose this one to avoid incurring unnecessary costs. You can find more details about the pricing of this service here.

If you have an existing **Resource group**, you can select it from the list, and if you haven't created one yet, which is most likely the case, then click **Create new**, which is what I'm going to do. You'll see the following dialog box.


A resource group is a container that holds related resources for an Azure solution.

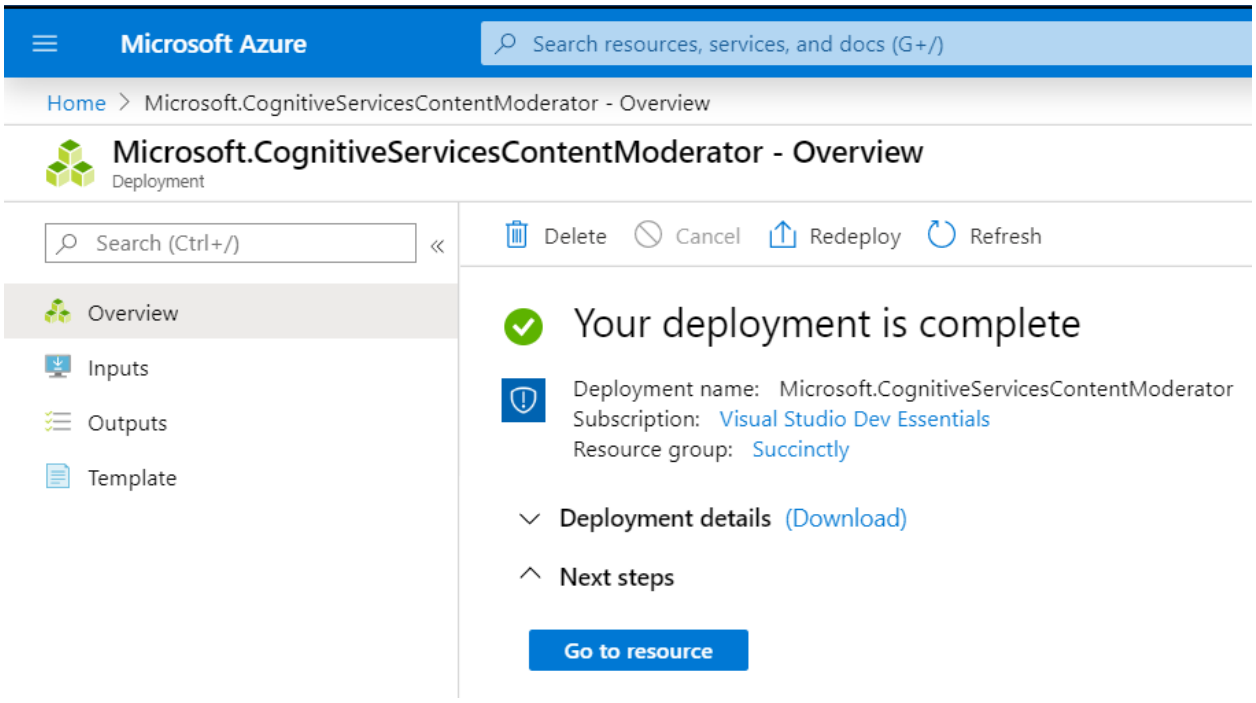
Name *

Succinctly

Figure 2-f: Create New Resource Dialog (Azure Portal)

Enter the required **Name** for the resource and click **OK**. I've named the resource **Succinctly**, as I'll use it for other Cognitive Services that we'll be exploring throughout this book. I recommend that you do the same, so it's easier to follow along.

Once that's done, click **Create**. This will create the Content Moderator service instance, and you will see a screen similar to the following one.


Microsoft.CognitiveServicesContentModerator - Overview
Deployment

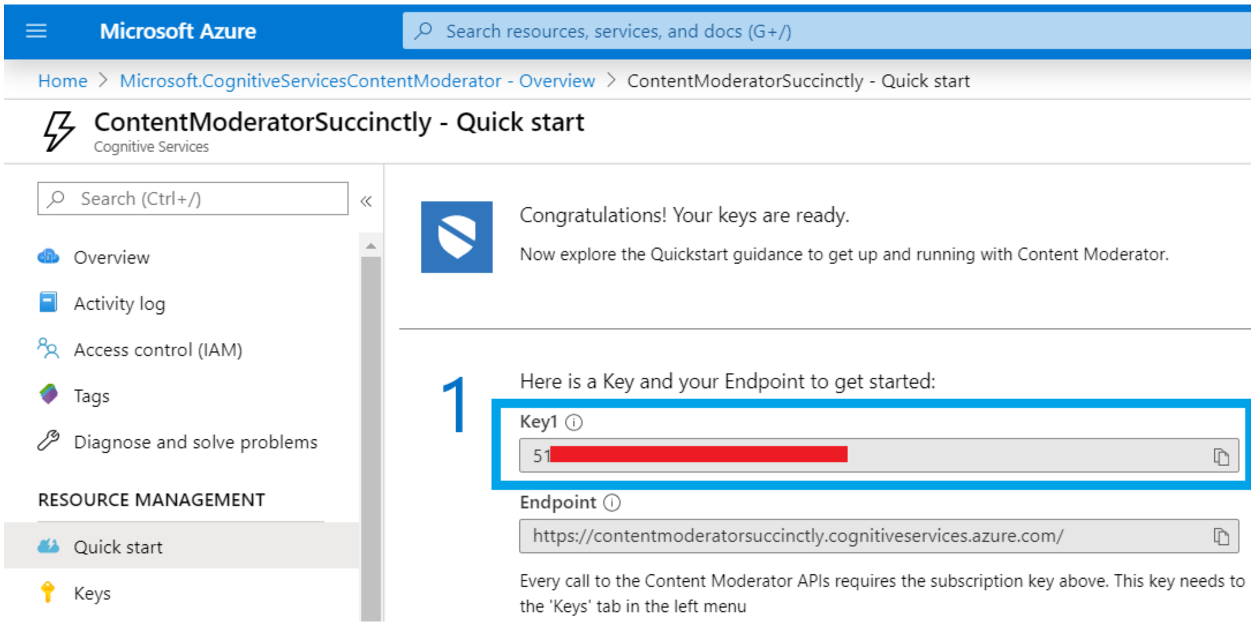
Your deployment is complete

Deployment name: Microsoft.CognitiveServicesContentModerator
Subscription: Visual Studio Dev Essentials
Resource group: Succinctly

Microsoft Azure
Search resources, services, and docs (G+/)
Home > Microsoft.CognitiveServicesContentModerator - Overview
Search (Ctrl+/)
Overview
Inputs
Outputs
Template
Delete
Cancel
Redeploy
Refresh
Deployment details (Download)
Next steps
Go to resource

Figure 2-g: Content Moderator Deployment Complete (Azure Portal)

With the instance created, we are now ready to start using the service. To do that, click **Go to resource** so we can get the key, which can be seen as follows.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Microsoft.CognitiveServicesContentModerator - Overview > ContentModeratorSuccinctly - Quick start
ContentModeratorSuccinctly - Quick start
Cognitive Services
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
RESOURCE MANAGEMENT
Quick start
Keys

Congratulations! Your keys are ready.

Now explore the Quickstart guidance to get up and running with Content Moderator.

1

Here is a Key and your Endpoint to get started:

Key1

51[redacted]

Endpoint

https://contentmoderatorsuccinctly.cognitiveservices.azure.com/

Every call to the Content Moderator APIs requires the subscription key above. This key needs to the 'Keys' tab in the left menu

Figure 2-h: Content Moderator—Quick Start (Azure Portal)

Something to notice is that I have not highlighted the **Endpoint** field, but just the **Key** field. The reason is that that the endpoint stated in the **Quick start** screen is the generic one, but depending on which Content Moderator API we use, the actual endpoint might vary.

To know which endpoint to use, it's best to refer to the Content Moderator API reference and check which API you want to use.

In our first example, we'll use the Image Moderation API and the API to moderate text. The endpoints for both APIs are almost identical, so we are now ready to start writing some code.

Toolset

Throughout this book, we'll be writing code with the Visual Studio 2019 Community Edition, using the C# programming language. Go ahead and download Visual Studio 2019 Community Edition and install it if you haven't yet. By using the default installation options, you'll be fine to proceed with the samples that will be presented throughout this book.

With Visual Studio 2019 open, choose the **Console App (.NET Framework)** option when creating a new project, which will be sufficient for our task.

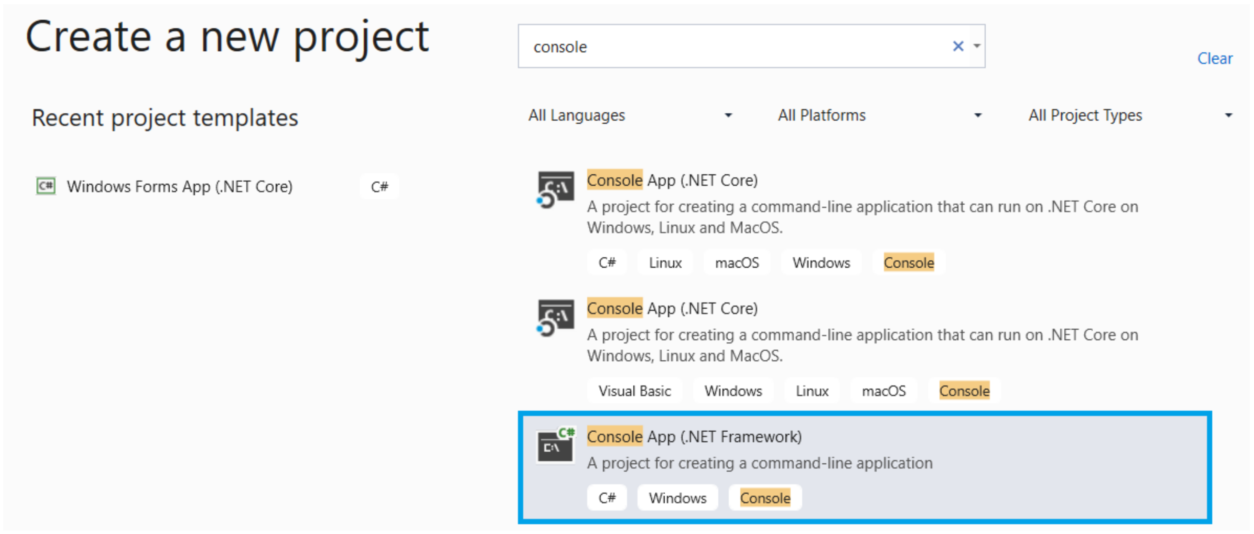

Create a new project
console
Clear
Recent project templates
Windows Forms App (.NET Core)
C#
All Languages
All Platforms
All Project Types
Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C#
Linux
macOS
Windows
Console
Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Visual Basic
Windows
Linux
macOS
Console
Console App (.NET Framework)
A project for creating a command-line application
C#
Windows
Console

Figure 2-i: Creating a New Project—Quick Start (Azure Portal)

Accessing the API

This is the [image]() we'll be initially moderating from [Pixabay](), which is a website that provides royalty-free images.

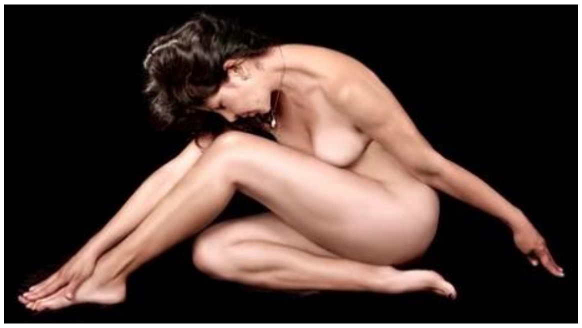


Figure 2-j: Image to Moderate (Courtesy of Pixabay)

Beyond this image, we are also going to moderate the following text, which contains some interesting information.

Code Listing 2-a: Text to Moderate

```
This is a crap test message from abcdef@abcd.com, 28128166778,
255.255.255.255,
```

The code in Listing 2-b is responsible for sending this image and text to the Content Moderator APIs and moderating them.

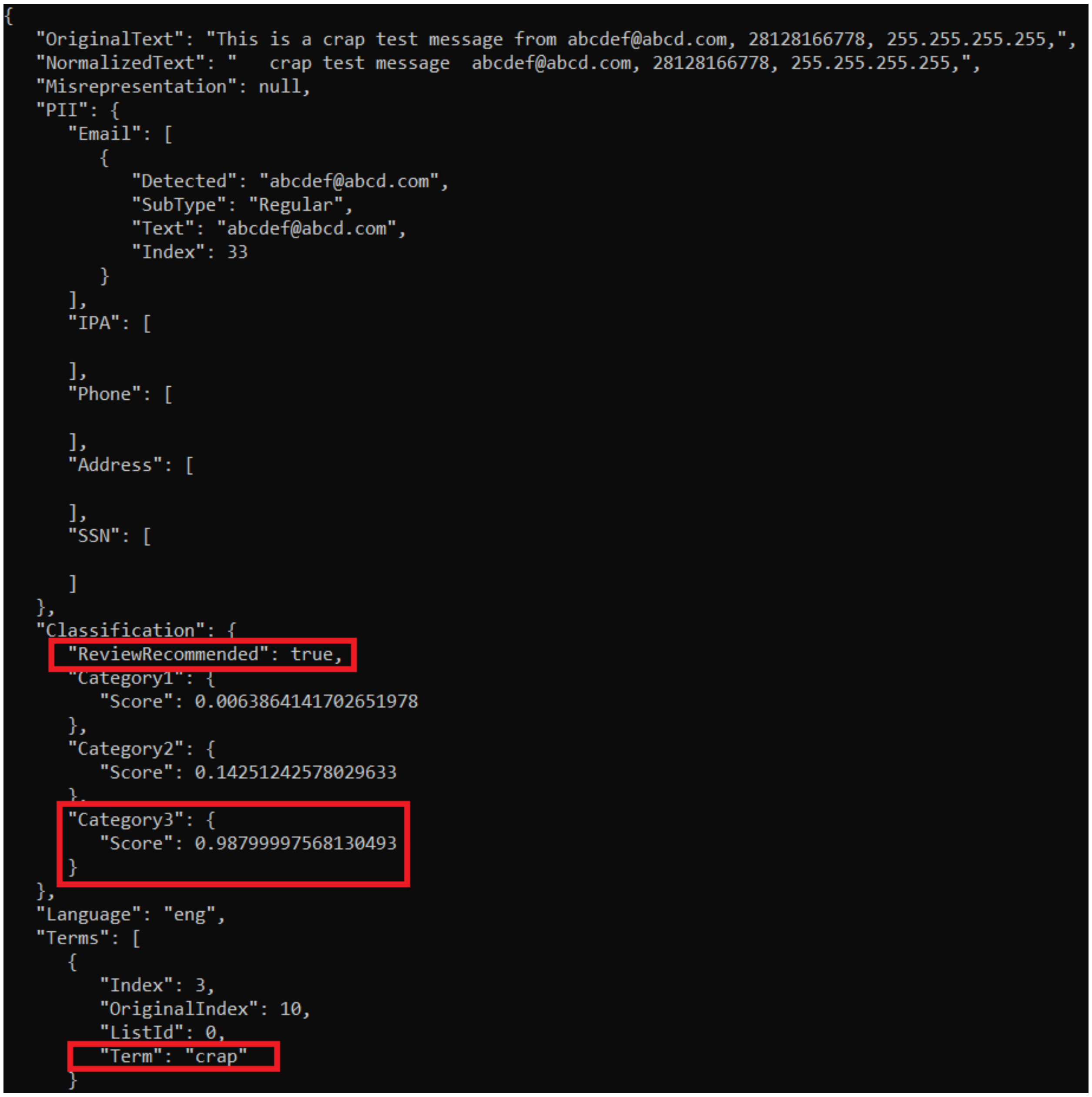
To better understand it, let's explore the complete code first, and then break it into smaller chunks.

Code Listing 2-b: Full API Access Code—Image and Text Moderation (Program.cs)

```
using System;
using System.Net.Http.Headers;
using System.Text;
using System.Net.Http;
using System.Web;
using System.Collections.Specialized;
using System.IO;
using System.Threading.Tasks;

namespace HttpClientDemo
{
    static class Program
    {
        // Azure Content Moderator Endpoint
        private const string cEndpoint =
        "https://eastus.api.cognitive.microsoft.com/contentmoderator/";
        private const string cModerate = "moderate/v1.0/";
        private const string cOcpApimSubscriptionKey =
        "Ocp-Apim-Subscription-Key";
        private const string cSubscriptionKey =
        "<< here goes your key >>"; // Change this!!

        // Image API
        private const string cImageApi = "ProcessImage/";
        // Text API
        private const string cTextApi = "ProcessText/";

        private const string cPath =
        @"C:\Test";
        // Change this path!!

        private static string cStrImage1 = $@"{cPath}\pic.jpg";
        private static string cStrText1 = $@"{cPath}\test.txt";

        static void Main()
        {
            ProcessRequest(cStrImage1, "image/jpeg",
              QryStrEvaluateImage(false));
            ProcessRequest(cStrText1, "text/plain",
              QryStrScreenText(false, true, "", true, ""));
            Console.ReadLine();
        }

        public static void ProcessRequest(string image,
          string contentType, string uri)
        {
            Task.Run(async () => {
                string res = await MakeRequest(image, contentType, uri);

                Console.WriteLine("\nResponse:\n");
                Console.WriteLine(JsonPrettyPrint(res));
            });
        }

        public static async Task<string> MakeRequest(string image,
          string contentType, string uri)
        {
            string contentString = string.Empty;
            HttpClient client = new HttpClient();
            client.DefaultRequestHeaders.Add(
              cOcpApimSubscriptionKey, cSubscriptionKey);

            HttpResponseMessage response = null;

            if (File.Exists(image) && uri != string.Empty &&
                contentType != string.Empty)
            {
                // This is important
                byte[] byteData = GetAsByteArray(image);
                using (var content = new ByteArrayContent(byteData))
                {
                    content.Headers.ContentType = new
                      MediaTypeHeaderValue(contentType);
                    response = await client.PostAsync(uri, content);
                    // This is important
                    contentString = await
                      response.Content.ReadAsStringAsync();
                }
            }

            return contentString;
        }

        public static byte[] GetAsByteArray(string filePath)
        {
            FileStream fileStream = new FileStream(filePath,
              FileMode.Open, FileAccess.Read);
            BinaryReader binaryReader = new BinaryReader(fileStream);
            return binaryReader.ReadBytes((int)fileStream.Length);
        }

        // Specific to the Image API
        public static string QryStrEvaluateImage(bool cacheImage)
        {
            NameValueCollection queryString =
              HttpUtility.ParseQueryString(string.Empty);
            queryString["CacheImage"] = cacheImage.ToString();

            return cEndpoint + cModerate + cImageApi +
              "Evaluate?" + queryString.ToString().ToLower();
        }

        // Specific to the Text API
        public static string QryStrScreenText(bool autoCorrect, bool pii,
          string listId, bool classify, string language)
        {
            NameValueCollection queryString =
              HttpUtility.ParseQueryString(string.Empty);

            queryString["autocorrect"] =
              autoCorrect.ToString().ToLower();
            queryString["PII"] = pii.ToString().ToLower();

            if (listId != string.Empty)
              queryString["listId"] = listId;
            queryString["classify"] = classify.ToString().ToLower();

            if (language != string.Empty)
              queryString["language"] = language;

            return cEndpoint + cModerate + cTextApi +
              "Screen?" + queryString.ToString();
        }

        public static string JsonPrettyPrint(string json)
        {
            if (string.IsNullOrEmpty(json))
                return string.Empty;

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

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

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

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

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

Checking the results

If we run this code, we should see the following output for the text file that was sent to the API for moderation.

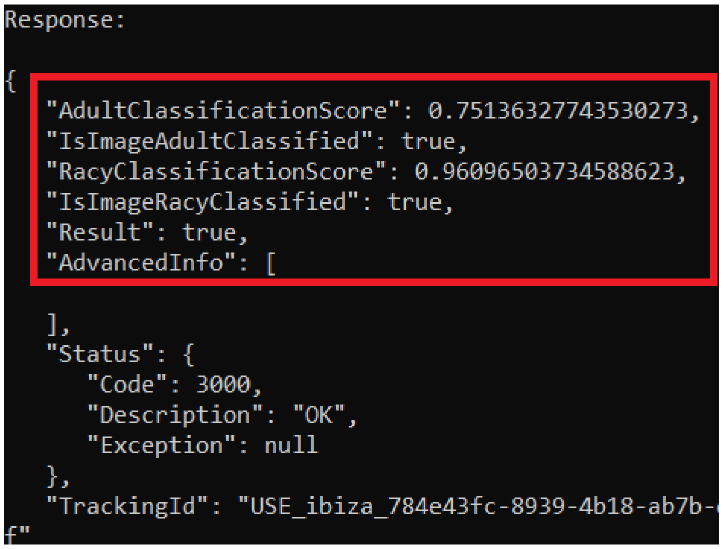


```
{
   "OriginalText": "This is a crap test message from abcdef@abcd.com, 28128166778, 255.255.255.255,",
   "NormalizedText": "   crap test message  abcdef@abcd.com, 28128166778, 255.255.255.255,",
   "Misrepresentation": null,
   "PII": {
      "Email": [
         {
            "Detected": "abcdef@abcd.com",
            "SubType": "Regular",
            "Text": "abcdef@abcd.com",
            "Index": 33
         }
      ],
      "IPA": [

      ],
      "Phone": [

      ],
      "Address": [

      ],
      "SSN": [

      ]
   },
   "Classification": {
      "ReviewRecommended": true,
      "Category1": {
         "Score": 0.0063864141702651978
      },
      "Category2": {
         "Score": 0.14251242578029633
      },
      "Category3": {
         "Score": 0.98799997568130493
      }
   },
   "Language": "eng",
   "Terms": [
      {
         "Index": 3,
         "OriginalIndex": 10,
         "ListId": 0,
         "Term": "crap"
      }
```

Figure 2-k: Console Output Results (Text Moderation)

Notice how the API has recommended that this text be reviewed, by giving it a high score and highlighting the term that contains profanity.

If you scroll down on the console output, you will see the results of the image moderation, which can be seen as follows.



```
Response:

{
  "AdultClassificationScore": 0.7513632774353027З,
  "IsImageAdultClassified": true,
  "RacyClassificationScore": 0.96096503734588623,
  "IsImageRacyClassified": true,
  "Result": true,
  "AdvancedInfo": [

  ],
  "Status": {
    "Code": 3000,
    "Description": "OK",
    "Exception": null
  },
  "TrackingId": "USE_ibiza_784e43fc-8939-4b18-ab7b-
f"
```

Figure 2-l: Console Output Results (Image Moderation)

Notice how this image has been classified as adult and racy content, scoring high on both. Although we wouldn't expect any other results, I'm sure you'll agree with me that this is quite impressive, given the limited code we've written to achieve this.

Main method and endpoint

To understand how this works, let's break the code into smaller chunks and explore what each part does.

Code Listing 2-c: API Access Code—Image and Text Moderation (Program.cs, Part 1)

```
using System;
using System.Net.Http.Headers;
using System.Text;
using System.Net.Http;
using System.Web;
using System.Collections.Specialized;
using System.IO;
using System.Threading.Tasks;

namespace HttpClientDemo
{
    static class Program
    {
        // Azure Content Moderator Endpoint
        private const string cEndpoint =
        "https://eastus.api.cognitive.microsoft.com/contentmoderator/";
        private const string cModerate = "moderate/v1.0/";
        private const string cOcpApimSubscriptionKey =
        "Ocp-Apim-Subscription-Key";
        private const string cSubscriptionKey =
        "<< here goes your key >>"; // Change this!!

        // Image API
        private const string cImageApi = "ProcessImage/";
        // Text API
        private const string cTextApi = "ProcessText/";

        private const string cPath =
        @"C:\Test";
        // Change this path!!

        private static string cStrImage1 = $@"{cPath}\pic.jpg";
        private static string cStrText1 = $@"{cPath}\test.txt";

        static void Main()
        {
            ProcessRequest(cStrImage1, "image/jpeg",
              QryStrEvaluateImage(false));
            ProcessRequest(cStrText1, "text/plain",
              QryStrScreenText(false, true, "", true, ""));
            Console.ReadLine();
        }

        // More code to follow…
    }
}
```

We start off by importing the required references and defining the constants that we need to access the API, such as the endpoint (**`cEndpoint`** combined with **`cModerate`**) and the key (**`cSubscriptionKey`**).

Note: Change the value of `<< here goes your key >>` ***with the value obtained from the Content Moderator subscription key (found on the Azure portal).***

Next, we define the access API methods for the Image (**`cImageApi`**) and Text (**`cTextApi`**) APIs. These are the API calls that are used to moderate the content.

After that, we define the local folder (**`cPath`**) in which the moderated image and text content resides on disk. You may change this value to any other local folder on your machine.

The variables **`cStrImage1`** and **`cStrText1`** refer to the actual names of the image and text files being moderated.

The **`Main`** method invokes the **`ProcessRequest`** method for both the image and text content, but with different query parameters for each: **`QryStrEvaluateImage`** for the image, and **`QryStrScreenText`** for the text.

Requests methods

Now, let’s explore the **`ProcessRequest`** and **`MakeRequest`** methods.

Code Listing 2-d: API Access Code—Image and Text Moderation (Program.cs, Part 2)

```
public static void ProcessRequest(string image, string contentType,
    string uri)
{
    Task.Run(async () => {
        string res = await MakeRequest(image, contentType, uri);

        Console.WriteLine("\nResponse:\n");
        Console.WriteLine(JsonPrettyPrint(res));
    });
}

public static async Task<string> MakeRequest(string image,
    string contentType, string uri)
{
    string contentString = string.Empty;
    HttpClient client = new HttpClient();

    client.DefaultRequestHeaders.Add(
        cOcpApimSubscriptionKey, cSubscriptionKey);

    HttpResponseMessage response = null;

    if (File.Exists(image) && uri != string.Empty &&
        contentType != string.Empty)
    {
        // This is important
        byte[] byteData = GetAsByteArray(image);
        using (var content = new ByteArrayContent(byteData))
        {
            content.Headers.ContentType = new
                MediaTypeHeaderValue(contentType);
            response = await client.PostAsync(uri, content);
            // This is important
            contentString = await response.Content.ReadAsStringAsync();
        }
    }

    return contentString;
}
```

The **`ProcessRequest`** method is very simple—all it does is invoke the **`MakeRequest`** method from within an anonymous **`async`** function that gets called by **`Task.Run`**. This done so the **`MakeRequest`** code doesn’t block execution of the other code in the program.

The **`MakeRequest`** method is the one that invokes the API. It does that by creating an **`HttpClient`** instance, to which **`cSubscriptionKey`** is passed as a header parameter of the HTTP request.

Then, if the actual file to moderate exists on disk (**`File.Exists(image)`**), and the URL (**`uri`**) of the request and **`contentType`** are not empty, the content of the file is sent to the API to be processed.

That content is first deserialized as an array of bytes (**`byte[]`**), which is what **`GetAsByteArray`** returns. That array of bytes is then transformed into a **`ByteArrayContent`** object, which is easier to submit to the API.

Before the request to the API can be submitted, an instance of **`MediaTypeHeaderValue`** must be created by passing **`contentType`** as a parameter.

The actual call to the API is performed by invoking the **`PostAsync`** method from the **`client`** object. This returns an object named **`response`**, which has type **`HttpResponseMessage`**.

The **`Content`** property of the **`response`** object is read as a string, and this is done by invoking the **`ReadAsStringAsync`** method, which returns the result that's written to the console as an output—this is what the **`MakeRequest`** method returns.

GetAsByteArray and query string methods

Now, let's have a look at the other methods that are responsible for enabling the call to the API, which we can see as follows.

Code Listing 2-e: API Access Code—Image and Text Moderation (Program.cs, Part 3)

```
public static byte[] GetAsByteArray(string filePath)
{
    FileStream fileStream = new FileStream(filePath,
        FileMode.Open, FileAccess.Read);
    BinaryReader binaryReader = new BinaryReader(fileStream);
    return binaryReader.ReadBytes((int)fileStream.Length);
}

// Specific to the Image API
public static string QryStrEvaluateImage(bool cacheImage)
{
    NameValueCollection queryString =
        HttpUtility.ParseQueryString(string.Empty);
    queryString["CacheImage"] = cacheImage.ToString();

    return cEndpoint + cModerate + cImageApi +
        "Evaluate?" + queryString.ToString().ToLower();
}

// Specific to the Text API
```

```
public static string QryStrScreenText(bool autoCorrect, bool pii,
    string listId, bool classify, string language)
{
    NameValueCollection queryString =
        HttpUtility.ParseQueryString(string.Empty);

    queryString["autocorrect"] =
        autoCorrect.ToString().ToLower();
    queryString["PII"] = pii.ToString().ToLower();

    if (listId != string.Empty)
        queryString["listId"] = listId;
    queryString["classify"] = classify.ToString().ToLower();

    if (language != string.Empty)
        queryString["language"] = language;

    return cEndpoint + cModerate + cTextApi +
        "Screen?" + queryString.ToString();
}
```

The **`GetAsByteArray`** method reads the file that gets sent to the API as a **`FileStream`** object, which gets passed to a **`BinaryReader`** instance, and the containing bytes are read using the **`ReadBytes`** method.

The **`QryStrEvaluateImage`** method is specifically for working with images, and its purpose is to create a query string that gets passed to the API that is suitable for processing images. It adds the **`CacheImage`** parameter and uses the API's **`Evaluate`** method.

The following is how the full URL to the API call would look, once the **`QryStrEvaluateImage`** method has been invoked:

`https://{endpoint}/contentmoderator/moderate/v1.0/ProcessImage/Evaluate[?CacheImage]`

Essentially, this is what the URL that invokes the API that performs image moderation looks like.

The **`QryStrScreenText`** method, on the other hand, is specifically for working with text. Its purpose is to create a query string that gets passed to the API that is suitable for processing text, which it's able to do by calling the **`Screen`** API method.

It uses several parameters that are not used by the **`QryStrEvaluateImage`** method, such as:

- **`Autocorrect`**: Used for automatically correcting misspelled words, when set to true.
- **`PII`**: Used for detecting personally identifiable information, when set to true.
- **`listId`**: Represents a list of words to be used for matching.
- **`classify`**: Enables text classification.
- **`language`**: Indicates the language to detect within the text; if nothing is specified, it defaults to English.

For text moderation, the URL would look like this once the **`QryStrScreenText`** method has been invoked:

`https://{endpoint}/contentmoderator/moderate/v1.0/ProcessText/Screen[?autocorrect][&PII][&listId][&classify][&language]`

Writing the results

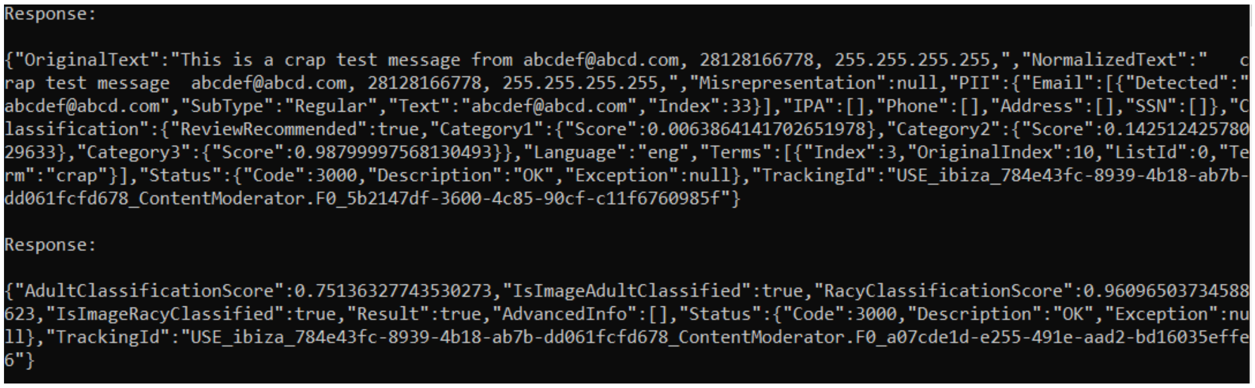
The final part of the code focuses on writing to the console the results returned by the calls to the API. It does this by invoking the **`JsonPrettyPrint`** method.

Code Listing 2-f: API Access Code—Image and Text Moderation (Program.cs, Part 4)

```
public static string JsonPrettyPrint(string json)
{
    if (string.IsNullOrEmpty(json))
        return string.Empty;

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

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

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

        if (quote)
            sb.Append(ch);
        else
        {
            switch (ch)
            {
                case '{':
                case '[':
                    sb.Append(ch);
                    sb.Append(Environment.NewLine);
```

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

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

In order to understand the value that this method adds to our application, it is important to look at how the results would appear if this method is not used. Let's see how the results are displayed when **`JsonPrettyPrint`** is not used.



```
Response:

{"OriginalText":"This is a crap test message from abcdef@abcd.com, 28128166778, 255.255.255.255,","NormalizedText":"   c
rap test message  abcdef@abcd.com, 28128166778, 255.255.255.255,","Misrepresentation":null,"PII":{"Email":[{"Detected":"
abcdef@abcd.com","SubType":"Regular","Text":"abcdef@abcd.com","Index":33}],"IPA":[],"Phone":[],"Address":[],"SSN":[]},"C
lassification":{"ReviewRecommended":true,"Category1":{"Score":0.0063864141702651978},"Category2":{"Score":0.142512425780
29633},"Category3":{"Score":0.98799997568130493}},"Language":"eng","Terms":[{"Index":3,"OriginalIndex":10,"ListId":0,"Te
rm":"crap"}],"Status":{"Code":3000,"Description":"OK","Exception":null},"TrackingId":"USE_ibiza_784e43fc-8939-4b18-ab7b-
dd061fcfd678_ContentModerator.F0_5b2147df-3600-4c85-90cf-c11f6760985f"}

Response:

{"AdultClassificationScore":0.75136327743530273,"IsImageAdultClassified":true,"RacyClassificationScore":0.96096503734588
623,"IsImageRacyClassified":true,"Result":true,"AdvancedInfo":[],"Status":{"Code":3000,"Description":"OK","Exception":nu
ll},"TrackingId":"USE_ibiza_784e43fc-8939-4b18-ab7b-dd061fcfd678_ContentModerator.F0_a07cde1d-e255-491e-aad2-bd16035effe
6"}
```

Figure 2-m: Console Output Results (without JsonPrettyPrint)

Note: If you would like to try this yourself, all you need to do is replace *`Console.WriteLine(JsonPrettyPrint(res));`* ***with*** *`Console.WriteLine(res);`* ***within the*** *`ProcessRequest`* ***method.***

As you have seen, the results are not very readable, which is the main reason why **`JsonPrettyPrint`** is used.

Since the **`JsonPrettyPrint`** method is not really part of the Content Moderator set of APIs and functionality, let's go over it very quickly.

Given that the result returned by the API is in JSON format, which tends to be quite long, we need to have a string that can accommodate the result. We can achieve this by creating an instance of the **`StringBuilder`** class.

We loop over each character within the JSON result, which is done with the following code: **`foreach (char ch in json)`**. Each character is appended to the **`StringBuilder`** class, with enough indentation (**`indentLength`**) and **`offset`** such that when the resultant string is written, it is displayed nicely.

That's it—this is how Content Moderator can be used to allow your application to make better decisions regarding the content it processes.

I'd recommend running this application with other images and text and check the results it returns. You'll have a ton of fun and learn a lot by checking the resultant patterns.

Summary

Content Moderator APIs allow you to infuse your apps with AI-powered decision-making capabilities easily and quickly.

There are two other Cognitive Services that are related to decision-making and are definitely worth exploring: Anomaly Detector and Personalizer. We won't be covering them in this book, but if these are AI topics that excite you, I suggest you explore them a bit.

If you feel inclined to play with either of them, I'm sure you'll have a lot of fun and also be impressed with their ease of use, accuracy, and sophistication.

Chapter 3 Language

Quick intro

Another very exciting aspect of Cognitive Services is its ability to understand and process language.

In this chapter, we'll be focusing on Text Analytics with Azure Cognitive Services, which provides a great way of identifying the language, sentiment, key phrases, and entities within text.

In my book *Skype Bots Succinctly*, we explore in-depth two other fundamental language services within Azure Cognitive Services: Language Understanding (also known as LUIS) and QnA Maker. If you would like to explore both services, feel free to check out this book—it will give you a good understanding of how both services work, and what you can achieve with them.

Without further ado, let's jump right into the Text Analytics service with Azure.

Removing unused resources

Azure is an amazing cloud platform, and even if you are using free resources, it's always a good practice to do housekeeping and remove resources that are no longer being used. If you are done experimenting with and using Content Moderator, it's a good idea to delete the resource—this is a practice I usually follow.

To do that, go to the Azure portal dashboard and click **All resources**, which you can see as follows.

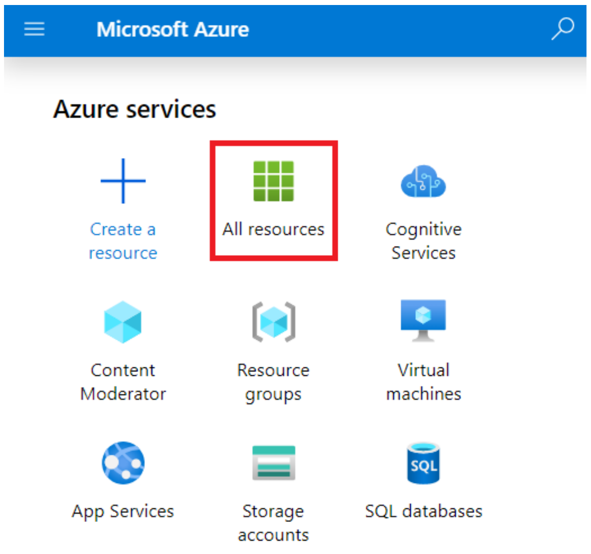
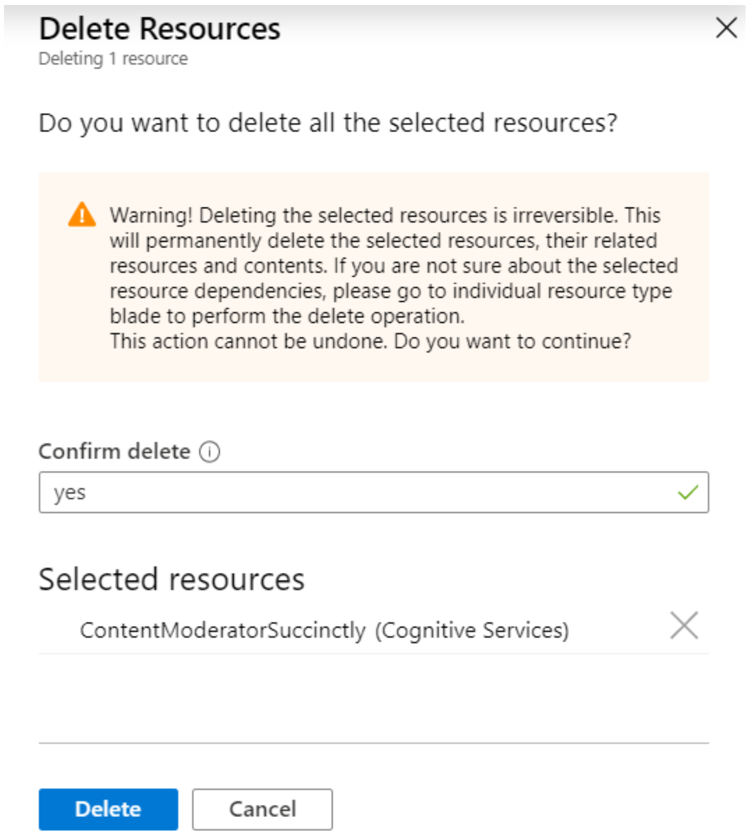


Figure 3-a: Azure Portal Dashboard

Clicking the **All resources** icon will bring you to the **All resources** screen, which will list all your active Azure resources. Select the resource you want to remove, and then click **Delete**. This will remove the resource from your Azure account. Before you can remove the resource, Azure will ask you to confirm this action, as shown in the following figure.



Delete Resources

Deleting 1 resource

Do you want to delete all the selected resources?

Warning! Deleting the selected resources is irreversible. This will permanently delete the selected resources, their related resources and contents. If you are not sure about the selected resource dependencies, please go to individual resource type blade to perform the delete operation.
This action cannot be undone. Do you want to continue?

Confirm delete

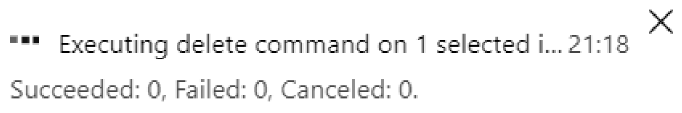
yes

Selected resources

ContentModeratorSuccinctly (Cognitive Services)

Figure 3-b: Delete Resources Confirmation (Azure Portal)

To confirm, you need to explicitly enter **yes** inside the **Confirm delete** text box. Once you've done that, the **Delete** button will become available, which you can then click to perform the operation. The execution of the process usually takes a few seconds. Depending on the type of resource, you should see a message similar to the following one.

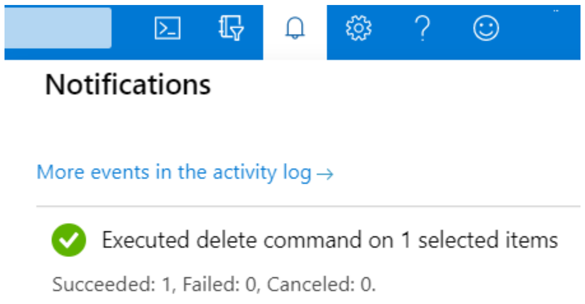


Executing delete command on 1 selected i... 21:18

Succeeded: 0, Failed: 0, Canceled: 0.

Figure 3-c: Deleting a Resource (Azure Portal)

Once the process has ended and the resource has been removed, a notification will appear under the Notifications section of the Azure portal, which we can see as follows.



Executed delete command on 1 selected items

Succeeded: 1, Failed: 0, Canceled: 0.

Figure 3-d: Resource Deleted—Confirmation (Azure Portal)

You might have to manually refresh the **All resources** screen section (referred to in Azure as a *blade*) because sometimes it doesn't automatically refresh after a resource has been removed. You can do this by clicking **Refresh**.

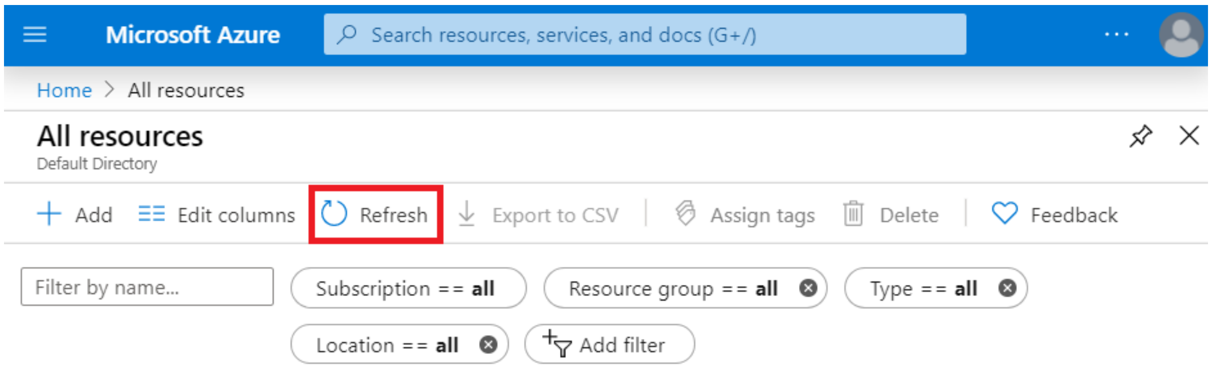
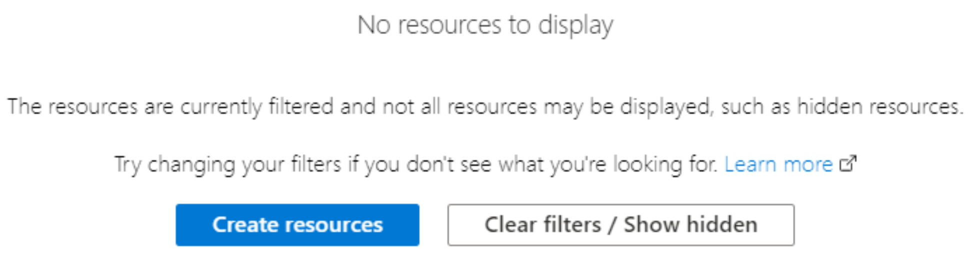


Figure 3-e: All Resources Blade—Upper Part of the Screen (Azure Portal)

With unnecessary resources removed, we can now focus on creating a new one, which is what we'll do next.

Adding Text Analytics

On the **All resources** screen, scroll to the bottom, and there you will find the **Create resources** button. This provides an easy way to create new Azure resources.


No resources to display

The resources are currently filtered and not all resources may be displayed, such as hidden resources.

Try changing your filters if you don't see what you're looking for. Learn more

Figure 3-f: All Resources Blade—Bottom Part of the Screen (Azure Portal)

Click **Create resources**, which will take us to the following screen.

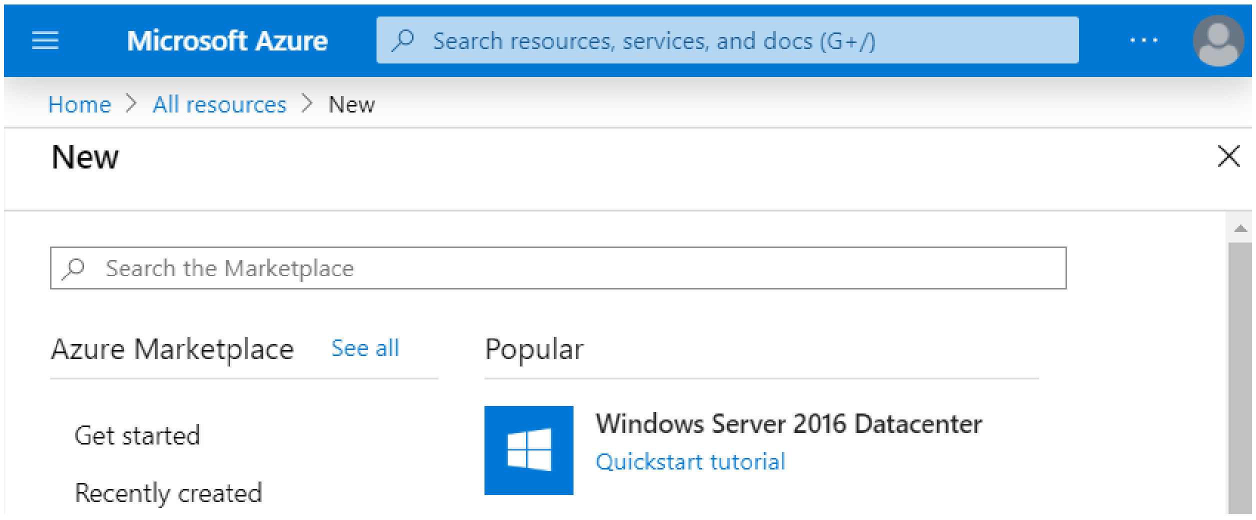


Figure 3-g: New Resources Blade (Azure Portal)

In the **Search the Marketplace** search bar, type the search term **Text Analytics** and choose this option from the list. This will take you to the following screen.

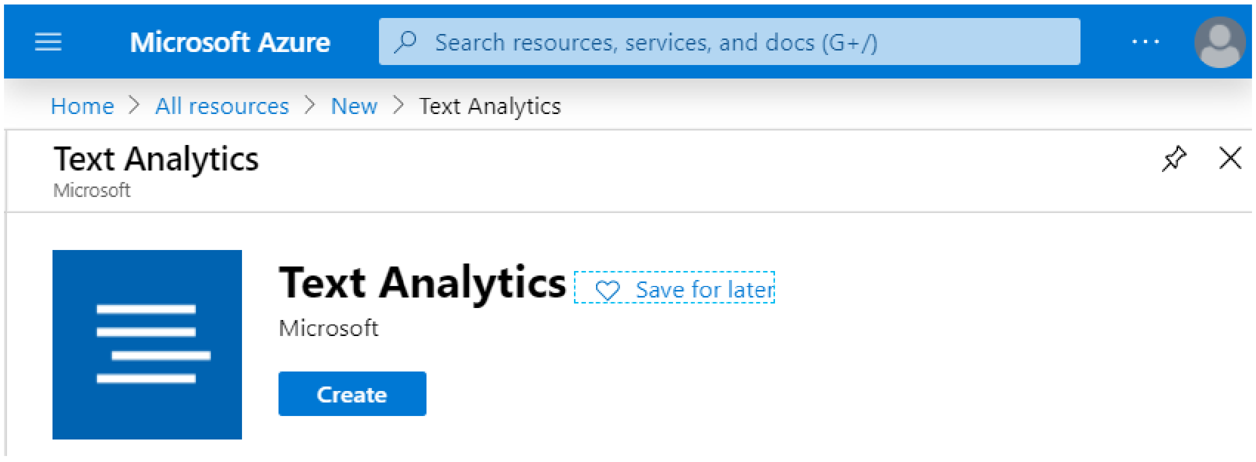
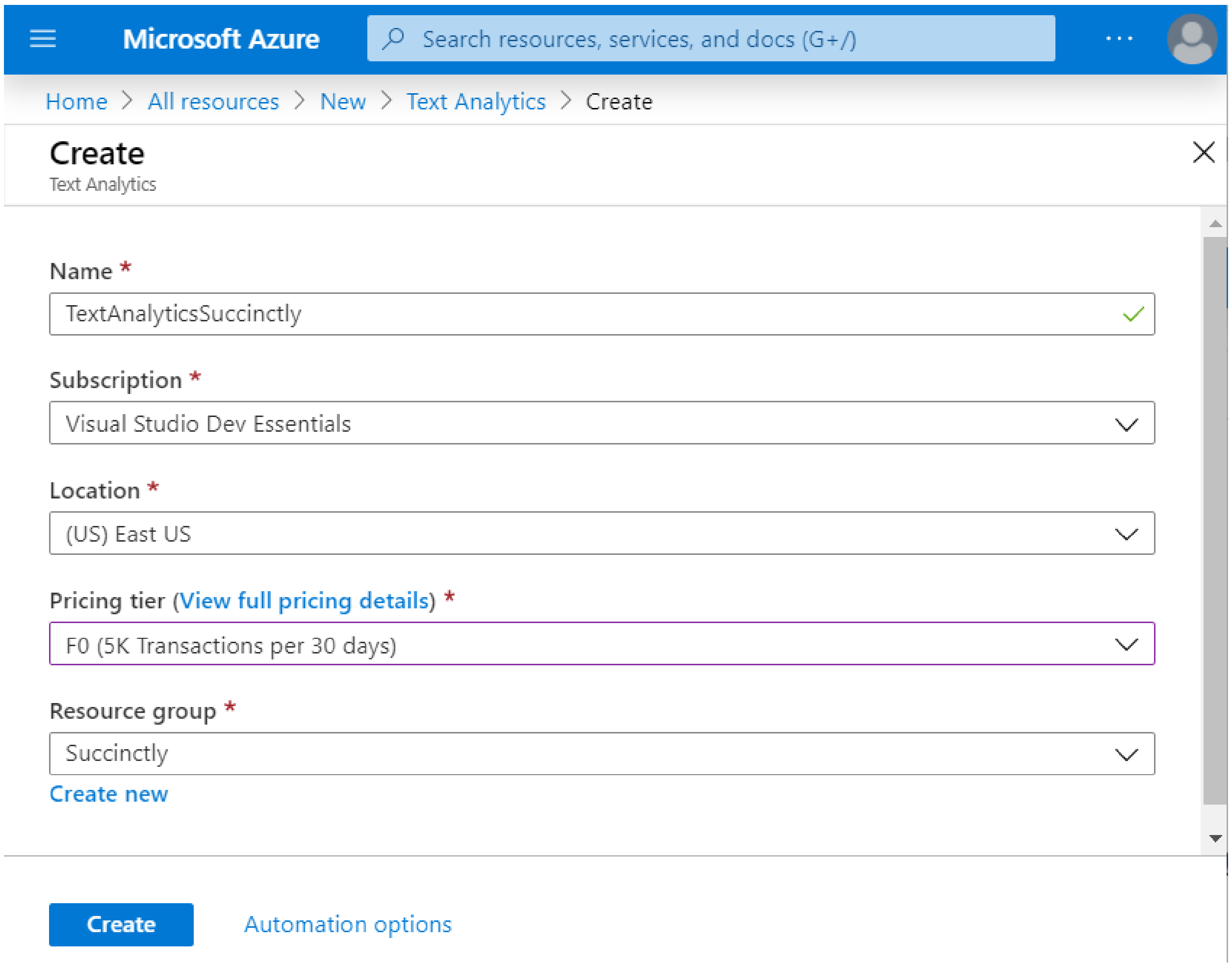

Microsoft Azure
Search resources, services, and docs (G+/)
Home > All resources > New > Text Analytics
Text Analytics
Microsoft
Text Analytics
Save for later
Microsoft
Create

Figure 3-h: Text Analytics (Azure Portal)

Click **Create** to create the Text Analytics service.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > All resources > New > Text Analytics > Create
Create
Text Analytics

Name *
TextAnalyticsSuccinctly

Subscription *
Visual Studio Dev Essentials

Location *
(US) East US

Pricing tier (View full pricing details) *
F0 (5K Transactions per 30 days)

Resource group *
Succinctly

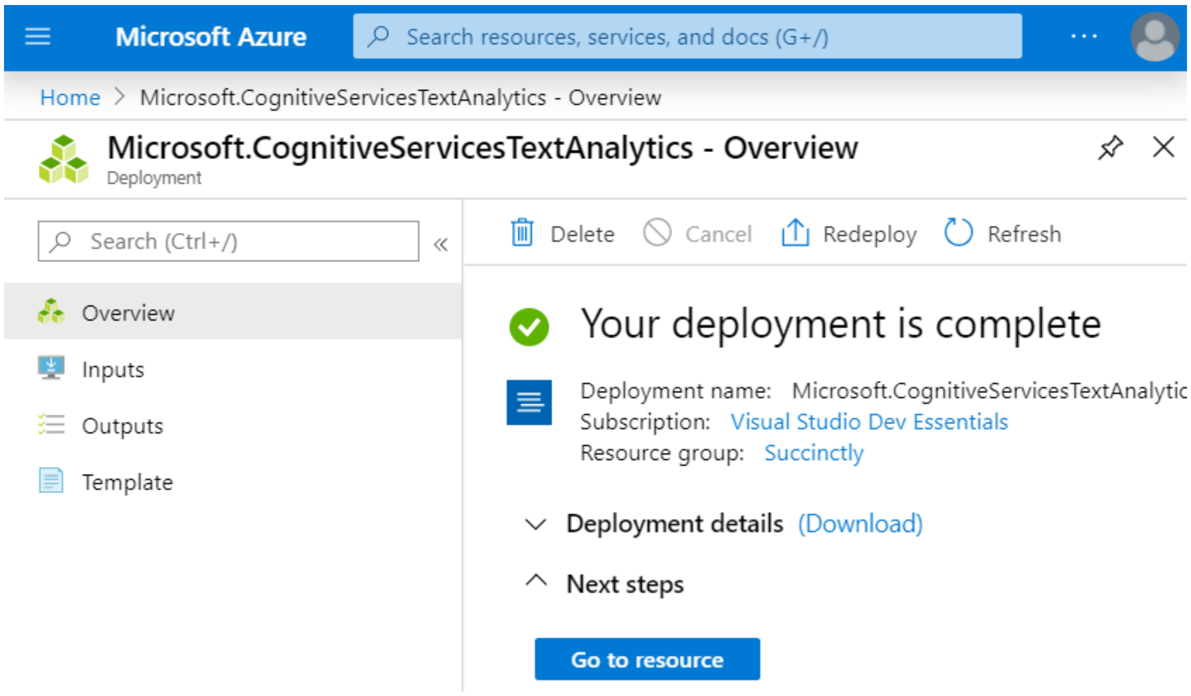
Create new
Create
Automation options

Figure 3-i: Create Text Analytics (Azure Portal)

Next, you'll need to enter the required field details. The most important field is the **Pricing tier**. Choose the **F0** option, which is the free pricing tier.

As you might remember, we previously created a resource group called **Succinctly**, so you can select that one from the list, or any other you might have created.

Once you're done, click **Create**. One the service has been created, you'll see a screen similar to the following one.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Microsoft.CognitiveServicesTextAnalytics - Overview
Microsoft.CognitiveServicesTextAnalytics - Overview
Deployment
Search (Ctrl+/)
Overview
Inputs
Outputs
Template
Delete
Cancel
Redeploy
Refresh

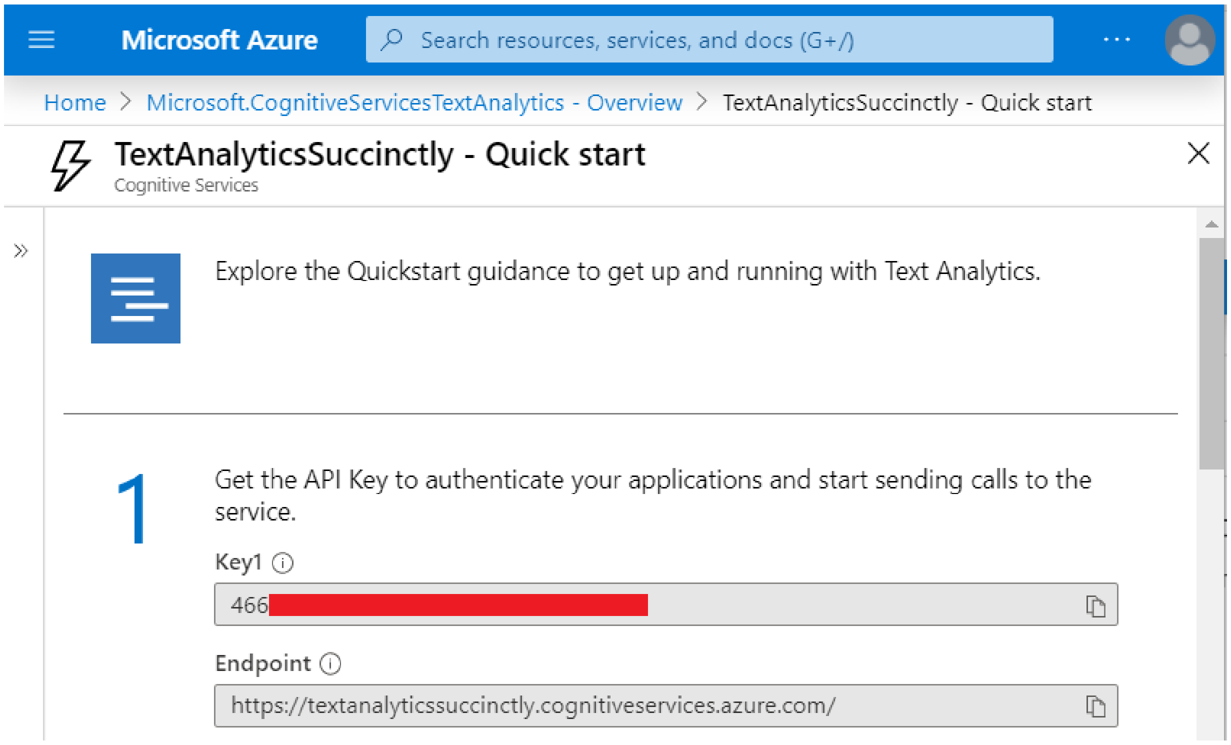
Your deployment is complete

Deployment name: Microsoft.CognitiveServicesTextAnalytic
Subscription: Visual Studio Dev Essentials
Resource group: Succinctly

Deployment details (Download)
Next steps
Go to resource

Figure 3-j: Text Analytics Created (Azure Portal)

To access the service, click **Go to resource**, which will take us to the **Quick start** screen.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Microsoft.CognitiveServicesTextAnalytics - Overview > TextAnalyticsSuccinctly - Quick start
TextAnalyticsSuccinctly - Quick start
Cognitive Services

Explore the Quickstart guidance to get up and running with Text Analytics.

1 Get the API Key to authenticate your applications and start sending calls to the service.

Key1: 466[redacted]

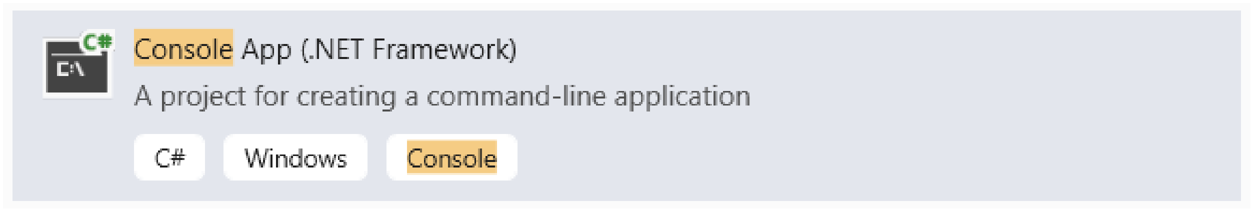
Endpoint: https://textanalyticssuccinctly.cognitiveservices.azure.com/

Figure 3-k: Text Analytics Created (Azure Portal)

We have our service up and running—now it’s time to use it.

Text Analytics app setup

With Visual Studio 2019 open, create a new **Console App (.NET Framework)** project, which you can do by choosing the following option.


Console App (.NET Framework)

A project for creating a command-line application

Figure 3-l: Console App (.NET Framework) Project Option

Once the project has been created—I'll name mine **TextAnalytics**—open **Program.cs** and add the following code so we can start to build up our logic within this file.

Code Listing 3-a: Program.cs—Setup and Authentication

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

using Microsoft.Rest;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics.Models;

namespace TextAnalytics
{
    class ApiKeyServiceClientCredentials : ServiceClientCredentials
    {
        private const string cKeyLbl = "Ocp-Apim-Subscription-Key";
        private readonly string subscriptionKey;

        public ApiKeyServiceClientCredentials(string subscriptionKey)
        {
            this.subscriptionKey = subscriptionKey;
        }

        public override Task ProcessHttpRequestAsync(HttpRequestMessage
            request, CancellationToken cancellationToken)
        {
            if (request != null)
            {
                request.Headers.Add(cKeyLbl, subscriptionKey);
                return base.ProcessHttpRequestAsync(
                    request, cancellationToken);
            }
            else return null;
        }
    }

    // Program class goes here

}
```

We start by referencing the required libraries that our application will need, which are:

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
```

Notice the references to these libraries:

```
using Microsoft.Rest;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics.Models;
```

You need to add these to the project by using the **Solution Explorer**. Right-click the **References** option and choose **Manage NuGet Packages**.

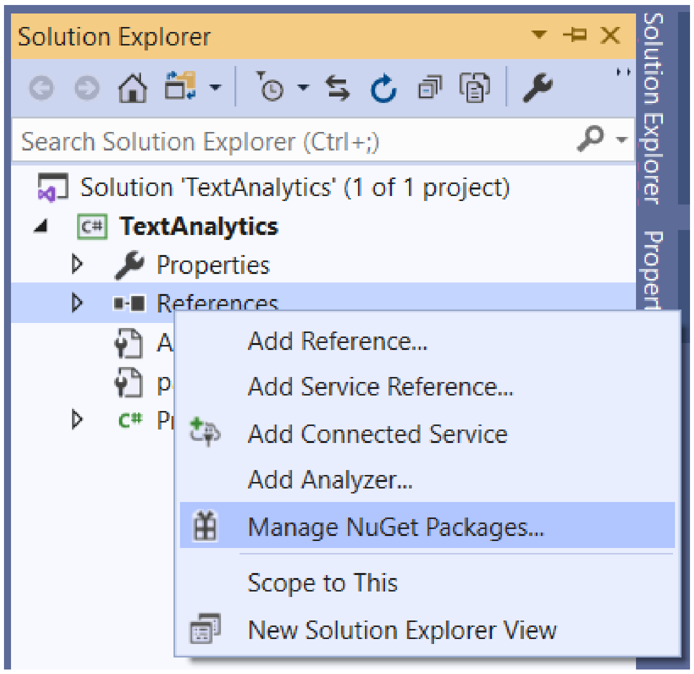
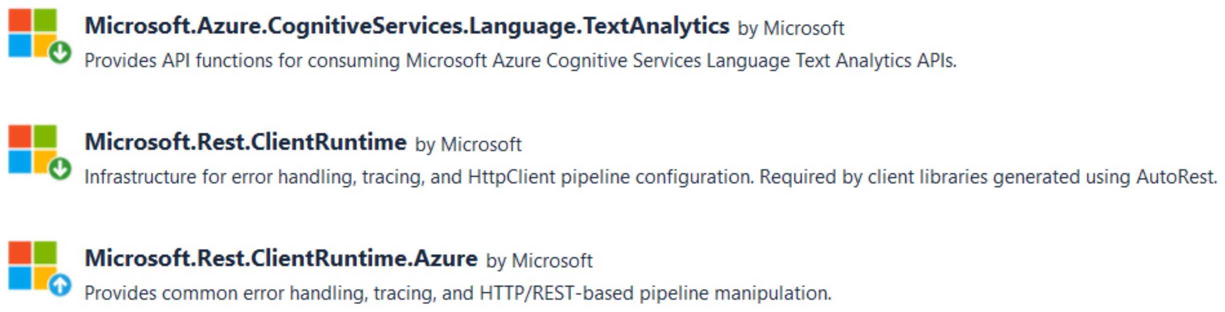


Figure 3-m: Adding Project References

Once you've opened the **NuGet Package Manager**, search for the following packages and install them.



Microsoft.Azure.CognitiveServices.Language.TextAnalytics by Microsoft
Provides API functions for consuming Microsoft Azure Cognitive Services Language Text Analytics APIs.

Microsoft.Rest.ClientRuntime by Microsoft
Infrastructure for error handling, tracing, and HttpClient pipeline configuration. Required by client libraries generated using AutoRest.

Microsoft.Rest.ClientRuntime.Azure by Microsoft
Provides common error handling, tracing, and HTTP/REST-based pipeline manipulation.

Figure 3-n: Packages to Be Installed

With these packages installed, let's continue to review our code—notice how we've defined an **`ApiKeyServiceClientCredentials`** class that inherits from **`ServiceClientCredentials`**, which we will use for authentication to the **`TextAnalytics`** service.

The constant **`cKeyLbl`** represents the label of the **`Ocp-Apim-Subscription-Key`** header parameter, which is passed to the **`TextAnalytics`** service. **`subscriptionKey`** is the value of the subscription key that is passed to the service to perform authentication.

The **`subscriptionKey`** value is initialized within the class constructor as follows:

```
public ApiKeyServiceClientCredentials(string subscriptionKey)
{
    this.subscriptionKey = subscriptionKey;
}
```

The **`ProcessHttpRequestAsync`** method is responsible for assigning to the **`request`** object headers, the value of **`subscriptionKey`** that corresponds to the **`Ocp-Apim-Subscription-Key`** header parameter. This method executes when the authentication to the service takes place.

Now that we've seen how the authentication works, let's add the **`Program`** class to **`TextAnalytics`**, which will contain the logic to interact with the service.

Text Analytics Logic

Within **Program.cs**, let's add the **`Program`** class, the subscription key, and the API service endpoint, as follows.

Code Listing 3-b: Program.cs—Text Analytics Logic

```
class Program
{
    private const string cKey = "<< Your Subscription Key goes here >>";
    private const string cEndpoint =
        "https://textanalyticssuccinctly.cognitiveservices.azure.com/";
}
```

Assign to **`cKey`** the value of the **Key1** and to **`cEndpoint`** the value of the **Endpoint** obtained from the Text Analytics service within the Azure portal, as seen in Figure 3-k.

Next, we need to add the following code to **Program.cs**.

Code Listing 3-c: Program.cs—Text Analytics Logic (InitApi Method)

```
class Program
{
    // Previous code

    private static TextAnalyticsClient InitApi(string key)
    {
        return new TextAnalyticsClient(new
            ApiKeyServiceClientCredentials(key))
            {
                Endpoint = cEndpoint
            };
    }
}
```

The **`InitApi`** method is responsible for creating and returning an instance of the **`TextAnalyticsClient`** class by passing an instance of **`ApiKeyServiceClientCredentials`**, which will perform the authentication to the Text Analytics service.

Let's have a look at the **`Main`** method of **Program.cs** so we can work our way back to each of the methods that will make up the logic of the app.

Code Listing 3-d: Program.cs—Text Analytics Logic (Main Method)

```
class Program
{
    // Previous code

    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

        string[] items = new string[] {
            // English text
            "Microsoft was founded by Bill Gates" +
            " and Paul Allen on April 4, 1975, " +
            "to develop and sell BASIC " +
            "interpreters for the Altair 8800",
            // Spanish text
            "La sede principal de Microsoft " +
            "se encuentra en la ciudad de " +
            "Redmond, a 21 kilómetros " +
            "de Seattle"
        };

        ProcessSentiment(items).Wait();
        ProcessRecognizeEntities(items).Wait();
        ProcessKeyPhrasesExtract(items).Wait();

        Console.ReadLine();
    }
}
```

Note: The** `Console.OutputEncoding = Encoding.UTF8;` **instruction is set in case you would like to experiment later with languages that aren't constrained to the ASCII character set, such as Arabic and Chinese, or languages that use the Cyrillic alphabet.

In this example, we want to run the Text Analytics service on the **items** array, which contains text in both **English** and **Spanish**.

The app's **Main** method invokes the **ProcessSentiment**, **ProcessRecognizeEntities**, and **ProcessKeyPhrasesExtract** methods.

The **ProcessSentiment** method is responsible for executing sentiment analysis on the text contained within the **items** array and returning a result.

The **ProcessRecognizeEntities** method is responsible for recognizing entities within the text contained in the **items** array.

The **ProcessKeyPhrasesExtract** method is responsible for extracting key phrases within the text contained in the **items** array.

Note: The** `.Wait();` **method instruction is appended to each of the calls to** `ProcessSentiment`, `ProcessRecognizeEntities`, **and** `ProcessKeyPhrasesExtract` **methods. This is because these methods are asynchronous, and before the results can be displayed, the program must wait for these methods to finalize their execution.

The **Console.ReadLine** instruction is placed just before the end of the **Main** method so that the results returned from these three methods can be displayed on the screen before the execution of the program finalizes.

Now that we understand how the **Main** method works, let's explore how each of these three methods works.

Sentiment analysis

Let's now explore the **`ProcessSentiment`** method to understand how sentiment analysis can be done with the Text Analytics service.

Code Listing 3-e: Program.cs—TextAnalytics Logic (ProcessSentiment Method)

```
class Program
{
    // Previous code

    private static async Task ProcessSentiment(string[] items)
    {
        string[] langs = await GetDetectLanguage(
            InitApi(cKey), GetLBI(items));

        RunSentiment(InitApi(cKey),
            GetMLBI(MergeItems(items, langs))).Wait();
        Console.WriteLine($"\t");
    }
}
```

The **`ProcessSentiment`** method receives the **`items`** array as a parameter. This contains the text that is going to be submitted to the Text Analytics service for sentiment analysis.

The **`GetDetectLanguage`** method is invoked first; for sentiment analysis to take place, the service must know what language it needs to perform the analysis on.

The **`GetDetectLanguage`** method requires two parameters. The first is an instance of TextAnalyticsClient, and the second is a variable of LanguageBatchInput type.

The **`TextAnalyticsClient`** object is returned by the **`InitApi`** method, which receives the subscription key (**`cKey`**), as follows.

Code Listing 3-f: Program.cs—TextAnalytics Logic (InitApi Method)

```
class Program
{
    // Previous code

    private static TextAnalyticsClient InitApi(string key)
    {
        return new TextAnalyticsClient(new
            ApiKeyServiceClientCredentials(key))
        {
            Endpoint = cEndpoint
        };
    }
}
```

As we can clearly observe, the **TextAnalyticsClient** instance is created by passing an **ApiKeyServiceClientCredentials** instance as a parameter, assigning the **key** and **Endpoint**.

The **GetLBI** method returns the **LanguageBatchInput** object, which is obtained from the **items** array. Let's have a look at the implementation of the **GetLBI** method.

Code Listing 3-g: Program.cs—TextAnalytics Logic (GetLBI Method)

```
class Program
{
    // Previous code

    private static LanguageBatchInput GetLBI(string[] items)
    {
        List<LanguageInput> lst = new List<LanguageInput>();

        for (int i = 0; i <= items.Length - 1; i++)
            lst.Add(new LanguageInput((i + 1).ToString(), items[i]));

        return new LanguageBatchInput(lst);
    }
}
```

The **GetLBI** method starts by creating a **LanguageInput** list (**lst**), which will store the language information returned by the method.

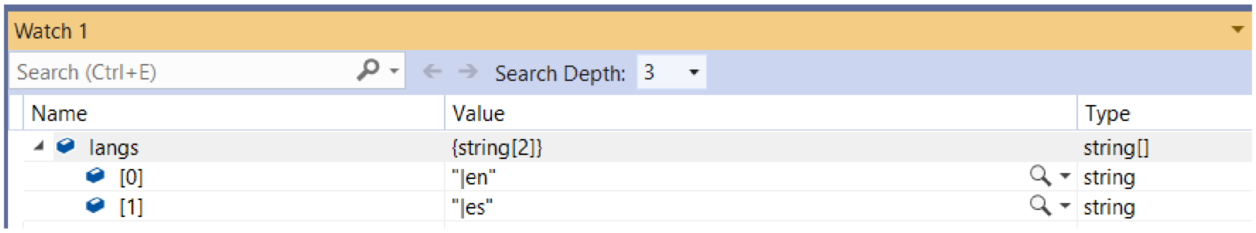
To do that, we loop through the **items** array and for each item, we create a **LanguageInput** instance, to which we pass an index as a string (**(i + 1).ToString()**) and the item itself (**items[i]**). Each **LanguageInput** instance is added to **lst**.

Why are we determining the language first? The reason is that the Text Analytics service needs to know the language of the text that is going to be analyzed. This is done to ensure that the result of the analysis can be as accurate as possible, which is the reason why we need to determine the languages of the text elements contained within the **items** array by invoking the **GetDetectLanguage** method before calling **RunSentiment**.

The language of each text element contained within the **items** array is kept within the **langs** array, which we can see as follows.

```
    string[] items = new string[] {
        // English text
        "Microsoft was founded by Bill Gates" +
        " and Paul Allen on April 4, 1975, " +
        "to develop and sell BASIC " +
        "interpreters for the Altair 8800",

        // Spanish text
        "La sede principal de Microsoft " +
        "se encuentra en la ciudad de " +
        "Redmond, a 21 kilómetros " +
        "de Seattle"
    };
```

Here are the results assigned to the **langs** array after executing the **GetDetectLanguage** method:



Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
langs	{string[2]}	string[]
[0]	"|en"	string
[1]	"|es"	string

Figure 3-o: Values of the langs Array

Let's have a look at the code of the **GetDetectLanguage** method to better understand what it does.

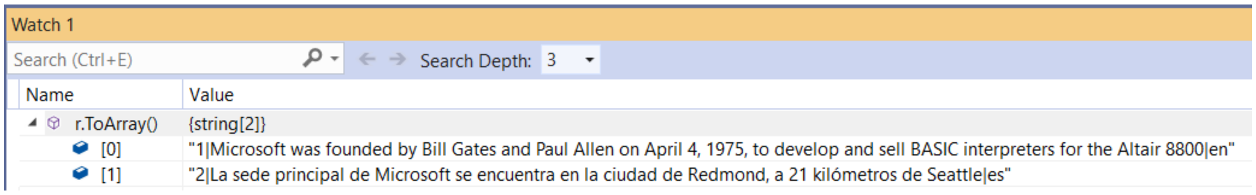
Code Listing 3-h: Program.cs—TextAnalytics Logic (GetDetectLanguage Method)

```
class Program
{
    // Previous code

    private static async Task<string[]> GetDetectLanguage(
        TextAnalyticsClient client, LanguageBatchInput docs)
    {
        List<string> ls = new List<string>();

        var res = await client.DetectLanguageBatchAsync(docs);

        foreach (var document in res.Documents)
            ls.Add("|" + document.DetectedLanguages[0].Iso6391Name);

        return ls.ToArray();
    }
}
```

This method takes the **TextAnalyticsClient** instance as a parameter, as it is required to be able to invoke the **DetectLanguageBatchAsync** method from the Text Analytics service, to which **docs** (the resultant object returned from the **GetLBI** method) is passed.

Then, we loop through each of the **Documents** found within the results (**res**) obtained from the call to the **DetectLanguageBatchAsync** method. This is done so that we can obtain the ISO 639-1 code of each of the languages detected. These are appended to **ls**, which is a **string** list that will be returned as an array by the **GetDetectLanguage** method.

Going back to the **ProcessSentiment** method code listing, we can see that these language values are merged into a single array with their corresponding text items. This is done by invoking the **MergeItems** method, which renders the following results when executed.



```
Watch 1
Search (Ctrl+E)        Search Depth: 3
Name            Value
◢ r.ToArray()   {string[2]}
    [0]         "1|Microsoft was founded by Bill Gates and Paul Allen on April 4, 1975, to develop and sell BASIC interpreters for the Altair 8800|en"
    [1]         "2|La sede principal de Microsoft se encuentra en la ciudad de Redmond, a 21 kilómetros de Seattle|es"
```

Figure 3-p: Values Returned by the MergeItems Method

To understand this better, let's explore the code of the **`MergeItems`** method, which we can see as follows.

Code Listing 3-i: Program.cs—TextAnalytics Logic (MergeItems Method)

```
class Program
{
    // Previous code

    private static string[] MergeItems(string[] a1,  string[] a2)
    {
        List<string> r = new List<string>();

        if (a2 == null || a1.Length == a2.Length)
            for (int i = 0; i <= a1.Length - 1; i++)
                r.Add($"{(i + 1).ToString()}|{a1[i]}{a2[i]}");

        return r.ToArray();
    }
}
```

The **`MergeItems`** method literally merges the values of the **`items`** array (**`a1[i]`**) with those of the **`langs`** array (**`a2[i]`**), and it appends an index (a sequential number—**`(i + 1).ToString()`**) at the beginning of each element of the resultant array. This sequential number is required by the Text Analytics service.

The resultant array (**`r.ToArray()`**) returned by the **`MergeItems`** method is passed to the **`GetMLBI`** method as a parameter, which returns a MultiLanguageBatchInput object. This is how the Text Analytics service expects the information in order to process it.

Let's now explore the code of the **`GetMLBI`** method to understand what it does.

Code Listing 3-j: Program.cs —TextAnalytics Logic (GetMLBI Method)

```
Class Program
{
    // Previous code

    private static MultiLanguageBatchInput GetMLBI(string[] items)
    {
        List<MultiLanguageInput> lst = new List<MultiLanguageInput>();
        foreach (string itm in items)
        {
            string[] p = itm.Split('|');
            lst.Add(new MultiLanguageInput(p[0], p[1], p[2]));
        }

        return new MultiLanguageBatchInput(lst);
    }
}
```

If you pay close attention, you'll notice that the code of the **`GetMLBI`** method is quite similar to the code of the **`GetLBI`** method.

We loop through each of the string elements (**`itm`**) contained within the **`items`** array and split each string element into parts—the splitting is done using the pipe (**|**) character.

Then, the parts obtained are passed as parameters when creating an instance of the MultiLanguageInput class.

For each string element (**`itm`**) contained within the **`items`** array, a **`MultiLanguageInput`** instance is created and added to a list of the same type.

The method then returns a **`MultiLanguageBatchInput`** instance from the resultant **`MultiLanguageInput`** list.

The returned **`MultiLanguageBatchInput`** instance will later be used by the **`RunSentiment`**, **`RunRecognizeEntities`**, and **`RunKeyPhrasesExtract`** methods.

Now, let's explore the **`RunSentiment`** method, which is the final piece of the puzzle required to understand how sentiment analysis is done using the Text Analytics service.

Code Listing 3-k: Program.cs—TextAnalytics Logic (RunSentiment Method)

```
Class Program
{
    // Previous code

    private static async Task RunSentiment(TextAnalyticsClient client,
        MultiLanguageBatchInput docs)
    {
        var res = await client.SentimentBatchAsync(docs);

        foreach (var document in res.Documents)
            Console.WriteLine($"Document ID: {document.Id}, " +
                $"Sentiment Score: {document.Score:0.00}");
    }
}
```

As we can see, the code is quite straightforward. The **`RunSentiment`** method receives a **`TextAnalyticsClient`** parameter, which is responsible for calling the Text Analytics service, through the **`client`** object.

The **`RunSentiment`** method also receives a **`MultiLanguageBatchInput`** parameter, which represents the text that is going to be sent to the Text Analytics service for processing. This occurs when the **`SentimentBatchAsync`** method is invoked.

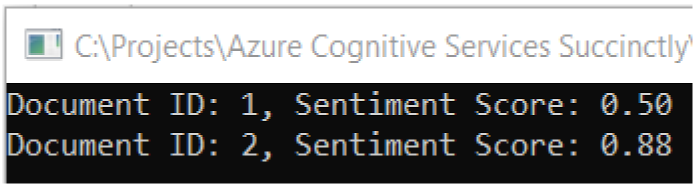
Once the result from the **`SentimentBatchAsync`** method is received (the value assigned to the **`res`** object), we loop through all the results by specifically checking the **`Documents`** property of the **`res`** object.

Each document's (**`document.Id`**) sentiment score is obtained by inspecting the **`document.Score`** property.

Now that we have this, we need to comment out the following two lines within the **`Main`** method. We can do this as follows.

```
ProcessRecognizeEntities(items).Wait(); // Comment out this line within Main
ProcessKeyPhrasesExtract(items).Wait(); // Comment out this line within Main
```

Awesome—that's all we need to perform sentiment analysis using the Text Analytics service. Let's run the code we've written and see what results we get.



```
Document ID: 1, Sentiment Score: 0.50
Document ID: 2, Sentiment Score: 0.88
```

Figure 3-q: Sentiment Analysis Results

Sentiment analysis full code

To fully appreciate what we've done, let's have a look at the full source code of **Program.cs** that we've created to perform sentiment analysis.

Code Listing 3-l: Program.cs—TextAnalytics Logic (Full Code)

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

using Microsoft.Rest;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics.Models;

namespace TextAnalytics
{
    class ApiKeyServiceClientCredentials : ServiceClientCredentials
    {
        private const string cKeyLbl = "Ocp-Apim-Subscription-Key";
        private readonly string subscriptionKey;

        public ApiKeyServiceClientCredentials(string subscriptionKey)
        {
            this.subscriptionKey = subscriptionKey;
        }

        public override Task ProcessHttpRequestAsync(
           HttpRequestMessage request, CancellationToken cancellationToken)
        {
            if (request != null)
            {
                request.Headers.Add(cKeyLbl, subscriptionKey);
                return base.ProcessHttpRequestAsync(
                    request, cancellationToken);
            }
            else return null;
        }
    }

    class Program
    {
        private const string cKey = "<< Key goes here >>";
        private const string cEndpoint =
            "https://textanalyticssuccinctly.cognitiveservices.azure.com/";

        private static TextAnalyticsClient InitApi(string key)
        {
            return new TextAnalyticsClient(new
                ApiKeyServiceClientCredentials(key))
                {
                    Endpoint = cEndpoint
                };
        }

        private static MultiLanguageBatchInput GetMLBI(string[] items)
        {
            List<MultiLanguageInput> lst = new List<MultiLanguageInput>();

            foreach (string itm in items)
            {
                string[] p = itm.Split('|');
                lst.Add(new MultiLanguageInput(p[0], p[1], p[2]));
            }

            return new MultiLanguageBatchInput(lst);
        }

        private static LanguageBatchInput GetLBI(string[] items)
        {
            List<LanguageInput> lst = new List<LanguageInput>();

            for (int i = 0; i <= items.Length - 1; i++)
                lst.Add(new LanguageInput((i + 1).ToString(), items[i]));

            return new LanguageBatchInput(lst);
        }

        private static async Task RunSentiment(TextAnalyticsClient client,
            MultiLanguageBatchInput docs)
        {
            var res = await client.SentimentBatchAsync(docs);

            foreach (var document in res.Documents)
                Console.WriteLine($"Document ID: {document.Id}, " +
                    $"Sentiment Score: {document.Score:0.00}");
        }

        private static async Task<string[]>
            GetDetectLanguage(TextAnalyticsClient client,
                LanguageBatchInput docs)
        {
            List<string> ls = new List<string>();

            var res = await client.DetectLanguageBatchAsync(docs);

            foreach (var document in res.Documents)
                ls.Add("|" + document.DetectedLanguages[0].Iso6391Name);

            return ls.ToArray();
        }

        private static string[] MergeItems(string[] a1,  string[] a2)
        {
            List<string> r = new List<string>();

            if (a2 == null || a1.Length == a2.Length)
                for (int i = 0; i <= a1.Length - 1; i++)
                    r.Add($"{(i + 1).ToString()}|{a1[i]}{a2[i]}");

            return r.ToArray();
        }

        private static async Task ProcessSentiment(string[] items)
        {
            string[] langs = await GetDetectLanguage(
                InitApi(cKey), GetLBI(items));

            RunSentiment(InitApi(cKey),
                GetMLBI(MergeItems(items, langs))).Wait();
            Console.WriteLine($"\t");
        }

        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.UTF8;

            string[] items = new string[] {
                "Microsoft was founded by Bill Gates" +
                " and Paul Allen on April 4, 1975, " +
                    "to develop and sell BASIC " +
                    "interpreters for the Altair 8800",

                "La sede principal de Microsoft " +
                "se encuentra en la ciudad de " +
                    "Redmond, a 21 kilómetros " +
                    "de Seattle"
            };

            ProcessSentiment(items).Wait();
            Console.ReadLine();
        }
    }
}
```

Recognizing entities

Now that we have seen how to perform sentiment analysis on text, let's see how we can recognize entities in them.

Let's start off by looking at the implementation of the **`ProcessRecognizeEntities`** method.

Code Listing 3-m: Program.cs—TextAnalytics Logic (ProcessRecognizeEntities Method)

```
class Program
{
    // Previous code

    private static async Task ProcessRecognizeEntities(string[] items)
    {
         string[] langs = await GetDetectLanguage(InitApi(cKey),
            GetLBI(items));

         RunRecognizeEntities(InitApi(cKey), GetMLBI(
             MergeItems(items, langs))).Wait();
         Console.WriteLine($"\t");
    }
}
```

As you can see, the code for this method is almost identical to the code of the **`ProcessSentiment`** method. The only difference is that it invokes the method **`RunRecognizeEntities`** instead, but the overall structure and logic is the same.

We've already looked at the code for the **`GetDetectLanguage`**, **`InitApi`**, **`GetLBI`**, **`GetMLBI`**, and **`MergeItems`** methods, so we are only missing the **`RunRecognizeEntities`** code. Let's have a look at it.

Code Listing 3-n: Program.cs—TextAnalytics Logic (RunRecognizeEntities Method)

```
class Program
{
    // Previous code

    private static async Task RunRecognizeEntities(
        TextAnalyticsClient client,
        MultiLanguageBatchInput docs)
    {
        var res = await client.EntitiesBatchAsync(docs);

        foreach (var document in res.Documents)
        {
            Console.WriteLine($"Document ID: {document.Id} ");
            Console.WriteLine("\tEntities:");

            foreach (var entity in document.Entities)
            {
                Console.WriteLine($"\t\t{entity.Name}");
                Console.WriteLine($"\t\tType: {entity.Type ?? "N/A"}");
                Console.WriteLine(
                    $"\t\tSubType: {entity.SubType ?? "N/A"}");

                foreach (var match in entity.Matches)
                    Console.WriteLine
                        ($"\t\tScore: {match.EntityTypeScore:F3}");

                Console.WriteLine($"\t");
            }
        }
    }
}
```

All the **`RunRecognizeEntities`** method does is execute the **`EntitiesBatchAsync`** method from the **`client`** object.

Once the result is returned—which contains the extracted entities found within the text—we loop through **`res.Documents`**. Each **`document`** object contains **`document.Entities`**.

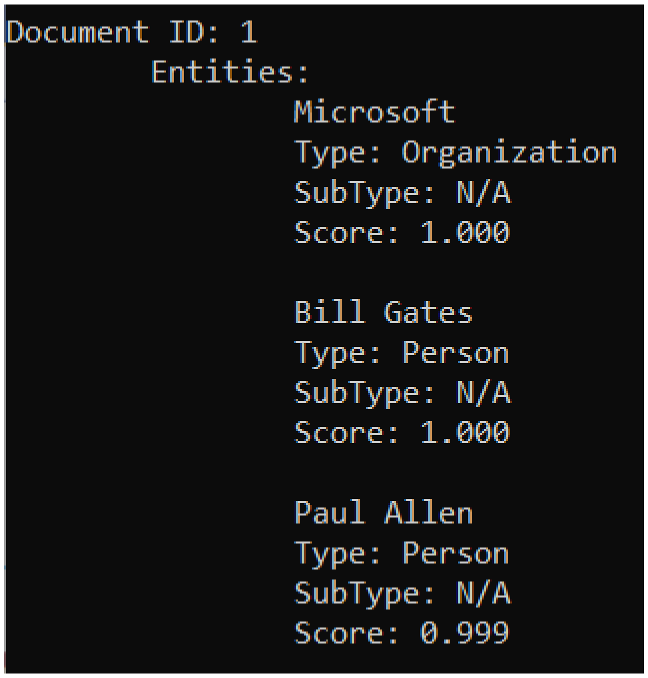
Then we loop through each **`entity`**. This way, we can print out each of the entity properties found, such as **`Name`**, **`Type`**, **`SubType`**, and **`EntityTypeScore`**, from **`entity.Matches`**.

That's all there is to it. As you can see, recognizing entities was super simple. Let's now invoke **`ProcessRecognizeEntities`** from the **`Main`** method.

Code Listing 3-o: Program.cs—TextAnalytics Logic (Main Method)

```
class Program
{
    // Previous code

    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

        string[] items = new string[] {
            "Microsoft was founded by Bill Gates" +
            " and Paul Allen on April 4, 1975, " +
            "to develop and sell BASIC " +
            "interpreters for the Altair 8800",

            "La sede principal de Microsoft " +
            "se encuentra en la ciudad de " +
            "Redmond, a 21 kilómetros " +
            "de Seattle"
        };

        ProcessRecognizeEntities(items).Wait();
        Console.ReadLine();
    }
}
```

Let's run the code to see what we get—here's a snippet of the results.



```
Document ID: 1
        Entities:
                Microsoft
                Type: Organization
                SubType: N/A
                Score: 1.000

                Bill Gates
                Type: Person
                SubType: N/A
                Score: 1.000

                Paul Allen
                Type: Person
                SubType: N/A
                Score: 0.999
```

Figure 3-r: Recognizing Entities Results

As you can see, the Text Analytics service was able to identify **Microsoft** as an organization, and **Bill Gates** and **Paul Allen** as persons.

Extracting key phrases

We are now ready to explore how to extract key phrases from text by using the Text Analytics service. Let's start off with the **`ProcessKeyPhrasesExtract`** method.

Code Listing 3-p: Program.cs—TextAnalytics Logic (ProcessKeyPhrasesExtract Method)

```
class Program
{
    // Previous code

    private static async Task ProcessKeyPhrasesExtract(string[] items)
    {
        string[] langs = await GetDetectLanguage(
            InitApi(cKey), GetLBI(items));

        RunKeyPhrasesExtract(InitApi(cKey),
            GetMLBI(MergeItems(items, langs))).Wait();
        Console.WriteLine($"\t");
    }
}
```

As you can see, the pattern is the same one we've seen before. The languages are first detected when the **GetDetectLanguage** method is invoked, and then the call to the Text Analytics service is made—which, in this case, is done by executing the **RunKeyPhrasesExtract** method.

As we've already seen in each of the other methods invoked by **ProcessKeyPhrasesExtract**, all we are missing is to look at the code behind **RunKeyPhrasesExtract**—which is what we'll do next.

Code Listing 3-q: Program.cs—TextAnalytics Logic (RunKeyPhrasesExtract Method)

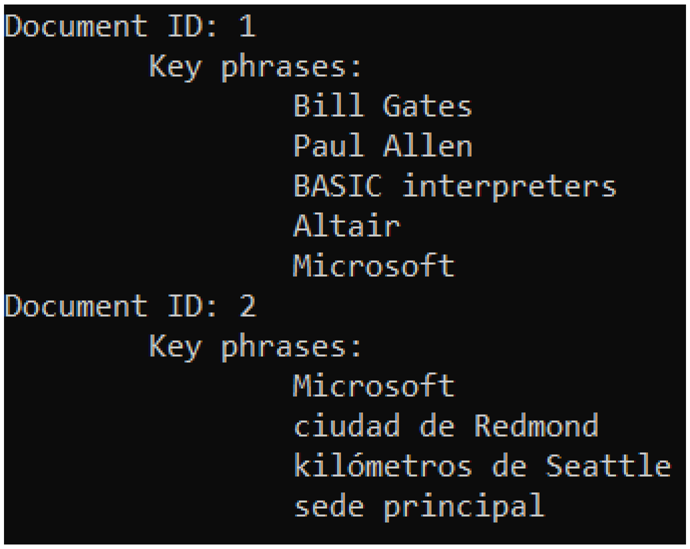
```
class Program
{
    // Previous code

    private static async Task RunKeyPhrasesExtract(
        TextAnalyticsClient client, MultiLanguageBatchInput docs)
    {
        var res = await client.KeyPhrasesBatchAsync(docs);

        foreach (var document in res.Documents)
        {
            Console.WriteLine($"Document ID: {document.Id} ");
            Console.WriteLine("\tKey phrases:");

            foreach (string keyphrase in document.KeyPhrases)
                Console.WriteLine($"\t\t{keyphrase}");
        }
    }
}
```

The **RunKeyPhrasesExtract** method receives as parameters a **TextAnalyticsClient** object, which is used to invoke the Text Analytics service, and the text to process, which is a **MultiLanguageBatchInput** object.

The call to the Text Analytics service is made by executing the **KeyPhrasesBatchAsync** method, and the results returned are assigned to the **res** object.

For each **document** contained within **res.Documents**, we can get each **keyphrase** by looping through the **document.KeyPhrases** object.

As you have seen, that was also very easy to do! Now, let's run the code to check what results we get.



```
Document ID: 1
        Key phrases:
                Bill Gates
                Paul Allen
                BASIC interpreters
                Altair
                Microsoft
Document ID: 2
        Key phrases:
                Microsoft
                ciudad de Redmond
                kilómetros de Seattle
                sede principal
```

Figure 3-s: Recognizing Key Phrases

Notice how the Text Analytics service has recognized various key phrases for the text submitted.

Text Analytics full code

The following listing shows the complete source code that we've written throughout this chapter for working with the Text Analytics service.

Code Listing 3-r: Program.cs—Text Analytics Logic (Full Source Code)

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

using Microsoft.Rest;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics.Models;

namespace TextAnalytics
{
    class ApiKeyServiceClientCredentials : ServiceClientCredentials
    {
        private const string cKeyLbl = "Ocp-Apim-Subscription-Key";
        private readonly string subscriptionKey;

        public ApiKeyServiceClientCredentials(string subscriptionKey)
        {
            this.subscriptionKey = subscriptionKey;
        }

        public override Task ProcessHttpRequestAsync(
            HttpRequestMessage request,
            CancellationToken cancellationToken)
        {
            if (request != null)
            {
                request.Headers.Add(cKeyLbl, subscriptionKey);
                return base.ProcessHttpRequestAsync(
                    request, cancellationToken);
            }
            else return null;
        }
    }

    class Program
    {
        private const string cKey = "<< Key goes here >>";
        private const string cEndpoint =
            "https://textanalyticssuccinctly.cognitiveservices.azure.com/";

        private static TextAnalyticsClient InitApi(string key)
        {
            return new TextAnalyticsClient(new
                ApiKeyServiceClientCredentials(key))
                {
                    Endpoint = cEndpoint
                };
        }

        private static MultiLanguageBatchInput GetMLBI(string[] items)
        {
            List<MultiLanguageInput> lst = new List<MultiLanguageInput>();

            foreach (string itm in items)
            {
                string[] p = itm.Split('|');
                lst.Add(new MultiLanguageInput(p[0], p[1], p[2]));
            }

            return new MultiLanguageBatchInput(lst);
        }

        private static LanguageBatchInput GetLBI(string[] items)
        {
            List<LanguageInput> lst = new List<LanguageInput>();
            for (int i = 0; i <= items.Length - 1; i++)
                lst.Add(new LanguageInput((i + 1).ToString(), items[i]));

            return new LanguageBatchInput(lst);
        }

        private static async Task RunSentiment(TextAnalyticsClient client,
            MultiLanguageBatchInput docs)
        {
            var res = await client.SentimentBatchAsync(docs);

            foreach (var document in res.Documents)
                Console.WriteLine($"Document ID: {document.Id}, " +
                    $"Sentiment Score: {document.Score:0.00}");
        }

        private static async Task<string[]> GetDetectLanguage(
            TextAnalyticsClient client,
            LanguageBatchInput docs)
        {
            List<string> ls = new List<string>();

            var res = await client.DetectLanguageBatchAsync(docs);

            foreach (var document in res.Documents)
                ls.Add("|" + document.DetectedLanguages[0].Iso6391Name);

            return ls.ToArray();
        }

        private static async Task RunRecognizeEntities(
            TextAnalyticsClient client,
            MultiLanguageBatchInput docs)
        {
            var res = await client.EntitiesBatchAsync(docs);

            foreach (var document in res.Documents)
            {
                Console.WriteLine($"Document ID: {document.Id} ");
                Console.WriteLine("\tEntities:");

                foreach (var entity in document.Entities)
                {
                    Console.WriteLine($"\t\t{entity.Name}");
                    Console.WriteLine($"\t\tType: {entity.Type ?? "N/A"}");
                    Console.WriteLine
                        ($"\t\tSubType: {entity.SubType ?? "N/A"}");

                    foreach (var match in entity.Matches)
                        Console.WriteLine
                            ($"\t\tScore: {match.EntityTypeScore:F3}");

                    Console.WriteLine($"\t");
                }
            }
        }

        private static async Task RunKeyPhrasesExtract(
            TextAnalyticsClient client,
            MultiLanguageBatchInput docs)
        {
            var res = await client.KeyPhrasesBatchAsync(docs);

            foreach (var document in res.Documents)
            {
                Console.WriteLine($"Document ID: {document.Id} ");
                Console.WriteLine("\tKey phrases:");

                foreach (string keyphrase in document.KeyPhrases)
                    Console.WriteLine($"\t\t{keyphrase}");
            }
        }

        private static string[] MergeItems(string[] a1,  string[] a2)
        {
            List<string> r = new List<string>();

            if (a2 == null || a1.Length == a2.Length)
                for (int i = 0; i <= a1.Length - 1; i++)
                    r.Add($"{(i + 1).ToString()}|{a1[i]}{a2[i]}");

            return r.ToArray();
        }

        private static async Task ProcessSentiment(string[] items)
        {
            string[] langs = await GetDetectLanguage(
                InitApi(cKey), GetLBI(items));

            RunSentiment(InitApi(cKey),
                GetMLBI(MergeItems(items, langs))).Wait();
            Console.WriteLine($"\t");
        }

        private static async Task ProcessRecognizeEntities(string[] items)
        {
            string[] langs = await GetDetectLanguage(
                InitApi(cKey), GetLBI(items));
            RunRecognizeEntities(InitApi(cKey),
                GetMLBI(MergeItems(items, langs))).Wait();
            Console.WriteLine($"\t");
        }

        private static async Task ProcessKeyPhrasesExtract(string[] items)
        {
            string[] langs = await GetDetectLanguage(
                InitApi(cKey), GetLBI(items));

            RunKeyPhrasesExtract(InitApi(cKey),
                GetMLBI(MergeItems(items, langs))).Wait();
            Console.WriteLine($"\t");
        }

        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.UTF8;

            string[] items = new string[] {
                "Microsoft was founded by Bill Gates" +
                " and Paul Allen on April 4, 1975, " +
                    "to develop and sell BASIC " +
                    "interpreters for the Altair 8800",

                "La sede principal de Microsoft " +
                "se encuentra en la ciudad de " +
                    "Redmond, a 21 kilómetros " +
                    "de Seattle"
            };

            ProcessSentiment(items).Wait();
            ProcessRecognizeEntities(items).Wait();
            ProcessKeyPhrasesExtract(items).Wait();

            Console.ReadLine();
        }
    }
}
```

Summary

In this chapter, we looked at how to use Text Analytics to identify languages, run sentiment analysis, and extract key phrases and entities within text.

As you have seen, the code was easy to understand and relatively easy to implement, and the results we obtained were great.

If you are done experimenting with the Text Analytics service, I encourage you to remove any unused Azure resources, especially if they are not in the free pricing tier.

In the next chapter, we'll explore how to work with some of the speech-processing capabilities that are provided by Azure Cognitive Services.

Chapter 4 Speech

Quick intro

Cognitive Services also provides the ability to integrate and add speech processing capabilities to your application, which include:

- **Speech to text**: Transcribes audible speech into readable and searchable text.
- **Text to speech**: Converts text to lifelike speech for more natural interfaces.
- **Speech translation**: Integrates real-time speech translation into your apps.
- **Speaker recognition**: Identifies and verifies the people speaking based on audio.

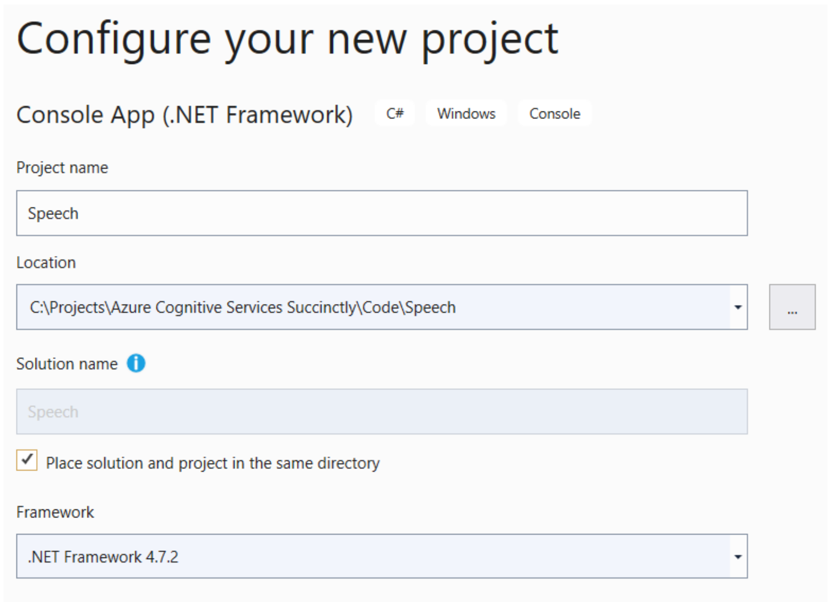
In this chapter, we'll be focusing on implementing text to speech with Azure Cognitive Services, which provides a great way to give a voice to your application, and we'll also explore how to convert speech to text.

Speech translation and speaker recognition are beyond the scope of this book, and probably deserve a full book to cover them in depth.

Nevertheless, text to speech and speech to text are two very exciting capabilities within Cognitive Services. So, without further ado, let's jump right into both.

Speech project

With Visual Studio 2019 open, create a new **Console App (.NET Framework)** project. I'll name mine **Speech**, which we can see as follows.


Configure your new project

Console App (.NET Framework)

Project name: Speech

Location: C:\Projects\Azure Cognitive Services Succinctly\Code\Speech

Solution name: Speech

[✓] Place solution and project in the same directory

Framework: .NET Framework 4.7.2

Figure 4-a: Creating the Speech Project

Within this project, we will organize the code for both the text-to-speech and speech-to-text logic.

Creating the Speech service

We are going to start our journey with **Cognitive Services** and the Speech SDK, and learn how to convert text to synthesized speech.

Before we can write any code, let's create a Speech service instance on the Azure portal. Here's a link to the official documentation.

In the Azure portal, navigate to **All resources** and click the **Create resource** or the **Add resource** button. Enter **Speech** in the **Search the Marketplace** search box and select the **Speech** option from the drop-down list, which will display the following.

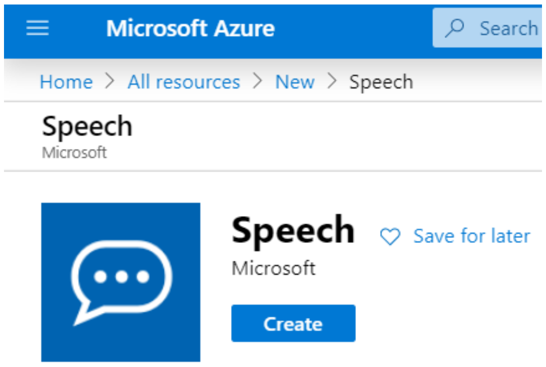
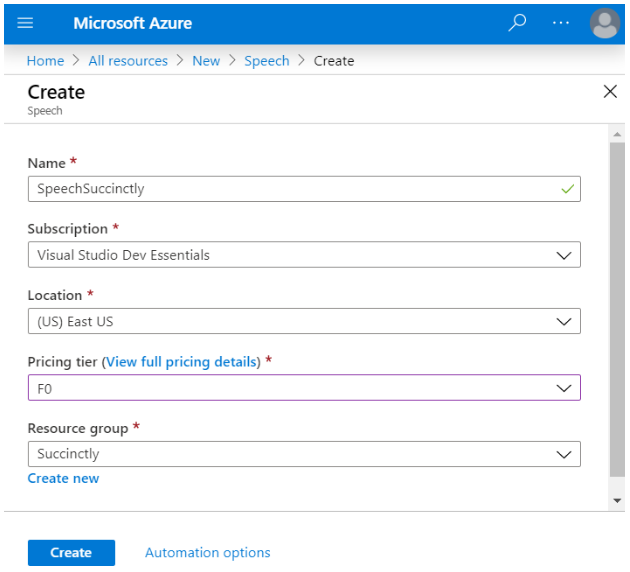


Figure 4-b: Speech Service

Next, click **Create**. This will display the following screen, which displays the fields that need to be filled.



Name *
SpeechSuccinctly

Subscription *
Visual Studio Dev Essentials

Location *
(US) East US

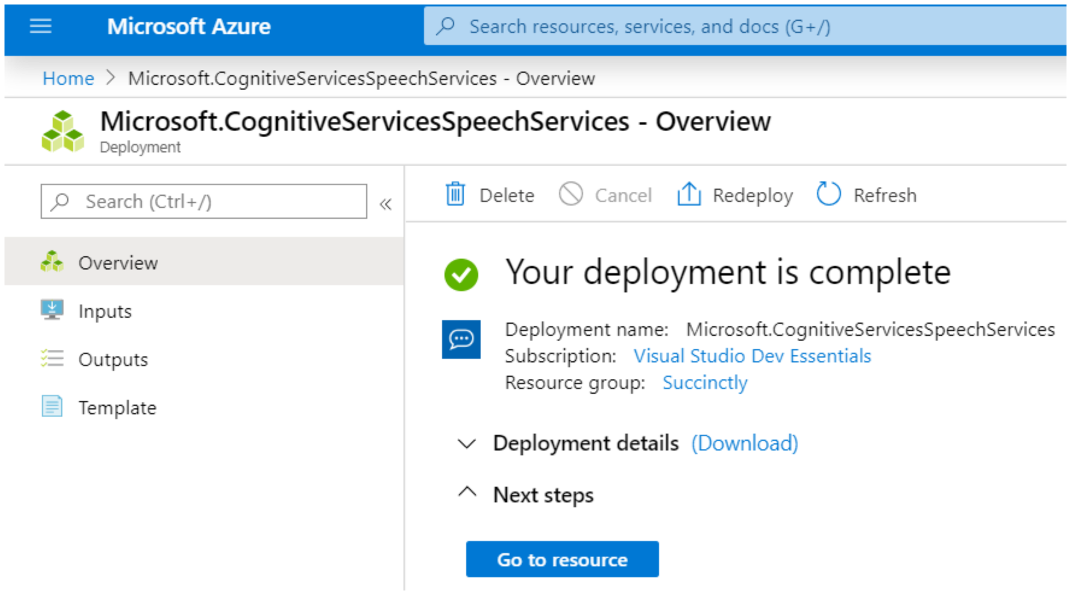
Pricing tier (View full pricing details) *
F0

Resource group *
Succinctly

Figure 4-c: Creating a Speech Service

Enter the required field values. Notice that I've chosen the **F0** option, which indicates the free pricing tier, and I've also reused the **Succinctly** resource group that had been previously created.

When you're done, click **Create** to finalize the creation of the Speech service. Once the deployment of the Azure resource has been finalized, you will see the following screen.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Microsoft.CognitiveServicesSpeechServices - Overview
Microsoft.CognitiveServicesSpeechServices - Overview
Deployment
Search (Ctrl+/)
Delete
Cancel
Redeploy
Refresh
Overview
Inputs
Outputs
Template

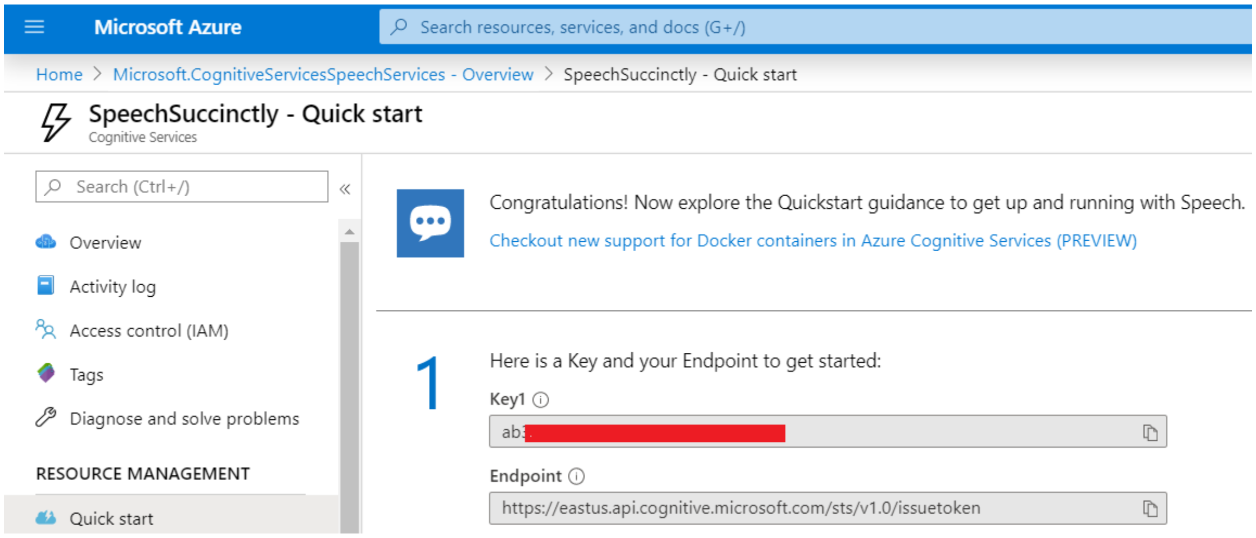
Your deployment is complete

Deployment name: Microsoft.CognitiveServicesSpeechServices
Subscription: Visual Studio Dev Essentials
Resource group: Succinctly

Deployment details (Download)
Next steps
Go to resource

Figure 4-d: Speech Service Ready

Click **Go to resource** to access the service's key and endpoint.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Microsoft.CognitiveServicesSpeechServices - Overview > SpeechSuccinctly - Quick start
SpeechSuccinctly - Quick start
Cognitive Services
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
RESOURCE MANAGEMENT
Quick start

Congratulations! Now explore the Quickstart guidance to get up and running with Speech.
Checkout new support for Docker containers in Azure Cognitive Services (PREVIEW)

1 Here is a Key and your Endpoint to get started:

Key1: ab[redacted]

Endpoint: https://eastus.api.cognitive.microsoft.com/sts/v1.0/issuetoken

Figure 4-e: Speech Service Quick Start Page

Now that we have created the Speech service, it's time to work on the Visual Studio project.

Setting up the Speech SDK

Before we can write any code, we need to set up the Speech SDK by adding it to our Visual Studio project. We can do this from the **Solution Explorer** by right-clicking **References** and selecting **Manage NuGet Packages**. Once you do that, you'll see the screen shown in Figure 4-f.

To install the Speech SDK (**Microsoft.CogntiveServices.Speech**), click the **Browse** tab, enter **speech** in the search bar, and then click **Install**.

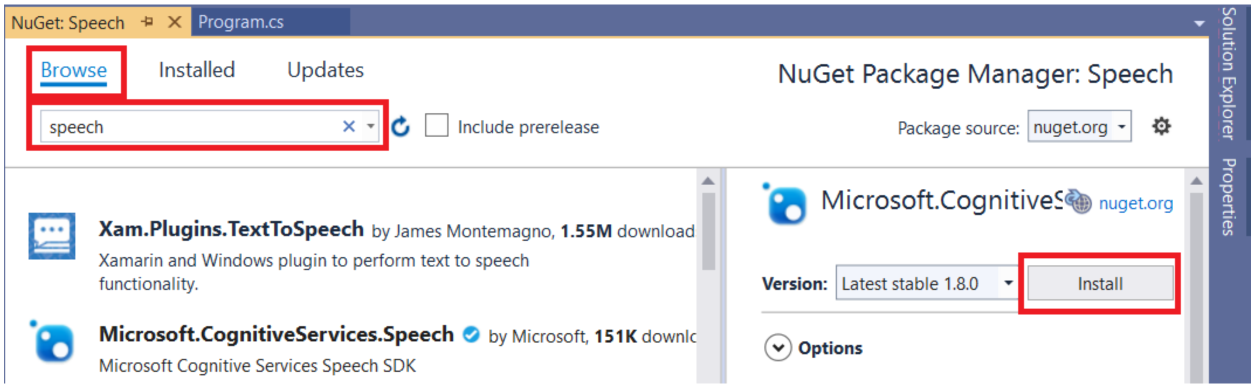

NuGet: Speech
Program.cs
Browse
Installed
Updates
NuGet Package Manager: Speech
speech
Include prerelease
Package source: nuget.org
Solution Explorer
Properties
Xam.Plugins.TextToSpeech by James Montemagno, 1.55M download
Xamarin and Windows plugin to perform text to speech functionality.
Microsoft.CognitiveServices.Speech by Microsoft, 151K downlc
Microsoft Cognitive Services Speech SDK
Microsoft.CognitiveS nuget.org
Version: Latest stable 1.8.0
Install
Options

Figure 4-f: NuGet Package Manager—Speech SDK

Once you've installed the SDK, we can start to write some code.

Text to speech

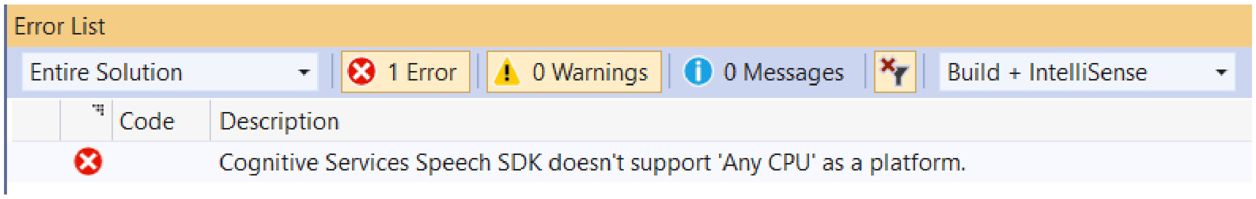
Listing 4-a shows the full code for the text-to-speech program, which creates a speech synthesizer using the default speaker as audio output.

Please make sure to replace `cKey` with the value of your Speech service subscription key, and if you're not using the East US (`eastus`) region, set the value of `cRegion` to another Azure region.

Code Listing 4-a: Text to Speech (Program.cs)

```
using System;
using System.Threading.Tasks;
using Microsoft.CognitiveServices.Speech;

namespace Speech
{
    class Program
    {
        private const string cKey = "<< Key goes here >>";
        private const string cRegion = "eastus"; // Azure region

        public static async Task TextToSpeechSynthesisAsync(string text)
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            // Speech synthesizer using the speaker as audio output
            using (var synthesizer = new SpeechSynthesizer(config))
            {
                using (var r = await synthesizer.SpeakTextAsync(text))
                {
                    if (r.Reason ==
                        ResultReason.SynthesizingAudioCompleted)
                        Console.WriteLine($"Speech synthesized " +
                            $"to speaker for text [{text}]");
                    else if (r.Reason == ResultReason.Canceled)
                    {
                        var cancellation =
                        SpeechSynthesisCancellationDetails.FromResult(r);
                        Console.WriteLine($"CANCELED: " +
                            $"Reason={cancellation.Reason}");

                        if (cancellation.Reason ==
                            CancellationReason.Error)
                        {
                            Console.WriteLine($"Cancelled with " +
                                $"Error Code {cancellation.ErrorCode}");
                            Console.WriteLine($"Cancelled with " +
                                $"Error Details " +
                                $"[{cancellation.ErrorDetails}]");
                        }
                    }
                }

                Console.WriteLine("Waiting to play " +
                    "back to the audio...");
                Console.ReadKey();
            }
        }

        static void Main()
        {
            TextToSpeechSynthesisAsync("Hey, how are you? " +
                "Are you going out now with Cathy?").Wait();
        }
    }
}
```

If you attempt to run this code, you'll probably get the following error.



Error List

Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense

Code	Description
	Cognitive Services Speech SDK doesn't support 'Any CPU' as a platform.

Figure 4-g: Any CPU Platform Error—Speech SDK

This is because, by default, Visual Studio set our project to compile to **Any CPU**, as you can see in the following figure.

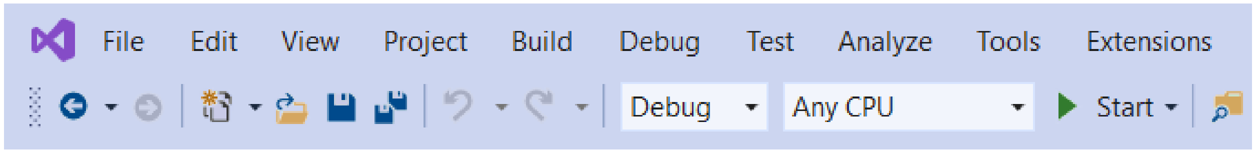


Figure 4-h: Any CPU Setting—Visual Studio 2019

The error is very descriptive. Essentially it means that you must choose a target platform to compile the code to. You can do this by clicking the drop-down arrow next to **Any CPU**.

Go ahead and do that—you should see something like the following options.

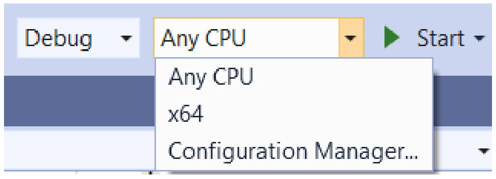
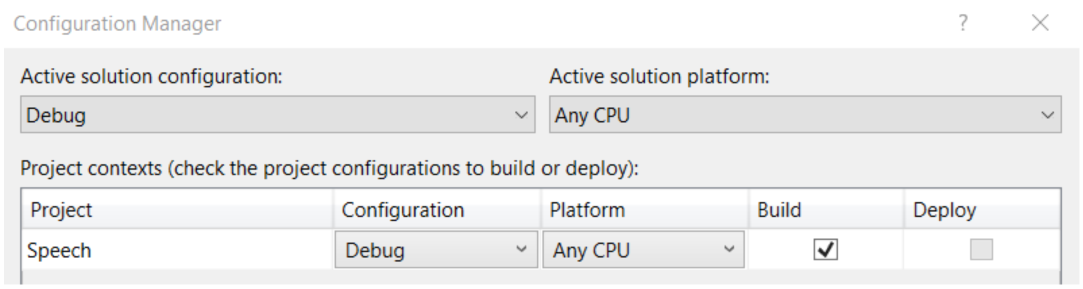


Figure 4-i: Additional Platform Settings—Visual Studio 2019

Then, click the **Configuration Manager** option to set up a target platform if you don't have one set up already.

In my case, I had previously set up **x64** as a target platform (which would compile my code to 64 bits), so I could have chosen this one from the list.

After you have clicked the **Configuration Manager** option, you should see a screen similar to the following one.



Configuration Manager

Active solution configuration: Debug

Active solution platform: Any CPU

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
Speech	Debug	Any CPU	✓	

Figure 4-j: Configuration Manager (Before Selection)—Visual Studio 2019

In the **Platform** drop-down menu, click **New**, which you can see in the following figure.

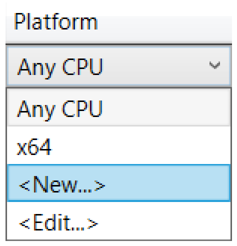
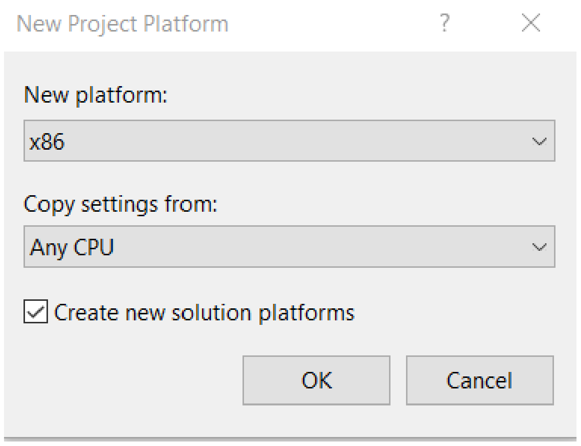


Figure 4-k: Platform Drop-Down—Visual Studio 2019

After you've clicked **New**, you'll see the screen shown in Figure 4-l. You can choose to use the default **x86** platform (32 bits) or select **x64** (64 bits) from the **New** platform drop-down list, depending on which operating system architecture you need to target.

If you are using a 32-bit operating system, you must choose **x86**. However, if your operating system is 64 bits, you can choose to target **x86** or **x64** (preferably the latter).



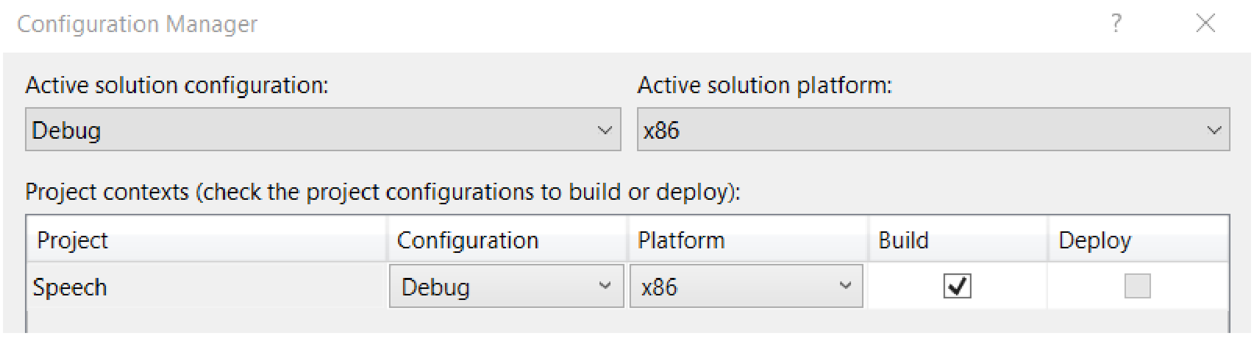
New platform: x86

Copy settings from: Any CPU

☑ Create new solution platforms

Figure 4-l: New Project Platform—Visual Studio 2019

Once you've selected an option, click **OK**. Now you should be able to see that the **Speech** project was assigned to the build option you selected, as shown here.



Active solution configuration: Debug

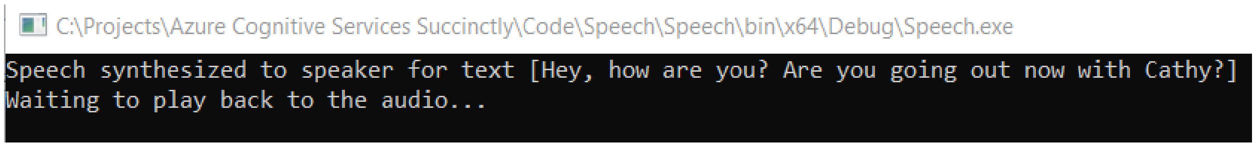
Active solution platform: x86

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
Speech	Debug	x86	✓	☐

Figure 4-m: Configuration Manager (After Selection)—Visual Studio 2019

With the right build option selected, you are now ready to compile your code. So, let’s run the project to see what we get.


C:\Projects\Azure Cognitive Services Succinctly\Code\Speech\Speech\bin\x64\Debug\Speech.exe

```
Speech synthesized to speaker for text [Hey, how are you? Are you going out now with Cathy?]
Waiting to play back to the audio...
```

Figure 4-n: Running the Code

If you have your computer speakers on, you should have heard the following text being synthesized to speech: "Hey, how are you? Are you going out now with Cathy?"

Text to audio

Now let’s take this code a step further and instead of directly synthesizing the written text to speech, we’ll have the code create an audio file that can be used to play the written text.

As we should be able to reuse most of the existing logic we already have, let’s refactor the code we’ve already written as follows.

Code Listing 4-b: Program.cs (Code Refactored)

```
using System;
using System.Threading.Tasks;
using Microsoft.CognitiveServices.Speech;

namespace Speech
{
    class Program
    {
        private const string cKey = "<< Key goes here >>";
        private const string cRegion = "eastus"; // Azure region

        public static async Task TextToSpeechSynthesisAsync(string text)
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (var synthesizer = new SpeechSynthesizer(config))
                await Synthesize(text, synthesizer);
        }

        private static async Task Synthesize(string text,
            SpeechSynthesizer synthesizer)
        {
            using (var r = await synthesizer.SpeakTextAsync(text))
            {
                if (r.Reason == ResultReason.SynthesizingAudioCompleted)
```

```
                    Console.WriteLine($"Speech synthesized " +
                        $"to speaker for text [{text}]");
                else if (r.Reason == ResultReason.Canceled)
                {
                    var cancellation =
                        SpeechSynthesisCancellationDetails.FromResult(r);
                    Console.WriteLine($"CANCELED: " +
                        $"Reason={cancellation.Reason}");

                    if (cancellation.Reason == CancellationReason.Error)
                    {
                        Console.WriteLine($"Cancelled with " +
                            $"Error Code {cancellation.ErrorCode}");
                        Console.WriteLine($"Cancelled with " +
                            $"Error Details " +
                            $"[{cancellation.ErrorDetails}]");
                    }
                }
            }

            Console.WriteLine("Waiting to play " +
                "back to the audio...");
            Console.ReadKey();
        }

        static void Main()
        {
            string txt = "Hey, how are you? " +
                "Are you going out now with Cathy?";

            TextToSpeechSynthesisAsync(txt).Wait();
        }
    }
}
```

As you might have noticed, all we've done is extract the lines of code that made up the core logic of the **TextToSpeechSynthesisAsync** method into a new method called **Synthesize**.

The reason we want to do this is that we are going to add a new method called **TextToAudioFileAsync**, which will also invoke the logic contained within the **Synthesize** method.

But before we can add the logic for the **TextToAudioFileAsync** method, let's reference a couple of additional namespaces our new code will need, which I've highlighted in bold in the following code listing.

Code Listing 4-c: Program.cs (Using Statements—New Items)

```
using System;
using System.Threading.Tasks;
using Microsoft.CognitiveServices.Speech;

using System.IO;
using Microsoft.CognitiveServices.Speech.Audio;
```

Now, let's add the code for the **`TextToAudioFileAsync`** method, which we can see as follows.

Code Listing 4-d: Program.cs (TextToAudioFileAsync Method)

```
public static async Task TextToAudioFileAsync(string fn, string text)
{
    var config = SpeechConfig.FromSubscription(cKey, cRegion);

    using (FileStream f = new FileStream(fn, FileMode.Create))
        using (BinaryWriter wr = new BinaryWriter(f))
            wr.Write(System.Text.Encoding.ASCII.GetBytes("RIFF"));

    using (var fo = AudioConfig.FromWavFileOutput(fn))
        using (var synthesizer = new SpeechSynthesizer(config, fo))
            await Synthesize(text, synthesizer);
}
```

Notice how the code of the **`TextToAudioFileAsync`** method looks almost identical to the code of the **`TextToSpeechSynthesisAsync`** method. I've highlighted the differences in bold.

The **`TextToAudioFileAsync`** method first creates an empty audio (.wav) file, which is done by creating a new **`FileStream`** instance, that gets passed the name of the file (**`fn`**) as a parameter.

This **`FileStream`** instance (**`f`**) is passed as a parameter when creating a new **`BinaryWriter`** instance. Then, the **`BinaryWriter`** object (**`wr`**) is used to write the content of the file, which is done by executing the **`System.Text.Encoding.ASCII`**.

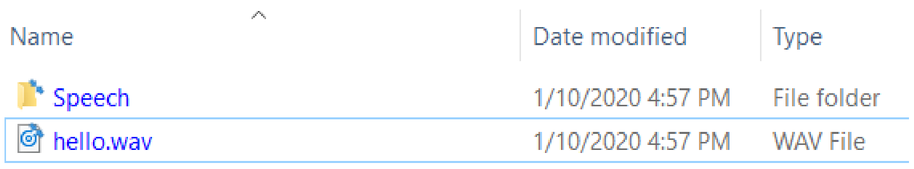
Then, the **`AudioConfig.FromWavFileOutput`** method is invoked by passing **`fn`** as a parameter. This results in a **`fo`** object, which is then passed to the new **`SpeechSynthesizer`** instance as a parameter.

The **`TextToAudioFileAsync`** method can be invoked from the **`Main`** method as follows.

Code Listing 4-e: Program.cs (Main Method)

```
static void Main()
{
    string txt = "Hey, how are you? " +
        "Are you going out now with Cathy?";

    string fn = @"C:\Test\hello.wav";

    TextToAudioFileAsync(txt, fn).Wait();
}
```

If we execute the program, we should get an audio file containing the synthesized written text. Let's have a look.



Name	Date modified	Type
Speech	1/10/2020 4:57 PM	File folder
hello.wav	1/10/2020 4:57 PM	WAV File

Figure 4-o: Newly Created Audio File

If you now double-click the file, you should be able to listen to the synthesized text. Isn't that cool? Personally, I think this is impressive, given that there is very little code involved—that's the power of Azure Cognitive Services.

Text to speech and audio code

Just to make sure that we have a complete overview of what we've done, the following listing shows the full source code of what we have written for achieving the text-to-speech and text-to-audio functionalities.

Code Listing 4-f: Program.cs (Full Source Code)

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.CognitiveServices.Speech;
using Microsoft.CognitiveServices.Speech.Audio;

namespace Speech
{
    class Program
    {
        private const string cKey = "<< Key goes here >>";
        private const string cRegion = "eastus"; // Azure region

        public static async Task TextToSpeechSynthesisAsync(string text)
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (var synthesizer = new SpeechSynthesizer(config))
                await Synthesize(text, synthesizer);
        }

        public static async Task TextToAudioFileAsync(string text,
            string fn)
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (FileStream f = new FileStream(fn, FileMode.Create))
                using (BinaryWriter wr = new BinaryWriter(f))
                    wr.Write(
                        System.Text.Encoding.ASCII.GetBytes("RIFF"));

            using (var fo = AudioConfig.FromWavFileOutput(fn))
                using (var synthesizer = new
                    SpeechSynthesizer(config, fo))
                    await Synthesize(text, synthesizer);
        }

        private static async Task Synthesize(string text,
            SpeechSynthesizer synthesizer)
        {
            using (var r = await synthesizer.SpeakTextAsync(text))
            {
                if (r.Reason == ResultReason.SynthesizingAudioCompleted)
                    Console.WriteLine($"Speech synthesized " +
                        $"to speaker for text [{text}]");
                else if (r.Reason == ResultReason.Canceled)
                {
                    var cancellation =
                        SpeechSynthesisCancellationDetails.FromResult(r);
                    Console.WriteLine($"CANCELED: " +
                        $"Reason={cancellation.Reason}");

                    if (cancellation.Reason == CancellationReason.Error)
                    {
                        Console.WriteLine($"Cancelled with " +
                            $"Error Code {cancellation.ErrorCode}");
                        Console.WriteLine($"Cancelled with " +
                            $"Error Details " +
                            $"[{cancellation.ErrorDetails}]");
                    }
                }
            }

            Console.WriteLine("Waiting to play " +
                "back to the audio...");
            Console.ReadKey();
        }
        static void Main()
        {
            string txt = "Hey, how are you? " +
                "Are you going out now with Cathy?";

            string fn = @"C:\Test\hello.wav";

            TextToSpeechSynthesisAsync(txt).Wait();
            TextToAudioFileAsync(txt, fn).Wait();
        }
    }
}
```

Speech to text

Now that we've covered how to convert text to speech and to save text to an audio file, it's now time to see how we can use the speech-to-text capability.

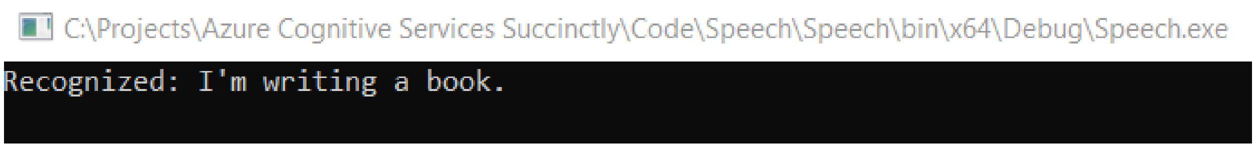
To do that, we'll continue to work on our **Program.cs** file. As you'll see shortly, the process is almost identical to what we have done, but instead of a **`SpeechSynthesizer`**, we'll be using a **`SpeechRecognizer`** class.

Let's add the following two new methods, which will be able to recognize speech and output text.

Code Listing 4-g: Program.cs (Full Source Code)

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.CognitiveServices.Speech;
using Microsoft.CognitiveServices.Speech.Audio;

namespace Speech
{
    class Program
    {
        private const string cKey = "<< Key goes here >>";
        private const string cRegion = "eastus"; // Azure region

        // Previous text to speech and audio methods

        public static async Task SpeechToTextAsync()
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (var recognizer = new SpeechRecognizer(config))
                await Recognize(recognizer);
        }

        private static async Task Recognize(SpeechRecognizer recognizer)
        {
            var result = await recognizer.RecognizeOnceAsync();

            if (result.Reason == ResultReason.RecognizedSpeech)
                Console.WriteLine($"Recognized: {result.Text}");
            else if (result.Reason == ResultReason.NoMatch)
                Console.WriteLine("Speech could not be recognized.");
            else if (result.Reason == ResultReason.Canceled)
            {
                var cancellation =
                    CancellationDetails.FromResult(result);
                Console.WriteLine
                    ($"Cancelled due to reason={cancellation.Reason}");

                if (cancellation.Reason == CancellationReason.Error)
                {
                    Console.WriteLine
                        ($"Error code={cancellation.ErrorCode}");
                    Console.WriteLine
                        ($"Error details={cancellation.ErrorDetails}");
                    Console.WriteLine
                        ($"Did you update the subscription info?");
                }
            }
        }

        static void Main()
        {
            SpeechToTextAsync().Wait();
            Console.ReadLine();
        }
    }
}
```

If we now run the program with this code and speak a few words—in my case, I'll say "I'm writing a book"—we should be able to see the code recognizing the spoken words. The following figure shows what mine recognized. How cool is that?


C:\Projects\Azure Cognitive Services Succinctly\Code\Speech\Speech\bin\x64\Debug\Speech.exe

```
Recognized: I'm writing a book.
```

Figure 4-p: My Spoken Words Recognized as Text

Let's try to understand what we have done here. Basically, the **SpeechToTextAsync** method creates a **SpeechRecognizer** instance and then invokes the **Recognize** method, passing that **SpeechRecognizer** object as a parameter.

The logic of the **Recognize** method is very similar to the logic of the **Synthesize** method. The main difference is that instead of calling the **SpeakTextAsync** method, the **RecognizeOnceAsync** method is invoked.

The other difference is that the conditional statements of the method check the value of **result.Reason** and compare that to the value of **ResultReason.RecognizedSpeech**, instead of **ResultReason.SynthesizingAudioCompleted** (as there isn't synthesized completed audio to check because it is being recognized instead).

Audio to text

Using the audio file that we previously created, let's write some code to read it and recognize the recorded speech as text.

So, let's expand **Program.cs** to do that. Let's create an **AudioToTextAsync** method that will read the content of the audio file and invoke the Speech service.

Code Listing 4-h: Program.cs (AudioToTextAsync Method)

```
public static async Task AudioToTextAsync(string fn)
{
    var config = SpeechConfig.FromSubscription(cKey, cRegion);

    using (var ai = AudioConfig.FromWavFileInput(fn))
        using (var recognizer = new SpeechRecognizer(config, ai))
            await Recognize(recognizer);
}
```

As you can see, the code of the **AudioToTextAsync** method looks almost identical to the **SpeechToTextAsync** method.

The differences are highlighted in bold. Basically, the name of the audio file (**fn**) that is going to be read is passed to the method.

The file is read by invoking the **AudioConfig.FromWavFileInput** method, which returns an audio input (**ai**) object that is passed as a parameter when a new instance of the **SpeechRecognizer** class is created.

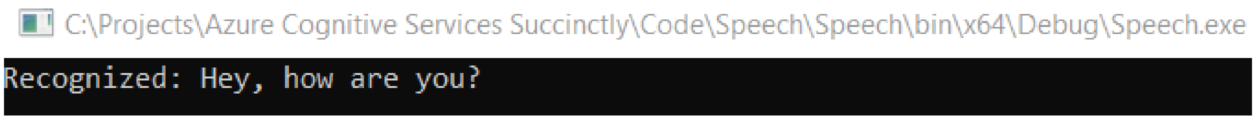
We can invoke the **AudioToTextAsync** method from the **Main** method as follows. Notice how I've left the original text that was recorded as comments.

Code Listing 4-i: Program.cs (Main Method)

```
static void Main()
```

```
{
    // string txt = "Hey, how are you? " +
    // "Are you going out now with Cathy?";

    string fn = @"C:\Test\hello.wav";

    AudioToTextAsync(fn).Wait();
    Console.ReadLine();
}
```

Let’s now execute the program and see what results we get.



```
C:\Projects\Azure Cognitive Services Succinctly\Code\Speech\Speech\bin\x64\Debug\Speech.exe
Recognized: Hey, how are you?
```

Figure 4-q: Recorded Audio Recognized as Text

We can see that the recorded audio has been recognized correctly. But what happened to the second sentence—"Are you going out now with Cathy?”

The **`RecognizeOnceAsync`** method within **`Recognize`** returns one utterance (sentence) only. It basically stops recognizing when it detects a pause (silence)—thus, the second utterance was not recognized.

To recognize all the sentences that have been recorded within the audio file, we need to use the **`StartContinuousRecognitionAsync`** method instead.

Let’s add some logic to achieve that. We can do this by creating a new method called **`AudioToTextContinuousAsync`**.

Code Listing 4-j: Program.cs (AudioToTextContinuousAsync Method)

```
public static async Task AudioToTextContinuousAsync(string fn)
{
    var config = SpeechConfig.FromSubscription(cKey, cRegion);

    using (var ai = AudioConfig.FromWavFileInput(fn))
        using (var recognizer = new SpeechRecognizer(config, ai))
            await RecognizeAll(recognizer);
}
```

As you can see, this method is almost identical to the **`AudioToTextAsync`** method; the only difference is that **`AudioToTextContinuousAsync`** invokes **`RecognizeAll`** instead of **`Recognize`**.

So, let’s have a look at the **`RecognizeAll`** method to see what it does.

Code Listing 4-k: Program.cs (RecognizeAll Method)

```
private static async Task RecognizeAll(SpeechRecognizer recognizer)
{
    var taskCompletetion = new TaskCompletionSource<int>();

    // Events
    recognizer.Recognizing += (sender, eventargs) =>
    {
        // Handle recognized intermediate result
    };

    recognizer.Recognized += (sender, eventargs) =>
    {
        if (eventargs.Result.Reason == ResultReason.RecognizedSpeech)
            Console.WriteLine($"Recognized: {eventargs.Result.Text}");
    };

    recognizer.Canceled += (sender, eventargs) =>
    {
        if (eventargs.Reason == CancellationReason.Error)
            Console.WriteLine("Error reading the audio file.");

        if (eventargs.Reason == CancellationReason.EndOfStream)
            Console.WriteLine("End of file.");

        taskCompletetion.TrySetResult(0);
    };

    recognizer.SessionStarted += (sender, eventargs) =>
    {
        // Started recognition session
    };

    recognizer.SessionStopped += (sender, eventargs) =>
    {
        // Ended recognition session
        taskCompletetion.TrySetResult(0);
    };

    // Starts recognition
    await recognizer.
       StartContinuousRecognitionAsync().ConfigureAwait(false);

    // Waits for completion
    Task.WaitAny(new[] { taskCompletetion.Task });

    // Stops recognition
    await recognizer.StopContinuousRecognitionAsync();
}
```

That’s quite a bit of code! What is happening here?

First, as this is a continuous async operation, we need to create a **`TaskCompletionSource`** instance, which we will use to wait for the completion of the task.

Then, to be able to follow along with what happens, we need to subscribe to various events, such as the **`Recognizing`**, **`Recognized`**, **`Canceled`**, **`SessionStarted`**, and **`SessionStopped`** events.

Next, we hook up the following lambda function, which displays each utterance recognized by the Speech service.

```
recognizer.Recognized += (sender, eventargs) =>
{
    if (eventargs.Result.Reason == ResultReason.RecognizedSpeech)
        Console.WriteLine($"Recognized: {eventargs.Result.Text}");
};
```

Each recognized utterance is passed to the lambda function as a parameter and is accessible through the **`eventargs.Result.Text`** property.

Then, on the **`Canceled`** event, we check if there’s a problem reading the audio file, or if we have reached the end of the file. This is achieved as follows.

```
recognizer.Canceled += (sender, eventargs) =>
{
    if (eventargs.Reason == CancellationReason.Error)
        Console.WriteLine("Error reading the audio file.");

    if (eventargs.Reason == CancellationReason.EndOfStream)
        Console.WriteLine("End of file.");

    taskCompletetion.TrySetResult(0);
};
```

Notice also how the **`taskCompletion`** object’s state is set to zero by executing this instruction, **`taskCompletion.TrySetResult(0)`**, in order to indicate that the continuous running process has nothing more to wait for.

On the **`SessionStopped`** event, we also need to set the **`taskCompletion`** object’s state to zero, which is done as follows.

```
recognizer.SessionStopped += (sender, eventargs) =>
{
    //Ended recognition session
    taskCompletetion.TrySetResult(0);
};
```

We could also add some logic to the **Recognizing** and **SessionStarted** events. These could be useful for processing intermediate results, such as adjusting the output text—for example, if the audio would contain profanity—by intercepting the text and changing the output, but in this case, we are simply outputting everything as is.

The final three instructions are where the magic happens.

```
// Starts recognition
await recognizer.StartContinuousRecognitionAsync().ConfigureAwait(false);

// Waits for completion
Task.WaitAny(new[] { taskCompletetion.Task });

// Stops recognition
await recognizer.StopContinuousRecognitionAsync();
```

The call to the **StartContinuousRecognitionAsync** method starts the continuous speech recognition. Then, the call to the **Task.WaitAny** method simply waits for the continuous recognition process to complete. Finally, the call to the **StopContinuousRecognitionAsync** method finalizes the continuous recognition process.

Before we run the program, let’s check the following full source code for **Program.cs** with all the changes made.

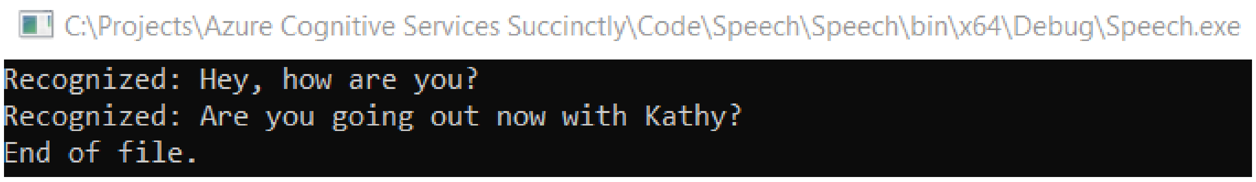
Code Listing 4-l: Program.cs (Full Source Code)

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.CognitiveServices.Speech;
using Microsoft.CognitiveServices.Speech.Audio;

namespace Speech
{
    class Program
    {
        private const string cKey = "<< Key goes here >>";
        private const string cRegion = "eastus"; // Azure region

        public static async Task TextToSpeechSynthesisAsync(string text)
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (var synthesizer = new SpeechSynthesizer(config))
                await Synthesize(text, synthesizer);
        }

        public static async Task TextToAudioFileAsync(string text,
            string fn)
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (FileStream f = new FileStream(fn, FileMode.Create))
                using (BinaryWriter wr = new BinaryWriter(f))
                    wr.Write(
                        System.Text.Encoding.ASCII.GetBytes("RIFF"));

            using (var fo = AudioConfig.FromWavFileOutput(fn))
                using (var synthesizer = new SpeechSynthesizer(
                    config, fo))
                    await Synthesize(text, synthesizer);
        }

        private static async Task Synthesize(string text,
            SpeechSynthesizer synthesizer)
        {
            using (var r = await synthesizer.SpeakTextAsync(text))
            {
                if (r.Reason == ResultReason.SynthesizingAudioCompleted)
                    Console.WriteLine($"Speech synthesized " +
                        $"to speaker for text [{text}]");
                else if (r.Reason == ResultReason.Canceled)
                {
                    var cancellation =
                        SpeechSynthesisCancellationDetails.FromResult(r);
                    Console.WriteLine($"CANCELED: " +
                        $"Reason={cancellation.Reason}");

                    if (cancellation.Reason == CancellationReason.Error)
                    {
                        Console.WriteLine($"Cancelled with " +
                            $"Error Code {cancellation.ErrorCode}");
                        Console.WriteLine($"Cancelled with " +
                            $"Error Details " +
                            $"[{cancellation.ErrorDetails}]");
                    }
                }
            }

            Console.WriteLine("Waiting to play " +
                "back to the audio...");
            Console.ReadKey();
        }

        public static async Task SpeechToTextAsync()
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (var recognizer = new SpeechRecognizer(config))
```

```
                await Recognize(recognizer);
        }

        public static async Task AudioToTextAsync(string fn)
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (var ai = AudioConfig.FromWavFileInput(fn))
                using (var recognizer = new SpeechRecognizer(config, ai))
                    await Recognize(recognizer);
        }

        public static async Task AudioToTextContinuousAsync(string fn)
        {
            var config = SpeechConfig.FromSubscription(cKey, cRegion);

            using (var ai = AudioConfig.FromWavFileInput(fn))
                using (var recognizer = new SpeechRecognizer(config, ai))
                    await RecognizeAll(recognizer);
        }

        private static async Task RecognizeAll(
            SpeechRecognizer recognizer)
        {
            var taskCompletetion = new TaskCompletionSource<int>();

            // Events
            recognizer.Recognizing += (sender, eventargs) =>
            {
                // Handle recognized intermediate result
            };

            recognizer.Recognized += (sender, eventargs) =>
            {
                if (eventargs.Result.Reason ==
                    ResultReason.RecognizedSpeech)
                    Console.WriteLine
                        ($"Recognized: {eventargs.Result.Text}");
            };

            recognizer.Canceled += (sender, eventargs) =>
            {
                if (eventargs.Reason == CancellationReason.Error)
                    Console.WriteLine("Error reading the audio file.");

                if (eventargs.Reason == CancellationReason.EndOfStream)
                    Console.WriteLine("End of file.");

                taskCompletetion.TrySetResult(0);
            };

            recognizer.SessionStarted += (sender, eventargs) =>
            {
                // Started recognition session
            };

            recognizer.SessionStopped += (sender, eventargs) =>
            {
                // Ended recognition session
                taskCompletetion.TrySetResult(0);
            };

            // Starts continuous recognition
            await recognizer.
                StartContinuousRecognitionAsync().ConfigureAwait(false);

            // Waits for completion
            Task.WaitAny(new[] { taskCompletetion.Task });

            // Stops recognition
            await recognizer.StopContinuousRecognitionAsync();
        }

        private static async Task Recognize(SpeechRecognizer recognizer)
        {
            var result = await recognizer.RecognizeOnceAsync();

            if (result.Reason == ResultReason.RecognizedSpeech)
                Console.WriteLine($"Recognized: {result.Text}");
            else if (result.Reason == ResultReason.NoMatch)
                Console.WriteLine("Speech could not be recognized.");
            else if (result.Reason == ResultReason.Canceled)
            {
                var cancellation =
                    CancellationDetails.FromResult(result);
                Console.WriteLine
                    ($"Cancelled due to reason={cancellation.Reason}");

                if (cancellation.Reason == CancellationReason.Error)
                {
                    Console.WriteLine
                        ($"Error code={cancellation.ErrorCode}");
                    Console.WriteLine
                        ($"Error details={cancellation.ErrorDetails}");
                    Console.WriteLine
                        ($"Did you update the subscription info?");
                }
            }
        }

        static void Main()
        {
            string txt = "Hey, how are you? " +
                "Are you going out now with Cathy?";

            string fn = @"C:\Test\hello.wav";

            // TextToSpeechSynthesisAsync(txt).Wait();
            // TextToAudioFileAsync(txt, fn).Wait();

            // SpeechToTextAsync().Wait();
            // AudioToTextAsync(fn).Wait();

            AudioToTextContinuousAsync(fn).Wait();
            Console.ReadLine();
        }
    }
}
```

Notice how I've commented out all the method calls within the **`Main`** method, except for the one that invokes **`AudioToTextContinuousAsync`**.

This way, we can execute the program and see if we are able to read the complete audio file. Let's give it a go.



```
C:\Projects\Azure Cognitive Services Succinctly\Code\Speech\Speech\bin\x64\Debug\Speech.exe
Recognized: Hey, how are you?
Recognized: Are you going out now with Kathy?
End of file.
```

Figure 4-r: Recorded Audio Recognized as Text (Continuous Processing)

The program was able to recognize all the audio as text correctly. The only difference is a **K** instead of a **C** for the word **Cathy**, which if you think about it, is really nothing, as it's pronounced the same way in both cases.

Summary

Throughout this chapter, we've explored how to convert text to speech and speech to text using the Speech SDK along with the Speech service from Azure. Considering what we've accomplished with relatively few lines of code, it's quite amazing what can be achieved.

There’s still quite a lot you can explore with this awesome Azure service, such as being able to perform intent recognition, speech translation, conversation transcription, and how to use the service to create voice assistants.

Beyond the Speech service, we still have a bit to explore with Cognitive Services, such as how to extract information from scanned documents using Computer Vision. We’ll also use some of these Computer Vision services to do other cool things, such as generate thumbnails from images and perform image analysis—exciting stuff!

Chapter 5 Vision

Quick intro

Computer vision is one of the most exciting aspects of AI, and for me, one of the coolest sets of services and APIs that Azure offers.

The Computer Vision API, as its name implies, is an AI service that analyzes content in images. It was the first vision service released by Azure.

It allows developers to create thumbnails, get insights from images, perform optical character recognition (OCR), detect and extract handwritten text in images, and identify field values from images and scanned documents.

Cognitive Services offers other vision services that give developers the ability to detect and identify faces in images and build custom vision models for specific domains.

In this chapter, we'll specifically focus on using the Computer Vision API and see how it can allow us to gather insights from images and extract text contained within them.

Without further ado, let's see how we can put the power of Cognitive Services at our disposal and use it to work with images.

Creating the Computer Vision API service

In order to start exploring the Computer Vision API, we need to create an Azure service instance of it.

To do that, go to the Azure portal, navigate to **All resources**, and click **Create resource**. Then, enter **Computer Vision** in the **Search the Marketplace** search box, and select the **Computer Vision** option from the drop-down menu, which will display the following.

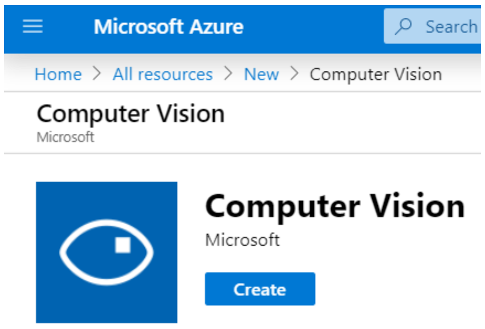
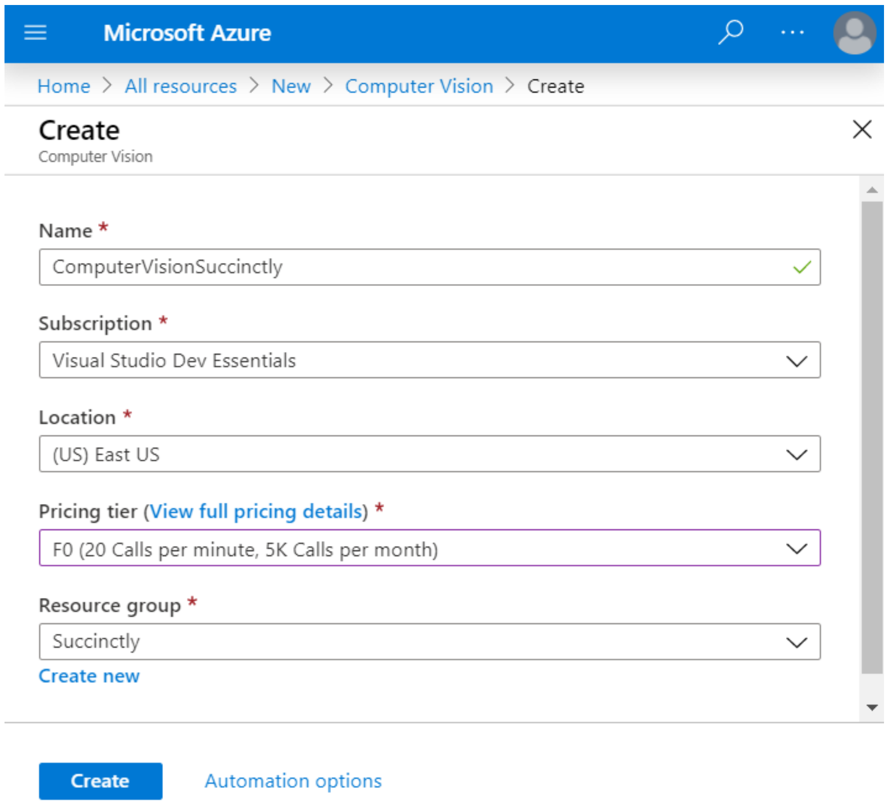


Figure 5-a: Computer Vision API Service

Then, click **Create**—this will display the following screen, which displays the fields that need to be filled in.



Name *
ComputerVisionSuccinctly

Subscription *
Visual Studio Dev Essentials

Location *
(US) East US

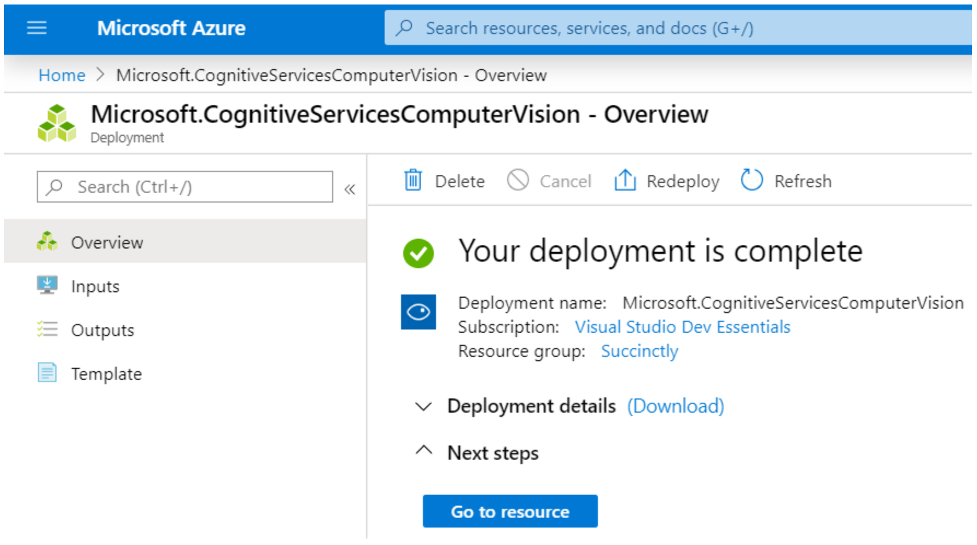
Pricing tier (View full pricing details) *
F0 (20 Calls per minute, 5K Calls per month)

Resource group *
Succinctly

Figure 5-b: Creating a Computer Vision API Service

Enter the required field values. Notice that I've chosen the **F0** option, which indicates the free pricing tier, and I've reused the **Succinctly Resource group** that had been previously created.

When you're done, click **Create** to finalize the creation of the Computer Vision API service. Once the deployment of the Azure resource has been finalized, you will see the following screen.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Microsoft.CognitiveServicesComputerVision - Overview
Microsoft.CognitiveServicesComputerVision - Overview
Deployment
Search (Ctrl+/)
Overview
Inputs
Outputs
Template
Delete
Cancel
Redeploy
Refresh

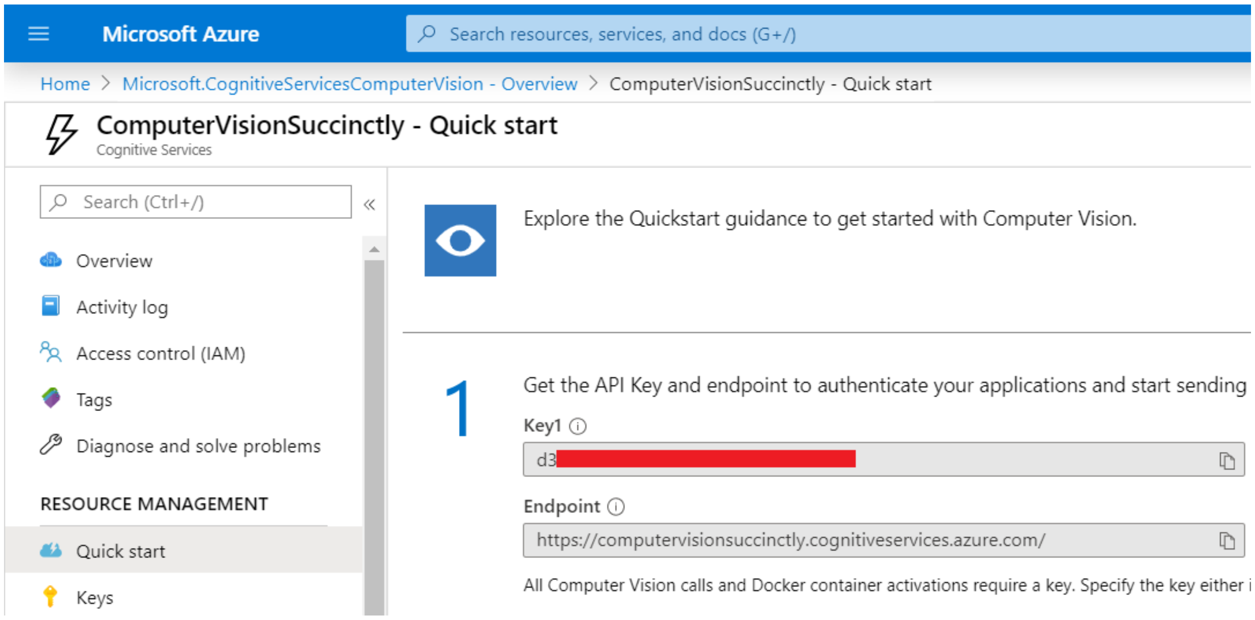
Your deployment is complete

Deployment name: Microsoft.CognitiveServicesComputerVision
Subscription: Visual Studio Dev Essentials
Resource group: Succinctly

Deployment details (Download)
Next steps
Go to resource

Figure 5-c: Computer Vision API Service Ready

Next, click **Go to resource** to get the key and endpoint.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Microsoft.CognitiveServicesComputerVision - Overview > ComputerVisionSuccinctly - Quick start
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
RESOURCE MANAGEMENT
Quick start
Keys

ComputerVisionSuccinctly - Quick start
Cognitive Services

Explore the Quickstart guidance to get started with Computer Vision.

1 Get the API Key and endpoint to authenticate your applications and start sending

Key1
d3[redacted]

Endpoint
https://computervisionsuccinctly.cognitiveservices.azure.com/

All Computer Vision calls and Docker container activations require a key. Specify the key either i

Figure 5-d: Computer Vision API Service Quick Start Page

Now that we have created the Computer Vision API service, it’s time to start to work on the Visual Studio projects we’ll be developing.

Creating VS projects

Since we’ll be learning various aspects of the Computer Vision API, I personally think that it’s better to keep each aspect of the Computer Vision API separated and organized into different Visual Studio projects.

By now, you know how to create console app (.NET Framework) projects with Visual Studio, so we’ll skip that explanation going forward, and start directly with each project’s code.

Accessing the API

Our first Visual Studio project within this chapter is going to focus on how to access the Computer Vision API. So, go ahead and create a **Console App (.NET Framework)** project named **AccessAPI** in Visual Studio.

This is the image that we’ll be analyzing using the Computer Vision API, which is available for download at Pixabay.

Feel free to download and save this image to a local folder (or alternatively, use any other). In my case, I’ll save it as **image.jpg** to a folder called **Test** on my local drive.

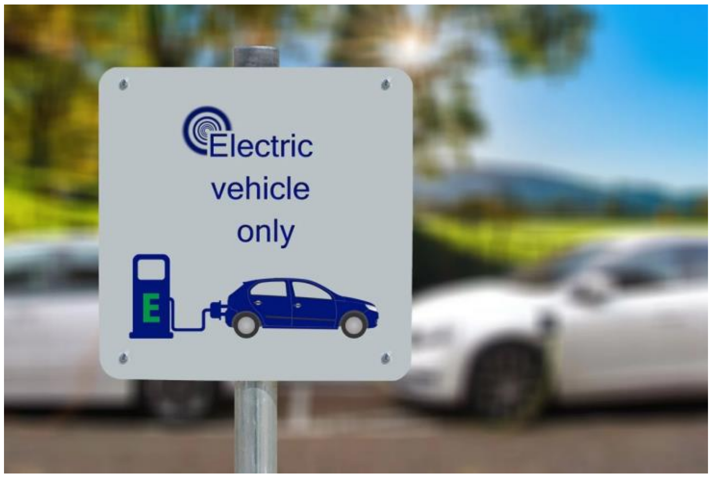


Figure 5-e: Image to Analyze (Courtesy of Pixabay)

Once the image is created, open the **Program.cs** file and replace its content with the following code.

Code Listing 5-a: Program.cs (AccessAPI Project)

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

namespace AccessAPI
{
    static class Program
    {
        private const string subscriptionKey =
            "<< Key goes here >>";
        private const string cEndpoint =
        "https://computervisionsuccinctly.cognitiveservices.azure.com/";
        private static string uriBase =
            $"{cEndpoint}vision/v2.1/analyze";

        static void Main()
        {
            string imageFilePath =
                @"C:\Test\image.jpg";

            MakeAnalysisRequest(imageFilePath);

            Console.WriteLine("\nPlease wait...\n");
            Console.ReadLine();
        }
        public static async void MakeAnalysisRequest(
            string imageFilePath)
        {
            HttpClient client = new HttpClient();
            client.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key",
                subscriptionKey);

            string requestParameters =
            "visualFeatures=Categories,Description,Color&language=en";
            string uri = uriBase + "?" + requestParameters;

            HttpResponseMessage response = null;
            byte[] byteData = GetImageAsByteArray(imageFilePath);

            using (ByteArrayContent content = new
                ByteArrayContent(byteData))
            {
                content.Headers.ContentType = new
                    MediaTypeHeaderValue("application/octet-stream");
                response = await client.PostAsync(uri, content);

                string contentString =
                    await response.Content.ReadAsStringAsync();

                Console.WriteLine("\nResponse:\n");
                Console.WriteLine(JsonPrettyPrint(contentString));
            }
        }

        public static byte[] GetImageAsByteArray(string imageFilePath)
        {
            FileStream fileStream = new FileStream(
                imageFilePath, FileMode.Open, FileAccess.Read);
            BinaryReader binaryReader = new BinaryReader(fileStream);

            return binaryReader.ReadBytes((int)fileStream.Length);
        }

        public static string JsonPrettyPrint(string json)
        {
            if (string.IsNullOrEmpty(json))
                return string.Empty;

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

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

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

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

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

Let’s analyze this code. First, make sure you assign the value of the Computer Vision API subscription key from the Azure portal to **subscriptionKey**.

Notice that the Computer Vision API endpoint has already been assigned to **cEndpoint**. However, the code will be using **uriBase** to invoke the service, and not **cEndpoint**.

The string assigned to **uriBase** contains the endpoint, plus the name of the API version (**v2.1**) and method (**analyze**); this API method performs the analysis on the image.

The **Main** method invokes the **MakeAnalysisRequest** method, which receives the image location (**imageFilePath**) as a parameter.

Now, let’s have a look at the **MakeAnalysisRequest** method. The call to the API is done through an instance of the **HttpClient** class.

The subscription key is added as a header parameter to the **HttpClient** instance by calling the **DefaultRequestHeaders.Add** method.

Following that, the specific analysis features that the API will check for are indicated by assigning **"visualFeatures=Categories,Description,Color&language=en"** to **requestParameters**.

In order to send the image to the API for analysis, the image must be converted to a **byte** array; this is what the **GetImageAsByteArray** method does.

The method does this by invoking the **ReadBytes** method from a **BinaryReader** object that was created with a **FileStream** instance obtained by reading the image file name (**imageFilePath**).

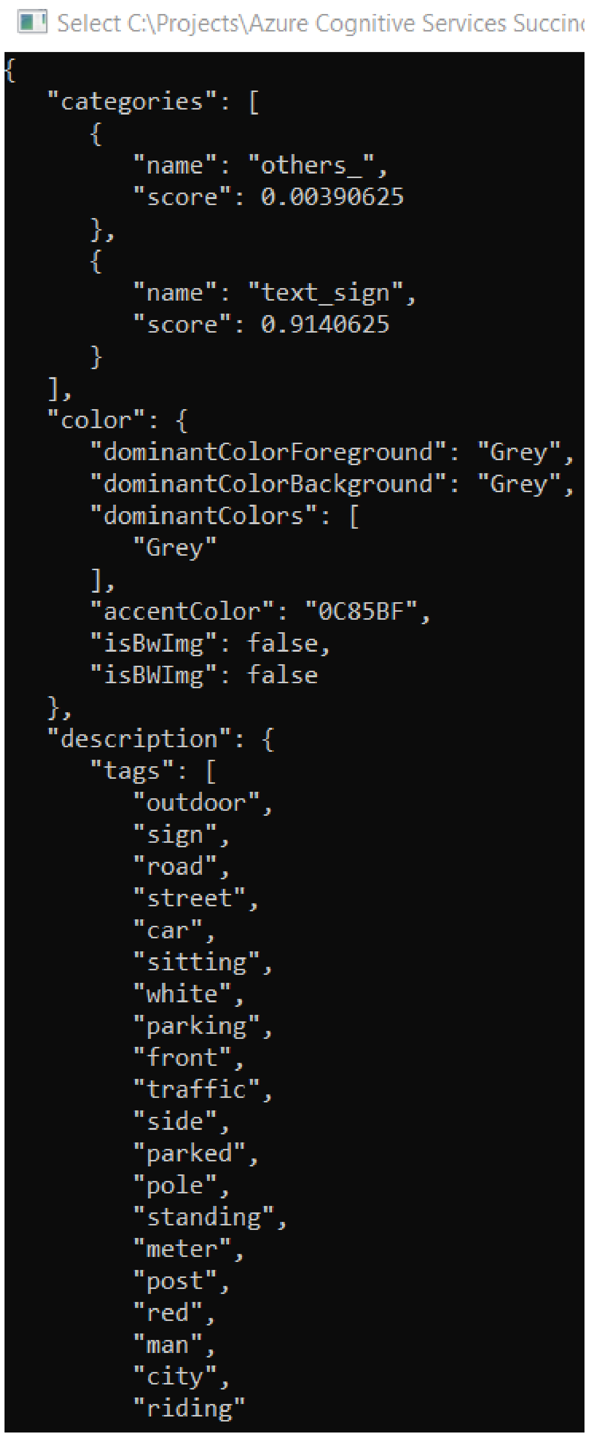
Then, the **byte** array obtained as a result of the call to the **GetImageAsByteArray** method is passed as a parameter when creating a new **ByteArrayContent** instance.

The **ByteArrayContent** object that contains the image information is passed to the API when the **PostAsync** method is invoked.

The result returned by the API is retrieved by calling the **ReadAsStringAsync** method from the **response.Content** object.

Previously we looked at the **JsonPrettyPrint** method—which, as you know, is simply used to print out the API results nicely—so we’ll skip that.

That’s all there is to it. As you have seen, the code wasn’t that complicated. If we execute this code, we get the following result.

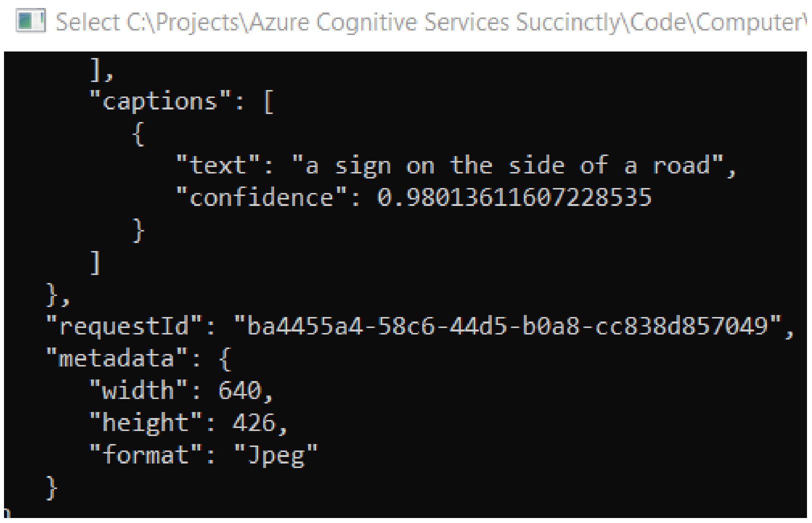

Select C:\Projects\Azure Cognitive Services Succin

```
{
   "categories": [
      {
         "name": "others_",
         "score": 0.00390625
      },
      {
         "name": "text_sign",
         "score": 0.9140625
      }
   ],
   "color": {
      "dominantColorForeground": "Grey",
      "dominantColorBackground": "Grey",
      "dominantColors": [
         "Grey"
      ],
      "accentColor": "0C85BF",
      "isBwImg": false,
      "isBWImg": false
   },
   "description": {
      "tags": [
         "outdoor",
         "sign",
         "road",
         "street",
         "car",
         "sitting",
         "white",
         "parking",
         "front",
         "traffic",
         "side",
         "parked",
         "pole",
         "standing",
         "meter",
         "post",
         "red",
         "man",
         "city",
         "riding"
```

Figure 5-f: Image Analysis Results (Part 1)

Notice how the Computer Vision API was able to detect that the picture corresponds to a sign, and also found the relevant colors of the image. Besides that, various tags that describe the image were identified.

If you scroll down, you'll also see that the Computer Vision API has labeled the image with descriptive text, which clearly indicates what it is, as you can see in the following figure.


Select C:\Projects\Azure Cognitive Services Succinctly\Code\Computer\

```
      ],
      "captions": [
         {
            "text": "a sign on the side of a road",
            "confidence": 0.98013611607228535
         }
      ]
   },
   "requestId": "ba4455a4-58c6-44d5-b0a8-cc838d857049",
   "metadata": {
      "width": 640,
      "height": 426,
      "format": "Jpeg"
   }
```

Figure 5-g: Image Analysis Results (Part 2)

In essence, the Computer Vision API received an image as a set of pixels and came up with a result that includes keywords and properties that describe what those pixels are, which is quite impressive.

Generating thumbnails

Now, let's create a new Visual Studio project, which we can use to generate thumbnails. I'll call this project **CreateThumbnails**.

After you've created the project, go to the **Solution Explorer**, right-click **References**, select **Manage NuGet Packages**, and search for **computer vision** in the search bar. Once the result appears, click **Install**.

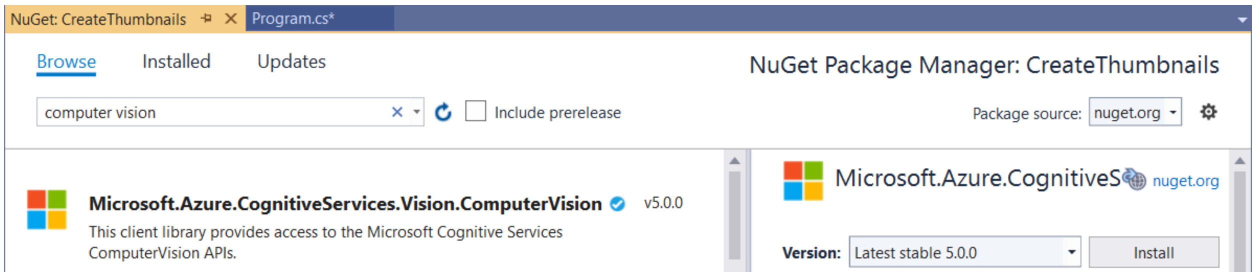

NuGet: CreateThumbnails
Program.cs*
Browse
Installed
Updates
NuGet Package Manager: CreateThumbnails
computer vision
Include prerelease
Package source: nuget.org
Microsoft.Azure.CognitiveServices.Vision.ComputerVision
v5.0.0
This client library provides access to the Microsoft Cognitive Services ComputerVision APIs.
Microsoft.Azure.CognitiveS
nuget.org
Version: Latest stable 5.0.0
Install

Figure 5-h: ComputerVision NuGet Package

We'll be using this package from now on to work with the Computer Vision API, which is easier than working directly with the API. So, go to the **Program.cs** file and add the following code.

Code Listing 5-b: Program.cs (CreateThumbnails Project)

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;

namespace CreateThumbnails
{
    class Program
    {
        const string API_key = "<< Key goes here >>";
        const string API_location =
        "https://computervisionsuccinctly.cognitiveservices.azure.com/";

        static void Main(string[] args)
        {
            string imgToAnalyze =
                @"C:\Test\image.jpg";

            SmartThumbnail(imgToAnalyze, 80, 80, true);

            Console.ReadKey();
        }

        public static ComputerVisionClient Authenticate(
            string key, string endpoint)
        {
            ComputerVisionClient client =
              new ComputerVisionClient(new
                  ApiKeyServiceClientCredentials(key))
                  { Endpoint = endpoint };

            return client;
        }

        public static byte[] ReadFully(Stream input)
        {
            byte[] buffer = new byte[16 * 1024];
            using (MemoryStream ms = new MemoryStream())
            {
                int read;
                while ((read = input.Read(buffer, 0, buffer.Length)) > 0)
                {
                    ms.Write(buffer, 0, read);
                }
                return ms.ToArray();
            }
        }
        public static void SmartThumbnail(string fname, int width, int
            height, bool smartCropping)
        {
            Task.Run(async () => {

                string imgname = Path.GetFileName(fname);
                Console.WriteLine($"Thumbnail for image: {imgname}");

                Stream thumbnail = await SmartThumbnailGeneration(
                    fname, width, height, smartCropping);

                string thumbnailFullPath = string.Format
                    ("{0}\\thumbnail_{1:yyyy-MMM-dd_hh-mm-ss}.jpg",
                    Path.GetDirectoryName(fname), DateTime.Now);

                using (BinaryWriter bw = new BinaryWriter(new
                    FileStream(thumbnailFullPath,
                    FileMode.OpenOrCreate, FileAccess.Write)))
                    bw.Write(ReadFully(thumbnail));

            }).Wait();
        }

        public static async Task<Stream> SmartThumbnailGeneration(
            string fname, int width, int height, bool smartCropping)
        {
            Stream thumbnail = null;
            ComputerVisionClient client = Authenticate(
                API_key, API_location);

            if (File.Exists(fname))
                using (Stream stream = File.OpenRead(fname))
                    thumbnail = await
                    client.GenerateThumbnailInStreamAsync(
                        width, height, stream, smartCropping);

            return thumbnail;
        }
    }
}
```

Let’s go over this code to understand what it does. The first thing we’ve done is add a reference to the Computer Vision SDK we added, which is what the following instruction indicates.

`using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;`

Next, we find the **`API_key`** and **`API_location`**, which will be used to access the service. Don’t forget to change the value of **`API_key`**.

Next, we define the **`Main`** method. All it does is specify the location of the image that is going to be used—which is the same one we used in the previous example (the image sign from Pixabay)—and invoke the **`SmartThumbnail`** method.

The **`SmartThumbnail`** method is where the thumbnail of the image gets created. This method calls the others, so let's start exploring it first.

The execution of the code contained within the **`SmartThumbnail`** method is wrapped around a **`Task.Run(async () => {}).Wait();`** construct. This is done so that the code execution is fully asynchronous and non-blocking.

The **`SmartThumbnailGeneration`** method is the one that invokes the Computer Vision API and creates the thumbnail **`Stream`** object that is returned to the **`SmartThumbnail`** method.

The resultant thumbnail file is written to disk using the **`Write`** method from the **`BinaryWriter`** instance, which is created using a **`FileStream`** object.

The **`Write`** method requires a **`byte`** array, so this is why the image needs to be converted from a **`Stream`** object to a **`byte`** array. This is done by invoking the **`ReadFully`** method.

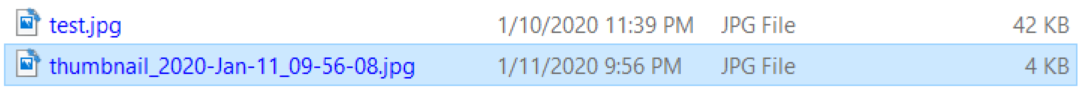
The thumbnail's file name is going to use this naming convention: **`thumbnail_{1:yyyy-MMM-dd_hh-mm-ss}.jpg`** (for example, **`thumbnail_2020-Jan-11_09-56-08.jpg`**).

Now, let's move our attention to the **`SmartThumbnailGeneration`** method. The first thing this method does is invoke **`Authenticate`**, which creates a **`ComputerVisionClient`** instance.

The **`SmartThumbnailGeneration`** method will be able to generate the thumbnail by invoking the **`GenerateThumbnailInStreamAsync`** method from the **`ComputerVisionClient`** instance. The image is sent to the Computer Vision API as a **`stream`** object.

The **`Authenticate`** method basically creates a **`ComputerVisionClient`** instance by passing an **`ApiKeyServiceClientCredentials`** object as a parameter.

Now, let's run the program and what we get. Go to the folder where **test.jpg** resides and check the resultant thumbnail file.



test.jpg	1/10/2020 11:39 PM	JPG File	42 KB
thumbnail_2020-Jan-11_09-56-08.jpg	1/11/2020 9:56 PM	JPG File	4 KB

Figure 5-i: Thumbnail File Created

Let's open the file (double-click it) to see what it looks like.

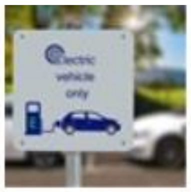


Figure 5-j: Thumbnail File Opened

Awesome—we now know how to use the Computer Vision API to create thumbnails. As you have seen, it wasn't difficult to do.

Optical character recognition

A very interesting feature of the Computer Vision API is the ability to recognize words and sentences from images. This is what we are going to explore now.

So, let's go ahead and create a new **Console .NET Framework** Visual Studio project—I'll call it **RecognizeOcr**.

Once you've created the project, go to the **NuGet Package Manager** and install the Computer Vision SDK, just like we did with the **CreateThumbnails** project.

Then, go to **Program.cs** and add the following code.

Code Listing 5-c: Program.cs (RecognizeOcr Project)

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

using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;

namespace RecognizeOcr
{
    class Program
    {
        const string API_key = "<< Key goes here >>";
        const string API_location =
        "https://computervisionsuccinctly.cognitiveservices.azure.com/";

        static void Main(string[] args)
        {
            string imgToAnalyze = @"C:\Test\receipt.jpg";

            TextExtractionCore(imgToAnalyze).Wait();
            Console.ReadLine();
        }

        public static ComputerVisionClient Authenticate(string key,
            string endpoint)
        {
            ComputerVisionClient client =
              new ComputerVisionClient(new
                  ApiKeyServiceClientCredentials(key))
```

```
                   { Endpoint = endpoint };

            return client;
        }

        public static List<string> GetWords(OcrLine line)
        {
            List<string> words = new List<string>();

            foreach (OcrWord w in line.Words)
                words.Add(w.Text);

            return words;
        }

        public static string GetLineAsString(OcrLine line)
        {
            List<string> words = GetWords(line);
            return
                words.Count > 0 ? string.Join(" ", words) : string.Empty;
        }

        public static async Task TextExtractionCore(string fname)
        {
            using (Stream stream = File.OpenRead(fname))
            {
                ComputerVisionClient client = Authenticate(
                    API_key, API_location);
                OcrResult ocrRes = await
                  client.RecognizePrintedTextInStreamAsync(true, stream);

                foreach (var localRegion in ocrRes.Regions)
                {
                    foreach (var line in localRegion.Lines)
                        Console.WriteLine(GetLineAsString(line));
                }
            }
        }
    }
}
```

Let’s explore what is going on here. The first thing we have done is add the references to the Computer Vision API with these two **using** statements.

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

Then, just like in the previous example, we have the **API_key** and **API_location** values. Don't forget to replace the value of **API_key** with the corresponding subscription key value for the Computer Vision API from the Azure portal.

Next, we have the **Main** method, which invokes **TextExtractionCore**—to which the image to be analyzed (**imgToAnalyze**) is passed as a parameter.

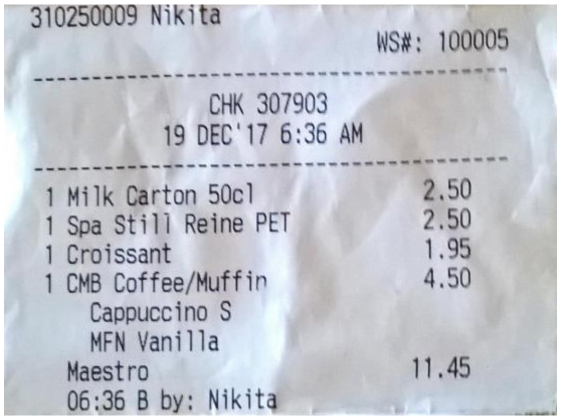
The **TextExtractionCore** method is quite straightforward. The image is opened as a **Stream** object, and the **Authenticate** method returns an instance of **ComputerVisionClient**.

Optical character recognition (OCR) is performed on the image (in order to extract the text contained within the image) by calling the **RecognizePrintedTextInStreamAsync** method from the **ComputerVisionClient** instance.

Then, for each of the **ocrRes.Regions** results returned, each OCR line is printed to the console by invoking the **GetLineAsString** method—which returns the words found on each line as a one-line string.

The **GetLineAsString** method basically loops through each **OcrWord** occurrence found within **line.Words** and returns a **List<string>** object.

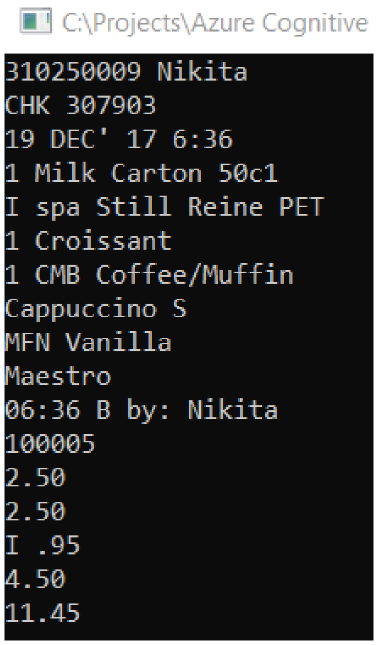
As you have seen, adding OCR detection capabilities was very easy to do. Before we run the program, let's have a look at the image we'll be testing—it's basically a scanned expense receipt.



```
310250009 Nikita
                              WS#: 100005
-----------------------------------------
               CHK 307903
            19 DEC'17 6:36 AM
-----------------------------------------
 1 Milk Carton 50cl               2.50
 1 Spa Still Reine PET            2.50
 1 Croissant                      1.95
 1 CMB Coffee/Muffin              4.50
     Cappuccino S
     MFN Vanilla
   Maestro                       11.45
   06:36 B by: Nikita
```

Figure 5-k: Scanned Receipt Image (receipt.jpg)

You can try with any other scanned receipt you have at hand—just make sure the image has a good resolution. Now, let's run the program to see what results we get.



```
C:\Projects\Azure Cognitive
310250009 Nikita
CHK 307903
19 DEC' 17 6:36
1 Milk Carton 50c1
I spa Still Reine PET
1 Croissant
1 CMB Coffee/Muffin
Cappuccino S
MFN Vanilla
Maestro
06:36 B by: Nikita
100005
2.50
2.50
I .95
4.50
11.45
```

Figure 5-l: Scanned Image Results

With just a few lines of code, we were able to extract the text contained within the scanned image receipt.

Adding field logic

Extracting text from images is great, but the real value comes when we are able to make sense of the information extracted. We achieve this by adding specific field logic.

Say we would like to find the date and the highest amount found within the extracted text—this way we could know the receipt's date and also the total value. This is what we are going to do now.

So, going back to **Program.cs**, let's add the following changes to the existing code, which I've highlighted in bold.

Code Listing 5-d: Modified Program.cs (RecognizeOcr Project)

```
// Previous using statements...

using System.Globalization;
using System.Text.RegularExpressions;

namespace RecognizeOcr
{
    class Program
    {
        // Previous code...

        public static async Task TextExtractionCore(string fname)
        {
            List<string> strList = new List<string>();

            using (Stream stream = File.OpenRead(fname))
            {
                ComputerVisionClient client =
                    Authenticate(API_key, API_location);
                OcrResult ocrRes = await
                    client.RecognizePrintedTextInStreamAsync
                        (true, stream);

                foreach (var localRegion in ocrRes.Regions)
                    foreach (var line in localRegion.Lines)
                        strList.Add(GetLineAsString(line));

                Console.WriteLine("Date: " +
                    GetDate(strList.ToArray()));
                Console.WriteLine("Highest amount: " +
                    HighestAmount(strList.ToArray()));
            }
        }

        public static string ParseDate(string str)
        {
            string result = string.Empty;
            string[] formats = new string[]
                { "dd MMM yy h:mm", "dd MMM yy hh:mm" };

            foreach (string fmt in formats)
            {
                try
                {
                    str = str.Replace("'", "");

                    if (DateTime.TryParseExact
                        (str, fmt, CultureInfo.InvariantCulture,
                        DateTimeStyles.None, out DateTime dateTime))
                    {
                        result = str;
                        break;
                    }
                }
                catch { }
            }

            return result;
        }

        public static string GetDate(string[] res)
        {
            string result = string.Empty;

            foreach (string l in res)
            {
                result = ParseDate(l);
                if (result != string.Empty) break;
            }

            return result;
        }

        public static string HighestAmount(string[] res)
        {
            string result = string.Empty;
            float highest = 0;

            Regex r = new Regex(@"[0-9]+\.[0-9]+");

            foreach (string l in res)
            {
                Match m = r.Match(l);

                if (m != null && m.Value != string.Empty &&
                    Convert.ToDouble(m.Value) > highest)
                    result = m.Value;
            }

            return result;
        }
    }
}
```

What have we done here? First, we added the following **`using`** statements, which we previously didn't have.

```
using System.Globalization;
using System.Text.RegularExpressions;
```

Then, we modified the **`TextExtractionCore`** method. First, we added the following instruction, which initializes a string **`List`**.

```
List<string> strList = new List<string>();
```

This list will be used to store the OCR results returned by the Computer Vision API, instead of writing them to the console. This is done as follows.

```
strList.Add(GetLineAsString(line));
```

Then, before the **`TextExtractionCore`** method finalizes, we invoke the **`GetDate`** method, which finds the date within the OCR results returned (**`strList.ToArray()`**) and invokes the **`HighestAmount`** method. As its name implies, the **`HighestAmount`** method returns the highest amount found within the OCR results returned.

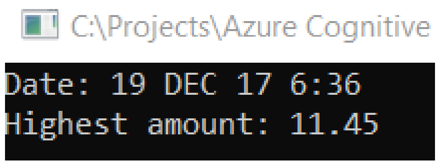
```
Console.WriteLine("Date: " + GetDate(strList.ToArray()));
Console.WriteLine("Highest amount: " + HighestAmount(strList.ToArray()));
```

The **`GetData`** method is actually very simple. All it does is loop through each line of text results found to check which line contains a date and retrieve it, which it's able to do by calling the **`ParseDate`** method.

The **`ParseDate`** method is able to retrieve the date from the results by looking for the following date string patterns: **`dd MMM yy h:mm`** and **`dd MMM yy hh:mm`**.

The **`HighestAmount`** method finds the highest amount contained within the text returned by the OCR results by looking for all occurrences that match the **`[0-9]+\.[0-9]+`** regular expression, and checking which of those has the highest numeric value.

Let's now run the code to see what we get.



```
Date: 19 DEC 17 6:36
Highest amount: 11.45
```

Figure 5-m: Scanned Image Results (Date and Highest Amount)

Cool—we can see that the program has been able to recognize the date and highest amount. Code Listing 5-e shows the full-blown code with all the modifications.

Code Listing 5-e: Full Program.cs (RecognizeOcr Project)

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;

namespace RecognizeOcr
{
    class Program
    {
        const string API_key = "<< Key goes here >>";
        const string API_location =
        "https://computervisionsuccinctly.cognitiveservices.azure.com/";

        static void Main(string[] args)
        {
            string imgToAnalyze = @"C:\Test\receipt.jpg";

            TextExtractionCore(imgToAnalyze).Wait();

            Console.ReadLine();
        }

        public static ComputerVisionClient Authenticate(
            string key, string endpoint)
        {
            ComputerVisionClient client =
              new ComputerVisionClient(new
              ApiKeyServiceClientCredentials(key))
              { Endpoint = endpoint };

            return client;
        }

        public static List<string> GetWords(OcrLine line)
        {
            List<string> words = new List<string>();

            foreach (OcrWord w in line.Words)
                words.Add(w.Text);

            return words;
        }

        public static string GetLineAsString(OcrLine line)
        {
            List<string> words = GetWords(line);
            return
              words.Count > 0 ? string.Join(" ", words) : string.Empty;
        }

        public static async Task TextExtractionCore(string fname)
        {
            List<string> strList = new List<string>();

            using (Stream stream = File.OpenRead(fname))
            {
                ComputerVisionClient client = Authenticate(
                  API_key, API_location);
                OcrResult ocrRes = await
                client.RecognizePrintedTextInStreamAsync(true, stream);

                foreach (var localRegion in ocrRes.Regions)
                    foreach (var line in localRegion.Lines)
                        strList.Add(GetLineAsString(line));

                Console.WriteLine("Date: " +
                    GetDate(strList.ToArray()));
                Console.WriteLine("Highest amount: " +
                    HighestAmount(strList.ToArray()));
            }
        }

        public static string ParseDate(string str)
        {
            string result = string.Empty;
            string[] formats = new string[]
                { "dd MMM yy h:mm", "dd MMM yy hh:mm" };

            foreach (string fmt in formats)
            {
                try
                {
                    str = str.Replace("'", "");

                    if (DateTime.TryParseExact(str, fmt,
                        CultureInfo.InvariantCulture,
                        DateTimeStyles.None, out DateTime dateTime))
                    {
                        result = str;
                        break;
                    }
                }
                catch { }
            }

            return result;
        }

        public static string GetDate(string[] res)
        {
            string result = string.Empty;

            foreach (string l in res)
            {
                result = ParseDate(l);
                if (result != string.Empty) break;
            }
            return result;
        }

        public static string HighestAmount(string[] res)
        {
            string result = string.Empty;
            float highest = 0;

            Regex r = new Regex(@"[0-9]+\.[0-9]+");

            foreach (string l in res)
            {
                Match m = r.Match(l);

                if (m != null && m.Value != string.Empty &&
                    Convert.ToDouble(m.Value) > highest)
                    result = m.Value;
            }

            return result;
        }
    }
}
```

Visual Studio projects: full code

You can download the complete source code for each of the Visual Studio projects created throughout this book from this link.

Final thoughts

Throughout this chapter, we've explored how to use the Computer Vision API to analyze images, generate thumbnails, extract text contained within images, and also get specific field values to make sense of the information extracted.

Still, the Computer Vision set of APIs in Azure offers even more possibilities, such as the ability to detect faces and create your own models to do things like recognize ordinary, day-to-day objects.

If you take a step back and think about it, it's quite impressive how with just a few lines of code, we are able to get an application to have vision-like capabilities.

The Azure team has done a fantastic job wrapping up this incredible functionality into a set of APIs that are both easy to use and learn in a relatively short amount of time.

We’ve now reached the end of this book, yet there’s so much more to explore about Cognitive Services.

A very interesting aspect of Cognitive Services is search, specifically the web search side of it, which empowers developers to build search engine capabilities within their apps.

For instance, you can leverage image search capabilities that use the same underlying technology as the Bing image search engine to retrieve images that correspond to specific terms or trending internet queries.

Regarding the search aspect of Cognitive Services, a complete book on the subject could be written, not only on how to search for images and perform visual searches, but also how to search for terms and entities, and how to combine search capabilities with other Azure services. I highly recommend that you explore these amazing features and APIs that Azure has to offer for search.

Beyond that, keep an eye on services that are still in preview mode. I’m very excited about Form Recognizer and Ink Recognizer—both have a very promising outlook.

I hope this book has given you sufficient insights on how to use the power of Cognitive Services to build applications that are infused with AI with very little code, which you can use to tackle real-world business problems within your work and your organization, and that your customers might need.

Thank you for reading this book, and until next time, take care.

All the best,

Ed