Azure AI Engineer

- Labs for Microsoft Learn

Responsible AI:

- Fairness
- Reliability and safety
- Privacy and security
- Inclusiveness
- Transparency
- Accountability

1. Azure cognitive services

- Single-service resource: individual resource. Different endpoint for each service, separate billing. Offers free tier
- Multi-service resource: language, computer vision, speech, etc

To consume the endpoint, we need:

- Endpoint URI
- Subscription key (sometimes synonym with API key)
- Resource location

```
# pip install azure-ai-textanalytics==5.0.0
credential = AzureKeyCredential(cog_key)
client = TextAnalyticsClient(endpoint=cog_endpoint, credential=credential)
```

Keys can also be stored in a keyvault. To do this, a Service Principal has to be created and given access to the keyvault.

2. Monitoring

- Azure pricing calculator (needs specific cognitive service API, region and pricing tier)
- View costs in Costs analysis
- Alerts: specify scope (resource), condition: signal type (e.g. Activity Log) or metric (e.g. #errors exceeding 10 in an hour)

To create diagnostics:

- Create resource for diagnostic log storage: azure log analytics, or azure storage
- configure diagnostic settings
  - Name
  - Categories of log event data
  - Details of the destinations in which you want to store the log data

it can take 1h or more for diagnostic data to start flowing to the destination. There, it can be viewed by running queries

3. Containers

Cognitive services can be deployed on-premises or in Azure. Deploying Cognitive Services in a container on-premises will also decrease the latency between the service and your local data, which can improve performance.

Containers can be deployed to the following options:

- A docker server
- An Azure Container Instance (ACI)
- An AKS cluster

To deploy and use a CS (cognitive services) containers:

1. The container image is downloaded and deployed to a container host (Docker server, ACI or AKS cluster)
2. Client apps submit data to the endpoint and retrieve results
3. Periodically, usage metrics for the containerized service are sent to the CS resource in Azure to calculate billing

Even when using a container, the CS resource must be provisioned in Azure for billing purposes. Client applications send their requests to the containerized service, meaning that potentially sensitive data is not sent to the Cognitive Services endpoint in Azure; but the container must be able to connect to the Cognitive Services resource in Azure periodically to send usage metrics for billing.

To deploy to container, must specify 3 settings: ApiKey, Billing, Eula (value of accept to state you accept the license of the container). No subscription key necessary

4. Language

4.1. Language detection

Input request. Max 5120 characters in text, max 1000 documents. Optionally, you can provide a countryHint to improve prediction performance.

In the code:

```
sentimentAnalysis = cog_client.detect_language(documents=[text])[0]
```

```
{
  "documents": [
    {
      "countryHint": "US",
      "id": "1",
      "text": "Hello world"
    }
  ]
}
```

Response json:

```
{
  "documents": [
   {
     "id": "1",
     "detectedLanguage": {
       "name": "English",
       "iso6391Name": "en",
       "confidenceScore": 1
```

```
      },
      "warnings": []
    }
  ],
  "errors": [],
  "modelVersion": "2020-04-01"
}
```

If there are encoding problems, or text isn't string, the result will be (Unknown) and score: NaN

4.2. Key-phrase extraction

In the code:

```
phrases = cog_client.extract_key_phrases(documents=[text])[0].key_phrases
for phrase in phrases:
    print('\t{}'.format(phrase))
```

Input JSON same as with language detection. Response:

```
{
  "documents": [
   {
     "id": "1",
     "keyPhrases": [
       "change",
       "world"
     ],
     "warnings": []
   }
  ],
  "errors": [],
  "modelVersion": "2020-04-01"
}
```

4.3. Sentiment analysis

In the code:

```
sentimentAnalysis = cog_client.analyze_sentiment(documents=[text])[0]
```

Input:

```
{
  "documents": [
    {
      "language": "en",
      "id": "1",
      "text": "Smile! Life is good!"
    }
  ]
}
```

Output:

```

  {
    "documents": [
     {
       "id": "1",
       "sentiment": "positive",
       "confidenceScores": {
         "positive": 0.99,
         "neutral": 0.01,
         "negative": 0.00
       },
       "sentences": [
         {
           "text": "Smile!",
           "sentiment": "positive",
           "confidenceScores": {
               "positive": 0.97,
               "neutral": 0.02,
               "negative": 0.01
             },
           "offset": 0,
           "length": 6
         },
         {
            "text": "Life is good!",
            "sentiment": "positive",
            "confidenceScores": {
               "positive": 0.98,
               "neutral": 0.02,
               "negative": 0.00
             },
           "offset": 7,
           "length": 13
         }
       ],
       "warnings": []
     }
    ],
    "errors": [],
    "modelVersion": "2020-04-01"
  }

```

- If sentences are neutral, overall sentiment is neutral
- If sentences are positive + neutral -> positive
- If sentences are negative + neutral -> negative
- If sentences are positive + negative -> mixed

4.4. Named entity recognition

In the code:

```
entities = cog_client.recognize_entities(documents=[text])[0].entities
for entity in entities:
    print(f"{entity.text}: {entity.category}")
```

Same input as sentiment analysis. Output:

```
{
  "documents":[
      {
          "id":"1",
          "entities":[
          {
            "text":"Joe",
            "category":"Person",
            "offset":0,
            "length":3,
            "confidenceScore":0.62
          },
          {
            "text":"London",
            "category":"Location",
            "subcategory":"GPE",
            "offset":12,
            "length":6,
            "confidenceScore":0.88
          },
          {
            "text":"Saturday",
            "category":"DateTime",
            "subcategory":"Date",
            "offset":22,
            "length":8,
            "confidenceScore":0.8
          }
        ],
        "warnings":[]
      }
  ],
  "errors":[],
  "modelVersion":"2021-01-15"
}
```

The ID attribute can be used when there are multiple entries in the document. It helps to identify the specific text portion where the entity was located. As an example, if the submitted request contains more than one entry for text, the ID is used to locate the entities for that text, within the results.

4.5. Entity linking

In the code:

```
entities = cog_client.recognize_linked_entities(documents=[text])[0].entities
for linked_entity in entities:
    print(f"{linked_entity.name}: {linked_entity.url}")
```

Identifying specific entities by providing reference links to Wikipedia articles. Output:

```
{
  "documents":
    [
      {
        "id":"1",
        "entities":[
          {
            "name":"Venus",
            "matches":[
              {
                "text":"Venus",
                "offset":6,
                "length":5,
                "confidenceScore":0.01
              }
            ],
            "language":"en",
            "id":"Venus",
            "url":"https://en.wikipedia.org/wiki/Venus",
            "dataSource":"Wikipedia"
          }
        ],
        "warnings":[]
      }
    ],
  "errors":[],
  "modelVersion":"2020-02-01"
}
```

4.6. Translation

To access the API, we need the subscription key and the location. Supports language detection and translation (as input we need the original text and the "to" language). Also supports transliteration (changing script)

When translating to languages without tokenization, the JSON response includes an alignment projection: "0:8-0:1 ..." saying that chars 0-8 in the original language correspond to chars 0-1 in the target language.

The response also includes sentLen, sentence length of the original text and the translated text.

Profanity filtering:

- NoAction
- Deleted
- Marked: replaced by **** or any other technique.

A custom model can be trained with custom translations. Your custom model is assigned a unique category Id, which you can specify in translate calls to your Translator resource by using the category parameter, causing translation to be performed by your custom model instead of the default model.

```
# Use the Translator detect function
path = '/detect'
url = translator_endpoint + path

# Build the request
params = {
    'api-version': '3.0'
}

headers = {
'Ocp-Apim-Subscription-Key': cog_key,
'Ocp-Apim-Subscription-Region': cog_region,
'Content-type': 'application/json'
}

body = [{
    'text': text
}]

# Send the request and get response
request = requests.post(url, params=params, headers=headers, json=body)
response = request.json()

# Parse JSON array and get language
language = response[0]["language"]


# Use the Translator translate function
path = '/translate'
url = translator_endpoint + path

# Build the request
params = {
    'api-version': '3.0',
    'from': source_language,
    'to': ['en']
}
```

```
headers = {
    'Ocp-Apim-Subscription-Key': cog_key,
    'Ocp-Apim-Subscription-Region': cog_region,
    'Content-type': 'application/json'
}

body = [{
    'text': text
}]

# Send the request and get response
request = requests.post(url, params=params, headers=headers, json=body)
response = request.json()

# Parse JSON array and get translation
translation = response[0]["translations"][0]["text"]
```

4.7. Question answering

1. Create knowledge base
    1. Option 1: REST API or SDK
    2. Option 2: Language Studio UI

To access this knowledge base, we need the ID, endpoint and authorization key.

Create a language resource in the azure portal, enable question answering feature, create/select azure cognitive search to host the knowledge base index. Create project in Language Studio and choose the data source

- URLs for webpages containing FAQs
- Files containing structured text from which questions and answers can be derived
- Pre-defined chit-chat datasets that include common conversational questions and responses

After defining the knowledeg base, train its natural language model and test it, and deploy it to a REST endpoint.

The response has this format:

```
{
  "answers": [
    {
      "questions": [
        "How can I cancel a reservation?"
      ],
      "answer": "Call us on 555 123 4567 to cancel a reservation.",
      "confidenceScore": 1.0,
      "id": 6,
      "source": "https://margies-travel.com/faq",
      "metadata": {},
      "dialog": {
        "isContextOnly": false,
        "prompts": []
      }
    }
  ]
}
```

To improve performance, use active learning:

- Implicit feedback: the service identifies questions with similar scores, they are automatically clustered as alternate phrase suggestions for the possible answers that you can accept/reject in the Suggestions page in Language Studio
- Explicit feedback: when asking a question you can ask for "top":3, in which case you get the top 3 questions associated with your question. You can pick the question that matches your most and use that, by referencing the qnaId of the suggestion you received earlier.

You can submit synonyms to the REST API by uploading a json file with a specific format.

Language Studio provides the option to easily create a bot that runs in the Azure Bot Service based on your knowledge base. To create a bot from your knowledge base, use Language Studio to deploy the bot and then use the Create Bot button to create a bot in your Azure subscription. You can then edit and customize your bot in the Azure portal.

4.8. Language understanding (LUIS)

It's the NLP service that enables language interactions between users and conversational AI agents. This can be encapsulated into a bot, but LUIS is the backend of the conversation.

Dashboard: main page, manage page: import app, rename, clone, export, delete

LUIS basics

When creating an app, first create intents, then utterances. When adding utterances, add utterances unrelated to the app into the None intent. These are sentences that will trigger no intents. You should always provide at least 15 example utterances for each intent.

- Domain: email, communication
- Utterance: turn on the light
- Entity: Paris, Lamp, Light
  - Numbers (2, 3, 5) are prebuilt entities
  - For currency, use prebuilt entities
- Intent: BookFlight

Entity types:

- Machine learned entities are the most flexible kind of entity, and should be used in most cases. Define a machine learned entity with a suitable name, and then associate words or phrases with it in training utterances. The model will learn to match the appropriate elements in the utterances with the entity
- List entities are useful for entities with a specific set of possible values - for example, days of the week. You can include synonyms in a list entity definition, so you could define a DayOfWeek entity that

includes the values "Sunday", "Monday", "Tuesday", and so on; each with synonyms like "Sun", "Mon", "Tue", and so on
- Regular Expression or RegEx entities are useful when an entity can be identified by matching a particular format of string. For example, a date in the format MM/DD/YYYY, or a flight number in the format AB-1234
- Pattern.any() entities are used with patterns

To create a new LUIS app version, clone an existing version and then make changes to the cloned app. If there are 1+ contributors, you can clone the base app for each contributor so you end up with 1 base app and 1+ cloned copies.

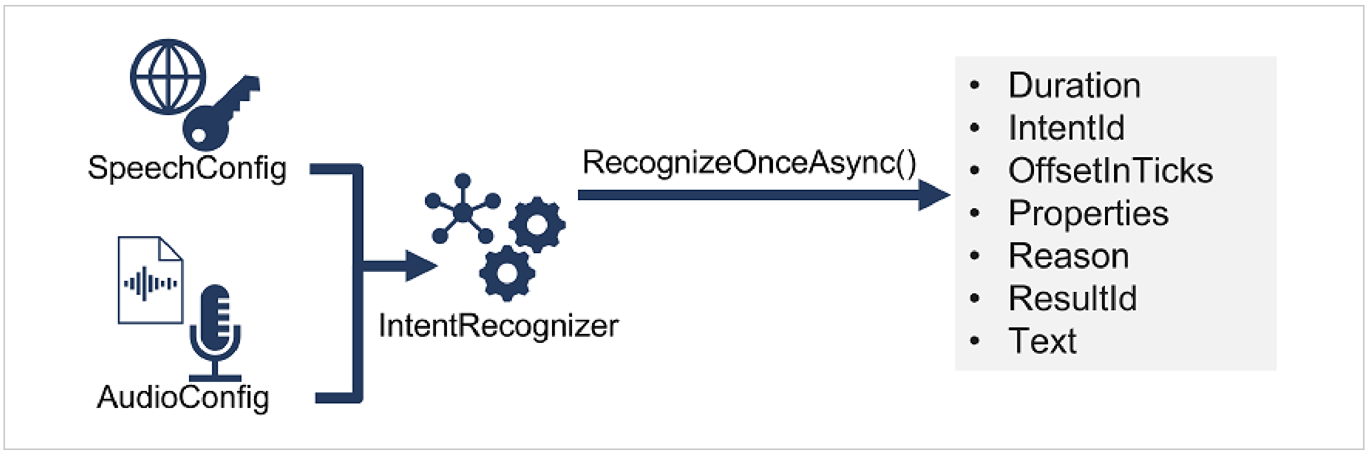
There are 3 LUIS websites based on the region: luis.ai (US), au.luis.ai (Australia), eu.luis.ai (Europe)

To deploy: staging, production. When deploying you can activate sentiment analysis, spelling correction, speech priming (allows users to interact with the LUIS app through speech).

Model can be exported to a .lu file and imported into another Language understanding app.

Intent recognition with Speech SDK

If the intent score is low or if the correct intent is not the top scoring intent, use pattern matching. Patterns can improve prediction scores on utterances that reveal patterns in the word order and choice of words used.



- SpeechConfig: info to connect to LU, not to the Speech resource. Location and key of the LU resource
- AudioConfig: input source for the speech

If the operation was successful, the Reason property has the enumerated value **RecognizedIntent**, and the IntentId property contains the top intent name. Full details of the Language Understanding prediction can be found in Properties, which includes the full JSON prediction.

Other possible values for Result include **RecognizedSpeech**, which indicates that the speech was successfully transcribed (the transcription is in the Text property), but no matching intent was identified. If the result is **NoMatch**, the audio was successfully parsed but no speech was recognized, and if the result is **Canceled**, an error occurred (in which case, you can check the Properties collection for the CancellationReason property to determine what went wrong.)

Accessing endpoint

To access endpoint, we need App ID, endpoint URL and primary key/secondary key. When calling the endpoint, we need:

- query
- show-all-intents: include all identified intents and their scores, or only the most likely intent
- verbose: bool
- log: to use active learning

LUIS in a container

1. Download the container image
2. Export the model for a container (.gz)
3. Run the container with required parameters

- Prediction endpoint for billing
- Prediction key
- EULA acceptance
- Mount points (input for exported models, output for logs)

4. Use container to predict intents
5. The conatiner sends usage metrics to the prediction resource for billing

LUIS app lifecycle

- Create LUIS app
- Edit schema
- Train the app
- Test the app
- Publish the app
- Evaluate the app

LUIS keys

- Authoring key: to programatically author LUIS apps
- Prediction key: query prediction endpoint requests beyond the 1000 requests provided by the starter resource

4.9. Bots

- Azure Bot Service: bot delivery and integration
- Bot Framework Service: REST interface
- Bot Framework SDK: bot development. Bot templates:
    - Empty bot: basic bot skeleton
    - Echo bot: simple "hello world" sample
    - Core bot: common bot functionality, e.g. integration with the language understanding service
- Bot framework emulator: local testing
- QnA Maker: helps create a knowledge base that can be used for a conversation between humans and AI agents
    - First provision a QnA maker resource, then create and populate the knowledge base

    - QnA maker can contain several knowledge bases, but the language of the first knowledge base defines the language for the rest of the bases within that resource

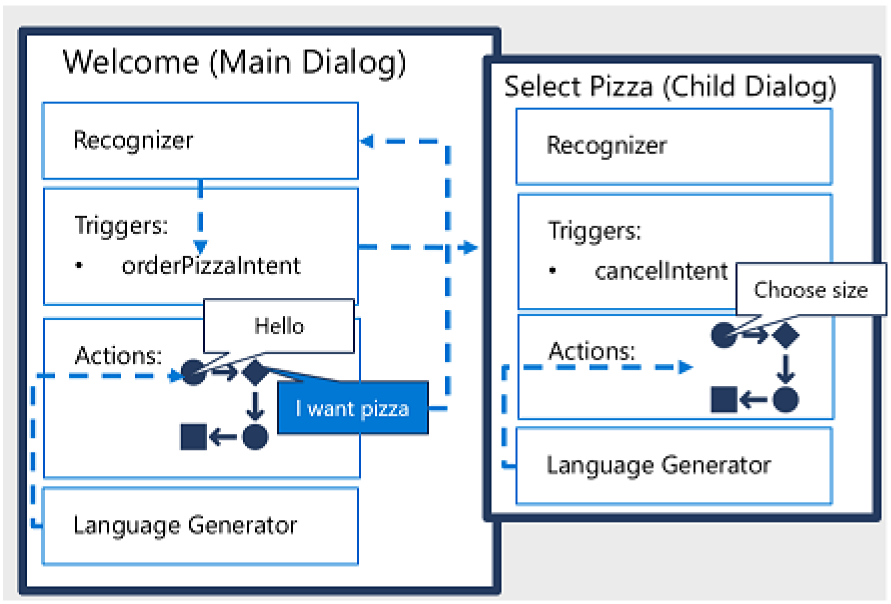
Bot framework units:

- Activity handlers: event methods that you can override to handle different kinds of activities. Includes:
    - Message received
    - Member joined the conversation
    - Member left the conversation
    - Message reaction received
    - Bot installed
- Turn context: an activity occurs within the context of a turn. Activity handler methods include a parameter for the turn context which you can use to access relevant information, for example to access the text of the message that was received
- Dialogues: more complex paterns for handling stateful, multi-turn conversations
    - Component dialog: can contain subdialogs. usually main dialog is a waterfall dialog, defining sequential steps (pizza ordering)
    - Adaptive dialog: more flexible flow, allowing for interruptions, cancellations, context switches
    - To build bots with complex dialogs, use **Bot framework composer**. This allows interruptions and context switches

Bots make use of an Adapter class that handles communication with the user's channel. The Bot Framework Service notifies your bot's adapter when an activity occurs in a channel by calling its **Process Activity** method, and the adapter creates a context for the turn and calls the bot's **Turn Handler** method to invoke the appropriate logic for the activity.

An adaptive dialog consists of:

- 1+ actions defining the flow of message activities in the dialog
- Language generator: formulates the output to the user
- Recognizer: interprets user input to determine semantic intent
- Trigger: fired by actions or based on the intent detected by the recognizer



If you need a bot and some users want to use Teams and others a web chat, create a knowledge base, create a bot for it and connect the web chat and Team channels for your bot.

User experience:

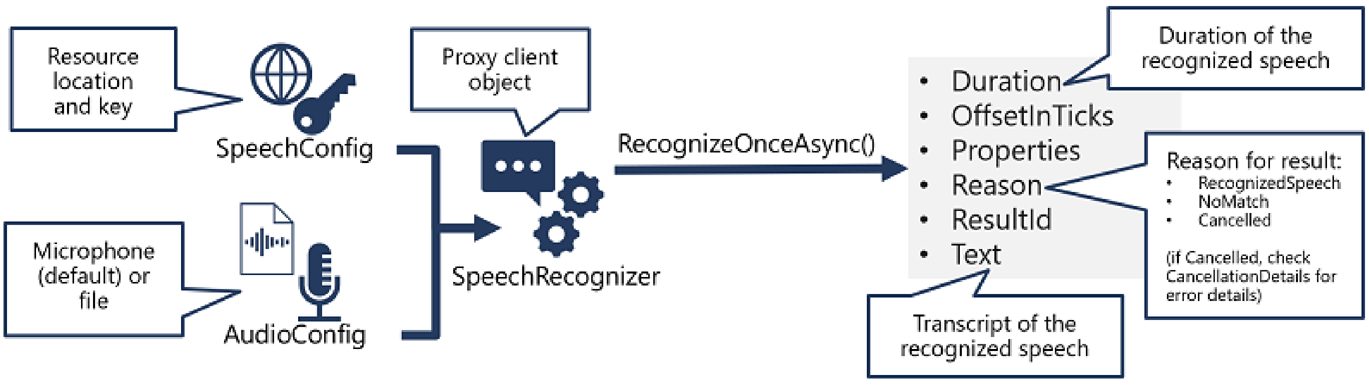
- Text
- Speech: the bot will have to access the Speech cognitive service
- Buttons
- Images
- Cards: visual, audio, selectable messages

5. Speech

Needs subscription key and location to access the API.

5.1. Speech to text

- The Speech-to-text API, which is the primary way to perform speech recognition. The endpoint for this API is `https://<LOCATION>.api.cognitive.microsoft.com/sts/v1.0`
- The Speech-to-text Short Audio API, which is optimized for short streams of audio (up to 60 seconds). The endpoint for this API is at `https://<LOCATION>.stt.speech.microsoft.com/speech/recognition/conversation/cognitiveservices/v1`


Resource location and key
SpeechConfig
Microphone (default) or file
AudioConfig
Proxy client object
SpeechRecognizer
RecognizeOnceAsync()
Duration
OffsetInTicks
Properties
Reason
ResultId
Text
Duration of the recognized speech
Reason for result:
RecognizedSpeech
NoMatch
Cancelled
(if Cancelled, check CancellationDetails for error details)
Transcript of the recognized speech

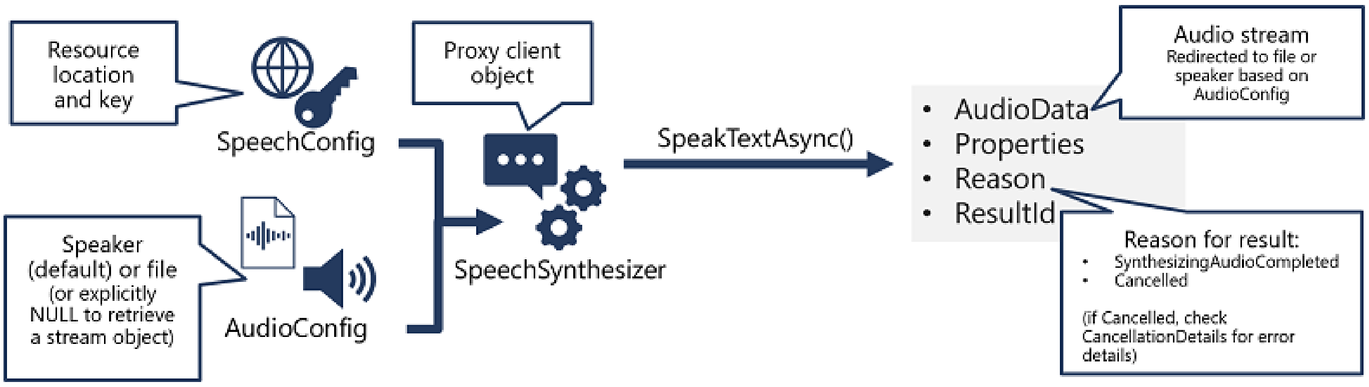
1. Use a SpeechConfig object to encapsulate the information required to connect to your Speech resource. Specifically, its location and key.
2. Optionally, use an AudioConfig to define the input source for the audio to be transcribed. By default, this is the default system microphone, but you can also specify an audio file.
3. Use the SpeechConfig and AudioConfig to create a SpeechRecognizer object. This object is a proxy client for the Speech-to-text API.
4. Use the methods of the SpeechRecognizer object to call the underlying API functions. For example, the RecognizeOnceAsync() method uses the Speech service to asynchronously transcribe a single spoken utterance.
5. Process the response from the Speech service. In the case of the RecognizeOnceAsync() method, the result is a SpeechRecognitionResult object that includes the following properties:

- Duration
- OffsetInTicks
- Properties

- Reason
- ResultId
- Text

5.2. Text to speech

- The Text-to-speech API, which is the primary way to perform speech synthesis. The endpoint for this API is `https://<LOCATION>.api.cognitive.microsoft.com/sts/v1.0`
    - Speech synthesis supports
        - Enhancing learning with multiple modes
        - Responding in multitasking scenarios
        - Improving accessibility
        - Delivering intuitive bots or assistants
- The Text-to-speech Long Audio API, which is designed to support batch operations that convert large volumes of text to audio - for example to generate an audio-book from the source text. The endpoint for this API is at `https://<LOCATION>.customvoice.api.speech.microsoft.com/api/texttospeech/v3.0/longaudiosynthesis`


Resource location and key
SpeechConfig
Speaker (default) or file (or explicitly NULL to retrieve a stream object)
AudioConfig
Proxy client object
SpeechSynthesizer
SpeakTextAsync()
AudioData
Properties
Reason
ResultId
Audio stream Redirected to file or speaker based on AudioConfig
Reason for result: SynthesizingAudioCompleted Cancelled (if Cancelled, check CancellationDetails for error details)

Audio format: you can pick the audio file type, the sample rate and the bit depth. The supported formats are indicated in the SDK using the *SpeechSynthesisOutputFormat* enumeration. To specify a output format:

```
speechConfig.SetSpeechSynthesisOutputFormat(SpeechSynthesisOutputFormat.Riff24Khz16BitMonoPcm);
```

For voices, you can pick standard voices (synthetic), neural (natural, created from DNNs) or custom. To specify a voice:

```
speechConfig.SpeechSynthesisVoiceName = "en-GB-George";
```

You can also use the *Speech Synthesis Markup Language (SSML)*. this includes:

- Specify a speaking style (excited, cheerful) when using a neural voice
- Insert pauses or silence
- Specify phonemes (pronouncing SQL as sequel)
- Adjust the prosody of the voice (affecting the pitch, timbre, and speaking rate).

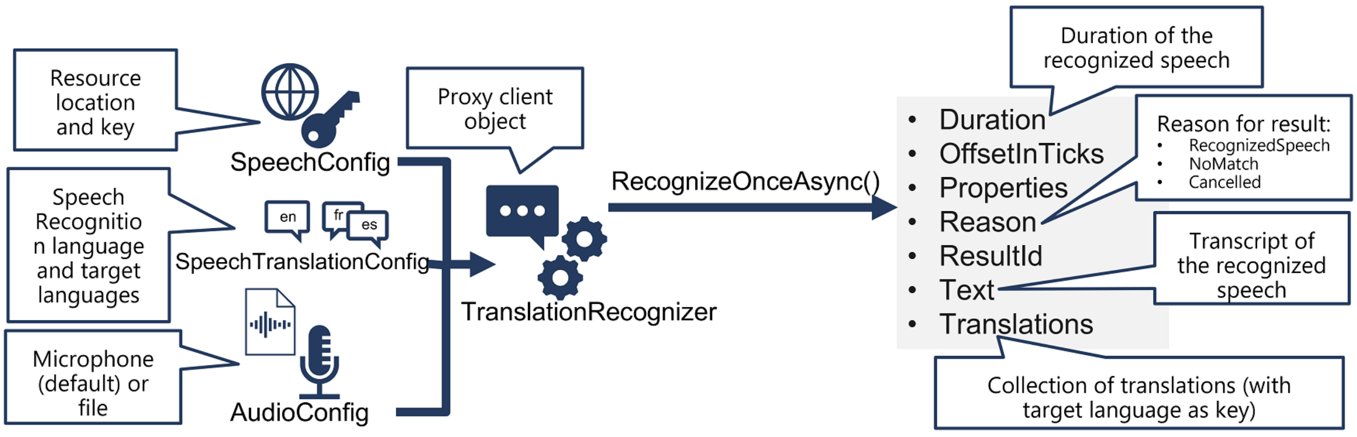
- Use common "say-as" rules, for example to specify that a given string should be expressed as a date, time, telephone number, or other form.
- Insert recorded speech or audio, for example to include a standard recorded message or simulate background noise.

```
<speak version="1.0" xmlns="http://www.w3.org/2001/10/synthesis"
                     xmlns:mstts="https://www.w3.org/2001/mstts" xml:lang="en-US">
    <voice name="en-US-AriaNeural">
        <mstts:express-as style="cheerful">
          I say tomato
        </mstts:express-as>
    </voice>
    <voice name="en-US-GuyNeural">
        I say <phoneme alphabet="sapi" ph="t ao m ae t ow"> tomato </phoneme>.
        <break strength="weak"/>Lets call the whole thing off!
    </voice>
</speak>

# to submit SSML description to the Speech device:
speechSynthesizer.SpeakSsmlAsync(ssml_string);
```

5.3. Speech translation

TranslationRecognizer does speech to text, it's the proxy for the Translation API of Speech service


Resource location and key
SpeechConfig
Speech Recognition language and target languages
en
fr
es
SpeechTranslationConfig
Microphone (default) or file
AudioConfig
Proxy client object
TranslationRecognizer
RecognizeOnceAsync()
Duration of the recognized speech
Duration
OffsetInTicks
Properties
Reason
ResultId
Text
Translations
Reason for result:
RecognizedSpeech
NoMatch
Cancelled
Transcript of the recognized speech
Collection of translations (with target language as key)

Supports speech-to-speech translation and speech-to-text. The engine knows that an utterance is finished when there's a pause in the audio.

To do speech-to-speech directly:

- Event-based synthesis
  - when translating to only 1 target language
  1. Specify the desired voice for the translated speech in the TranslationConfig.
  2. Create an event handler for the TranslationRecognizer object's Synthesizing event.
  3. In the event handler, use the GetAudio() method of the Result parameter to retrieve the byte stream of translated audio.
- Manual synthesis

  - doesn't require to implement an event handler
  - Supports 1+ target languages
  - Manual synthesis of translations is essentially just the combination of two separate operations in which you:
    - Use a TranslationRecognizer to translate spoken input into text transcriptions in one or more target languages
    - Iterate through the Translations dictionary in the result of the translation operation, using a SpeechSynthesizer to synthesize an audio stream for each language

6. Vision

6.1. Computer vision

- Image analysis
  - Upload image, specify the visual features you want to include in the analysis by adding a description. Returns JSON with description, captions, confidence scores
- Generate smart-cropped thumbnail
  - Thumbnails can have a width and height up to 1024 pixels, with a recommended minimum size of 50x50 pixel
  - When you run a curl command in Azure CLI, thumbnails are stored in Azure Cloud shell storage accoutn with the filename thumbnail.jpg
- Video analysis in the Video Analyzer for Media
  - Can extract: people, language, brands, animated characters
  - Can be used through the portal, widget or API
  - First upload the video and index it
  - For recognizing custom entities (specific people), create a custom model containing a Person for each person, with example images of their faces
  - Videos are split into shots by the video indexer. Each shot includes metadata, e.g. keyframes. A scene is a single event within the vide, it groups consecutive shots that are related. It has a start time, end time and thumbnail.
    - Video specs: videofilename less than 80 characters, upload size less than 30GB, max duration 4h
    - In the video indexer subscription, for an API request you need the Account ID and the API key
    - Video Indexer identifies temporal segments within the video to improve how you browse and edit indexed videos. The key aspects are extracted based on changes in colour, contrast, and other semantic properties
  - Video insights contain: transcript, ocr, keywords, labels, faces, brands, sentiments, emotions
- Custom vision
  - Needs 2 resources: training and prediction. A single cognitive resource can be used for both, or mix custom vision / cognitive resource
  - Can be used for image classification and object detection
  - Supports active learning
  - Needs training key for training API and prediction key for prediction API
  - Each time a model is trained, a new iteration is created
- Facial analysis
  - Computer vision service: gender, age

  - Face service: bounding box of face, facial feature analysis (age, gender, emotional state, head pose, hair color, facial hair, glasses), face comparison, facial recognition
  - Data privacy and security, transparency, fairness and inclusiveness
  - When a face is detected by the Face service, an ID is assigned to it. Next images are compared to the previous image for facial recognition
  - Facial recognition builds on the facial detection API
  - Face API tasks falls into 5 categories
    - Detection: detect human faces in an image
    - Verification: check the likelihood that 2 faces belong to the same person
    - Identification: search and identify faces
    - Similarity: find similar faces
    - Grouping: organize unidentified faces into groups based on similarity
  - Storage limits:
    - up to 1000 distinct faces
    - Up to 10000 persons
    - Up to 248 faces
- Optical character recognition
  - OCR API
    - read small-medium volumes of text from images
    - Supports multiple languages
    - Resultes are returned immediately from a single function call
    - Input: image URL / binary image data, language, and optional input parameter detectOrientation
    - In the result, the text is broken down by region, then line, and then word
  - Read API
    - Small-large volumes of text from images and PDF documents
    - More accuracy than OCR API
    - Can read printed text in multiple languages, and handwritten text in English
    - To recognize text: `recognizeText?mode=` mode= to choose printed/handwritten
    - The initial function call returns an asynchronous operation ID, which must be used in a subsequent call to retrieve the results
    - Input: image URL / binary image data, language
    - In the result, the text is broken down by page, then line, and then word
- Form recognizer
  - OCR to extract semantic meaning from forms
  - Form OCR test tool (FOTT) UI: supports prebuilt models, layout and custom training
  - Create Cognitive service resource or Form recognizer resource
  - Services:
    - Layout: takes input of jpeg, png, pdf and tiff (less than 50MB). Image dimensions must be between 50x50 and 10kx10k. Total size of training dataset must be less than 500 pages. Returns json file with location of text in bbox, text content, tables etc
    - Prebuilt models: detect and extract info from document images. Support for: receipts, business cards, invoices
    - Custom models: can be trained by calling the Train Custom Model API. Can be used unsupervised(at least 5 samples) / supervised (needs files, .ocr.json, .labels.json, .fields.json)

7. Cognitive search

- Index documents and data from a range of sources.
- Use cognitive skills to enrich index data.
- Store extracted insights in a knowledge store for analysis and integration.

Service tiers:

- Free
- Basic: max 15 indexes, 2GB of index data
- Standard: enterprise-scale solutions. S, S2, S3, S3HD
- Storage optimized: L1, L2. More latency but large indexes
- Replica: instances of the device, increase for more capacity
- Partition: used to divide an index into multiple storaeg locations, enabling you to split I/O operations such as querying or rebuilding an index

The combination of replicas and partitions you configure determines the search units used by your solution. Put simply, the number of search units is the number of replicas multiplied by the number of partitions (R x P = SU). For example, a resource with four replicas and three partitions is using 12 search units.

Search components:

- Data source: unstructured files in Azure blob, tables in SQL database, documents in Cosmos DB
- Skillset: AI skills can be applied to indexing to enrich the source data with new info: language, key phrases, sentiment score, specific locations/people/etc, AI generated description of images
  - The **Shaper** skill maps the enriched fields extracted by a given skillset to the desired structure for the knowledge store data
  - If there is a custom skill implemented as an Azure function, add a WebApiSkill to a skillset and reference the function's URI
- Indexer: the engine that drives the indexing
  - An indexer is automatically run when it is created, and can be scheduled to run at regular intervals or run on demand to add more documents to the index
  - In some cases, such as when you add new fields to an index or new skills to a skillset, you may need to reset the index before re-running the indexer
- Index: the searchable result of the indexing. It's a collection of JSON documents

When indexing, one document is created for each indexed entity. During indexing, an enrichment pipeline iteratively builds the documents that combine metadata from the data source with enriched fields extracted by cognitive skills.

```
documents
  metadata_storage_name
  metadata_author
  content
  normalized_images
      image0
```

```
            Text
        image1
            Text
    language
    merged_content
```

7.1. Search an index

Full text search

- Simple - An intuitive syntax that makes it easy to perform basic searches that match literal query terms submitted by a user
- Full - An extended syntax that supports complex filtering, regular expressions, and other more sophisticated queries
  - based on the Lucene query syntax, which provides a rich set of query operations for searching, filtering, and sorting data in indexes.

Query processing has 4 stages:

1. Query parsing
   1. The search expression is evaluated and reconstructed as a tree of appropriate subqueries. Subqueries might include term queries (finding specific individual words in the search expression - for example hotel), phrase queries (finding multi-term phrases specified in quotation marks in the search expression - for example, "free parking"), and prefix queries (finding terms with a specified prefix - for example air*, which would match aircon, air-conditioning, and airport)
2. Lexical analysis
   1. The query terms are analyzed and refined based on linguistic rules. For example, text is converted to lower case, non-essential stopwords (such as "the", "a", "is", and so on) are removed, words are converted to their root form (for example, "comfortable" may be simplified to "comfort"), and composite words are split into their constituent terms
3. Document retrieval
   1. The query terms are matched against the indexed terms, and the set of matching documents is identified
4. Scoring
   1. A relevance score is assigned to each result based on a term frequency/inverse document frequency (TF/IDF) calculation

7.2. Filtering results

Filters can be applied in 2 ways, and can be applied to any filterable field in the index

- By including filter criteria in a simple search expression
- By providing an OData filter expression as a $filter parameter with a full syntax search expression

```
// simple search
search=London+author='Reviewer'
queryType=Simple
```

```
// full search
search=London
$filter=author eq 'Reviewer'
queryType=Full
```

7.3. Filtering with facets

For fields that have a small number of discrete values. To use facets you must specify facetable fields in the index.

```
// example 1
search=*
facet=author

// example 2
search=*
$filter=author eq 'selected-facet-value-here'
```

To sort results

```
search=*
$orderby=last_modified desc
```

- Enrichment pipeline: enhance the index with insights derived from source data (using NLP and CV to generate descriptions of images)
- Enrichment pipeline can be persisted in a knowledge store for further analysis

8. Decision

- Anomaly detection
- Text moderation
  - Images: min 128 pixels and max file size 4MB
  - Category: score between 0 and 1
  - Analyze text to look for unwanted content
  - Classify the potentially offensive content
    - Category 1: sexually explicit or adult language
    - Category 2: sexually suggestive or mature language
    - Category 3: offensive language
  - Returns a boolean for "reviewRecommended"
  - The "ListId" identifies a specific term found in a custom term list, if such list is available
  - Get insights into the potential PII that's being shared so that you can protect it. If PII values are found, the API returns info about the text and the index location within the text
  - But it doesn't support automatically blocking repeat offenders of submitting spam content
- Content personalization