Table of Contents

Microsoft Speech API overview

4/19/2018 • 2 min to read • Edit Online

The cloud-based Microsoft Speech API provides developers an easy way to create powerful speech-enabled features in their applications, like voice command control, user dialog using natural speech conversation, and speech transcription and dictation. The Microsoft Speech API supports both *Speech to Text* and *Text to Speech* conversion.

- **Speech to Text** API converts human speech to text that can be used as input or commands to control your application.
- **Text to Speech** API converts text to audio streams that can be played back to the user of your application.

Speech to text (speech recognition)

Microsoft speech recognition API *transcribes* audio streams into text that your application can display to the user or act upon as command input. It provides two ways for developers to add Speech to their apps: REST APIs **or** Websocket-based client libraries.

- REST APIs: Developers can use HTTP calls from their apps to the service for speech recognition.
- Client libraries: For advanced features, developers can download Microsoft Speech client libraries, and link into their apps. The client libraries are available on various platforms (Windows, Android, iOS) using different languages (C#, Java, JavaScript, ObjectiveC). Unlike the REST APIs, the client libraries utilize Websocket-based procotol.

USE CASES	REST APIS	CLIENT LIBRARIES
Convert a short spoken audio, for example, commands (audio length < 15 s) without interim results	Yes	Yes
Convert a long audio (> 15 s)	No	Yes
Stream audio with interim results desired	No	Yes
Understand the text converted from audio using LUIS	No	Yes

Whichever approach developers choose (REST APIs or client libraries), Microsoft speech service supports the following:

- Advanced speech recognition technologies from Microsoft that are used by Cortana, Office Dictation, Office Translator, and other Microsoft products.
- Real-time continuous recognition. The speech recognition API enables users to transcribe audio into text in real time, and supports to receive the intermediate results of the words that have been recognized so far. The speech service also supports end-of-speech detection. In addition, users can choose additional formatting capabilities, like capitalization and punctuation, masking profanity, and text normalization.
- Supports optimized speech recognition results for *interactive*, *conversation*, and *dictation* scenarios. For user scenarios which require customized language models and acoustic models, Custom Speech Service allows you to create speech models that tailored to your application and your users.

- Support many spoken languages in multiple dialects. For the full list of supported languages in each recognition mode, see recognition languages.
- Integration with language understanding. Besides converting the input audio into text, the *Speech to Text* provides applications an additional capability to understand what the text means. It uses the Language Understanding Intelligent Service(LUIS) to extract intents and entities from the recognized text.

Next steps

- Get started to use Microsoft speech recognition service with REST APIs or client libraries.
- Check out sample applications in your preferred programming language.
- Go to the Reference section to find Microsoft Speech Protocol details and API references.

Text to speech (speech synthesis)

Text to Speech APIs use REST to convert structured text to an audio stream. The APIs provide fast text to speech conversion in various voices and languages. In addition users also have the ability to change audio characteristics like pronunciation, volume, pitch etc. using SSML tags.

Next steps

- Get started to use Microsoft text to speech service: Text to Speech API Reference. For the full list of languages and voices supported by Text to Speech, see Supported Locales and Voice Fonts.

Get started with Speech Service

4/19/2018 • 1 min to read • Edit Online

To use the Microsoft speech recognition service to convert audio to text, see the Speech Recognition REST APIs or the client libraries.

To use the Microsoft text-to-speech service, see the Text to Speech API reference.

Get started with speech recognition by using the REST API

4/19/2018 • 5 min to read • Edit Online

With cloud-based Speech Service, you can develop applications by using the REST API to convert spoken audio to text.

Prerequisites

Subscribe to the Speech API, and get a free trial subscription key

The Speech API is part of Cognitive Services (previously Project Oxford). You can get free trial subscription keys from the Cognitive Services subscription page. After you select the Speech API, select **Get API Key** to get the key. It returns a primary and secondary key. Both keys are tied to the same quota, so you can use either key.

> **IMPORTANT**
>
> - Get a subscription key. Before you can access the REST API, you must have a subscription key.
> - Use your subscription key. In the following REST samples, replace YOUR_SUBSCRIPTION_KEY with your own subscription key.
> - Refer to the authentication page for how to get a subscription key.

Prerecorded audio file

In this example, we use a recorded audio file to illustrate how to use the REST API. Record an audio file of yourself saying a short phrase. For example, say "What is the weather like today?" or "Find funny movies to watch." The speech recognition API also supports external microphone input.

> **NOTE**
>
> The example requires that audio is recorded as a WAV file with **PCM single channel (mono), 16 KHz**.

Build a recognition request, and send it to the speech recognition service

The next step for speech recognition is to send a POST request to the Speech HTTP endpoints with the proper request header and body.

Service URI

The speech recognition service URI is defined based on recognition modes and recognition languages:

```
https://speech.platform.bing.com/speech/recognition/<RECOGNITION_MODE>/cognitiveservices/v1?language=
<LANGUAGE_TAG>&format=<OUTPUT_FORMAT>
```

`<RECOGNITION_MODE>` specifies the recognition mode and must be one of the following values: `interactive`, `conversation`, or `dictation`. It's a required resource path in the URI. For more information, see Recognition modes.

`<LANGUAGE_TAG>` is a required parameter in the query string. It defines the target language for audio conversion: for example, `en-US` for English (United States). For more information, see Recognition languages.

`<OUTPUT_FORMAT>` is an optional parameter in the query string. Its allowed values are `simple` and `detailed`. By default, the service returns results in `simple` format. For more information, see Output format.

Some examples of service URIs are listed in the following table.

RECOGNITION MODE	LANGUAGE	OUTPUT FORMAT	SERVICE URI
`interactive`	pt-BR	Default	https://speech.platform.bing.com/speech/recognition/interactive/cognitiveservices/v1?language=pt-BR
`conversation`	en-US	Detailed	https://speech.platform.bing.com/speech/recognition/conversation/cognitiveservices/v1?language=en-US&format=detailed
`dictation`	fr-FR	Simple	https://speech.platform.bing.com/speech/recognition/dictation/cognitiveservices/v1?language=fr-FR&format=simple

NOTE

The service URI is needed only when your application uses REST APIs to call the speech recognition service. If you use one of the client libraries, you usually don't need to know which URI is used. The client libraries might use different service URIs, which are applicable only for a specific client library. For more information, see the client library of your choice.

Request headers

The following fields must be set in the request header:

- `Ocp-Apim-Subscription-Key`: Each time that you call the service, you must pass your subscription key in the `Ocp-Apim-Subscription-Key` header. Speech Service also supports passing authorization tokens instead of subscription keys. For more information, see Authentication.
- `Content-type`: The `Content-type` field describes the format and codec of the audio stream. Currently, only WAV file and PCM Mono 16000 encoding is supported. The Content-type value for this format is `audio/wav; codec=audio/pcm; samplerate=16000`.

The `Transfer-Encoding` field is optional. If you set this field to `chunked`, you can chop the audio into small chunks. For more information, see Chunked transfer.

The following is a sample request header:

```
POST https://speech.platform.bing.com/speech/recognition/interactive/cognitiveservices/v1?language=en-US&format=detailed HTTP/1.1
Accept: application/json;text/xml
Content-Type: audio/wav; codec=audio/pcm; samplerate=16000
Ocp-Apim-Subscription-Key: YOUR_SUBSCRIPTION_KEY
Host: speech.platform.bing.com
Transfer-Encoding: chunked
Expect: 100-continue
```

Send a request to the service

The following example shows how to send a speech recognition request to Speech REST endpoints. It uses the `interactive` recognition mode.

> **NOTE**
> Replace `YOUR_AUDIO_FILE` with the path to your prerecorded audio file. Replace `YOUR_SUBSCRIPTION_KEY` with your own subscription key.

- PowerShell
- curl
- C#

```
$SpeechServiceURI =
'https://speech.platform.bing.com/speech/recognition/interactive/cognitiveservices/v1?language=en-
us&format=detailed'

# $OAuthToken is the authorization token returned by the token service.
$RecoRequestHeader = @{
  'Ocp-Apim-Subscription-Key' = 'YOUR_SUBSCRIPTION_KEY';
  'Transfer-Encoding' = 'chunked';
  'Content-type' = 'audio/wav; codec=audio/pcm; samplerate=16000'
}

# Read audio into byte array
$audioBytes = [System.IO.File]::ReadAllBytes("YOUR_AUDIO_FILE")

$RecoResponse = Invoke-RestMethod -Method POST -Uri $SpeechServiceURI -Headers $RecoRequestHeader -Body
$audioBytes

# Show the result
$RecoResponse
```

Process the speech recognition response

After processing the request, Speech Service returns the results in a response as JSON format.

> **NOTE**
> If the previous code returns an error, see Troubleshooting to locate the possible cause.

The following code snippet shows an example of how you can read the response from the stream.

- PowerShell
- curl
- C#

```
# show the response in JSON format
ConvertTo-Json $RecoResponse
```

The following sample is a JSON response:

```
OK
{
  "RecognitionStatus": "Success",
  "Offset": 22500000,
  "Duration": 21000000,
  "NBest": [{
    "Confidence": 0.941552162,
    "Lexical": "find a funny movie to watch",
    "ITN": "find a funny movie to watch",
    "MaskedITN": "find a funny movie to watch",
    "Display": "Find a funny movie to watch."
  }]
}
```

Limitations

The REST API has some limitations:

- It supports audio stream only up to 15 seconds.
- It doesn't support intermediate results during recognition. Users receive only the final recognition result.

To remove these limitations, use Speech client libraries. Or you can work directly with the Speech WebSocket protocol.

What's next

- To see how to use the REST API in C#, Java, etc., see these sample applications.
- To locate and fix errors, see Troubleshooting.
- To use more advanced features, see how to get started by using Speech client libraries.

License

Get started with Speech Service client libraries

4/19/2018 • 1 min to read • Edit Online

Besides making direct HTTP requests via a REST API, Speech Service provides developers with Speech client libraries in different languages. The Speech client libraries:

- Support more advanced capabilities in speech recognition, such as intermediate results in real time, long audio stream (up to 10 minutes), and continuous recognition.
- Provide a simple and idiomatic API in the language of your preference.
- Hide low-level communication details.

Currently, the following Speech client libraries are available:

- C# desktop library
- C# service library
- JavaScript library
- Java library for Android
- Objective-C library for iOS

Additional resources

- The samples page provides complete samples to use Speech client libraries.
- If you need a client library that's not yet supported, you can create your own SDK. Implement the Speech WebSocket protocol on the platform and use the language of your choice.

License

Get started with the Speech Recognition API in C# for .NET on Windows

4/19/2018 • 4 min to read • Edit Online

This page shows how to develop a basic Windows application that uses the Speech Recognition API to convert spoken audio to text. Using the client library allows for real-time streaming, which means that when your client application sends audio to the service, it simultaneously and asynchronously receives partial recognition results back.

Developers who want to use Speech Service from apps that run on any device can use the C# desktop library. To use the library, install the NuGet package Microsoft.ProjectOxford.SpeechRecognition-x86 for a 32-bit platform and the NuGet package Microsoft.ProjectOxford.SpeechRecognition-x64 for a 64-bit platform. For the client library API reference, see Microsoft Speech C# desktop library.

The following sections describe how to install, build, and run the C# sample application by using the C# desktop library.

Prerequisites

Platform requirements

The following sample was developed for Windows 8+ and .NET Framework 4.5+ by using Visual Studio 2015, Community Edition.

Get the sample application

Clone the sample from the Speech C# desktop library sample repository.

Subscribe to the Speech Recognition API, and get a free trial subscription key

The Speech API is part of Cognitive Services (previously Project Oxford). You can get free trial subscription keys from the Cognitive Services subscription page. After you select the Speech API, select **Get API Key** to get the key. It returns a primary and secondary key. Both keys are tied to the same quota, so you can use either key.

> **IMPORTANT**
>
> - Get a subscription key. Before you use the Speech client libraries, you must have a subscription key.
> - Use your subscription key. With the provided C# desktop sample application, paste your subscription key into the text box when you run the sample. For more information, see Run the sample application.

Step 1: Install the sample application

1. Start Visual Studio 2015, and select **File** > **Open** > **Project/Solution**.
2. Browse to the folder where you saved the downloaded Speech Recognition API files. Select **Speech** > **Windows**, and then select the Sample-WP folder.
3. Double-click to open the Visual Studio 2015 Solution (.sln) file named SpeechToText-WPF-Samples.sln. The solution opens in Visual Studio.

Step 2: Build the sample application

1. If you want to use *recognition with intent*, you first need to sign up for the Language Understanding Intelligent Service (LUIS). Then use the endpoint URL of your LUIS app to set the value of the key `LuisEndpointUrl` in the app.config file in the samples/SpeechRecognitionServiceExample folder. For more information on the endpoint URL of the LUIS app, see Publish your app.

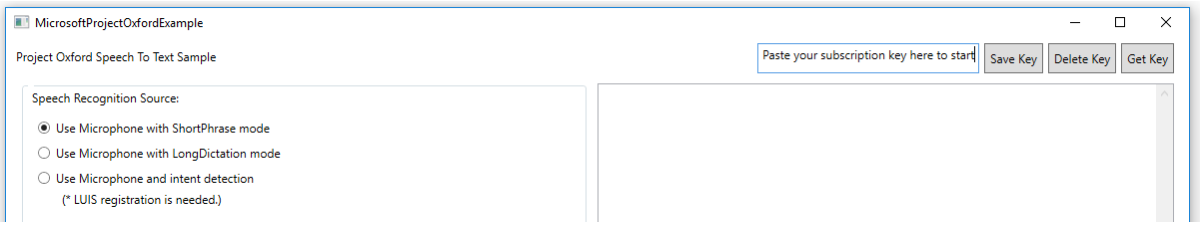
   > **TIP**
   > Replace the `&` character in the LUIS endpoint URL with `&` to ensure that the URL is correctly interpreted by the XML parser.

2. Press Ctrl+Shift+B, or select **Build** on the ribbon menu. Then select **Build Solution**.

Step 3: Run the sample application

1. After the build is finished, press F5 or select **Start** on the ribbon menu to run the sample.

2. Go to the **Project Oxford Speech to Text Sample** window. Paste your subscription key into the **Paste your subscription key here to start** text box as shown. To persist your subscription key on your PC or laptop, select **Save Key**. To delete the subscription key from the system, select **Delete Key** to remove it from your PC or laptop.



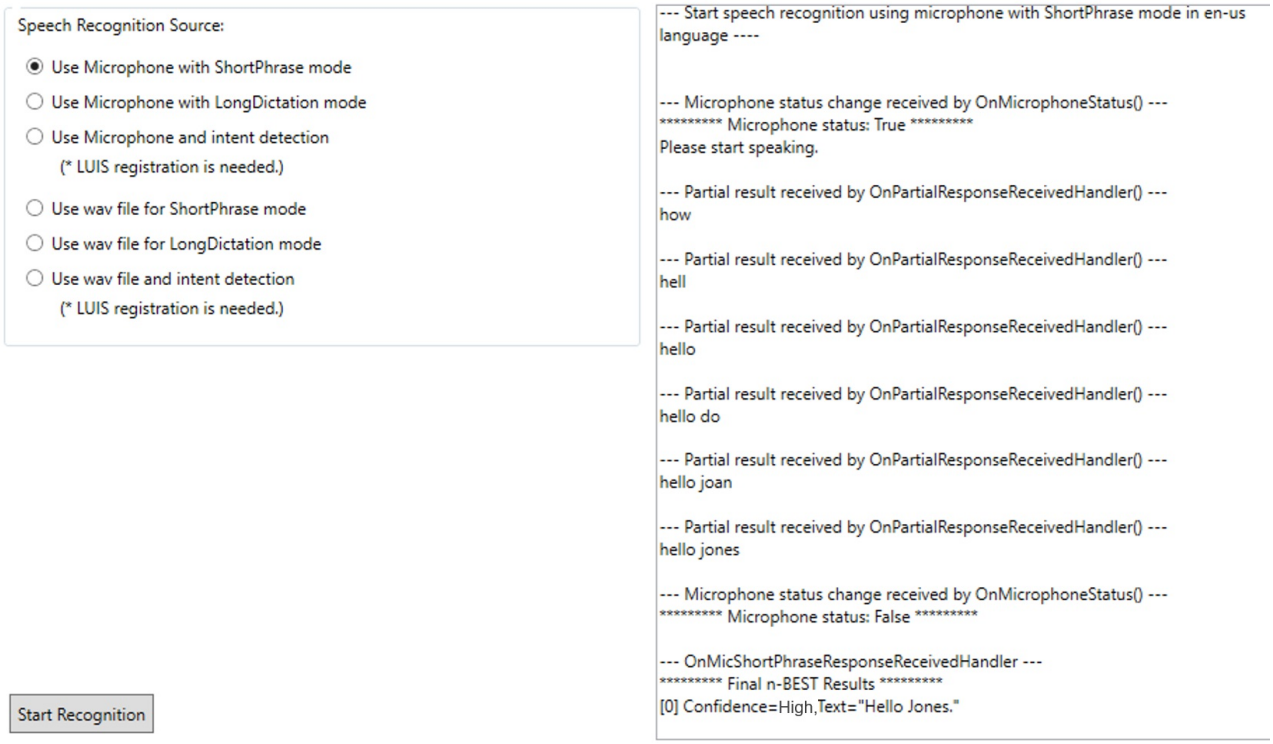
3. Under **Speech Recognition Source**, choose one of the six speech sources, which fall into two main input categories:

   - Use your computer's microphone or an attached microphone to capture speech.
   - Play an audio file.

   Each category has three recognition modes:

   - **ShortPhrase mode**: An utterance up to 15 seconds long. As data is sent to the server, the client receives multiple partial results and one final result with multiple n-best choices.
   - **LongDictation mode**: An utterance up to two minutes long. As data is sent to the server, the client receives multiple partial results and multiple final results, based on where the server indicates sentence pauses.
   - **Intent detection**: The server returns additional structured information about the speech input. To use intent detection, you need to first train a model by using LUIS.

Use sample audio files with this sample application. Find the files in the repository you downloaded with this sample under the samples/SpeechRecognitionServiceExample folder. These sample audio files run automatically if no other files are chosen when you select **Use wav file for Shortphrase mode** or **Use wav file for Longdictation mode** as your speech input. Currently, only WAV audio format is supported.



Samples explained

Recognition events

- **Partial Results events**: This event gets called every time Speech Service predicts what you might be saying, even before you finish speaking (if you use `MicrophoneRecognitionClient`) or finish sending data (if you use `DataRecognitionClient`).
- **Error events**: Called when the service detects an error.
- **Intent events**: Called on "WithIntent" clients (only in ShortPhrase mode) after the final recognition result is parsed into a structured JSON intent.
- **Result events**:
  - In `ShortPhrase` mode, this event is called and returns the n-best results after you finish speaking.
  - In `LongDictation` mode, the event handler is called multiple times, based on where the service identifies sentence pauses.
  - **For each of the n-best choices**, a confidence value and a few different forms of the recognized text are returned. For more information, see Output format.

Event handlers are already pointed out in the code in the form of code comments.

Related topics

- Microsoft Speech desktop library reference
- Get started with the Microsoft Speech Recognition API in Java on Android
- Get started with the Microsoft Speech Recognition API in Objective-C on iOS
- Get started with the Microsoft Speech Recognition API in JavaScript
- Get started with the Microsoft Speech Recognition API via REST

Get started with the speech recognition service library in C# for .NET Windows

4/19/2018 • 5 min to read • Edit Online

The service library is for developers who have their own cloud service and want to call Speech Service from their service. If you want to call the speech recognition service from device-bound apps, do not use this SDK. (Use other client libraries or REST APIs for that.)

To use the C# service library, install the NuGet package Microsoft.Bing.Speech. For the library API reference, see the Microsoft Speech C# service library.

The following sections describe how to install, build, and run the C# sample application by using the C# service library.

Prerequisites

Platform requirements

The following example was developed for Windows 8+ and .NET 4.5+ Framework by using Visual Studio 2015, Community Edition.

Get the sample application

Clone the sample from the Speech C# service library sample repository.

Subscribe to the Speech Recognition API, and get a free trial subscription key

The Speech API is part of Cognitive Services (previously Project Oxford). You can get free trial subscription keys from the Cognitive Services subscription page. After you select the Speech API, select **Get API Key** to get the key. It returns a primary and secondary key. Both keys are tied to the same quota, so you can use either key.

> **IMPORTANT**
>
> - Get a subscription key. Before you can use the Speech client libraries, you must have a subscription key.
> - Use your subscription key. With the provided C# service library sample application, you need to provide your subscription key as one of the command-line parameters. For more information, see Run the sample application.

Step 1: Install the sample application

1. Start Visual Studio 2015, and select **File** > **Open** > **Project/Solution**.
2. Double-click to open the Visual Studio 2015 Solution (.sln) file named SpeechClient.sln. The solution opens in Visual Studio.

Step 2: Build the sample application

Press Ctrl+Shift+B, or select **Build** on the ribbon menu. Then select **Build Solution**.

Step 3: Run the sample application

1. After the build is finished, press F5 or select **Start** on the ribbon menu to run the example.

2. Open the output directory for the sample, for example, SpeechClientSample\bin\Debug. Press Shift+Right-click, and select **Open command window here**.

3. Run `SpeechClientSample.exe` with the following arguments:

   - Arg[0]: Specify an input audio WAV file.
   - Arg[1]: Specify the audio locale.
   - Arg[2]: Specify the recognition modes: *Short* for the `ShortPhrase` mode and *Long* for the `LongDictation` mode.
   - Arg[3]: Specify the subscription key to access the speech recognition service.

Samples explained

Recognition modes

- `ShortPhrase` mode: An utterance up to 15 seconds long. As data is sent to the server, the client receives multiple partial results and one final best result.
- `LongDictation` mode: An utterance up to 10 minutes long. As data is sent to the server, the client receives multiple partial results and multiple final results, based on where the server indicates sentence pauses.

Supported audio formats

The Speech API supports audio/WAV by using the following codecs:

- PCM single channel
- Siren
- SirenSR

Preferences

To create a SpeechClient, you need to first create a Preferences object. The Preferences object is a set of parameters that configures the behavior of the speech service. It consists of the following fields:

- `SpeechLanguage`: The locale of the audio sent to the speech service.
- `ServiceUri`: The endpoint used to call the speech service.
- `AuthorizationProvider`: An IAuthorizationProvider implementation used to fetch tokens in order to access the speech service. Although the sample provides a Cognitive Services authorization provider, we highly recommend that you create your own implementation to handle token caching.
- `EnableAudioBuffering`: An advanced option. See Connection management.

Speech input

The SpeechInput object consists of two fields:

- **Audio**: A stream implementation of your choice from which the SDK pulls audio. It can be any stream that supports reading.

  > **NOTE**
  > The SDK detects the end of the stream when the stream returns **0** in read.

- **RequestMetadata**: Metadata about the speech request. For more information, see the reference.

Speech request

After you have instantiated a SpeechClient and SpeechInput objects, use RecognizeAsync to make a request to Speech Service.

```
var task = speechClient.RecognizeAsync(speechInput);
```

After the request finishes, the task returned by RecognizeAsync finishes. The last RecognitionResult is the end of the recognition. The task can fail if the service or the SDK fails unexpectedly.

Speech recognition events

Partial results event

This event gets called every time Speech Service predicts what you might be saying, even before you finish speaking (if you use `MicrophoneRecognitionClient`) or finish sending data (if you use `DataRecognitionClient`). You can subscribe to the event by using `SpeechClient.SubscribeToPartialResult()`. Or you can use the generic events subscription method `SpeechClient.SubscribeTo<RecognitionPartialResult>()`.

RETURN FORMAT	DESCRIPTION
LexicalForm	This form is optimal for use by applications that need raw, unprocessed speech recognition results.
DisplayText	The recognized phrase with inverse text normalization, capitalization, punctuation, and profanity masking applied. Profanity is masked with asterisks after the initial character, for example, "d***." This form is optimal for use by applications that display the speech recognition results to a user.
Confidence	The level of confidence the recognized phrase represents for the associated audio as defined by the speech recognition server.
MediaTime	The current time relative to the start of the audio stream (in 100-nanosecond units of time).
MediaDuration	The current phrase duration/length relative to the audio segment (in 100-nanosecond units of time).

Result event

When you finish speaking (in `ShortPhrase` mode), this event is called. You're provided with n-best choices for the result. In `LongDictation` mode, the event can be called multiple times, based on where the server indicates sentence pauses. You can subscribe to the event by using `SpeechClient.SubscribeToRecognitionResult()`. Or you can use the generic events subscription method `SpeechClient.SubscribeTo<RecognitionResult>()`.

RETURN FORMAT	DESCRIPTION
RecognitionStatus	The status on how the recognition was produced. For example, was it produced as a result of successful recognition or as a result of canceling the connection, etc.
Phrases	The set of n-best recognized phrases with the recognition confidence.

For more information on recognition results, see Output format.

Speech recognition response

Speech response example:

```
--- Partial result received by OnPartialResult
---what
--- Partial result received by OnPartialResult
--what's
--- Partial result received by OnPartialResult
---what's the web
--- Partial result received by OnPartialResult
---what's the weather like
---***** Phrase Recognition Status = [Success]
***What's the weather like? (Confidence:High)
What's the weather like? (Confidence:High)
```

Connection management

The API utilizes a single WebSocket connection per request. For optimal user experience, the SDK attempts to reconnect to Speech Service and start the recognition from the last RecognitionResult that it received. For example, if the audio request is two minutes long, the SDK received a RecognitionEvent at the one-minute mark, and a network failure occurred after five seconds, the SDK starts a new connection that starts from the one-minute mark.

> **NOTE**
> The SDK doesn't seek back to the one-minute mark because the stream might not support seeking. Instead, the SDK keeps an internal buffer that it uses to buffer the audio and clears the buffer as it receives RecognitionResult events.

Buffering behavior

By default, the SDK buffers audio so that it can recover when a network interrupt occurs. In a scenario where it's preferable to discard the audio lost during the network disconnect and restart the connection, it's best to disable audio buffering by setting `EnableAudioBuffering` in the Preferences object to `false`.

Related topics

Microsoft Speech C# service library reference

Get started with the Speech Recognition API in JavaScript

4/19/2018 • 1 min to read • Edit Online

You can develop applications that convert spoken audio to text by using the Speech Recognition API. The JavaScript client library uses the Speech Service WebSocket protocol, which allows you to talk and receive transcribed text simultaneously. This article helps you to get started with the Speech Recognition API in JavaScript.

Prerequisites

Subscribe to the Speech Recognition API, and get a free trial subscription key

The Speech API is part of Cognitive Services. You can get free trial subscription keys from the Cognitive Services subscription page. After you select the Speech API, select **Get API Key** to get the key. It returns a primary and secondary key. Both keys are tied to the same quota, so you can use either key.

> **IMPORTANT**
> Get a subscription key. Before you can use Speech client libraries, you must have a subscription key.

Get started

In this section we will walk you through the necessary steps to load a sample HTML page. The sample is located in our github repository. You can **open the sample directly** from the repository, or **open the sample from a local copy** of the repository.

> **NOTE**
> Some browsers block microphone access on un-secure origin. So, it is recommended to host the 'sample'/'your app' on https to get it working on all supported browsers.

Open the sample directly

Acquire a subscription key as described above. Then open the link to the sample. This will load the page into your default browser (Rendered using htmlPreview).

Open the sample from a local copy

To try the sample locally, clone this repository:

```
git clone https://github.com/Azure-Samples/SpeechToText-WebSockets-Javascript
```

compile the TypeScript sources and bundle/browserfy them into a single JavaScript file (npm needs to be installed on your machine). Change into the root of the cloned repository and run the commands:

```
cd SpeechToText-WebSockets-Javascript && npm run bundle
```

Open `samples\browser\Sample.html` in your favorite browser.

Next steps

More information on how to include the SDK into your own webpage is available here.

Remarks

- The Speech Recognition API supports three recognition modes. You can switch the mode by updating the **Setup()** function found in the Sample.html file. The sample sets the mode to `Interactive` by default. To change the mode, update the parameter `SR.RecognitionMode.Interactive` to another mode. For example, change the parameter to `SR.RecognitionMode.Conversation`.
- For a complete list of supported languages, see Supported languages.

Related topics

- JavaScript Speech Recognition API sample repository
- Get started with the REST API

Get started with speech recognition in Java on Android

4/19/2018 • 4 min to read • Edit Online

With the Speech Recognition API, you can develop Android applications that use cloud-based Speech Service to convert spoken audio to text. The API supports real-time streaming, so your application can simultaneously and asynchronously receive partial recognition results at the same time it's sending audio to the service.

This article uses a sample application to demonstrate how to use the Speech client library for Android to develop speech-to-text applications in Java for Android devices.

Prerequisites

Platform requirements

The sample is developed by Android Studio for Windows in Java.

Get the client library and sample application

The Speech client library and samples for Android are available in the Speech client SDK for Android. You can find the buildable sample under the samples/SpeechRecoExample directory. You can find the two libraries you need to use in your own apps in SpeechSDK/libs under the armeabi and the x86 folder. The size of the libandroid_platform.so file is 22 MB, but it's reduced to 4 MB at deployment time.

Subscribe to the Speech API, and get a free trial subscription key

The Speech API is part of Cognitive Services (previously Project Oxford). You can get free trial subscription keys from the Cognitive Services subscription page. After you select the Speech API, select **Get API Key** to get the key. It returns a primary and secondary key. Both keys are tied to the same quota, so you can use either key.

If you want to use *recognition with intent*, you also need to sign up for the Language Understanding Intelligent Service (LUIS).

> **IMPORTANT**
>
> - Get a subscription key. Before you can use Speech client libraries, you must have a subscription key.
> - Use your subscription key. With the provided Android sample application, update the file samples/SpeechRecoExample/res/values/strings.xml with your subscription keys. For more information, see Build and run samples.

Use the Speech client library

To use the client library in your application, follow the instructions.

You can find the client library reference for Android in the docs folder of the Speech client SDK for Android.

Build and run samples

To learn how to build and run samples, see this README page.

Samples explained

Create recognition clients

The code in the following sample shows how to create recognition client classes based on user scenarios:

```
void initializeRecoClient()
    {
        String language = "en-us";

        String subscriptionKey = this.getString(R.string.subscription_key);
        String luisAppID = this.getString(R.string.luisAppID);
        String luisSubscriptionID = this.getString(R.string.luisSubscriptionID);

        if (m_isMicrophoneReco && null == m_micClient) {
            if (!m_isIntent) {
                m_micClient = SpeechRecognitionServiceFactory.createMicrophoneClient(this,
                                                                                     m_recoMode,
                                                                                     language,
                                                                                     this,
                                                                                     subscriptionKey);
            }
            else {
                MicrophoneRecognitionClientWithIntent intentMicClient;
                intentMicClient = SpeechRecognitionServiceFactory.createMicrophoneClientWithIntent(this,
                                                                                                   language,
                                                                                                   this,

subscriptionKey,
                                                                                                   luisAppID,

luisSubscriptionID);
                m_micClient = intentMicClient;

            }
        }
        else if (!m_isMicrophoneReco && null == m_dataClient) {
            if (!m_isIntent) {
                m_dataClient = SpeechRecognitionServiceFactory.createDataClient(this,
                                                                                m_recoMode,
                                                                                language,
                                                                                this,
                                                                                subscriptionKey);
            }
            else {
                DataRecognitionClientWithIntent intentDataClient;
                intentDataClient = SpeechRecognitionServiceFactory.createDataClientWithIntent(this,
                                                                                               language,
                                                                                               this,
                                                                                               subscriptionKey,
                                                                                               luisAppID,

luisSubscriptionID);
                m_dataClient = intentDataClient;
            }
        }
    }
```

The client library provides pre-implemented recognition client classes for typical scenarios in speech recognition:

- `DataRecognitionClient`: Speech recognition with PCM data (for example, from a file or audio source). The data is broken up into buffers, and each buffer is sent to Speech Service. No modification is done to the buffers, so the user can apply their own silence detection if desired. If the data is provided from WAV files, you can send data from the file right to Speech Service. If you have raw data, for example, audio coming over Bluetooth, you first send a format header to Speech Service followed by the data.
- `MicrophoneRecognitionClient`: Speech recognition with audio coming from the microphone. Make sure the microphone is turned on and the data from the microphone is sent to the speech recognition service. A built-in

"Silence Detector" is applied to the microphone data before it's sent to the recognition service.

- `DataRecognitionClientWithIntent` and `MicrophoneRecognitionClientWithIntent`: These clients return, in addition to recognition text, structured information about the intent of the speaker, which can be used to drive further actions by your applications. To use "Intent," you need to first train a model by using LUIS.

Recognition language

When you use `SpeechRecognitionServiceFactory` to create the client, you must select a language. For the complete list of languages supported by Speech Service, see Supported languages.

`SpeechRecognitionMode`

You also need to specify `SpeechRecognitionMode` when you create the client with `SpeechRecognitionServiceFactory`:

- `ShortPhrase`: An utterance up to 15 seconds long. As data is sent to the service, the client receives multiple partial results and one final result with multiple n-best choices.
- `LongDictation`: An utterance up to two minutes long. As data is sent to the service, the client receives multiple partial results and multiple final results, based on where the service identifies sentence pauses.

Attach event handlers

You can attach various event handlers to the client you created:

- **Partial Results events**: This event gets called every time Speech Service predicts what you might be saying, even before you finish speaking (if you use `MicrophoneRecognitionClient`) or finish sending data (if you use `DataRecognitionClient`).
- **Error events**: Called when the service detects an error.
- **Intent events**: Called on "WithIntent" clients (only in `ShortPhrase` mode) after the final recognition result is parsed into a structured JSON intent.
- **Result events**:
  - In `ShortPhrase` mode, this event is called and returns n-best results after you finish speaking.
  - In `LongDictation` mode, the event handler is called multiple times, based on where the service identifies sentence pauses.
  - **For each of the n-best choices**, a confidence value and a few different forms of the recognized text are returned. For more information, see Output format.

Related topics

- Client library reference for Android
- Get started with the Microsoft Speech API in C# for Windows in .NET
- Get started with the Microsoft Speech API in Objective-C on iOS
- Get started with the Microsoft Speech API in JavaScript
- Get started with the Microsoft Speech API via REST

Get started with the Speech Recognition API in Objective-C on iOS

4/19/2018 • 4 min to read • Edit Online

With the Speech Recognition API, you can develop iOS applications that use cloud-based Speech Service to convert spoken audio to text. The API supports real-time streaming, so your application can simultaneously and asynchronously receive partial recognition results at the same time it's sending audio to the service.

This article uses a sample application to demonstrate the basics of how to get started with the Speech Recognition API to develop an iOS application. For a complete API reference, see the Speech SDK client library reference.

Prerequisites

Platform requirements

Make sure that the Mac XCode IDE is installed.

Get the client library and examples

The Speech client library and examples for iOS are available on the Speech client SDK for iOS.

Subscribe to the Speech Recognition API, and get a free trial subscription key

The Speech API is part of Cognitive Services (previously Project Oxford). You can get free trial subscription keys from the Cognitive Services subscription page. After you select the Speech API, select **Get API Key** to get the key. It returns a primary and secondary key. Both keys are tied to the same quota, so you can use either key.

If you want to use *recognition with intent*, you also need to sign up for the Language Understanding Intelligent Service (LUIS).

> **IMPORTANT**
>
> - Get a subscription key. Before you can use Speech client libraries, you must have a subscription key.
> - Use your subscription key. With the provided iOS sample application, you need to update the file Samples/SpeechRecognitionServerExample/settings.plist with your subscription key. For more information, see Build and run samples.

Use the Speech client library

To add the client library into an XCode project, follow these instructions.

To find the client library reference for iOS, see this webpage.

Build and run samples

For information on how to build and run samples, see this README page.

Samples explained

Create recognition clients

The following code in the sample shows how to create recognition client classes based on user scenarios:

```
{
    NSString* language = @"en-us";

    NSString* path = [[NSBundle mainBundle] pathForResource:@"settings" ofType:@"plist"];
    NSDictionary* settings = [[NSDictionary alloc] initWithContentsOfFile:path];

    NSString* primaryOrSecondaryKey = [settings objectForKey:(@"primaryKey")];
    NSString* luisAppID = [settings objectForKey:(@"luisAppID")];
    NSString* luisSubscriptionID = [settings objectForKey:(@"luisSubscriptionID")];

    if (isMicrophoneReco) {
        if (!isIntent) {
            micClient = [SpeechRecognitionServiceFactory createMicrophoneClient:(recoMode)
                                                                   withLanguage:(language)
                                                                        withKey:(primaryOrSecondaryKey)
                                                                   withProtocol:(self)];
        }
        else {
            MicrophoneRecognitionClientWithIntent* micIntentClient;
            micIntentClient = [SpeechRecognitionServiceFactory createMicrophoneClientWithIntent:(language)
                                                                                        withKey:
(primaryOrSecondaryKey)
                                                                                 withLUISAppID:(luisAppID)
                                                                                    withLUISSecret:
(luisSubscriptionID)
                                                                                      withProtocol:(self)];
            micClient = micIntentClient;
        }
    }
    else {
        if (!isIntent) {
            dataClient = [SpeechRecognitionServiceFactory createDataClient:(recoMode)
                                                              withLanguage:(language)
                                                                   withKey:(primaryOrSecondaryKey)
                                                              withProtocol:(self)];
        }
        else {
            DataRecognitionClientWithIntent* dataIntentClient;
            dataIntentClient = [SpeechRecognitionServiceFactory createDataClientWithIntent:(language)
                                                                                    withKey:
(primaryOrSecondaryKey)
                                                                             withLUISAppID:(luisAppID)
                                                                                withLUISSecret:
(luisSubscriptionID)
                                                                               withProtocol:(self)];
            dataClient = dataIntentClient;
        }
    }
}
```

The client library provides pre-implemented recognition client classes for typical scenarios in speech recognition:

- `DataRecognitionClient`: Speech recognition with PCM data (for example, from a file or audio source). The data is broken up into buffers, and each buffer is sent to Speech Service. No modification is done to the buffers, so users can apply their own silence detection, if desired. If the data is provided from WAV files, you can send data from the file right to the server. If you have raw data, for example, audio coming over Bluetooth, you first send a format header to the server followed by the data.
- `MicrophoneRecognitionClient`: Speech recognition with audio coming from the microphone. Make sure the microphone is turned on and that data from the microphone is sent to the speech recognition service. A built-in "Silence Detector" is applied to the microphone data before it's sent to the recognition service.
- `DataRecognitionClientWithIntent` and `MicrophoneRecognitionClientWithIntent`: In addition to recognition text, these clients return structured information about the intent of the speaker, which your applications can use to drive further actions. To use "Intent," you need to first train a model by using LUIS.

Recognition language

When you use `SpeechRecognitionServiceFactory` to create the client, you must select a language. For the complete list of languages supported by Speech Service, see Supported languages.

SpeechRecognitionMode

You also need to specify `SpeechRecognitionMode` when you create the client with `SpeechRecognitionServiceFactory`:

- `SpeechRecognitionMode_ShortPhrase`: An utterance up to 15 seconds long. As data is sent to the service, the client receives multiple partial results and one final result with multiple n-best choices.
- `SpeechRecognitionMode_LongDictation`: An utterance up to two minutes long. As data is sent to the service, the client receives multiple partial results and multiple final results, based on where the server identifies sentence pauses.

Attach event handlers

You can attach various event handlers to the client you created:

- **Partial Results events**: This event gets called every time that Speech Service predicts what you might be saying, even before you finish speaking (if you use `MicrophoneRecognitionClient`) or finish sending data (if you use `DataRecognitionClient`).
- **Error events**: Called when the service detects an error.
- **Intent events**: Called on "WithIntent" clients (only in ShortPhrase mode) after the final recognition result is parsed into a structured JSON intent.
- **Result events**:
  - In `SpeechRecognitionMode_ShortPhrase` mode, this event is called and returns n-best results after you finish speaking.
  - In `SpeechRecognitionMode_LongDictation` mode, the event handler is called multiple times, based on where the service identifies sentence pauses.
  - **For each of the n-best choices**, a confidence value and a few different forms of the recognized text are returned. For more information, see Output format.

Related topics

- Client library reference for iOS
- Get started with Microsoft speech recognition and/or Intent in Java on Android
- Get started with the Microsoft Speech API in JavaScript
- Get started with the Microsoft Speech API via REST

Microsoft speech client samples

4/19/2018 • 1 min to read • Edit Online

Microsoft Speech Service provides end-to-end samples showing how to use Microsoft speech recognition API in different use cases, for example command recognition, continuous recognition, and intent detection. All samples are available on GitHub, and can be downloaded by the following links: The README.md in each repository as well as the client libraries page provide details about how to build and run the samples.

- REST API samples
- JavaScript samples
- C# Desktop client samples
- C# service samples
- Java on Android samples
- ObjectiveC on iOS samples

Basic concepts

4/19/2018 • 18 min to read • Edit Online

This page describes some basic concepts in Microsoft speech recognition service. We recommend you to read this page before using Microsoft speech recognition API in your application.

Understanding speech recognition

If this is the first time you're creating a speech-enabled application, or if it's the first time you're adding speech capabilities to an existing application, this section helps you get started. If you already have some experience with speech-enabled applications, you may choose to just skim this section, or you may skip it entirely if you're an old hand at speech and you want to get right to the protocol details.

Audio streams

Foremost among the basic concepts of speech is the *audio stream*. Unlike a keystroke, which occurs at a single point in time and contains a single piece of information, a spoken request is spread over hundreds of milliseconds and contains many kilobytes of information. The duration of spoken utterances presents some difficulty to developers looking to provide a streamlined and elegant speech experience for their application. Today's computers and algorithms perform speech transcription in approximately half of the duration of the utterance, so a 2-second utterance can be transcribed in roughly 1 second, but any application that experiences a 1-second delay in processing user is neither streamlined nor elegant.

Fortunately, there are ways of "hiding" the transcription time by performing transcription on one part of the utterance while the user is speaking another part. For example, by splitting a 1-second utterance into 10 chunks of 100 milliseconds and by performing transcription on each chunk in turn, over 450 of the total 500 milliseconds required for transcription can be "hidden" so that the user is unaware transcription is being performed while he/she is speaking. When thinking about this example, remember that the service is performing transcription on the previous 100 milliseconds of audio while the user is speaking the next 100, so when the user stops speaking, the service will only have to transcribe roughly 100 milliseconds of audio to produce a result.

To achieve this user experience, spoken audio information is collected in chunks and transcribed as the user speaks. These audio chunks collectively from the *audio stream*, and the process of sending these audio chunks to the service is called *audio streaming*. Audio streaming is an important part of any speech-enabled application; tuning the chunk size and optimizing the streaming implementation are some of the most impactful ways of improving your application's user experience.

Microphones

People process spoken audio using their ears, but inanimate hardware uses microphones. To enable speech in any application, you need to integrate with the microphone providing the audio stream for your application.

The APIs for your microphone must allow you to start and stop receiving audio bytes from the microphone. You need to decide what user actions will trigger the microphone to start listening for speech. You may choose to have a button press trigger the start of listening, or you may choose to have a *key word* or *wake word* spotter always listening to the microphone and to use the output of that module to trigger sending audio to the Microsoft Speech Service.

End of speech

Detecting *when* a speaker has *stopped* speaking seems simple enough for humans but is a rather difficult problem outside of laboratory conditions. It is not enough to simply look for pure silence after an utterance, since there is often a lot of ambient noise to complicate things. The Microsoft Speech Service does an excellent job of quickly detecting when a user has stopped speaking, and the service can inform your application of this fact, but this

arrangement does mean that your application is the last to know when the user stop speaking. This isn't at all like other forms of input where your application is the *first* to know when the user's input starts *and* ends.

Asynchronous service responses

The fact that your application needs to be informed of when user input is complete doesn't impose any performance penalties or programming difficulties on your application, but it does require that you think about speech requests differently from the input request/response patterns with which you are familiar. Since your application won't know when the user stops speaking, your application must continue to stream audio to the service while simultaneously and asynchronously waiting on a response from the service. This pattern is unlike other request/response web protocols like HTTP. In those protocols, you must complete a request before receiving any response; in the Microsoft Speech Service protocol, you receive responses *while you are still streaming audio for the request*.

> **NOTE**
> This feature is not supported when using Speech HTTP REST API.

Turns

Speech is a carrier of information. When you speak, you are trying to convey information that is in your possession to someone who is listening for that information. When conveying information, you usually take turns speaking and listening. Likewise, your speech-enabled application interacts with users by alternately listening and responding, although your application usually does most of the listening. The user's spoken input and the service response to this input is called a *turn*. A *turn* starts when the user speaks and ends when your application has completed handling of the speech service response.

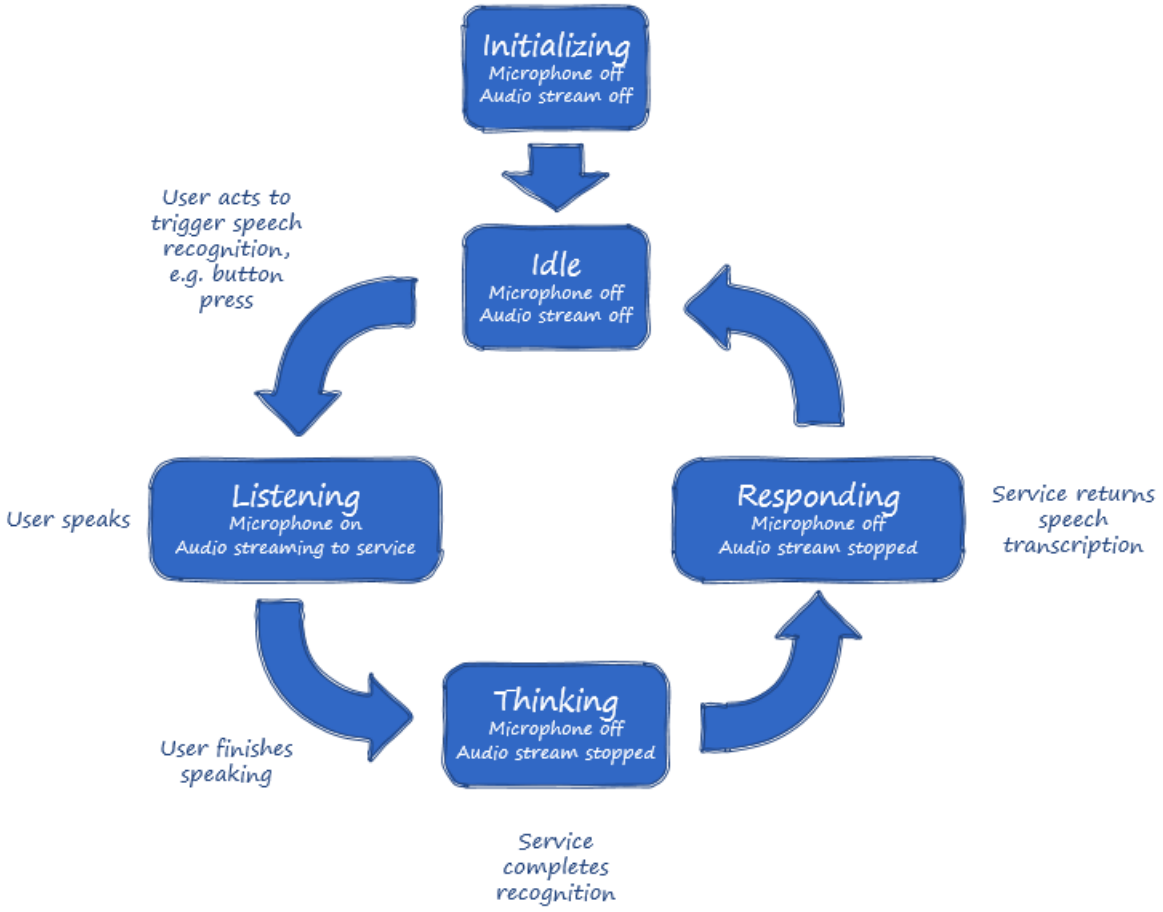
Telemetry

Creating a speech-enabled device or application can be challenging, even for experienced developers. Stream-based protocols often seem daunting at first glance, and important details like silence detection may be completely new. With so many messages needing to be successfully sent and received to complete a single request/response pair, it is *very* important to collect complete and accurate data about those messages. The Microsoft Speech Service protocol provides for the collection of this data. You should make every effort to supply the required data as accurately as possible; by supplying complete and accurate data, you will be helping yourself -- should you ever need help from the Microsoft Speech Service Team in troubleshooting your client implementation, the quality of the telemetry data you have gathered will be critical for problem analysis.

> **NOTE**
> This feature is not supported when using speech recognition REST API.

Speech application states

The steps you take to enable speech input in your application are a little different than the steps for other forms of input such as mouse clicks or finger taps. You must keep track of when your application is listening to the microphone and sending data to the speech service, when it is waiting for a response from the service and when it is in an idle state. The relationship between these states is shown in the diagram below.



Since the Microsoft Speech Service participates in some of the states, the service protocol defines messages that help your application transition between states. Your application needs to interpret and act on these protocol messages to track and manage the speech application states.

Using the speech recognition service from your apps

Microsoft speech recognition service provides two ways for developers to add Speech to their apps.

- REST APIs: Developers can use HTTP calls from their apps to the service for speech recognition.
- Client libraries: For advanced features, developers can download Microsoft Speech client libraries, and link into their apps. The client libraries are available on various platforms (Windows, Android, iOS) using different languages (C#, Java, JavaScript, ObjectiveC).

USE CASES	REST APIS	CLIENT LIBRARIES
Convert a short spoken audio, for example, commands (audio length < 15 s) without interim results	Yes	Yes
Convert a long audio (> 15 s)	No	Yes
Stream audio with interim results desired	No	Yes
Understand the text converted from audio using LUIS	No	Yes

If your language or platform does not yet have an SDK, you can create your own implementation based on the

protocol documentation.

Recognition modes

There are three modes of recognition: `interactive`, `conversation`, and `dictation`. The recognition mode adjusts speech recognition based on how the users are likely to speak. Choose the appropriate recognition mode for your application.

> **NOTE**
>
> Recognition modes might have different behaviors in the REST protocol than they do in the WebSocket protocol. For example, the REST API does not support continuous recognition, even in conversation or dictation mode.

> **NOTE**
>
> These modes are applicable when you directly use the REST or WebSocket protocol. The client libraries use different parameters to specify recognition mode. For more information, see the client library of your choice.

The Microsoft Speech Service returns only one recognition phrase result for all recognition modes. There is a limit of 15 seconds for any single utterance.

Interactive mode

In `interactive` mode, a user makes short requests and expects the application to perform an action in response.

The following characteristics are typical of interactive mode applications:

- Users know they are speaking to a machine and not to another human.
- Application users know ahead of time what they want to say, based on what they want the application to do.
- Utterances typically last about 2-3 seconds.

Conversation mode

In `conversation` mode, users are engaged in a human-to-human conversation.

The following characteristics are typical of conversation mode applications:

- Users know that they are talking to another person.
- Speech recognition enhances the human conversations by allowing one or both participants to see the spoken text.
- Users do not always plan what they want to say.
- Users frequently use slang and other informal speech.

Dictation mode

In `dictation` mode, users recite longer utterances to the application for further processing.

The following characteristics are typical of dictation mode applications:

- Users know that they are talking to a machine.
- Users are shown the speech recognition text results.
- Users often plan what they want to say and use more formal language.
- Users employ full sentences that last 5-8 seconds.

> **NOTE**
> In dictation and conversation modes, the Microsoft Speech Service does not return partial results. Instead, the service returns stable phrase results after silence boundaries in the audio stream. Microsoft might enhance the speech protocol to improve the user experience in these continuous recognition modes.

Recognition languages

The *recognition language* specifies the language that your application user speaks. Specify the *recognition language* with the *language* URL query parameter on the connection. The value of the *language* query parameter uses the IETF language tag BCP 47, and **must** be one of the languages that are supported by speech recognition API. The complete list of languages supported by the Speech Service can be found in the page Supported Languages.

The Microsoft Speech Service rejects invalid connection requests by displaying an `HTTP 400 Bad Request` response. An invalid request is one that:

- Does not include a *language* query parameter value.
- Includes a *language* query parameter that is incorrectly formatted.
- Includes a *language* query parameter that is not one of the support languages.

You may choose to build an application that supports one or all of the languages that are supported by the service.

Example

In the following example, an application uses *conversation* speech recognition mode for a US English speaker.

```
https://speech.platform.bing.com/speech/recognition/conversation/cognitiveservices/v1?language=en-US
```

Transcription responses

The transcription responses return the converted text from audio to clients. A transcription response contains the following fields:

- `RecognitionStatus` specifies the status of the recognition. The possible values are given in the table below.

STATUS	DESCRIPTION
Success	The recognition was successful and the DisplayText field is present
NoMatch	Speech was detected in the audio stream, but no words from the target language were matched. See [NoMatch Recognition Status(#nomatch-recognition-status) for more details
InitialSilenceTimeout	The start of the audio stream contained only silence, and the service timed out waiting for speech
BabbleTimeout	The start of the audio stream contained only noise, and the service timed out waiting for speech
Error	The recognition service encountered an internal error and could not continue

- `DisplayText` represents the recognized phrase after capitalization, punctuation, and inverse-text-

normalization have been applied and profanity has been masked with asterisks. The DisplayText field is present *only* if the `RecognitionStatus` field has the value `Success`.

- `Offset` specifies the offset (in 100-nanosecond units) at which the phrase was recognized, relative to the start of the audio stream.
- `Duration` specifies the duration (in 100-nanosecond units) of this speech phrase.

A transcription response returns more information if desired. See output format for how to return more detailed outputs.

Microsoft Speech Service supports additional transcription process that includes adding capitalization and punctuation, masking profanity, and normalizing text to common forms. For example, if a user speaks a phrase represented by the words "remind me to buy six iPhones", Microsoft's Speech Services will return the transcribed text "Remind me to buy 6 iPhones." The process that converts the word "six" to the number "6" is called *Inverse Text Normalization* (*ITN* for short).

NoMatch recognition status

The transcription response returns `NoMatch` in `RecognitionStatus` when the Microsoft Speech Service detects speech in the audio stream but is unable to match that speech to the language grammar being used for the request. For example, a *NoMatch* condition might occur if a user says something in German when the recognizer expects US English as the spoken language. The waveform pattern of the utterance would indicate the presence of human speech, but none of the words spoken would match the US English lexicon being used by the recognizer.

Another *NoMatch* condition occurs when the recognition algorithm is unable to find an accurate match for the sounds contained in the audio stream. When this condition happens, the Microsoft Speech Service may produce *speech.hypothesis* messages that contain *hypothesized text* but will produce a *speech.phrase* message in which the *RecognitionStatus* is *NoMatch*. This condition is normal; you must not make any assumptions about the accuracy or fidelity of the text in the *speech.hypothesis* message. Furthermore, you must not assume that because the Microsoft Speech Service produces *speech.hypothesis* messages that the service is able to produce a *speech.phrase* message with *RecognitionStatus Success*.

Output format

Microsoft Speech Service can return a variety of payload formats in transcription responses. All payloads are JSON structures.

You can control the phrase result format by specifying the `format` URL query parameter. By default, the service returns `simple` results.

FORMAT	DESCRIPTION
`simple`	A simplified phrase result containing the recognition status and the recognized text in display form.
`detailed`	A recognition status and N-best list of phrase results where each phrase result contains all four recognition forms and a confidence score.

The `detailed` format contains N-best values, in addition to `RecognitionStatus`, `Offset`, and `duration`, in the response.

N-best values

Listeners, whether human or machine, can never be certain that they heard *exactly* what was spoken. A listener can assign a *probability* only to a particular interpretation of an utterance.

In normal conditions, when speaking to others with whom they frequently interact, people have a high probability of recognizing the words that were spoken. Machine-based speech listeners strive to achieve similar accuracy levels and, under the right conditions, they achieve parity with humans.

The algorithms that are used in speech recognition explore alternative interpretations of an utterance as part of normal processing. Usually, these alternatives are discarded as the evidence in favor of a single interpretation becomes overwhelming. In less than optimal conditions, however, the speech recognizer finishes with a list of alternate possible interpretations. The top *N* alternatives in this list are called the *N-best list*. Each alternative is assigned a confidence score. Confidence scores range from 0 to 1. A score of 1 represents the highest level of confidence. A score of 0 represents the lowest level of confidence.

> **NOTE**
>
> The number of entries in the N-best list vary across multiple utterances. The number of entries can vary across multiple recognitions of the *same* utterance. This variation is a natural and expected outcome of the probabilistic nature of the speech recognition algorithm.

Each entry returned in the N-best list contains

- `Confidence`, which represents the confidence scores of this entry.
- `Lexical`, which is the lexical form of the recognized text.
- `ITN`, which is the ITN form of the recognized text.
- `MaskedITN`, which is the masked ITN form of the recognized text.
- `Display`, which is the display form of the recognized text.

Confidence scores

Confidence scores are integral to speech recognition systems. The Microsoft Speech Service obtains confidence scores from a *confidence classifier*. Microsoft trains the confidence classifier over a set of features that are designed to maximally discriminate between correct and incorrect recognition. Confidence scores are evaluated for individual words and entire utterances.

If you choose to use the confidence scores that are returned by the service, be aware of the following behavior:

- Confidence scores can be compared only within the same recognition mode and language. Do not compare scores between different languages or different recognition modes. For example, a confidence score in interactive recognition mode has *no* correlation to a confidence score in dictation mode.
- Confidence scores are best used on a restricted set of utterances. There is naturally a great degree of variability in the scores for a large set of utterances.

If you choose to use a confidence score value as a *threshold* on which your application acts, use speech recognition to establish the threshold values.

- Execute speech recognition on a representative sample of utterances for your application.
- Collect the confidence scores for each recognition in the sample set.
- Base your threshold value on some percentile of confidence for that sample.

No single threshold value is appropriate for all applications. An acceptable confidence score for one application might be unacceptable for another application.

lexical form

The lexical form is the recognized text, exactly how it occurred in the utterance and without punctuation or capitalization. For example, the lexical form of the address "1020 Enterprise Way" would be *ten twenty enterprise way*, assuming that it was spoken that way. The lexical form of the sentence "Remind me to buy 5 pencils" is *remind me to buy five pencils*.

The lexical form is most appropriate for applications that need to perform non-standard text normalization. The lexical form is also appropriate for applications that need unprocessed recognition words.

Profanity is never masked in the lexical form.

ITN form

Text normalization is the process of converting text from one form to another "canonical" form. For example, the phone number "555-1212" might be converted to the canonical form *five five five one two one two*. *Inverse* text normalization (ITN) reverses this process, converting the words "five five five one two one two" to the inverted canonical form *555-1212*. The ITN form of a recognition result does not include capitalization or punctuation.

The ITN form is most appropriate for applications that act on the recognized text. For example, an application that allows a user to speak search terms and then uses these terms in a web query would use the ITN form. Profanity is never masked in the ITN form. To mask profanity, use the *Masked ITN form*.

Masked ITN form

Because profanity is naturally a part of spoken language, the Microsoft Speech Service recognizes such words and phrases when they are spoken. Profanity might not, however, be appropriate for all applications, especially those applications with a restricted, non-adult user audience.

The masked ITN form applies profanity masking to the inverse text normalization form. To mask profanity, set the value of the profanity parameter value to `masked`. When profanity is masked, words that are recognized as part of the language's profanity lexicon are replaced with asterisks. For example: *remind me to buy 5 **** pencils*. The masked ITN form of a recognition result does not include capitalization or punctuation.

> **NOTE**
> If the profanity query parameter value is set to `raw`, the masked ITN form is the same as the ITN form. Profanity is *not* masked.

Display form

Punctuation and capitalization signal where to put emphasis, where to pause, and so on, which makes text easier to understand. The display form adds punctuation and capitalization to recognition results, making it the most appropriate form for applications that display the spoken text.

Because the display form extends the masked ITN form, you can set the profanity parameter value to `masked` or `raw`. If the value is set to `raw`, the recognition results include any profanity spoken by the user. If the value is set to `masked`, words recognized as part of the language's profanity lexicon are replaced with asterisks.

Sample responses

All payloads are JSON structures.

The payload format of the `simple` phrase result:

```
{
  "RecognitionStatus": "Success",
  "DisplayText": "Remind me to buy 5 pencils.",
  "Offset": "1236645672289",
  "Duration": "1236645672289"
}
```

The payload format of the `detailed` phrase result:

```
{
  "RecognitionStatus": "Success",
  "Offset": "1236645672289",
  "Duration": "1236645672289",
  "NBest": [
      {
        "Confidence" : "0.87",
        "Lexical" : "remind me to buy five pencils",
        "ITN" : "remind me to buy 5 pencils",
        "MaskedITN" : "remind me to buy 5 pencils",
        "Display" : "Remind me to buy 5 pencils.",
      },
      {
        "Confidence" : "0.54",
        "Lexical" : "rewind me to buy five pencils",
        "ITN" : "rewind me to buy 5 pencils",
        "MaskedITN" : "rewind me to buy 5 pencils",
        "Display" : "Rewind me to buy 5 pencils.",
      }
  ]
}
```

Profanity-handling in speech recognition

The Microsoft Speech Service recognizes all forms of human speech, including words and phrases that many people would classify as "profanity." You can control how the service handles profanity by using the *profanity* query parameter. By default, the service masks profanity in *speech.phrase* results and does not return *speech.hypothesis* messages that contain profanity.

***PROFANITY* VALUE**	**DESCRIPTION**
`masked`	Masks profanity with asterisks. This behavior is the default.
`removed`	Removes profanity from all results.
`raw`	Recognizes and returns profanity in all results.

Profanity value `Masked`

To mask profanity, set the *profanity* query parameter to the value *masked*. When the *profanity* query parameter has this value or is not specified for a request, the service *masks* profanity. The service performs masking by replacing profanity in the recognition results with asterisks. When you specify profanity-masking handling, the service does not return *speech.hypothesis* messages that contain profanity.

Profanity value `Removed`

When the *profanity* query parameter has the value *removed*, the service removes profanity from both *speech.phrase* and *speech.hypothesis* messages. The results are the same *as if the profanity words were not spoken*.

Profanity-only utterances

A user might speak *only* profanity when an application has configured the service to remove profanity. For this scenario, if the recognition mode is *dictation* or *conversation*, the service does not return a *speech.result*. If the recognition mode is *interactive*, the service returns a *speech.result* with the status code *NoMatch*.

Profanity value `Raw`

When the *profanity* query parameter has the value *raw*, the service does not remove or mask profanity in either the *speech.phrase* or *speech.hypothesis* messages.

Authenticate to the Speech API

4/19/2018 • 5 min to read • Edit Online

Speech Service supports authentication by using:

- A subscription key.
- An authorization token.

Use a subscription key

To use Speech Service, you must first subscribe to the Speech API that's part of Cognitive Services (previously Project Oxford). You can get free trial subscription keys from the Cognitive Services subscription page. After you select the Speech API, select **Get API Key** to get the key. It returns a primary and secondary key. Both keys are tied to the same quota, so you can use either key.

For long-term use or an increased quota, sign up for an Azure account.

To use the Speech REST API, you need to pass the subscription key in the `Ocp-Apim-Subscription-Key` field in the request header.

NAME	FORMAT	DESCRIPTION
Ocp-Apim-Subscription-Key	ASCII	YOUR_SUBSCRIPTION_KEY

The following is an example of a request header:

```
POST https://speech.platform.bing.com/speech/recognition/interactive/cognitiveservices/v1?language=en-
US&format=detailed HTTP/1.1
Accept: application/json;text/xml
Content-Type: audio/wav; codec=audio/pcm; samplerate=16000
Ocp-Apim-Subscription-Key: YOUR_SUBSCRIPTION_KEY
Host: speech.platform.bing.com
Transfer-Encoding: chunked
Expect: 100-continue
```

IMPORTANT

If you use client libraries in your application, verify that you can get the authorization token with your subscription key, as described in the following section. The client libraries use the subscription key to get an authorization token and then use the token for authentication.

Use an authorization token

Alternatively, you can use an authorization token for authentication as proof of authentication/authorization. To get this token, you must first obtain a subscription key from the Speech API, as described in the preceding section.

Get an authorization token

After you have a valid subscription key, send a POST request to the token service of Cognitive Services. In the response, you receive the authorization token as a JSON Web Token (JWT).

> **NOTE**
> The token has an expiration of 10 minutes. To renew the token, see the following section.

The token service URI is located here:

```
https://api.cognitive.microsoft.com/sts/v1.0/issueToken
```

The following code sample shows how to get an access token. Replace `YOUR_SUBSCRIPTION_KEY` with your own subscription key:

- PowerShell
- curl
- C#

```
$FetchTokenHeader = @{
  'Content-type'='application/x-www-form-urlencoded';
  'Content-Length'= '0';
  'Ocp-Apim-Subscription-Key' = 'YOUR_SUBSCRIPTION_KEY'
}

$OAuthToken = Invoke-RestMethod -Method POST -Uri https://api.cognitive.microsoft.com/sts/v1.0/issueToken -
Headers $FetchTokenHeader

# show the token received
$OAuthToken
```

The following is a sample POST request:

```
POST https://api.cognitive.microsoft.com/sts/v1.0/issueToken HTTP/1.1
Ocp-Apim-Subscription-Key: YOUR_SUBSCRIPTION_KEY
Host: api.cognitive.microsoft.com
Content-type: application/x-www-form-urlencoded
Content-Length: 0
Connection: Keep-Alive
```

If you cannot get an authorization token from the token service, check whether your subscription key is still valid. If you are using a free trial key, go to the Cognitive Services subscription page, click on "Log in" to login using the account that you used for applying the free trial key, and check whether the subscription key is expired or exceeds the quota.

Use an authorization token in a request

Each time you call the Speech API, you need to pass the authorization token in the `Authorization` header. The `Authorization` header must contain a JWT access token.

The following example shows how to use an authorization token when you call the Speech REST API.

> **NOTE**
> Replace `YOUR_AUDIO_FILE` with the path to your prerecorded audio file. Replace `YOUR_ACCESS_TOKEN` with the authorization token you got in the previous step Get an authorization token.

- PowerShell
- curl

- C#

```
$SpeechServiceURI =
'https://speech.platform.bing.com/speech/recognition/interactive/cognitiveservices/v1?language=en-
us&format=detailed'

# $OAuthToken is the authrization token returned by the token service.
$RecoRequestHeader = @{
  'Authorization' = 'Bearer '+ $OAuthToken;
  'Transfer-Encoding' = 'chunked'
  'Content-type' = 'audio/wav; codec=audio/pcm; samplerate=16000'
}

# Read audio into byte array
$audioBytes = [System.IO.File]::ReadAllBytes("YOUR_AUDIO_FILE")

$RecoResponse = Invoke-RestMethod -Method POST -Uri $SpeechServiceURI -Headers $RecoRequestHeader -Body
$audioBytes

# Show the result
$RecoResponse
```

Renew an authorization token

The authorization token expires after a certain time period (currently 10 minutes). You need to renew the authorization token before it expires.

The following code is an example implementation in C# of how to renew the authorization token:

```
    /*
     * This class demonstrates how to get a valid O-auth token.
     */
    public class Authentication
    {
        public static readonly string FetchTokenUri = "https://api.cognitive.microsoft.com/sts/v1.0";
        private string subscriptionKey;
        private string token;
        private Timer accessTokenRenewer;

        //Access token expires every 10 minutes. Renew it every 9 minutes.
        private const int RefreshTokenDuration = 9;

        public Authentication(string subscriptionKey)
        {
            this.subscriptionKey = subscriptionKey;
            this.token = FetchToken(FetchTokenUri, subscriptionKey).Result;

            // renew the token on set duration.
            accessTokenRenewer = new Timer(new TimerCallback(OnTokenExpiredCallback),
                                           this,
                                           TimeSpan.FromMinutes(RefreshTokenDuration),
                                           TimeSpan.FromMilliseconds(-1));
        }

        public string GetAccessToken()
        {
            return this.token;
        }

        private void RenewAccessToken()
        {
            this.token = FetchToken(FetchTokenUri, this.subscriptionKey).Result;
            Console.WriteLine("Renewed token.");
        }
```

```
        private void OnTokenExpiredCallback(object stateInfo)
        {
            try
            {
                RenewAccessToken();
            }
            catch (Exception ex)
            {
                Console.WriteLine(string.Format("Failed renewing access token. Details: {0}", ex.Message));
            }
            finally
            {
                try
                {
                    accessTokenRenewer.Change(TimeSpan.FromMinutes(RefreshTokenDuration),
TimeSpan.FromMilliseconds(-1));
                }
                catch (Exception ex)
                {
                    Console.WriteLine(string.Format("Failed to reschedule the timer to renew access token.
Details: {0}", ex.Message));
                }
            }
        }

        private async Task<string> FetchToken(string fetchUri, string subscriptionKey)
        {
            using (var client = new HttpClient())
            {
                client.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", subscriptionKey);
                UriBuilder uriBuilder = new UriBuilder(fetchUri);
                uriBuilder.Path += "/issueToken";

                var result = await client.PostAsync(uriBuilder.Uri.AbsoluteUri, null);
                Console.WriteLine("Token Uri: {0}", uriBuilder.Uri.AbsoluteUri);
                return await result.Content.ReadAsStringAsync();
            }
        }
    }
```

Speech recognition modes

4/19/2018 • 1 min to read • Edit Online

Microsoft's *Speech to Text* APIs support multiple modes of speech recognition. Choose the mode that produces the best recognition results for your application.

MODE	DESCRIPTION
interactive	"Command and control" recognition for interactive user application scenarios. Users speak short phrases intended as commands to an application.
dictation	Continuous recognition for dictation scenarios. Users speak longer sentences that are displayed as text. Users adopt a more formal speaking style.
conversation	Continuous recognition for transcribing conversations between humans. Users adopt a less formal speaking style and may alternate between longer sentences and shorter phrases.

NOTE

These modes are applicable when you use the REST APIs. The client libraries use different parameters to specify recognition mode. For more information, see the client library of your choice.

For more information, see the Recognition Modes page.

Chunked transfer encoding

4/19/2018 • 1 min to read • Edit Online

To transcribe speech to text, Microsoft speech recognition API allows you to send the audio as one whole chunk or to chop the audio into small chunks. For efficient audio streaming and reducing transcription latency, it is recommended that you use chunked transfer encoding to stream the audio to the service. Other implementations may result in higher user-perceived latency. For more information, see the Audio Streams page.

> **NOTE**
>
> You may not upload more than 10 seconds of audio in any one request and the total request duration cannot exceed 14 seconds.

> **NOTE**
>
> You need to specify the chunked transfer encoding only if you use the REST APIs to call the speech service. Applications that use client libraries do not need to configure the chunked transfer encoding.

The following code shows how to set the chunked transfer encoding and to send an audio file being chunked into 1024-byte chunks.

```
HttpWebRequest request = null;
request = (HttpWebRequest)HttpWebRequest.Create(requestUri);
request.SendChunked = true;
request.Accept = @"application/json;text/xml";
request.Method = "POST";
request.ProtocolVersion = HttpVersion.Version11;
request.Host = @"speech.platform.bing.com";
request.ContentType = @"audio/wav; codec=audio/pcm; samplerate=16000";
request.Headers["Ocp-Apim-Subscription-Key"] = "YOUR_SUBSCRIPTION_KEY";

using (fs = new FileStream(audioFile, FileMode.Open, FileAccess.Read))
{

    /*
    * Open a request stream and write 1024 byte chunks in the stream one at a time.
    */
    byte[] buffer = null;
    int bytesRead = 0;
    using (Stream requestStream = request.GetRequestStream())
    {
        /*
        * Read 1024 raw bytes from the input audio file.
        */
        buffer = new Byte[checked((uint)Math.Min(1024, (int)fs.Length))];
        while ((bytesRead = fs.Read(buffer, 0, buffer.Length)) != 0)
        {
            requestStream.Write(buffer, 0, bytesRead);
        }

        // Flush
        requestStream.Flush();
    }
}
```

Speech Service WebSocket protocol

4/19/2018 • 32 min to read • Edit Online

Speech Service is a cloud-based platform that features the most advanced algorithms available for converting spoken audio to text. The Speech Service protocol defines the connection setup between client applications and the service and the speech recognition messages exchanged between counterparts (client-originated Messages and service-originated messages). In addition, telemetry messages and error handling are described.

Connection establishment

The Speech Service protocol follows the WebSocket standard specification IETF RFC 6455. A WebSocket connection starts out as an HTTP request that contains HTTP headers that indicate the client's desire to upgrade the connection to a WebSocket instead of using HTTP semantics. The server indicates its willingness to participate in the WebSocket connection by returning an HTTP `101 Switching Protocols` response. After the exchange of this handshake, both client and service keep the socket open and begin using a message-based protocol to send and receive information.

To begin the WebSocket handshake, the client application sends an HTTPS GET request to the service. It includes standard WebSocket upgrade headers along with other headers that are specific to speech.

```
GET /speech/recognition/interactive/cognitiveservices/v1 HTTP/1.1
Host: speech.platform.bing.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
Authorization: t=EwCIAgALBAAUWkziSCJKS1VkhugDegv7L0eAAJqBYKKTzpPZOeGk7RfZmdBhYY28jl&p=
X-ConnectionId: A140CAF92F71469FA41C72C7B5849253
Origin: https://speech.platform.bing.com
```

The service responds with:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: upgrade
Sec-WebSocket-Key: 2PTTXbeeBXlrrUNsY15n01d/Pcc=
Set-Cookie: SpeechServiceToken=AAAAABAAWTC8ncb8COL; expires=Wed, 17 Aug 2016 15:39:06 GMT; domain=bing.com;
path="/"
Date: Wed, 17 Aug 2016 15:03:52 GMT
```

All speech requests require the TLS encryption. The use of unencrypted speech requests is not supported. The following TLS version is supported:

- TLS 1.2

Connection identifier

Speech Service requires that all clients include a unique ID to identify the connection. Clients *must* include the *X-ConnectionId* header when they start a WebSocket handshake. The *X-ConnectionId* header must be a universally unique identifier (UUID) value. WebSocket upgrade requests that do not include the *X-ConnectionId*, do not specify a value for the *X-ConnectionId* header, or do not include a valid UUID value are rejected by the service with an HTTP `400 Bad Request` response.

Authorization

In addition to the standard WebSocket handshake headers, speech requests require an *Authorization* header. Connection requests without this header are rejected by the service with an HTTP `403 Forbidden` response.

The *Authorization* header must contain a JSON Web Token (JWT) access token.

For information about how to subscribe and obtain API keys that are used to retrieve valid JWT access tokens, see the Cognitive Services subscription page.

The API key is passed to the token service. For example:

```
POST https://api.cognitive.microsoft.com/sts/v1.0/issueToken
Content-Length: 0
```

The following header information is required for token access.

NAME	FORMAT	DESCRIPTION
Ocp-Apim-Subscription-Key	ASCII	Your subscription key

The token service returns the JWT access token as `text/plain`. Then the JWT is passed as a `Base64 access_token` to the handshake as an *Authorization* header prefixed with the string `Bearer`. For example:

```
Authorization: Bearer [Base64 access_token]
```

Cookies

Clients *must* support HTTP cookies as specified in RFC 6265.

HTTP redirection

Clients *must* support the standard redirection mechanisms specified by the HTTP protocol specification.

Speech endpoints

Clients *must* use an appropriate endpoint of Speech Service. The endpoint is based on recognition mode and language. The table shows some examples.

MODE	PATH	SERVICE URI
Interactive	/speech/recognition/interactive/cognitiveservices/v1	https://speech.platform.bing.com/speech/recognition/interactive/cognitiveservices/v1?language=pt-BR
Conversation	/speech/recognition/conversation/cognitiveservices/v1	https://speech.platform.bing.com/speech/recognition/conversation/cognitiveservices/v1?language=en-US
Dictation	/speech/recognition/dictation/cognitiveservices/v1	https://speech.platform.bing.com/speech/recognition/dictation/cognitiveservices/v1?language=fr-FR

For more information, see the Service URI page.

Report connection problems

Clients should immediately report all problems encountered while making a connection. The message protocol for reporting failed connections is described in Connection failure telemetry.

Connection duration limitations

When compared with typical web service HTTP connections, WebSocket connections last a *long* time. Speech

Service places limitations on the duration of the WebSocket connections to the service:

- The maximum duration for any active WebSocket connection is 10 minutes. A connection is active if either the service or the client sends WebSocket messages over that connection. The service terminates the connection without warning when the limit is reached. Clients should develop user scenarios that do not require the connection to remain active at or near the maximum connection lifetime.
- The maximum duration for any inactive WebSocket connection is 180 seconds. A connection is inactive if neither the service nor the client sent a WebSocket message over the connection. After the maximum inactive lifetime is reached, the service terminates the inactive WebSocket connection.

Message types

After a WebSocket connection is established between the client and the service, both the client and the service can send messages. This section describes the format of these WebSocket messages.

IETF RFC 6455 specifies that WebSocket messages can transmit data by using either a text or a binary encoding. The two encodings use different on-the-wire formats. Each format is optimized for efficient encoding, transmission, and decoding of the message payload.

Text WebSocket messages

Text WebSocket messages carry a payload of textual information that consists of a section of headers and a body separated by the familiar double-carriage-return newline pair used for HTTP messages. And, like HTTP messages, text WebSocket messages specify headers in *name: value* format separated by a single-carriage-return newline pair. Any text included in a text WebSocket message *must* use UTF-8 encoding.

Text WebSocket messages must specify a message path in the header *Path*. The value of this header must be one of the speech protocol message types defined later in this document.

Binary WebSocket messages

Binary WebSocket messages carry a binary payload. In the Speech Service protocol, audio is transmitted to and received from the service by using binary WebSocket messages. All other messages are text WebSocket messages.

Like text WebSocket messages, binary WebSocket messages consist of a header and a body section. The first 2 bytes of the binary WebSocket message specify, in big-endian order, the 16-bit integer size of the header section. The minimum header section size is 0 bytes. The maximum size is 8,192 bytes. The text in the headers of binary WebSocket messages *must* use US-ASCII encoding.

Headers in a binary WebSocket message are encoded in the same format as in text WebSocket messages. The *name:value* format is separated by a single-carriage-return newline pair. Binary WebSocket messages must specify a message path in the header *Path*. The value of this header must be one of the speech protocol message types defined later in this document.

Both text and binary WebSocket messages are used in the Speech Service protocol.

Client-originated messages

After the connection is established, both the client and the service can start to send messages. This section describes the format and payload of messages that client applications send to Speech Service. The section Service-originated messages presents the messages that originate in Speech Service and are sent to the client applications.

The main messages sent by the client to the services are `speech.config`, `audio`, and `telemetry` messages. Before we consider each message in detail, the common required message headers for all these messages are described.

Required message headers

The following headers are required for all client-originated messages.

HEADER	VALUE
Path	The message path as specified in this document
X-RequestId	UUID in "no-dash" format
X-Timestamp	Client UTC clock time stamp in ISO 8601 format

X-RequestId header

Client-originated requests are uniquely identified by the *X-RequestId* message header. This header is required for all client-originated messages. The *X-RequestId* header value must be a UUID in "no-dash" form, for example, *123e4567e89b12d3a456426655440000*. It *cannot* be in the canonical form *123e4567-e89b-12d3-a456-426655440000*. Requests without an *X-RequestId* header or with a header value that uses the wrong format for UUIDs cause the service to terminate the WebSocket connection.

X-Timestamp header

Each message sent to Speech Service by a client application *must* include an *X-Timestamp* header. The value for this header is the time when the client sends the message. Requests without an *X-Timestamp* header or with a header value that uses the wrong format cause the service to terminate the WebSocket connection.

The *X-Timestamp* header value must be of the form 'yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffZ' where 'fffffff' is a fraction of a second. For example, '12.5' means '12 + 5/10 seconds' and '12.526' means '12 plus 526/1000 seconds'. This format complies with ISO 8601 and, unlike the standard HTTP *Date* header, it can provide millisecond resolution. Client applications might round time stamps to the nearest millisecond. Client applications need to ensure that the device clock accurately tracks time by using a Network Time Protocol (NTP) server.

Message `speech.config`

Speech Service needs to know the characteristics of your application to provide the best possible speech recognition. The required characteristics data includes information about the device and OS that powers your application. You supply this information in the `speech.config` message.

Clients *must* send a `speech.config` message immediately after they establish the connection to Speech Service and before they send any `audio` messages. You need to send a `speech.config` message only once per connection.

FIELD	DESCRIPTION
WebSocket message encoding	Text
Body	The payload as a JSON structure

Required message headers

HEADER NAME	VALUE
Path	`speech.config`
X-Timestamp	Client UTC clock time stamp in ISO 8601 format
Content-Type	application/json; charset=utf-8

As with all client-originated messages in the Speech Service protocol, the `speech.config` message *must* include an *X-Timestamp* header that records the client UTC clock time when the message was sent to the service. The `speech.config` message *does not* require an *X-RequestId* header because this message isn't associated with a

particular speech request.

Message payload

The payload of the `speech.config` message is a JSON structure that contains information about the application. The following example shows this information. Client and device context information is included in the *context* element of the JSON structure.

```
{
  "context": {
    "system": {
      "version": "2.0.12341",
    },
    "os": {
      "platform": "Linux",
      "name": "Debian",
      "version": "2.14324324"
    },
    "device": {
      "manufacturer": "Contoso",
      "model": "Fabrikan",
      "version": "7.341"
      }
    },
  }
}
```

System element

The system.version element of the `speech.config` message contains the version of the speech SDK software used by the client application or device. The value is in the form *major.minor.build.branch*. You can omit the *branch* component if it's not applicable.

OS element

FIELD	DESCRIPTION	USAGE
os.platform	The OS platform that hosts the application, for example, Windows, Android, iOS, or Linux	Required
os.name	The OS product name, for example, Debian or Windows 10	Required
os.version	The version of the OS in the form *major.minor.build.branch*	Required

Device element

FIELD	DESCRIPTION	USAGE
device.manufacturer	The device hardware manufacturer	Required
device.model	The device model	Required
device.version	The device software version provided by the device manufacturer. This value specifies a version of the device that can be tracked by the manufacturer.	Required

Message `audio`

Speech-enabled client applications send audio to Speech Service by converting the audio stream into a series of

audio chunks. Each chunk of audio carries a segment of the spoken audio that's to be transcribed by the service. The maximum size of a single audio chunk is 8,192 bytes. Audio stream messages are *Binary WebSocket messages*.

Clients use the `audio` message to send an audio chunk to the service. Clients read audio from the microphone in chunks and send these chunks to Speech Service for transcription. The first `audio` message must contain a well-formed header that properly specifies that the audio conforms to one of the encoding formats supported by the service. Additional `audio` messages contain only the binary audio stream data read from the microphone.

Clients might optionally send an `audio` message with a zero-length body. This message tells the service that the client knows that the user stopped speaking, the utterance is finished, and the microphone is turned off.

Speech Service uses the first `audio` message that contains a unique request identifier to signal the start of a new request/response cycle or *turn*. After the service receives an `audio` message with a new request identifier, it discards any queued or unsent messages that are associated with any previous turn.

FIELD	DESCRIPTION
WebSocket message encoding	Binary
Body	The binary data for the audio chunk. Maximum size is 8,192 bytes.

Required message headers

The following headers are required for all `audio` messages.

HEADER	VALUE
Path	`audio`
X-RequestId	UUID in "no-dash" format
X-Timestamp	Client UTC clock time stamp in ISO 8601 format
Content-Type	The audio content type. The type must be either *audio/x-wav* (PCM) or *audio/silk* (SILK).

Supported audio encodings

This section describes the audio codecs supported by Speech Service.

PCM

Speech Service accepts uncompressed pulse code modulation (PCM) audio. Audio is sent to the service in WAV format, so the first audio chunk *must* contain a valid Resource Interchange File Format (RIFF) header. If a client initiates a turn with an audio chunk that does *not* include a valid RIFF header, the service rejects the request and terminates the WebSocket connection.

PCM audio *must* be sampled at 16 kHz with 16 bits per sample and one channel (*riff-16khz-16bit-mono-pcm*). Speech Service doesn't support stereo audio streams and rejects audio streams that do not use the specified bit rate, sample rate, or number of channels.

Opus

Opus is an open, royalty-free, highly versatile audio codec. Speech Service supports Opus at a constant bit rate of `32000` or `16000`. Only the `OGG` container for Opus is currently supported that is specified by the `audio/ogg` mime type.

To use Opus, modify the JavaScript sample and change the `RecognizerSetup` method to return.

```
return SDK.CreateRecognizerWithCustomAudioSource(
            recognizerConfig,
            authentication,
            new SDK.MicAudioSource(
                    new SDK.OpusRecorder(
                    {
                        mimeType: "audio/ogg",
                        bitsPerSecond: 32000
                    }
                )
          ));
```

Detect end of speech

Humans do not explicitly signal when they're finished speaking. Any application that accepts speech as input has two choices for handling the end of speech in an audio stream: service end-of-speech detection and client end-of-speech detection. Of these two choices, service end-of-speech detection usually provides a better user experience.

Service end-of-speech detection

To build the ideal hands-free speech experience, applications allow the service to detect when the user has finished speaking. Clients send audio from the microphone as *audio* chunks until the service detects silence and responds back with a `speech.endDetected` message.

Client end-of-speech detection

Client applications that allow the user to signal the end of speech in some way also can give the service that signal. For example, a client application might have a "Stop" or "Mute" button that the user can press. To signal end-of-speech, client applications send an *audio* chunk message with a zero-length body. Speech Service interprets this message as the end of the incoming audio stream.

Message `telemetry`

Client applications *must* acknowledge the end of each turn by sending telemetry about the turn to Speech Service. Turn-end acknowledgment allows Speech Service to ensure that all messages necessary for completion of the request and its response were properly received by the client. Turn-end acknowledgment also allows Speech Service to verify that the client applications are performing as expected. This information is invaluable if you need help troubleshooting your speech-enabled application.

Clients must acknowledge the end of a turn by sending a `telemetry` message soon after receiving a `turn.end` message. Clients should attempt to acknowledge the `turn.end` as soon as possible. If a client application fails to acknowledge the turn end, Speech Service might terminate the connection with an error. Clients must send only one `telemetry` message for each request and response identified by the *X-RequestId* value.

FIELD	DESCRIPTION
WebSocket message encoding	Text
Path	`telemetry`
X-Timestamp	Client UTC clock time stamp in ISO 8601 format
Content-Type	`application/json`
Body	A JSON structure that contains client information about the turn

The schema for the body of the `telemetry` message is defined in the Telemetry schema section.

Telemetry for interrupted connections

If the network connection fails for any reason during a turn and the client does *not* receive a `turn.end` message from the service, the client sends a `telemetry` message. This message describes the failed request the next time the client makes a connection to the service. Clients do not have to immediately attempt a connection to send the `telemetry` message. The message might be buffered on the client and sent over a future user-requested connection. The `telemetry` message for the failed request *must* use the *X-RequestId* value from the failed request. It might be sent to the service as soon as a connection is established, without waiting to send or receive for other messages.

Service-originated messages

This section describes the messages that originate in Speech Service and are sent to the client. Speech Service maintains a registry of client capabilities and generates the messages required by each client, so not all clients receive all the messages that are described here. For brevity, messages are referenced by the value of the *Path* header. For example, we refer to a WebSocket text message with the *Path* value `speech.hypothesis` as a speech.hypothesis message.

Message `speech.startDetected`

The `speech.startDetected` message indicates that Speech Service detected speech in the audio stream.

FIELD	DESCRIPTION
WebSocket message encoding	Text
Path	`speech.startDetected`
Content-Type	application/json; charset=utf-8
Body	The JSON structure that contains information about the conditions when the start of speech was detected. The *Offset* field in this structure specifies the offset (in 100-nanosecond units) when speech was detected in the audio stream, relative to the start of the stream.

Sample message

```
Path: speech.startDetected
Content-Type: application/json; charset=utf-8
X-RequestId: 123e4567e89b12d3a456426655440000

{
  "Offset": 100000
}
```

Message `speech.hypothesis`

During speech recognition, Speech Service periodically generates hypotheses about the words the service recognized. Speech Service sends these hypotheses to the client approximately every 300 milliseconds. The `speech.hypothesis` is suitable *only* for enhancing the user speech experience. You must not take any dependency on the content or accuracy of the text in these messages.

The `speech.hypothesis` message is applicable to those clients that have some text rendering capability and want to provide near-real-time feedback of the in-progress recognition to the person who is speaking.

FIELD	DESCRIPTION
WebSocket message encoding	Text
Path	`speech.hypothesis`
X-RequestId	UUID in "no-dash" format
Content-Type	application/json
Body	The speech hypothesis JSON structure

Sample message

```
Path: speech.hypothesis
Content-Type: application/json; charset=utf-8
X-RequestId: 123e4567e89b12d3a456426655440000

{
  "Text": "this is a speech hypothesis",
  "Offset": 0,
  "Duration": 23600000
}
```

The *Offset* element specifies the offset (in 100-nanosecond units) when the phrase was recognized, relative to the start of the audio stream.

The *Duration* element specifies the duration (in 100-nanosecond units) of this speech phrase.

Clients must not make any assumptions about the frequency, timing, or text contained in a speech hypothesis or the consistency of text in any two speech hypotheses. The hypotheses are just snapshots into the transcription process in the service. They do not represent a stable accumulation of transcription. For example, a first speech hypothesis might contain the words "fine fun," and the second hypothesis might contain the words "find funny." Speech Service doesn't perform any post-processing (for example, capitalization, punctuation) on the text in the speech hypothesis.

Message `speech.phrase`

When Speech Service determines that it has enough information to produce a recognition result that won't change, the service produces a `speech.phrase` message. Speech Service produces these results after it detects that the user has finished a sentence or phrase.

FIELD	DESCRIPTION
WebSocket message encoding	Text
Path	`speech.phrase`
Content-Type	application/json
Body	The speech phrase JSON structure

The speech phrase JSON schema includes the following fields: `RecognitionStatus`, `DisplayText`, `Offset`, and `Duration`. For more information about these fields, see Transcription responses.

Sample message

```
Path: speech.phrase
Content-Type: application/json; charset=utf-8
X-RequestId: 123e4567e89b12d3a456426655440000

{
  "RecognitionStatus": "Success",
  "DisplayText": "Remind me to buy 5 pencils.",
  "Offset": 0,
  "Duration": 12300000
}
```

Message `speech.endDetected`

The `speech.endDetected` message specifies that the client application should stop streaming audio to the service.

FIELD	DESCRIPTION
WebSocket message encoding	Text
Path	`speech.endDetected`
Body	The JSON structure that contains the offset when the end of speech was detected. The offset is represented in 100-nanosecond units offset from the start of audio that's used for recognition.
Content-Type	application/json; charset=utf-8

Sample message

```
Path: speech.endDetected
Content-Type: application/json; charset=utf-8
X-RequestId: 123e4567e89b12d3a456426655440000

{
  "Offset": 0
}
```

The *Offset* element specifies the offset (in 100-nanosecond units) when the phrase was recognized, relative to the start of the audio stream.

Message `turn.start`

The `turn.start` signals the start of a turn from the perspective of the service. The `turn.start` message is always the first response message you receive for any request. If you don't receive a `turn.start` message, assume that the state of the service connection is invalid.

FIELD	DESCRIPTION
WebSocket message encoding	Text
Path	`turn.start`
Content-Type	application/json; charset=utf-8
Body	JSON structure

Sample message

```
Path: turn.start
Content-Type: application/json; charset=utf-8
X-RequestId: 123e4567e89b12d3a456426655440000

{
  "context": {
    "serviceTag": "7B33613B91714B32817815DC89633AFA"
  }
}
```

The body of the `turn.start` message is a JSON structure that contains context for the start of the turn. The *context* element contains a *serviceTag* property. This property specifies a tag value that the service has associated with the turn. This value can be used by Microsoft if you need help troubleshooting failures in your application.

Message `turn.end`

The `turn.end` signals the end of a turn from the perspective of the service. The `turn.end` message is always the last response message you receive for any request. Clients can use the receipt of this message as a signal for cleanup activities and transitioning to an idle state. If you don't receive a `turn.end` message, assume that the state of the service connection is invalid. In those cases, close the existing connection to the service and reconnect.

FIELD	DESCRIPTION
WebSocket message encoding	Text
Path	`turn.end`
Body	None

Sample message

```
Path: turn.end
X-RequestId: 123e4567e89b12d3a456426655440000
```

Telemetry schema

The body of the *telemetry* message is a JSON structure that contains client information about a turn or an attempted connection. The structure is made up of client time stamps that record when client events occur. Each time stamp must be in ISO 8601 format as described in the section titled "X-Timestamp header." *Telemetry* messages that do not specify all the required fields in the JSON structure or that do not use the correct time stamp format might cause the service to terminate the connection to the client. Clients *must* supply valid values for all required fields. Clients *should* supply values for optional fields whenever appropriate. The values shown in samples in this section are for illustration only.

Telemetry schema is divided into the following parts: received message time stamps and metrics. The format and usage of each part is specified in the following sections.

Received message time stamps

Clients must include time-of-receipt values for all messages that they receive after successfully connecting to the service. These values must record the time when the client *received* each message from the network. The value should not record any other time. For example, the client should not record the time when it *acted* on the message. The received message time stamps are specified in an array of *name:value* pairs. The name of the pair specifies the *Path* value of the message. The value of the pair specifies the client time when the message was received. Or, if more than one message of the specified name was received, the value of the pair is an array of time stamps that indicates when those messages were received.

```
  "ReceivedMessages": [
    { "speech.hypothesis": [ "2016-08-16T15:03:48.172Z", "2016-08-16T15:03:48.331Z", "2016-08-16T15:03:48.881Z" ] },
    { "speech.endDetected": "2016-08-16T15:03:49.721Z" },
    { "speech.phrase": "2016-08-16T15:03:50.001Z" },
    { "turn.end": "2016-08-16T15:03:51.021Z" }
  ]
```

Clients *must* acknowledge the receipt of all messages sent by the service by including time stamps for those messages in the JSON body. If a client fails to acknowledge the receipt of a message, the service might terminate the connection.

Metrics

Clients must include information about events that occurred during the lifetime of a request. The following metrics are supported: `Connection`, `Microphone`, and `ListeningTrigger`.

Metric `Connection`

The `Connection` metric specifies details about connection attempts by the client. The metric must include time stamps of when the WebSocket connection was started and finished. The `Connection` metric is required *only for the first turn of a connection*. Subsequent turns are not required to include this information. If a client makes multiple connection attempts before a connection is established, information about *all* the connection attempts should be included. For more information, see Connection failure telemetry.

FIELD	DESCRIPTION	USAGE
Name	`Connection`	Required
Id	The connection identifier value that was used in the *X-ConnectionId* header for this connection request	Required
Start	The time when the client sent the connection request	Required
End	The time when the client received notification that the connection was established successfully or, in error cases, rejected, refused, or failed	Required
Error	A description of the error that occurred, if any. If the connection was successful, clients should omit this field. The maximum length of this field is 50 characters.	Required for error cases, omitted otherwise

The error description should be at most 50 characters and ideally should be one of the values listed in the following table. If the error condition doesn't match one of these values, clients can use a succinct description of the error condition by using CamelCasing without white space. The ability to send a *telemetry* message requires a connection to the service, so only transient or temporary error conditions can be reported in the *telemetry* message. Error conditions that *permanently* block a client from establishing a connection to the service prevent the client from sending any message to the service, including *telemetry* messages.

ERROR	USAGE

ERROR	USAGE
DNSfailure	The client was unable to connect to the service because of a DNS failure in the network stack.
NoNetwork	The client attempted a connection, but the network stack reported that no physical network was available.
NoAuthorization	The client connection failed while attempting to acquire an authorization token for the connection.
NoResources	The client ran out of some local resource (for example, memory) while trying to make a connection.
Forbidden	The client was unable to connect to the service because the service returned an HTTP `403 Forbidden` status code on the WebSocket upgrade request.
Unauthorized	The client was unable to connect to the service because the service returned an HTTP `401 Unauthorized` status code on the WebSocket upgrade request.
BadRequest	The client was unable to connect to the service because the service returned an HTTP `400 Bad Request` status code on the WebSocket upgrade request.
ServerUnavailable	The client was unable to connect to the service because the service returned an HTTP `503 Server Unavailable` status code on the WebSocket upgrade request.
ServerError	The client was unable to connect to the service because the service returned an `HTTP 500` Internal Error status code on the WebSocket upgrade request.
Timeout	The client's connection request timed out without a response from the service. The *End* field contains the time when the client timed out and stopped waiting for the connection.
ClientError	The client terminated the connection because of some internal client error.

Metric `Microphone`

The `Microphone` metric is required for all speech turns. This metric measures the time on the client during which audio input is being actively used for a speech request.

Use the following examples as guidelines for recording *Start* time values for the `Microphone` metric in your client application:

- A client application requires that a user must press a physical button to start the microphone. After the button press, the client application reads the input from the microphone and sends it to Speech Service. The *Start* value for the `Microphone` metric records the time after the button push when the microphone is initialized and ready to provide input. The *End* value for the `Microphone` metric records the time when the client application stopped streaming audio to the service after it received the `speech.endDetected` message from the service.
- A client application uses a keyword spotter that is "always" listening. Only after the keyword spotter detects

a spoken trigger phrase does the client application collect the input from the microphone and send it to Speech Service. The *Start* value for the `Microphone` metric records the time when the keyword spotter notified the client to start using input from the microphone. The *End* value for the `Microphone` metric records the time when the client application stopped streaming audio to the service after it received the `speech.endDetected` message from the service.

- A client application has access to a constant audio stream and performs silence/speech detection on that audio stream in a *speech detection module*. The *Start* value for the `Microphone` metric records the time when the *speech detection module* notified the client to start using input from the audio stream. The *End* value for the `Microphone` metric records the time when the client application stopped streaming audio to the service after it received the `speech.endDetected` message from the service.
- A client application is processing the second turn of a multi-turn request and is informed by a service response message to turn on the microphone to gather input for the second turn. The *Start* value for the `Microphone` metric records the time when the client application enables the microphone and starts using input from that audio source. The *End* value for the `Microphone` metric records the time when the client application stopped streaming audio to the service after it received the `speech.endDetected` message from the service.

The *End* time value for the `Microphone` metric records the time when the client application stopped streaming audio input. In most situations, this event occurs shortly after the client received the `speech.endDetected` message from the service. Client applications might verify that they're properly conforming to the protocol by ensuring that the *End* time value for the `Microphone` metric occurs later than the receipt time value for the `speech.endDetected` message. And, because there is usually a delay between the end of one turn and the start of another turn, clients might verify protocol conformance by ensuring that the *Start* time of the `Microphone` metric for any subsequent turn correctly records the time when the client *started* using the microphone to stream audio input to the service.

FIELD	DESCRIPTION	USAGE
Name	Microphone	Required
Start	The time when the client started using audio input from the microphone or other audio stream or received a trigger from the keyword spotter	Required
End	The time when the client stopped using the microphone or audio stream	Required
Error	A description of the error that occurred, if any. If the microphone operations were successful, clients should omit this field. The maximum length of this field is 50 characters.	Required for error cases, omitted otherwise

Metric `ListeningTrigger`

The `ListeningTrigger` metric measures the time when the user executes the action that initiates speech input. The `ListeningTrigger` metric is optional, but clients that can provide this metric are encouraged to do so.

Use the following examples as guidelines for recording *Start* and *End* time values for the `ListeningTrigger` metric in your client application.

- A client application requires that a user must press a physical button to start the microphone. The *Start* value for this metric records the time of the button push. The *End* value records the time when the button push finishes.

- A client application uses a keyword spotter that is "always" listening. After the keyword spotter detects a spoken trigger phrase, the client application reads the input from the microphone and sends it to Speech Service. The *Start* value for this metric records the time when the keyword spotter received audio that was then detected as the trigger phrase. The *End* value records the time when the last word of the trigger phrase was spoken by the user.
- A client application has access to a constant audio stream and performs silence/speech detection on that audio stream in a *speech detection module*. The *Start* value for this metric records the time that the *speech detection module* received audio that was then detected as speech. The *End* value records the time when the *speech detection module* detected speech.
- A client application is processing the second turn of a multi-turn request and is informed by a service response message to turn on the microphone to gather input for the second turn. The client application should *not* include a `ListeningTrigger` metric for this turn.

FIELD	DESCRIPTION	USAGE
Name	ListeningTrigger	Optional
Start	The time when the client listening trigger started	Required
End	The time when the client listening trigger finished	Required
Error	A description of the error that occurred, if any. If the trigger operation was successful, clients should omit this field. The maximum length of this field is 50 characters.	Required for error cases, omitted otherwise

Sample message

The following sample shows a telemetry message with both ReceivedMessages and Metrics parts:

```
Path: telemetry
Content-Type: application/json; charset=utf-8
X-RequestId: 123e4567e89b12d3a456426655440000
X-Timestamp: 2016-08-16T15:03:54.183Z

{
  "ReceivedMessages": [
    { "speech.hypothesis": [ "2016-08-16T15:03:48.171Z", "2016-08-16T15:03:48.331Z", "2016-08-
16T15:03:48.881Z" ] },
    { "speech.endDetected": "2016-08-16T15:03:49.721Z" },
    { "speech.phrase": "2016-08-16T15:03:50.001Z" },
    { "turn.end": "2016-08-16T15:03:51.021Z" }
  ],
  "Metrics": [
    {
      "Name": "Connection",
      "Id": "A140CAF92F71469FA41C72C7B5849253",
      "Start": "2016-08-16T15:03:47.921Z",
      "End": "2016-08-16T15:03:48.000Z",
    },
    {
      "Name": "ListeningTrigger",
      "Start": "2016-08-16T15:03:48.776Z",
      "End": "2016-08-16T15:03:48.777Z",
    },
    {
      "Name": "Microphone",
      "Start": "2016-08-16T15:03:47.921Z",
      "End": "2016-08-16T15:03:51.921Z",
    },
  ],
}
```

Error handling

This section describes the kinds of error messages and conditions that your application needs to handle.

HTTP status codes

During the WebSocket upgrade request, Speech Service might return any of the standard HTTP status codes, such as `400 Bad Request`, etc. Your application must correctly handle these error conditions.

Authorization errors

If incorrect authorization is provided during the WebSocket upgrade, Speech Service returns an HTTP `403 Forbidden` status code. Among the conditions that can trigger this error code are:

- Missing *Authorization* header
- Invalid authorization token
- Expired authorization token

The `403 Forbidden` error message doesn't indicate a problem with Speech Service. This error message indicates a problem with the client application.

Protocol violation errors

If Speech Service detects any protocol violations from a client, the service terminates the WebSocket connection after returning a *status code* and *reason* for the termination. Client applications can use this information to troubleshoot and fix the violations.

Incorrect message format

If a client sends a text or binary message to the service that is not encoded in the correct format given in this

specification, the service closes the connection with a *1007 Invalid Payload Data* status code.

The service returns this status code for a variety of reasons, as shown in the following examples:

- "Incorrect message format. Binary message has invalid header size prefix." The client sent a binary message that has an invalid header size prefix.
- "Incorrect message format. Binary message has invalid header size." The client sent a binary message that specified an invalid header size.
- "Incorrect message format. Binary message headers decoding into UTF-8 failed." The client sent a binary message that contains headers that were not correctly encoded in UTF-8.
- "Incorrect message format. Text message contains no data." The client sent a text message that contains no body data.
- "Incorrect message format. Text message decoding into UTF-8 failed." The client sent a text message that was not correctly encoded in UTF-8.
- "Incorrect message format. Text message contains no header separator." The client sent a text message that did not contain a header separator or used the wrong header separator.

Missing or empty headers

If a client sends a message that doesn't have the required headers *X-RequestId* or *Path*, the service closes the connection with a *1002 Protocol Error* status code. The message is "Missing/Empty header. {Header name}."

RequestId values

If a client sends a message that specifies an *X-RequestId* header with an incorrect format, the service closes the connection and returns a *1002 Protocol Error* status. The message is "Invalid request. X-RequestId header value was not specified in no-dash UUID format."

Audio encoding errors

If a client sends an audio chunk that initiates a turn and the audio format or encoding doesn't conform to the required specification, the service closes the connection and returns a *1007 Invalid Payload Data* status code. The message indicates the format encoding error source.

RequestId reuse

After a turn is finished, if a client sends a message that reuses the request identifier from that turn, the service closes the connection and returns a *1002 Protocol Error* status code. The message is "Invalid request. Reuse of request identifiers is not allowed."

Connection failure telemetry

To ensure the best possible user experience, clients must inform Speech Service of the time stamps for important checkpoints within a connection by using the *telemetry* message. It's equally important that clients inform the service of connections that were attempted but failed.

For each connection attempt that failed, clients create a *telemetry* message with a unique *X-RequestId* header value. Because the client was unable to establish a connection, the *ReceivedMessages* field in the JSON body can be omitted. Only the `Connection` entry in the *Metrics* field is included. This entry includes the start and end time stamps as well as the error condition that was encountered.

Connection retries in telemetry

Clients should distinguish *retries* from *multiple connection attempts* by the event that triggers the connection attempt. Connection attempts that are carried out programmatically without any user input are retries. Multiple connection attempts that are carried out in response to user input are multiple connection attempts. Clients give each user-triggered connection attempt a unique *X-RequestId* and *telemetry* message. Clients reuse the *X-RequestId* for programmatic retries. If multiple retries were made for a single connection attempt, each retry

attempt is included as a `Connection` entry in the *telemetry* message.

For example, suppose a user speaks the keyword trigger to start a connection and the first connection attempt fails because of DNS errors. However, a second attempt that is made programmatically by the client succeeds. Because the client retried the connection without requiring additional input from the user, the client uses a single *telemetry* message with multiple `Connection` entries to describe the connection.

As another example, suppose a user speaks the keyword trigger to start a connection and this connection attempt fails after three retries. The client then gives up, stops attempting to connect to the service, and informs the user that something went wrong. The user then speaks the keyword trigger again. This time, suppose the client connects to the service. After connecting, the client immediately sends a *telemetry* message to the service that contains three `Connection` entries that describe the connection failures. After receiving the `turn.end` message, the client sends another *telemetry* message that describes the successful connection.

Error message reference

HTTP status codes

HTTP STATUS CODE	DESCRIPTION	TROUBLESHOOTING
400 Bad Request	The client sent a WebSocket connection request that was incorrect.	Check that you supplied all the required parameters and HTTP headers and that the values are correct.
401 Unauthorized	The client did not include the required authorization information.	Check that you're sending the *Authorization* header in the WebSocket connection.
403 Forbidden	The client sent authorization information, but it was invalid.	Check that you're not sending an expired or invalid value in the *Authorization* header.
404 Not Found	The client attempted to access a URL path that is not supported.	Check that you're using the correct URL for the WebSocket connection.
500 Server Error	The service encountered an internal error and could not satisfy the request.	In most cases, this error is transient. Retry the request.
503 Service Unavailable	The service was unavailable to handle the request.	In most cases, this error is transient. Retry the request.

WebSocket error codes

WEBSOCKETSSTATUS CODE	DESCRIPTION	TROUBLESHOOTING
1000 Normal Closure	The service closed the WebSocket connection without an error.	If the WebSocket closure was unexpected, reread the documentation to ensure that you understand how and when the service can terminate the WebSocket connection.
1002 Protocol Error	The client failed to adhere to protocol requirements.	Ensure that you understand the protocol documentation and are clear about the requirements. Read the previous documentation about error reasons to see if you're violating protocol requirements.

WEBSOCKETSSTATUS CODE	DESCRIPTION	TROUBLESHOOTING
1007 Invalid Payload Data	The client sent an invalid payload in a protocol message.	Check the last message that you sent to the service for errors. Read the previous documentation about payload errors.
1011 Server Error	The service encountered an internal error and could not satisfy the request.	In most cases, this error is transient. Retry the request.

Related topics

See a JavaScript SDK that is an implementation of the WebSocket-based Speech Service protocol.

Bing text to speech API

4/30/2018 • 9 min to read • Edit Online

Introduction

With the Bing text to speech API, your application can send HTTP requests to a cloud server, where text is instantly synthesized into human-sounding speech and returned as an audio file. This API can be used in many different contexts to provide real-time text-to-speech conversion in a variety of different voices and languages.

Voice synthesis request

Authorization token

Every voice synthesis request requires a JSON Web Token (JWT) access token. The JWT access token is passed through in the speech request header. The token has an expiry time of 10 minutes. For information about subscribing and obtaining API keys that are used to retrieve valid JWT access tokens, see Cognitive Services Subscription.

The API key is passed to the token service. For example:

```
POST https://api.cognitive.microsoft.com/sts/v1.0/issueToken
Content-Length: 0
```

The required header information for token access is as follows.

NAME	FORMAT	DESCRIPTION
Ocp-Apim-Subscription-Key	ASCII	Your subscription key

The token service returns the JWT access token as `text/plain`. Then the JWT is passed as a `Base64 access_token` to the speech endpoint as an authorization header prefixed with the string `Bearer`. For example:

```
Authorization: Bearer [Base64 access_token]
```

Clients must use the following endpoint to access the text-to-speech service:

```
https://speech.platform.bing.com/synthesize
```

> **NOTE**
> Until you have acquired an access token with your subscription key as described earlier, this link generates a `403 Forbidden` response error.

HTTP headers

The following table shows the HTTP headers that are used for voice synthesis requests.

HEADER	VALUE	COMMENTS
Content-Type	application/ssml+xml	The input content type.

HEADER	VALUE	COMMENTS
X-Microsoft-OutputFormat	**1.** ssml-16khz-16bit-mono-tts **2.** raw-16khz-16bit-mono-pcm **3.** audio-16khz-16kbps-mono-siren **4.** riff-16khz-16kbps-mono-siren **5.** riff-16khz-16bit-mono-pcm **6.** audio-16khz-128kbitrate-mono-mp3 **7.** audio-16khz-64kbitrate-mono-mp3 **8.** audio-16khz-32kbitrate-mono-mp3	The output audio format.
X-Search-AppId	A GUID (hex only, no dashes)	An ID that uniquely identifies the client application. This can be the store ID for apps. If one is not available, the ID can be user generated for an application.
X-Search-ClientID	A GUID (hex only, no dashes)	An ID that uniquely identifies an application instance for each installation.
User-Agent	Application name	The application name is required and must be fewer than 255 characters.
Authorization	Authorization token	See the Authorization token section.

Input parameters

Requests to the Bing text to speech API are made using HTTP POST calls. The headers are specified in the previous section. The body contains Speech Synthesis Markup Language (SSML) input that represents the text to be synthesized. For a description of the markup used to control aspects of speech such as the language and gender of the speaker, see the SSML W3C Specification.

> **NOTE**
> The maximum size of the SSML input that is supported is 1,024 characters, including all tags.

Example: voice output request

An example of a voice output request is as follows:

```
POST /synthesize
HTTP/1.1
Host: speech.platform.bing.com

X-Microsoft-OutputFormat: riff-8khz-8bit-mono-mulaw
Content-Type: application/ssml+xml
Host: speech.platform.bing.com
Content-Length: 197
Authorization: Bearer [Base64 access_token]

<speak version='1.0' xml:lang='en-US'><voice xml:lang='en-US' xml:gender='Female' name='Microsoft Server
Speech Text to Speech Voice (en-US, ZiraRUS)'>Microsoft Bing Voice Output API</voice></speak>
```

Voice output response

The Bing text to speech API uses HTTP POST to send audio back to the client. The API response contains the audio stream and the codec, and it matches the requested output format. The audio returned for a given request must not exceed 15 seconds.

Example: successful synthesis response

The following code is an example of a JSON response to a successful voice synthesis request. The comments and formatting of the code are for purposes of this example only and are omitted from the actual response.

```
HTTP/1.1 200 OK
Content-Length: XXX
Content-Type: audio/x-wav

Response audio payload
```

Example: synthesis failure

The following example code shows a JSON response to a voice-synthesis query failure:

```
HTTP/1.1 400 XML parser error
Content-Type: text/xml
Content-Length: 0
```

Error responses

ERROR	DESCRIPTION
HTTP/400 Bad Request	A required parameter is missing, empty, or null, or the value passed to either a required or optional parameter is invalid. One reason for getting the "invalid" response is passing a string value that is longer than the allowed length. A brief description of the problematic parameter is included.
HTTP/401 Unauthorized	The request is not authorized.
HTTP/413 RequestEntityTooLarge	The SSML input is larger than what is supported.
HTTP/502 BadGateway	There is a network-related problem or a server-side issue.

An example of an error response is as follows:

```
HTTP/1.0 400 Bad Request
Content-Length: XXX
Content-Type: text/plain; charset=UTF-8

Voice name not supported
```

Changing voice output via SSML

Microsoft Text-to-Speech API supports SSML 1.0 as defined in W3C Speech Synthesis Markup Language (SSML) Version 1.0. This section shows examples of changing certain characteristics of generated voice output like speaking rate, pronunciation etc. by using SSML tags.

1. Adding break

```
<speak version='1.0' xmlns="http://www.w3.org/2001/10/synthesis" xml:lang='en-US'><voice
name='Microsoft Server Speech Text to Speech Voice (en-US, BenjaminRUS)'> Welcome to use Microsoft
Cognitive Services <break time="100ms" /> Text-to-Speech API.</voice> </speak>
```

2. Change speaking rate

```
<speak version='1.0' xmlns="http://www.w3.org/2001/10/synthesis" xml:lang='en-US'><voice
name='Microsoft Server Speech Text to Speech Voice (en-US, JessaRUS)'><prosody rate="+30.00%">Welcome to
use Microsoft Cognitive Services Text-to-Speech API.</prosody></voice> </speak>
```

3. Pronunciation

```
<speak version='1.0' xmlns="http://www.w3.org/2001/10/synthesis" xml:lang='en-US'><voice
name='Microsoft Server Speech Text to Speech Voice (en-US, JessaRUS)'> <phoneme alphabet="ipa"
ph="t&#x259;mei&#x325;&#x27E;ou&#x325;"> tomato </phoneme></voice> </speak>
```

4. Change volume

```
<speak version='1.0' xmlns="http://www.w3.org/2001/10/synthesis" xml:lang='en-US'><voice
name='Microsoft Server Speech Text to Speech Voice (en-US, JessaRUS)'><prosody volume="+20.00%">Welcome
to use Microsoft Cognitive Services Text-to-Speech API.</prosody></voice> </speak>
```

5. Change pitch

```
<speak version='1.0' xmlns="http://www.w3.org/2001/10/synthesis" xml:lang='en-US'><voice
name='Microsoft Server Speech Text to Speech Voice (en-US, JessaRUS)'>Welcome to use <prosody
pitch="high">Microsoft Cognitive Services Text-to-Speech API.</prosody></voice> </speak>
```

6. Change prosody contour

```
<speak version='1.0' xmlns="http://www.w3.org/2001/10/synthesis" xml:lang='en-US'><voice
name='Microsoft Server Speech Text to Speech Voice (en-US, JessaRUS)'><prosody contour="(80%,+20%)
(90%,+30%)" >Good morning.</prosody></voice> </speak>
```

NOTE

Note the audio data has to be 8k or 16k wav filed in the following format: **CRC code** (CRC-32): 4 bytes (DWORD) with valid range 0x00000000 ~ 0xFFFFFFFF; **Audio format flag**: 4 bytes (DWORD) with valid range 0x00000000 ~ 0xFFFFFFFF; **Sample count**: 4 bytes (DWORD) with valid range 0x00000000 ~ 0x7FFFFFFF; **Size of binary body**: 4 bytes (DWORD) with valid range 0x00000000 ~ 0x7FFFFFFF; **Binary body**: n bytes.

Sample application

For implementation details, see the Visual C#.NET text-to-speech sample application.

Supported locales and voice fonts

The following table identifies some of the supported locales and related voice fonts.

LOCALE	GENDER	SERVICE NAME MAPPING
ar-EG*	Female	"Microsoft Server Speech Text to Speech Voice (ar-EG, Hoda)"
ar-SA	Male	"Microsoft Server Speech Text to Speech Voice (ar-SA, Naayf)"

LOCALE	GENDER	SERVICE NAME MAPPING
bg-BG	Male	"Microsoft Server Speech Text to Speech Voice (bg-BG, Ivan)"
ca-ES	Female	"Microsoft Server Speech Text to Speech Voice (ca-ES, HerenaRUS)"
cs-CZ	Male	"Microsoft Server Speech Text to Speech Voice (cs-CZ, Jakub)"
da-DK	Female	"Microsoft Server Speech Text to Speech Voice (da-DK, HelleRUS)"
de-AT	Male	"Microsoft Server Speech Text to Speech Voice (de-AT, Michael)"
de-CH	Male	"Microsoft Server Speech Text to Speech Voice (de-CH, Karsten)"
de-DE	Female	"Microsoft Server Speech Text to Speech Voice (de-DE, Hedda) "
de-DE	Female	"Microsoft Server Speech Text to Speech Voice (de-DE, HeddaRUS)"
de-DE	Male	"Microsoft Server Speech Text to Speech Voice (de-DE, Stefan, Apollo) "
el-GR	Male	"Microsoft Server Speech Text to Speech Voice (el-GR, Stefanos)"
en-AU	Female	"Microsoft Server Speech Text to Speech Voice (en-AU, Catherine) "
en-AU	Female	"Microsoft Server Speech Text to Speech Voice (en-AU, HayleyRUS)"
en-CA	Female	"Microsoft Server Speech Text to Speech Voice (en-CA, Linda)"
en-CA	Female	"Microsoft Server Speech Text to Speech Voice (en-CA, HeatherRUS)"
en-GB	Female	"Microsoft Server Speech Text to Speech Voice (en-GB, Susan, Apollo)"
en-GB	Female	"Microsoft Server Speech Text to Speech Voice (en-GB, HazelRUS)"
en-GB	Male	"Microsoft Server Speech Text to Speech Voice (en-GB, George, Apollo)"
en-IE	Male	"Microsoft Server Speech Text to Speech Voice (en-IE, Sean)"

LOCALE	GENDER	SERVICE NAME MAPPING
en-IN	Female	"Microsoft Server Speech Text to Speech Voice (en-IN, Heera, Apollo)"
en-IN	Female	"Microsoft Server Speech Text to Speech Voice (en-IN, PriyaRUS)"
en-IN	Male	"Microsoft Server Speech Text to Speech Voice (en-IN, Ravi, Apollo)"
en-US	Female	"Microsoft Server Speech Text to Speech Voice (en-US, ZiraRUS)"
en-US	Female	"Microsoft Server Speech Text to Speech Voice (en-US, JessaRUS)"
en-US	Male	"Microsoft Server Speech Text to Speech Voice (en-US, BenjaminRUS)"
es-ES	Female	"Microsoft Server Speech Text to Speech Voice (es-ES, Laura, Apollo)"
es-ES	Female	"Microsoft Server Speech Text to Speech Voice (es-ES, HelenaRUS)"
es-ES	Male	"Microsoft Server Speech Text to Speech Voice (es-ES, Pablo, Apollo)"
es-MX	Female	"Microsoft Server Speech Text to Speech Voice (es-MX, HildaRUS)"
es-MX	Male	"Microsoft Server Speech Text to Speech Voice (es-MX, Raul, Apollo)"
fi-FI	Female	"Microsoft Server Speech Text to Speech Voice (fi-FI, HeidiRUS)"
fr-CA	Female	"Microsoft Server Speech Text to Speech Voice (fr-CA, Caroline)"
fr-CA	Female	"Microsoft Server Speech Text to Speech Voice (fr-CA, HarmonieRUS)"
fr-CH	Male	"Microsoft Server Speech Text to Speech Voice (fr-CH, Guillaume)"
fr-FR	Female	"Microsoft Server Speech Text to Speech Voice (fr-FR, Julie, Apollo)"
fr-FR	Female	"Microsoft Server Speech Text to Speech Voice (fr-FR, HortenseRUS)"
fr-FR	Male	"Microsoft Server Speech Text to Speech Voice (fr-FR, Paul, Apollo)"

LOCALE	GENDER	SERVICE NAME MAPPING
he-IL	Male	"Microsoft Server Speech Text to Speech Voice (he-IL, Asaf)"
hi-IN	Female	"Microsoft Server Speech Text to Speech Voice (hi-IN, Kalpana, Apollo)"
hi-IN	Female	"Microsoft Server Speech Text to Speech Voice (hi-IN, Kalpana)"
hi-IN	Male	"Microsoft Server Speech Text to Speech Voice (hi-IN, Hemant)"
hr-HR	Male	"Microsoft Server Speech Text to Speech Voice (hr-HR, Matej)"
hu-HU	Male	"Microsoft Server Speech Text to Speech Voice (hu-HU, Szabolcs)"
id-ID	Male	"Microsoft Server Speech Text to Speech Voice (id-ID, Andika)"
it-IT	Male	"Microsoft Server Speech Text to Speech Voice (it-IT, Cosimo, Apollo)"
ja-JP	Female	"Microsoft Server Speech Text to Speech Voice (ja-JP, Ayumi, Apollo)"
ja-JP	Male	"Microsoft Server Speech Text to Speech Voice (ja-JP, Ichiro, Apollo)"
ja-JP	Female	"Microsoft Server Speech Text to Speech Voice (ja-JP, HarukaRUS)"
ja-JP	Female	"Microsoft Server Speech Text to Speech Voice (ja-JP, LuciaRUS)"
ja-JP	Male	"Microsoft Server Speech Text to Speech Voice (ja-JP, EkaterinaRUS)"
ko-KR	Female	"Microsoft Server Speech Text to Speech Voice (ko-KR, HeamiRUS)"
ms-MY	Male	"Microsoft Server Speech Text to Speech Voice (ms-MY, Rizwan)"
nb-NO	Female	"Microsoft Server Speech Text to Speech Voice (nb-NO, HuldaRUS)"
nl-NL	Female	"Microsoft Server Speech Text to Speech Voice (nl-NL, HannaRUS)"
pl-PL	Female	"Microsoft Server Speech Text to Speech Voice (pl-PL, PaulinaRUS)"

LOCALE	GENDER	SERVICE NAME MAPPING
pt-BR	Female	"Microsoft Server Speech Text to Speech Voice (pt-BR, HeloisaRUS)"
pt-BR	Male	"Microsoft Server Speech Text to Speech Voice (pt-BR, Daniel, Apollo)"
pt-PT	Female	"Microsoft Server Speech Text to Speech Voice (pt-PT, HeliaRUS)"
ro-RO	Male	"Microsoft Server Speech Text to Speech Voice (ro-RO, Andrei)"
ru-RU	Female	"Microsoft Server Speech Text to Speech Voice (ru-RU, Irina, Apollo)"
ru-RU	Male	"Microsoft Server Speech Text to Speech Voice (ru-RU, Pavel, Apollo)"
sk-SK	Male	"Microsoft Server Speech Text to Speech Voice (sk-SK, Filip)"
sl-SI	Male	"Microsoft Server Speech Text to Speech Voice (sl-SI, Lado)"
sv-SE	Female	"Microsoft Server Speech Text to Speech Voice (sv-SE, HedvigRUS)"
ta-IN	Male	"Microsoft Server Speech Text to Speech Voice (ta-IN, Valluvar)"
th-TH	Male	"Microsoft Server Speech Text to Speech Voice (th-TH, Pattara)"
tr-TR	Female	"Microsoft Server Speech Text to Speech Voice (tr-TR, SedaRUS)"
vi-VN	Male	"Microsoft Server Speech Text to Speech Voice (vi-VN, An)"
zh-CN	Female	"Microsoft Server Speech Text to Speech Voice (zh-CN, HuihuiRUS)"
zh-CN	Female	"Microsoft Server Speech Text to Speech Voice (zh-CN, Yaoyao, Apollo)"
zh-CN	Male	"Microsoft Server Speech Text to Speech Voice (zh-CN, Kangkang, Apollo)"
zh-HK	Female	"Microsoft Server Speech Text to Speech Voice (zh-HK, Tracy, Apollo)"
zh-HK	Female	"Microsoft Server Speech Text to Speech Voice (zh-HK, TracyRUS)"

LOCALE	GENDER	SERVICE NAME MAPPING
zh-HK	Male	"Microsoft Server Speech Text to Speech Voice (zh-HK, Danny, Apollo)"
zh-TW	Female	"Microsoft Server Speech Text to Speech Voice (zh-TW, Yating, Apollo)"
zh-TW	Female	"Microsoft Server Speech Text to Speech Voice (zh-TW, HanHanRUS)"
zh-TW	Male	"Microsoft Server Speech Text to Speech Voice (zh-TW, Zhiwei, Apollo)"

*ar-EG supports Modern Standard Arabic (MSA).

> **NOTE**
>
> Note that the previous service names **Microsoft Server Speech Text to Speech Voice (cs-CZ, Vit)** and **Microsoft Server Speech Text to Speech Voice (en-IE, Shaun)** will be deprecated after 3/31/2018, in order to optimize the Bing Speech API's capabilities. Please update your code with the updated names.

Troubleshooting and support

Post all questions and issues to the Bing Speech Service MSDN forum. Include complete details, such as:

- An example of the full request string.
- If applicable, the full output of a failed request, which includes log IDs.
- The percentage of requests that are failing.

Supported languages

4/19/2018 • 1 min to read • Edit Online

Interactive and dictation mode

The Microsoft speech recognition API supports the following languages in `interactive` and `dictation` modes.

CODE	LANGUAGE	CODE	LANGUAGE
ar-EG	Arabic (Egypt), modern standard	hi-IN	Hindi (India)
ca-ES	Catalan (Spain)	it-IT	Italian (Italy)
da-DK	Danish (Denmark)	ja-JP	Japanese (Japan)
de-DE	German (Germany)	ko-KR	Korean (Korea)
en-AU	English (Australia)	nb-NO	Norwegian (Bokmål) (Norway)
en-CA	English (Canada)	nl-NL	Dutch (Netherlands)
en-GB	English (United Kingdom)	pl-PL	Polish (Poland)
en-IN	English (India)	pt-BR	Portuguese (Brazil)
en-NZ	English (New Zealand)	pt-PT	Portuguese (Portugal)
en-US	English (United States)	ru-RU	Russian (Russia)
es-ES	Spanish (Spain)	sv-SE	Swedish (Sweden)
es-MX	Spanish (Mexico)	zh-CN	Chinese (Mandarin, simplified)
fi-FI	Finnish (Finland)	zh-HK	Chinese (Hong Kong SAR)
fr-CA	French (Canada)	zh-TW	Chinese (Mandarin, Taiwanese)
fr-FR	French (France)		

Conversation mode

The Microsoft speech recognition API supports the following languages in `conversation` modes.

CODE	LANGUAGE	CODE	LANGUAGE
ar-EG	Arabic (Egypt), modern standard	It-IT	Italian (Italy)
de-DE	German (Germany)	ja-JP	Japanese (Japan)
en-US	English (United States)	pt-BR	Portuguese (Brazil)
es-ES	Spanish (Spain)	ru-RU	Russian (Russia)
fr-FR	French (France)	zh-CN	Chinese (Mandarin, simplified)

Troubleshooting

4/19/2018 • 3 min to read • Edit Online

Error `HTTP 403 Forbidden`

When using speech recognition API, it returns an `HTTP 403 Forbidden` error.

Cause

This error is often caused by authentication issues. Connection requests without valid `Ocp-Apim-Subscription-Key` or `Authorization` header are rejected by the service with an `HTTP 403 Forbidden` response.

If you are using subscription key for authentication, the reason could be

- the subscription key is missing or invalid
- the usage quota of the subscription key is exceeded
- the `Ocp-Apim-Subscription-Key` field is not set in the request header, when REST API is called

If you are using authorization token for authentication, the following reasons could cause the error.

- the `Authorization` header is missing in the request when using REST
- the authorization token specified in the Authorization header is invalid
- the authorization token is expired. The access token has an expiry of 10 minutes

For more information about authentication, see the Authentication page.

Troubleshooting steps

Verify that your subscription key is valid

You can run the following command for verification. Note to replace *YOUR_SUBSCRIPTION_KEY* with your own subscription key. If your subscription key is valid, you receive in the response the authorization token as a JSON Web Token (JWT). Otherwise you get an error in response.

> **NOTE**
> Replace `YOUR_SUBSCRIPTION_KEY` with your own subscription key.

- Powershell
- curl

```
$FetchTokenHeader = @{
  'Content-type'='application/x-www-form-urlencoded';
  'Content-Length'= '0';
  'Ocp-Apim-Subscription-Key' = 'YOUR_SUBSCRIPTION_KEY'
}

$OAuthToken = Invoke-RestMethod -Method POST -Uri https://api.cognitive.microsoft.com/sts/v1.0/issueToken -
Headers $FetchTokenHeader

# show the token received
$OAuthToken
```

Make sure that you use the same subscription key in your application or in the REST request as that is used above.

Verify the authorization token

This step is only needed, if you use authorization token for authentication. Run the following command to verify that the authorization token is still valid. The command makes a POST request to the service, and expects a response message from the service. If you still receive HTTP `403 Forbidden` error, double-check the access token is set correctly and not expired.

> **IMPORTANT**
> The token has an expiry of 10 minutes.

> **NOTE**
> Replace `YOUR_AUDIO_FILE` with the path to your prerecorded audio file, and `YOUR_ACCESS_TOKEN` with the authorization token returned in the previous step.

- Powershell
- curl

```
$SpeechServiceURI =
'https://speech.platform.bing.com/speech/recognition/interactive/cognitiveservices/v1?language=en-
us&format=detailed'

# $OAuthToken is the authrization token returned by the token service.
$RecoRequestHeader = @{
  'Authorization' = 'Bearer '+ $OAuthToken;
  'Transfer-Encoding' = 'chunked'
  'Content-type' = 'audio/wav; codec=audio/pcm; samplerate=16000'
}

# Read audio into byte array
$audioBytes = [System.IO.File]::ReadAllBytes("YOUR_AUDIO_FILE")

$RecoResponse = Invoke-RestMethod -Method POST -Uri $SpeechServiceURI -Headers $RecoRequestHeader -Body
$audioBytes

# Show the result
$RecoResponse
```

Error `HTTP 400 Bad Request`

This reason is usually that the request body contains invalid audio data. Currently we only support WAV file.

Error `HTTP 408 Request Timeout`

The error is most likely because that no audio data is sent to the service and the service returns this error after timeout. For REST API, the audio data should be put in the request body.

The `RecognitionStatus` in the response is `InitialSilenceTimeout`

Audio data is usually the reason causing the issue. For example,

- the audio has a long silence time at the beginning. The service will stop the recognition after some number of seconds and returns `InitialSilenceTimeout`.
- the audio uses unsupported codec format, which makes the audio data be treated as silence.

Support

4/19/2018 • 1 min to read • Edit Online

If you have questions, feedback, or suggestions about Speech Service, you can reach out to us via GitHub.

- REST API-related questions
- JavaScript library
- C# desktop library-related questions
- C# service library-related questions
- Java library on Android-related questions
- Objective-C library on iOS
- General questions: Cognitive Services UserVoice forum