Analyze text with Azure AI Language

The Azure AI Language service enables you to create intelligent apps and services that **extract semantic information from text**.

Learning objectives

In this module, you'll learn how to use the Azure AI Language service to:

- **Detect language** from text
- Analyze text **sentiment**
- **Extract key phrases, entities, and linked entities**

Introduction

Every day, the world generates a vast quantity of data; much of it text-based in the form of emails, social media posts, online reviews, business documents, and more. Artificial intelligence techniques that apply statistical and semantic models enable you to create applications that extract meaning and insights from this text-based data.

The Azure AI Language provides an API for **common text analysis techniques** that you can easily integrate into your own application code.

In this module, you will learn how to use Azure AI Language to:

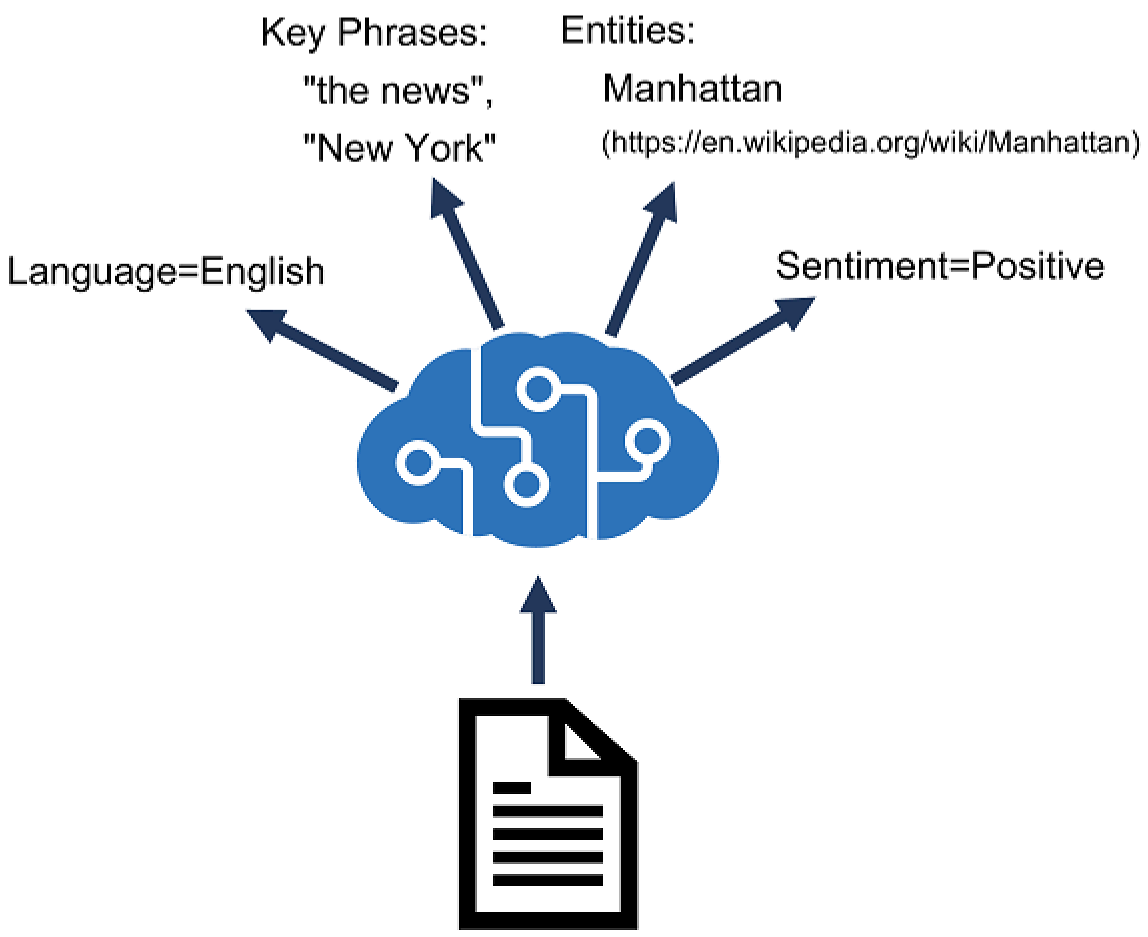
- Detect language from text.
- Analyze text sentiment.
- Extract key phrases, entities, and linked entities.

Provision an Azure AI Language resource

Azure AI Language is designed to help you extract information from text. It provides functionality that you can use for tasks like:

- **Language detection** - determining the language in which text is written.

- **Key phrase extraction** - identifying **important words and phrases** in the text that indicate the **main points**.
- **Sentiment analysis** - quantifying how positive or negative the text is.
- **Named entity recognition** - detecting references to entities, including **people, locations, time periods, organizations**, and more.
- **Entity linking** - identifying specific entities by providing **reference links to Wikipedia articles**.



Azure resources for text analysis

To use Azure AI Language to analyze text, you must provision a resource for it in your Azure subscription. You can **provision a resource through Azure AI Foundry portal**.

After you have provisioned a suitable resource in your Azure subscription, you can use its endpoint and one of its keys to call the Azure AI Language APIs from your code. You can call the Azure AI

Language APIs by **submitting requests in JSON format to the REST interface**, or by **using any of the available programming language-specific SDKs**.

> Note: The code examples in the subsequent units in this module show the JSON requests and responses exchanged with the REST interface. When using an SDK, the JSON requests are abstracted by appropriate objects and methods that encapsulate the same data values. You'll get a chance to try the SDK for C# or Python for yourself in the exercise later in the module.

Detect language

The **Azure AI Language detection API** evaluates text input and, for each document submitted, returns **language identifiers** with **a score indicating the strength of the analysis**.

This capability is useful for content stores that collect arbitrary text, where language is unknown. Another scenario could involve a chat bot. If a user starts a session with the chat bot, **language detection can be used to determine which language they are using** and allow you to configure your bot responses in the appropriate language.

You can parse the results of this analysis to determine which language is used in the input document. The response also returns a score, which reflects **the confidence of the model (a value between 0 and 1)**.

Language detection can work with documents or single phrases. It's important to note that the **document size must be under 5,120 characters**. The **size limit is per document** and **each collection is restricted to 1,000 items (IDs)**. A sample of a properly formatted JSON payload that you might submit to the service in the request body is shown here, including a collection of documents, each containing a unique id and the text to be analyzed. Optionally, you can provide a countryHint to improve prediction performance.

```json
{
    "kind": "LanguageDetection",
    "parameters": {
        "modelVersion": "latest"
    },
    "analysisInput":{
        "documents":[
              {
                "id": "1",
                "text": "Hello world",
                "countryHint": "US"
              },
              {
                "id": "2",
                "text": "Bonjour tout le monde"
              }
        ]
    }
}
```

The service will return a JSON response that contains a result for each document in the request body, **including the predicted language and a value indicating the confidence level of the prediction**. The confidence level is a value ranging from 0 to 1 with values closer to 1 being a higher confidence level. Here's an example of a standard JSON response that maps to the above request JSON.

```
{   "kind": "LanguageDetectionResults",
    "results": {
        "documents": [
          {
            "detectedLanguage": {
              "confidenceScore": 1,
              "iso6391Name": "en",
              "name": "English"
            },
            "id": "1",
            "warnings": []
          },
          {
            "detectedLanguage": {
              "confidenceScore": 1,
              "iso6391Name": "fr",
              "name": "French"
            },
            "id": "2",
            "warnings": []
          }
        ],
        "errors": [],
        "modelVersion": "2022-10-01"
    }
}
```

In our sample, all of the languages show a confidence of 1, mostly because the text is relatively simple and easy to identify the language for.

Multilingual Content

If you pass in a document that has multilingual content, the service will behave a bit differently. **Mixed language content within the same document returns the language with the largest representation in the content**, **but with a lower positive rating**, reflecting the marginal strength of that assessment. In the following example, the input is a blend of English, Spanish, and French. The analyzer uses statistical analysis of the text to determine the predominant language.

Textual Content not Parsable

The last condition to consider is when there is ambiguity as to the language content. The scenario might happen if you submit textual content that the analyzer is not able to parse, for example because of character encoding issues when converting the text to a string variable. As a result, the response for the language name and ISO code will indicate (unknown) and the score value will be returned as `0`. The following example shows how the response would look.

Extract key phrases

Key phrase extraction is the process of **evaluating the text of a document, or documents, and then identifying the main points around the context of the document(s)**.

Key phrase extraction **works best for larger documents** (**the maximum size that can be analyzed is 5,120 characters**).

As with language detection, the REST interface enables you to submit one or more documents for analysis.

Analyze sentiment

Sentiment analysis is used to **evaluate how positive or negative a text document is**, which can be useful in various workloads, such as:

- Evaluating a movie, book, or product by quantifying sentiment based on reviews.
- Prioritizing customer service responses to correspondence received through email or social media messaging.

When using Azure AI Language to evaluate sentiment, the response includes **overall document sentiment and individual sentence sentiment** for each document submitted to the service.

For example, you could submit a single document for sentiment analysis like this:

Sentence sentiment is based on confidence scores for positive, negative, and neutral classification values between 0 and 1.

Overall document sentiment is based on sentences:

- If all sentences are neutral, the overall sentiment is neutral.
- If sentence classifications include only positive and neutral, the overall sentiment is positive.
- If the sentence classifications include only negative and neutral, the overall sentiment is negative.
- If the sentence classifications include positive and negative, the overall sentiment is **mixed**.

Extract entities

Named Entity Recognition identifies entities that are mentioned in the text. Entities are grouped into **categories and subcategories**, for example:

- Person
- Location
- DateTime
- Organization
- Address
- Email
- URL

> Note: For a full list of categories, see the documentation.

Input for entity recognition is similar to input for other Azure AI Language API functions:

```
{
  "kind": "EntityRecognition",
  "parameters": {
    "modelVersion": "latest"
  },
  "analysisInput": {
    "documents": [
      {
        "id": "1",
        "language": "en",
        "text": "Joe went to London on Saturday"
      }
    ]
  }
}
```

The response includes a list of categorized entities found in each document:

```
{
    "kind": "EntityRecognitionResults",
     "results": {
          "documents":[
              {
                  "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
                  }
                ],
                "id":"1",
                "warnings":[]
              }
          ],
          "errors":[],
          "modelVersion":"2021-01-15"
    }
}
```

To learn more about entities see the Build a conversational language understanding model module.

Extract linked entities

In some cases, the same name might be applicable to more than one entity. For example, does an instance of the word "Venus" refer to the planet or the goddess from mythology?

Entity linking can be used to **disambiguate entities of the same name by referencing an article in a knowledge base**. Wikipedia provides the knowledge base for the **Text Analytics service**. Specific article links are determined based on entity context within the text.

For example, "I saw Venus shining in the sky" is associated with the Planet Venus; while "Venus, the goddess of beauty" is associated with Godness Venus.

As with all Azure AI Language service functions, you can submit one or more documents for analysis:

```
{
  "kind": "EntityLinking",
  "parameters": {
    "modelVersion": "latest"
  },
  "analysisInput": {
    "documents": [
      {
        "id": "1",
        "language": "en",
        "text": "I saw Venus shining in the sky"
      }
    ]
  }
}
```

The response includes the entities identified in the text along with links to associated articles:

```json
{
  "kind": "EntityLinkingResults",
  "results": {
    "documents": [
      {
        "id": "1",
        "entities": [
          {
            "bingId": "89253af3-5b63-e620-9227-f839138139f6",
            "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": "2021-06-01"
  }
}
```

Exercise - Analyze text

In this exercise, you use the Azure AI Language SDK to develop a client application that analyzes text.

Analyze Text

Azure Language supports analysis of text, including language detection, sentiment analysis, key phrase extraction, and entity recognition.

For example, suppose a travel agency wants to process hotel reviews that have been submitted to the company’s web site. By using the Azure AI Language, they can determine the language each review is written in, the sentiment (positive, neutral, or negative) of the reviews, key phrases that might indicate the main topics discussed in the review, and named entities, such as places, landmarks, or people mentioned in the reviews.

Module assessment

1. How should you create an application that monitors the comments on your company's web site and flags any negative posts? **use the Azure AI service to perform sentiment analysis of the comments**.
2. You are analyzing text that contains the word "Paris". How might you determine if this word refers to the French city or the character in Homer's "The Iliad"? **Use the Azure AI Language Service to extract linked entities**.

Summary

In this module, you learned how to use Azure AI Language to:

- **Detect language** from text.
- Analyze text **sentiment**.
- Extract **key phrases, entities, and linked entities**.

To learn more about Azure AI Language and some of the concepts covered in this module, you can explore the following documentation pages:

- Azure AI Language documentation
- Build a conversational language understanding model
- Create a custom named entity extraction solution

Create question answering solutions with Azure AI Language

The question answering capability of the Azure AI Language service makes it easy to build applications in which **users ask questions using natural language and receive appropriate answers**.

Learning objectives

After completing this module, you will be able to:

- Understand question answering and how it compares to language understanding.
- Create, test, publish, and consume a **knowledge base**.
- Implement **multi-turn conversation** and **active learning**.
- Create a question answering bot to interact with using natural language.

Introduction

A common pattern for "intelligent" applications is to enable users to ask questions using natural language, and receive appropriate answers. In effect, this kind of solution brings **conversational intelligence to a traditional frequently asked questions (FAQ) publication**.

In this module, you will learn how to use Azure AI Language to **create a knowledge base of question and answer pairs** that can support an application or bot.

After completing this module, you'll be able to:

- Understand question answering and how it compares to language understanding.
- Create, test, publish and consume a knowledge base.
- Implement multi-turn conversation and active learning.
- Create a question answering bot to interact with using natural language.

Understand question answering

Azure AI Language includes a question answering capability, which enables you to define **a knowledge base of question and answer pairs** that can be queried using natural language input. The knowledge base can be published to a REST endpoint and consumed by client applications, commonly bots.

The knowledge base can be created from existing sources, including:

- **Web sites containing frequently asked question (FAQ) documentation**.
- Files containing structured text, such as brochures or user guides.
- Built-in chit chat question and answer pairs that encapsulate common conversational exchanges.

> Note: The question answering capability of Azure AI Language is a newer version of the QnA Service, which still exists as a standalone service. To learn how to migrate a QnA Maker knowledge base to Azure AI Language, see the migration guide.

Compare question answering to Azure AI Language understanding

A question answering knowledge base is a form of language model, which raises the question of when to use question answering, and **when to use the conversational language understanding capabilities of Azure AI Language**.

The two features are similar in that they both enable you to define a language model that can be queried using natural language expressions. However, there are some differences in the use cases that they are designed to address, as shown in the following table:

	Question answering	**Language understanding**
Usage pattern	User submits a question, expecting an answer	User submits an utterance, expecting an appropriate response or action
Query processing	Service uses natural language understanding to match the question to an answer in the knowledge base	Service uses natural language understanding to interpret the utterance, match it to an intent, and identify entities

	Question answering	Language understanding
Response	Response is **a static answer to a known question**	Response indicates the most likely intent and referenced entities
Client logic	Client application typically presents the answer to the user	Client application is responsible for performing appropriate action based on the detected intent

The two services are in fact complementary. You can build **comprehensive natural language solutions that combine language understanding models and question answering knowledge bases**.

Create a knowledge base

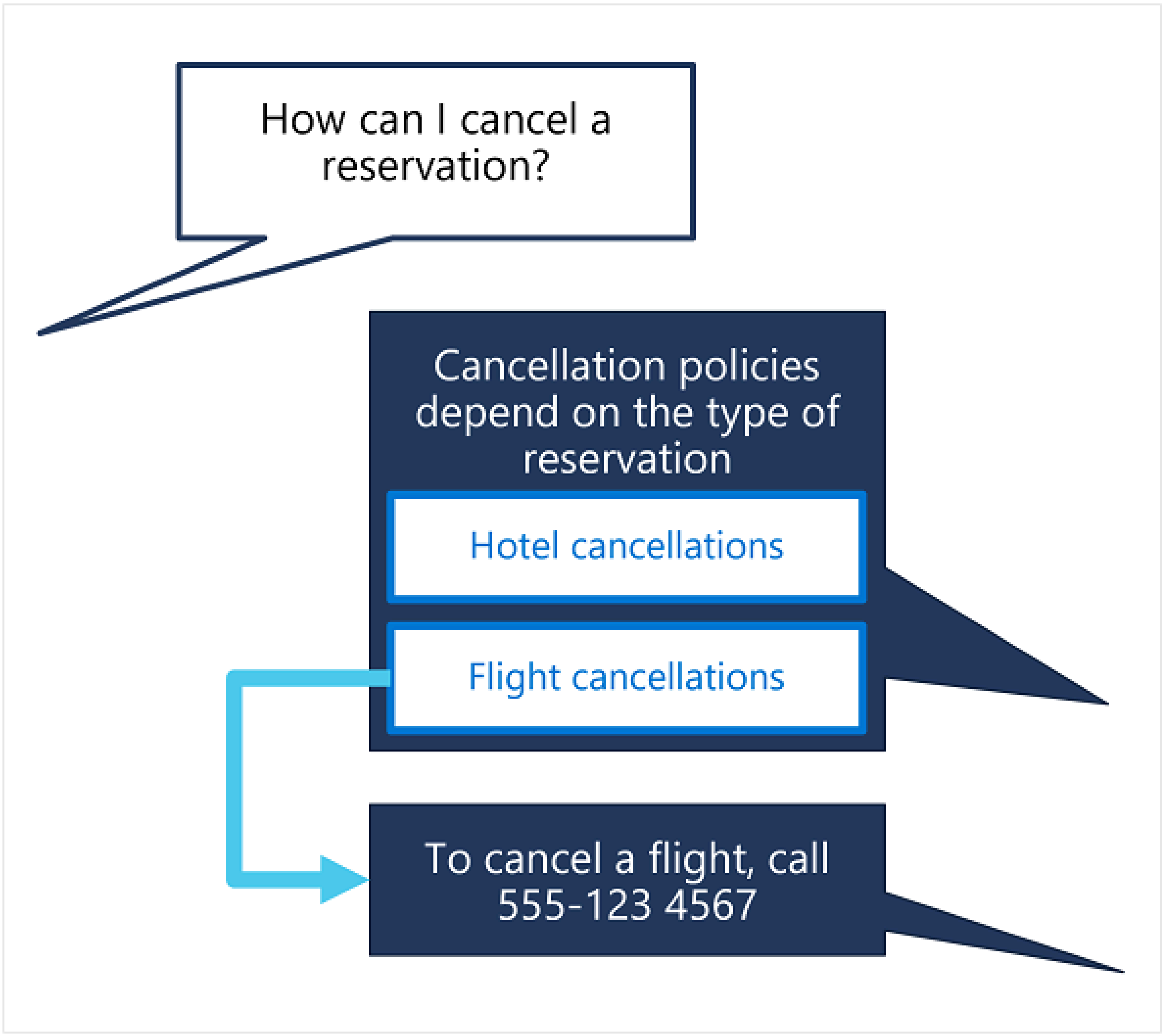
To create a question answering solution, you can use the REST API or SDK to write code that defines, trains, and publishes the knowledge base. However, it's more common to use the Language Studio web interface to define and manage a knowledge base.

To create a knowledge base you:

1. Sign in to Azure portal.
2. Search for Azure AI services using the search field at the top of the portal.
3. Select Create under the Language Service resource.
4. Create a resource in your Azure subscription:
   - Enable the question answering feature.
   - Create or select an Azure AI Search resource to host the knowledge base index.
5. In Language Studio, select your Azure AI Language resource and create a Custom question answering project.
6. Add one or more data sources to populate the knowledge base:
   - URLs for web pages containing FAQs.
   - Files containing structured text from which questions and answers can be derived.
   - Predefined chit-chat datasets that include common conversational questions and responses in a specified style.
7. Edit question and answer pairs in the portal.

Implement multi-turn conversation

Although you can often create an effective knowledge base that consists of individual question and answer pairs, sometimes you might need to **ask follow-up questions to elicit more information from a user before presenting a definitive answer**. This kind of interaction is referred to as a **multi-turn conversation**.


How can I cancel a reservation?
Cancellation policies depend on the type of reservation
Hotel cancellations
Flight cancellations
To cancel a flight, call 555-123 4567

You can **enable multi-turn responses when importing questions and answers from an existing web page or document based on its structure**, or you **can explicitly define follow-up prompts and responses for existing question and answer pairs**.

For example, suppose an initial question for a travel booking knowledge base is "How can I cancel a reservation?". A reservation might refer to a hotel or a flight, so a follow-up prompt is required to clarify this detail. The answer might consist of text such as "Cancellation policies depend on the type of reservation" and include follow-up prompts with links to answers about canceling flights and canceling hotels.

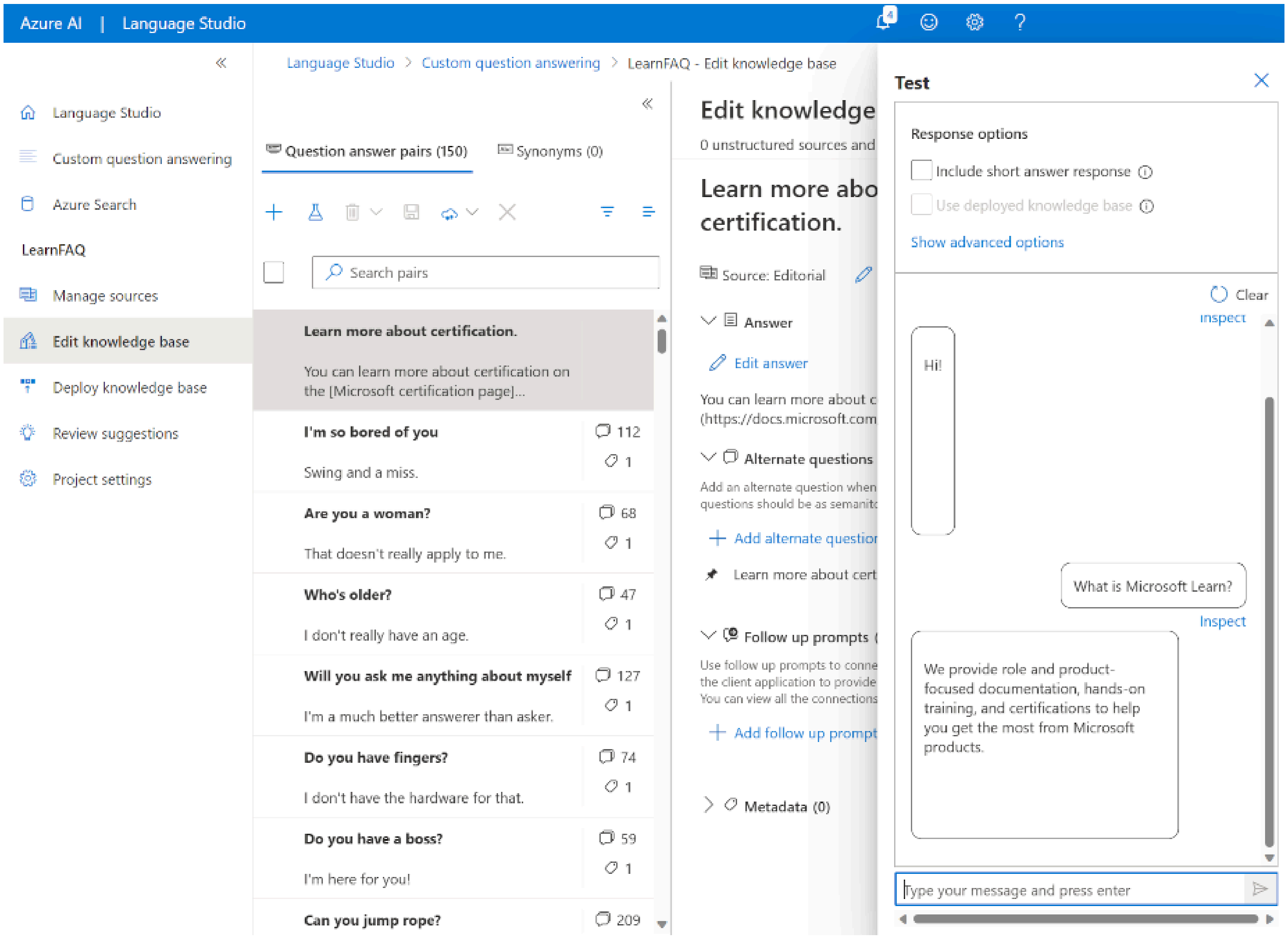
When you define a follow-up prompt for multi-turn conversation, you can link to an existing answer in the knowledge base or define a new answer specifically for the follow-up. You can also restrict the linked answer so that it is only ever displayed in the context of the multi-turn conversation initiated by the original question.

Test and publish a knowledge base

After you have defined a knowledge base, you can train its natural language model, and test it before publishing it for use in an application or bot.

Testing a knowledge base

You can test your knowledge base interactively in Language Studio, submitting questions and reviewing the answers that are returned. You can inspect the results to view their confidence scores as well as other potential answers.



Azure AI | Language Studio
Language Studio
Custom question answering
Azure Search
LearnFAQ
Manage sources
Edit knowledge base
Deploy knowledge base
Review suggestions
Project settings
Language Studio > Custom question answering > LearnFAQ - Edit knowledge base
Question answer pairs (150)
Synonyms (0)
Search pairs

Question	Answer		
Learn more about certification.	You can learn more about certification on the [Microsoft certification page]...		
I'm so bored of you	Swing and a miss.	112	1
Are you a woman?	That doesn't really apply to me.	68	1
Who's older?	I don't really have an age.	47	1
Will you ask me anything about myself	I'm a much better answerer than asker.	127	1
Do you have fingers?	I don't have the hardware for that.	74	1
Do you have a boss?	I'm here for you!	59	1
Can you jump rope?		209	

Edit knowledge
0 unstructured sources and
Learn more abo
certification.
Source: Editorial
Answer
Edit answer
You can learn more about c
(https://docs.microsoft.com
Alternate questions
Add an alternate question when
questions should be as semantic
Add alternate question
Learn more about cert
Follow up prompts (
Use follow up prompts to conne
the client application to provide
You can view all the connections
Add follow up prompt
Metadata (0)

Test
Response options
Include short answer response
Use deployed knowledge base
Show advanced options
Clear
Inspect
Hi!
What is Microsoft Learn?
Inspect
We provide role and product-focused documentation, hands-on training, and certifications to help you get the most from Microsoft products.
Type your message and press enter

Deploying a knowledge base

When you're happy with the performance of your knowledge base, you can **deploy it to a REST endpoint** that client applications can use to submit questions and receive answers. You can deploy it directly from **Language Studio**.

Use a knowledge base

To consume the published knowledge base, you can use the REST interface.

The minimal request body for the function contains a question, like this:

```
{
  "question": "What do I need to do to cancel a reservation?",
  "top": 2,
  "scoreThreshold": 20,
  "strictFilters": [
    {
      "name": "category",
      "value": "api"
    }
  ]
}
```

Property	Description
`question`	Question to send to the knowledge base.
`top`	Maximum number of answers to be returned.
`scoreThreshold`	*Score threshold for answers returned*.
`strictFilters`	Limit to only answers that contain the specified metadata.

The response includes the closest question match that was found in the knowledge base, along with the **associated answer**, the **confidence score**, and other metadata about the question and answer pair:

```json
{
  "answers": [
    {
      "score": 27.74823341616769,
      "id": 20,
      "answer": "Call us on 555 123 4567 to cancel a reservation.",
      "questions": [
        "How can I cancel a reservation?"
      ],
      "metadata": [
        {
          "name": "category",
          "value": "api"
        }
      ]
    }
  ]
}
```

Improve question answering performance

After creating and testing a knowledge base, you can **improve its performance with active learning and by defining synonyms**.

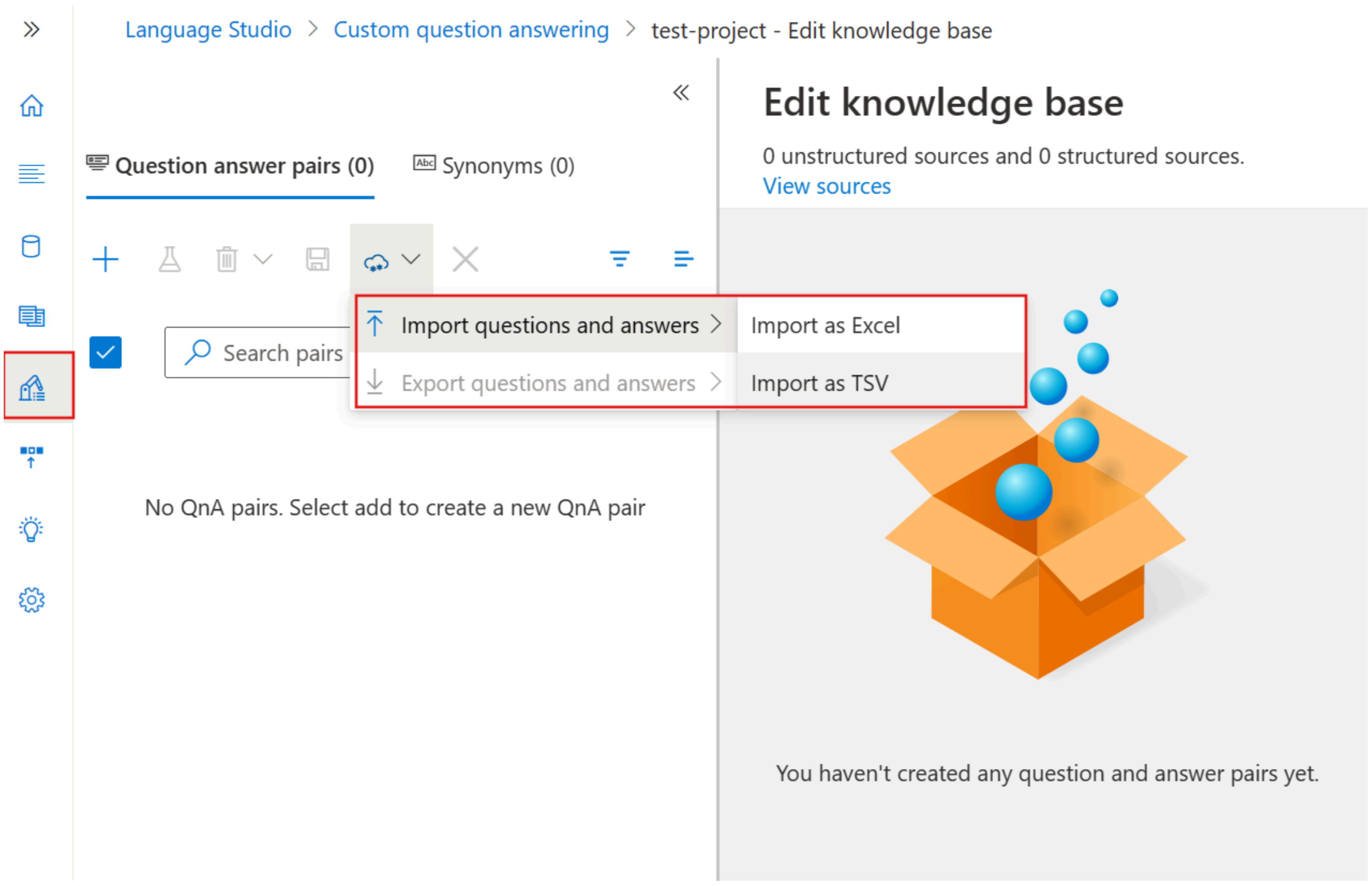
Use active learning

Active learning can help you make continuous improvements to get better at answering user questions correctly over time. **People often ask questions that are phrased differently, but ultimately have the same meaning**. Active learning can help in situations like this because it enables you to consider alternate questions to each question and answer pair. Active learning is enabled by default.

To use active learning, you can do the following:

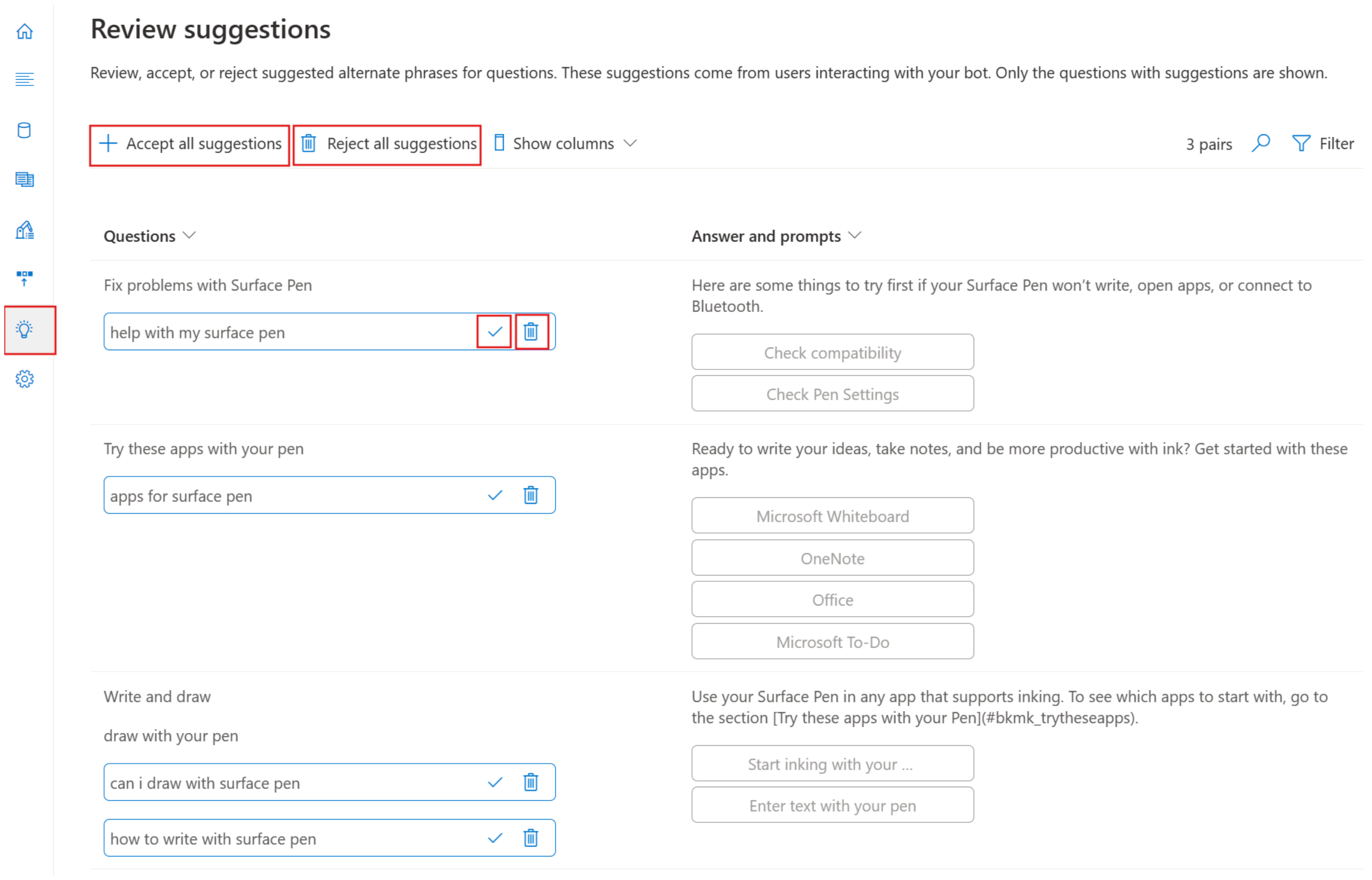
Create your question and answer pairs

You create pairs of questions and answers in Language Studio for your project. You can also import a file that contains question and answer pairs to upload in bulk.


Language Studio > Custom question answering > test-project - Edit knowledge base
Question answer pairs (0)
Synonyms (0)
Search pairs
Import questions and answers
Export questions and answers
Import as Excel
Import as TSV
No QnA pairs. Select add to create a new QnA pair
Edit knowledge base
0 unstructured sources and 0 structured sources.
View sources
You haven't created any question and answer pairs yet.

Review suggestions

Active learning then begins to offer alternate questions for each question in your question and answer pairs. You access this from the Review suggestions pane:



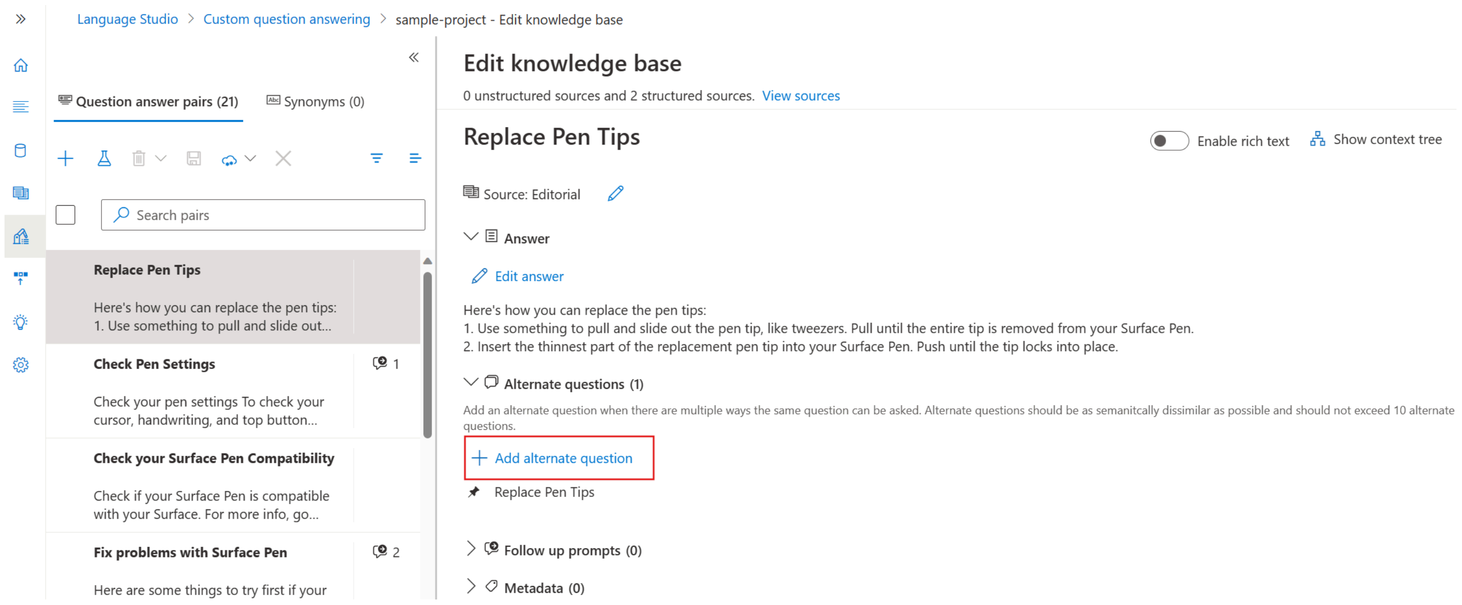
Review suggestions

Review, accept, or reject suggested alternate phrases for questions. These suggestions come from users interacting with your bot. Only the questions with suggestions are shown.

Questions	Answer and prompts
Fix problems with Surface Pen help with my surface pen	Here are some things to try first if your Surface Pen won't write, open apps, or connect to Bluetooth. Check compatibility Check Pen Settings
Try these apps with your pen apps for surface pen	Ready to write your ideas, take notes, and be more productive with ink? Get started with these apps. Microsoft Whiteboard OneNote Office Microsoft To-Do
Write and draw draw with your pen can i draw with surface pen how to write with surface pen	Use your Surface Pen in any app that supports inking. To see which apps to start with, go to the section [Try these apps with your Pen](#bkmk_trytheseapps). Start inking with your ... Enter text with your pen

You review, and then accept or reject these alternate phrases suggested for each question by selecting the checkmark or delete symbol next to the alternate phrase. You can bulk accept or reject suggestions using the Accept all suggestions or Reject all suggestions option at the top.

You can also manually add alternate questions when you select Add alternate question for a pair in the Edit knowledge base pane:



Language Studio > Custom question answering > sample-project - Edit knowledge base
Question answer pairs (21)
Synonyms (0)
Search pairs
Enable rich text
Show context tree

Replace Pen Tips
Here's how you can replace the pen tips: 1. Use something to pull and slide out...

Check Pen Settings 1
Check your pen settings To check your cursor, handwriting, and top button...

Check your Surface Pen Compatibility
Check if your Surface Pen is compatible with your Surface. For more info, go...

Fix problems with Surface Pen 2
Here are some things to try first if your

Edit knowledge base

0 unstructured sources and 2 structured sources. View sources

Replace Pen Tips

Source: Editorial

Answer

Edit answer

Here's how you can replace the pen tips:
1. Use something to pull and slide out the pen tip, like tweezers. Pull until the entire tip is removed from your Surface Pen.
2. Insert the thinnest part of the replacement pen tip into your Surface Pen. Push until the tip locks into place.

Alternate questions (1)

Add an alternate question when there are multiple ways the same question can be asked. Alternate questions should be as semanitcally dissimilar as possible and should not exceed 10 alternate questions.

+ Add alternate question

Replace Pen Tips

Follow up prompts (0)

Metadata (0)

Note: To learn more about active learning, see Enrich your project with active learning.

Define synonyms

Synonyms are useful when questions submitted by users might **include multiple different words to mean the same thing**. For example, a travel agency customer might refer to a "reservation" or a "booking". By defining these as synonyms, the question answering service can find an appropriate answer regardless of which term an individual customer uses.

To define synonyms, you **use the REST API to submit synonyms in the following JSON format**:

```
{
    "synonyms": [
        {
            "alterations": [
                "reservation",
                "booking"
                ]
        }
    ]
}
```

> Note: To learn more about synonyms, see the Improve quality of response with synonyms.

Exercise - Create a question answering solution

Create a Question Answering Solution

One of the most common conversational scenarios is providing support through a knowledge base of frequently asked questions (FAQs). Many organizations publish FAQs as documents or web pages, which works well for a small set of question and answer pairs, but large documents can be difficult and time-consuming to search.

Azure AI Language includes a question answering capability that enables you to create a knowledge base of question and answer pairs that can be queried using natural language input, and is most commonly used as a resource that a bot can use to look up answers to questions submitted by users.

Module assessment

1. You want to create a knowledge base from an existing FAQ document. What should you do? Create a new knowledge base, importing the existing FAQ document.
2. How can you add a multi-turn context for a question in an existing knowledge base? Add a follow-up prompt to the question.
3. How can you enable users to use your knowledge base through email? Create a bot based on your knowledge base and configure an email channel.

Summary

In this module, you have learned how to use the question answering capability of Azure AI Language to create a knowledge base of question and answer pairs that can support an application or bot.

Now that you've completed this module, you can:

- Understand **question answering** and how it compares to **language understanding**.
- Create, test, publish and consume a **knowledge base**.
- Implement **multi-turn conversation** and **active learning**.
- Create a **question answering bot** to interact with using natural language.

To learn more about the question answering capability of Azure AI Language, see the Question answering documentation.

Build a conversational language understanding model

The Azure AI Language **conversational language understanding service (CLU)** enables you to train a model that apps can use to extract meaning from natural language.

Learning objectives

After completing this module, you'll be able to:

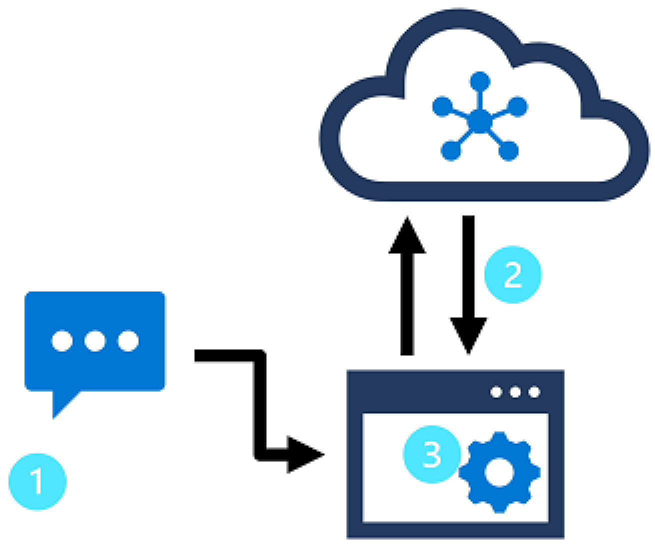
- Provision Azure resources for Azure AI Language resource
- Define intents, utterances, and entities
- Use patterns to differentiate similar utterances
- Use pre-built entity components
- Train, test, publish, and review an Azure AI Language model

Introduction

Video

Natural language processing (NLP) is a common AI problem in which software must be able to work with text or speech in the natural language form that a human user would write or speak. Within the broader area of NLP, **natural language understanding (NLU)** deals with the problem of determining semantic meaning from natural language - usually by using a trained language model.

A common design pattern for a natural language understanding solution looks like this:



In this design pattern:

1. An app accepts natural language input from a user.
2. A language model is used to determine semantic meaning (the user's intent).
3. The app performs an appropriate action.

Azure AI Language enables developers to build apps based on language models that can be trained with a relatively small number of samples to discern a user's intended meaning.

In this module, you'll learn how to use the service to create **a natural language understanding app** using Azure AI Language.

After completing this module, you'll be able to:

- Provision an Azure AI Language resource.
- Define intents, entities, and utterances.
- Use patterns to differentiate similar utterances.
- Use pre-built entity components.
- Train, test, publish, and review a model.

Understand prebuilt capabilities of the Azure AI Language service

The Azure AI Language service provides various features for understanding human language. You can use each feature to better communicate with users, better understand incoming communication, or use them together to provide more insight into what the user is saying, intending, and asking about.

Azure AI Language service features fall into two categories: **Pre-configured features**, and **Learned features**. Learned features require building and training a model to correctly predict appropriate labels, which is covered in upcoming units of this module.

This unit covers most of the capabilities of the Azure AI Language service, but head over to the Azure AI Language service documentation for a full list, including quickstarts and a full explanation of everything available.

Using these features in your app requires sending your query to the appropriate endpoint. The endpoint used to query a specific feature varies, but all of them are prefixed with the Azure AI Language resource you created in your Azure account, either when building your REST request or defining your client using an SDK. Examples of each can be found in the next unit.

Pre-configured features

The Azure AI Language service provides certain features without any model labeling or training. Once you create your resource, you can send your data and use the returned results within your app.

The following features are all pre-configured.

Summarization

Summarization is available for both documents and conversations, and will summarize the text into key sentences that are predicted to encapsulate the input's meaning.

Named entity recognition

Named entity recognition can **extract and identify entities, such as people, places, or companies**, allowing your app to recognize different types of entities for improved natural language responses. For example, given the text "The waterfront pier is my favorite Seattle attraction", Seattle would be identified and categorized as a location.

Personally identifiable information (PII) detection

PII detection allows you to **identify, categorize, and redact information that could be considered sensitive**, such as email addresses, home addresses, IP addresses, names, and protected health information. For example, if the text "email@contoso.com" was included in the query, the entire email address can be identified and redacted.

Key phrase extraction

Key phrase extraction is a feature that quickly pulls the main concepts out of the provided text. For example, given the text "Text Analytics is one of the features in Azure AI Services.", the service would extract "Azure AI Services" and "Text Analytics".

Sentiment analysis

Sentiment analysis identifies how **positive or negative** a string or document is. For example, given the text "Great hotel. Close to plenty of food and attractions we could walk to", the service would identify that as *positive* with a relatively high confidence score.

Language detection

Language detection takes one or more documents, and identifies the language for each. For example, if the text of one of the documents was "Bonjour", the service would identify that as *French*.

Learned features

Learned features require you to label data, train, and deploy your model to make it available to use in your application. These features allow you to customize what information is predicted or extracted.

> Note: Quality of data greatly impacts the model's accuracy. Be intentional about what data is used, how well it is tagged or labeled, and how varied the training data is. For details, see recommendations for labeling data, which includes valuable guidelines for tagging data. Also see the evaluation metrics that can assist in learning where your model needs improvement.

Conversational language understanding (CLU)

CLU is one of the core custom features offered by Azure AI Language. CLU helps users to build **custom natural language understanding models to predict overall intent and extract important information from incoming utterances**. CLU does require data to be tagged by the user to teach it how to predict intents and entities accurately.

The exercise in this module will be building a CLU model and using it in your app.

Custom named entity recognition

Custom entity recognition takes custom labeled data and extracts specified entities from unstructured text. For example, if you have various contract documents that you want to extract involved parties from, you can train a model to recognize how to predict them.

Custom text classification

Custom text classification enables users to classify text or documents as custom defined groups. For example, you can train a model to look at news articles and identify the category they should fall into, such as News or Entertainment.

Question answering

Question answering is a mostly pre-configured feature that provides answers to questions provided as input. The data to answer these questions comes from documents like FAQs or manuals.

For example, say you want to make a virtual chat assistant on your company website to answer common questions. You could use a company FAQ as the input document to create the question and answer pairs. Once deployed, your chat assistant can pass input questions to the service, and get the answers as a result.

For a complete list of capabilities and how to use them, see the Azure AI Language documentation.

Understand resources for building a conversational language understanding model

To use the Language Understanding service to develop a NLP solution, you'll need to **create a Language resource in Azure**. That resource will be used for both authoring your model and processing prediction requests from client applications.

> Tip: This module's lab covers building a model for conversational language understanding. For more focused modules on custom text classification and custom named entity recognition, see the custom solution modules in the Develop natural language solutions learning path.

Build your model

For features that require a model for prediction, you'll need to build, train and deploy that model before using it to make a prediction. This building and training will teach the Azure AI Language service what to look for.

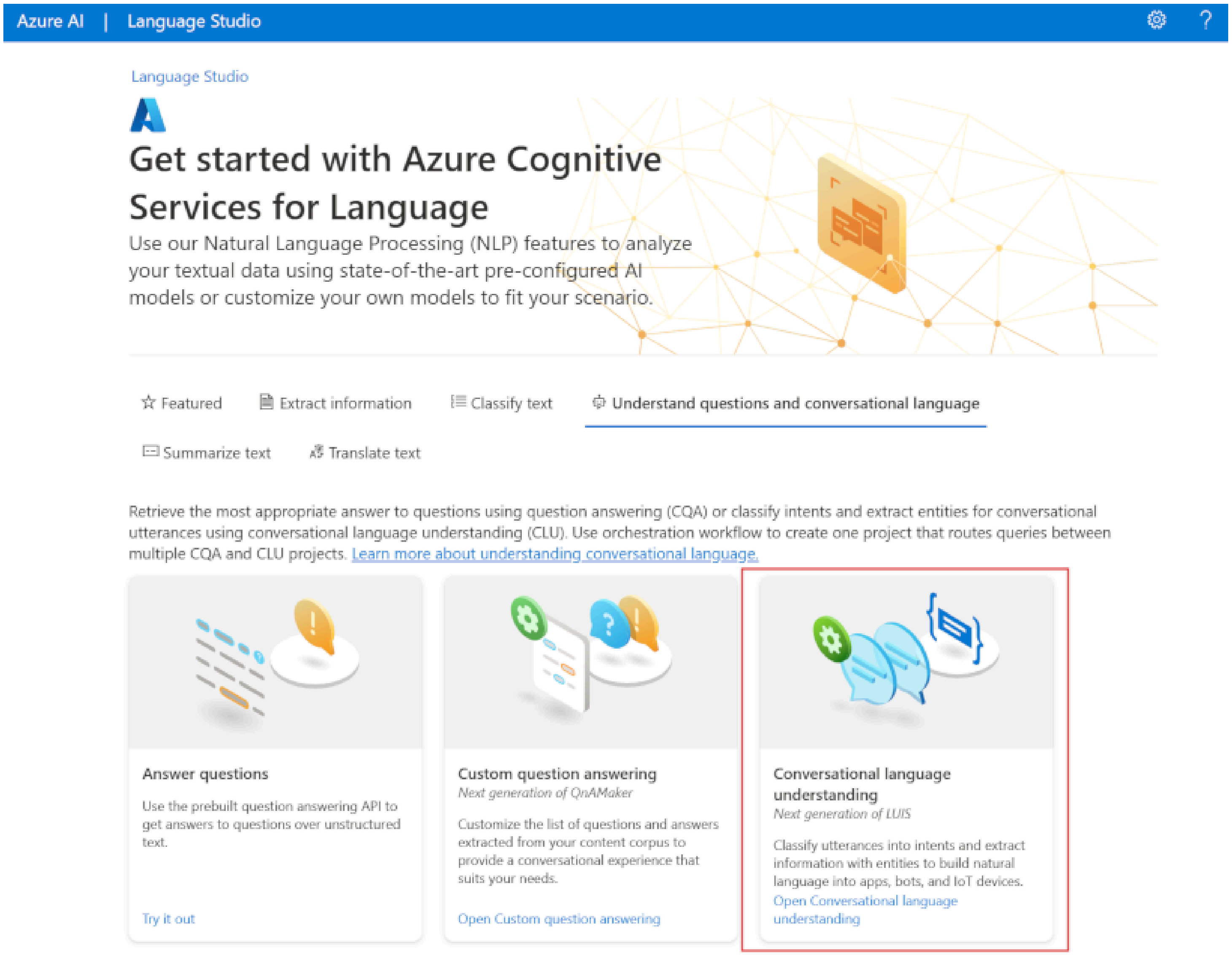
First, you'll need to create your Azure AI Language resource in the Azure portal. Then:

1. Search for **Azure AI services**.
2. Find and select **Language Service**.
3. Select *Create* under the **Language Service**.
4. Fill out the necessary details, choosing the region closest to you geographically (for best performance) and giving it a unique name.

Once that resource has been created, you'll need **a key and the endpoint**. You can find that on the left side under Keys and Endpoint of the resource overview page.

Use Language Studio

For a more visual method of building, training, and deploying your model, you can use Language Studio to achieve each of these steps. On the main page, you can choose to create a Conversational language understanding project. Once the project is created, then go through the same process as above to build, train, and deploy your model.


Azure AI | Language Studio

Language Studio

Get started with Azure Cognitive Services for Language

Use our Natural Language Processing (NLP) features to analyze your textual data using state-of-the-art pre-configured AI models or customize your own models to fit your scenario.

Retrieve the most appropriate answer to questions using question answering (CQA) or classify intents and extract entities for conversational utterances using conversational language understanding (CLU). Use orchestration workflow to create one project that routes queries between multiple CQA and CLU projects. Learn more about understanding conversational language.

Answer questions

Use the prebuilt question answering API to get answers to questions over unstructured text.

Try it out

Custom question answering

Next generation of QnAMaker

Customize the list of questions and answers extracted from your content corpus to provide a conversational experience that suits your needs.

Open Custom question answering

Conversational language understanding

Next generation of LUIS

Classify utterances into intents and extract information with entities to build natural language into apps, bots, and IoT devices.

Open Conversational language understanding

The lab in this module will walk through using Language Studio to build your model. If you'd like to learn more, see the Language Studio quickstart.

Use the REST API

One way to build your model is through the REST API. The pattern would be to create your project, import data, train, deploy, then use your model.

These tasks are done **asynchronously**; you'll need to submit a request to the appropriate URI for each step, and then **send another request to get the status of that job**.

For example, if you want to deploy a model for a conversational language understanding project, you'd submit the deployment job, and then check on the deployment job status.

Authentication

For each call to your Azure AI Language resource, you authenticate the request by providing the following header.

Key	Value
Ocp-Apim-Subscription-Key	The key to your resource

Request deployment

Submit a POST request to the following endpoint.

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYM
```

Placeholder	Value	Example
`{ENDPOINT}`	The endpoint of your Azure AI Language resource	`https://<your-subdomain>.cognitiveservices.azure.com`
`{PROJECT-NAME}`	The name for your project. This value is case-sensitive	`myProject`
`{DEPLOYMENT-NAME}`	The name for your deployment. This value is case-sensitive	`staging`
`{API-VERSION}`	The version of the API you're calling	`2022-05-01`

Include the following `body` with your request.

```
{
  "trainedModelLabel": "{MODEL-NAME}",
}
```

Placeholder	Value
`MODEL-NAME`	The model name that will be assigned to your deployment. This value is case-sensitive.

Successfully submitting your request will receive a `202` response, with a response header of operation-location. This header will have a URL with which to request the status, formatted like this:

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYM
```

Get deployment status

Submit a `GET` request to the URL from the response header above. The values will already be filled out based on the initial deployment request.

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYM
```

Placeholder	Value
`ENDPOINT`	The endpoint for authenticating your API request
`PROJECT-NAME`	The name for your project (case-sensitive)
`DEPLOYMENT-NAME`	The name for your deployment (case-sensitive)
`JOB-ID`	The ID for locating your model's training status, found in the header value detailed above in the deployment request
`API-VERSION`	The version of the API you're calling

The response body will give the deployment `status` details. The `status` field will have the value of *succeeded* when the deployment is complete.

```json
{
    "jobId":"{JOB-ID}",
    "createdDateTime":"String",
    "lastUpdatedDateTime":"String",
    "expirationDateTime":"String",
    "status":"running"
}
```

For a full walkthrough of each step with example requests, see the conversational understanding quickstart.

Query your model

To query your model for a prediction, you can **use SDKs in C# or Python**, or **use the REST API**.

Query using SDKs

To query your model using an SDK, you **first need to create your client**. Once you have your client, you **then use it to call the appropriate endpoint**.

```
language_client = TextAnalyticsClient(
            endpoint=endpoint,
            credential=credentials)
response = language_client.extract_key_phrases(documents = documents)[0]
```

Other language features, such as the conversational language understanding, require the request be built and sent differently.

```
result = client.analyze_conversation(
    task={
        "kind": "Conversation",
        "analysisInput": {
            "conversationItem": {
                "participantId": "1",
                "id": "1",
                "modality": "text",
                "language": "en",
                "text": query
            },
            "isLoggingEnabled": False
        },
        "parameters": {
            "projectName": cls_project,
            "deploymentName": deployment_slot,
            "verbose": True
        }
    }
)
```

Query using the REST API

To query your model using REST, create a `POST` request to the appropriate URL with the appropriate body specified. For built in features such as language detection or sentiment analysis, you'll query the `analyze-text` **endpoint**.

> Tip: Remember each request needs to be authenticated with your Azure AI Language resource key in the `Ocp-Apim-Subscription-Key` header

```
{ENDPOINT}/language/:analyze-text?api-version={API-VERSION}
```

Placeholder	Value
`ENDPOINT`	The endpoint for authenticating your API request
`API-VERSION`	The version of the API you're calling

Within the body of that request, you must specify the `kind` parameter, which tells the service what type of language understanding you're requesting.

If you want to detect the language, for example, the JSON body would look something like the following.

```
{
    "kind": "LanguageDetection",
    "parameters": {
        "modelVersion": "latest"
    },
    "analysisInput":{
        "documents":[
            {
                "id":"1",
                "text": "This is a document written in English."
            }
        ]
    }
}
```

Other language features, such as the conversational language understanding, require the request be routed to a different endpoint. For example, the conversational language understanding request would be sent to the following.

```
{ENDPOINT}/language/:analyze-conversations?api-version={API-VERSION}
```

Placeholder	Value
`ENDPOINT`	The endpoint for authenticating your API request
`API-VERSION`	The version of the API you're callings

That request would include a JSON body similar to the following.

```
{
  "kind": "Conversation",
  "analysisInput": {
    "conversationItem": {
      "id": "1",
      "participantId": "1",
      "text": "Sample text"
    }
  },
  "parameters": {
    "projectName": "{PROJECT-NAME}",
    "deploymentName": "{DEPLOYMENT-NAME}",
    "stringIndexType": "TextElement_V8"
  }
}
```

Placeholder	Value
`PROJECT-NAME`	The name of the project where you built your model
`DEPLOYMENT-NAME`	The name of your deployment

Sample response

The query response from an SDK will in the object returned, which varies depending on the feature (such as in `response.key_phrases` or `response.Value`). The REST API will return JSON that would be similar to the following.

```json
{
    "kind": "KeyPhraseExtractionResults",
    "results": {
        "documents": [{
            "id": "1",
            "keyPhrases": ["modern medical office", "Dr. Smith", "great staff"],
            "warnings": []
        }],
        "errors": [],
        "modelVersion": "{VERSION}"
    }
}
```

For other models like conversational language understanding, a sample response to your query would be similar to the following.

```json
{
  "kind": "ConversationResult",
  "result": {
    "query": "String",
    "prediction": {
      "topIntent": "intent1",
      "projectKind": "Conversation",
      "intents": [
        {
          "category": "intent1",
          "confidenceScore": 1
        },
        {
          "category": "intent2",
          "confidenceScore": 0
        }
      ],
      "entities": [
        {
          "category": "entity1",
          "text": "text",
          "offset": 7,
          "length": 4,
          "confidenceScore": 1
        }
      ]
    }
  }
}
```

The SDKs for both Python and C# return JSON that is very similar to the REST response.

For full documentation on features, including examples and how-to guides, see the Azure AI Language documentation documentation pages.

Define intents, utterances, and entities

Utterances are the phrases that a user might enter when interacting with an application that uses your language model. An **intent** represents **a task or action the user wants to perform**, or more simply the **meaning of an utterance**. You create a model by defining **intents** and associating them with one or more **utterances**.

For example, consider the following list of intents and associated utterances:

- GetTime:
  - "What time is it?"
  - "What is the time?"
  - "Tell me the time"
- GetWeather:
  - "What is the weather forecast?"
  - "Do I need an umbrella?"
  - "Will it snow?"
- TurnOnDevice
  - "Turn the light on."
  - "Switch on the light."
  - "Turn on the fan"
- None:
  - "Hello"
  - "Goodbye"

In your model, you must define the **intents** that you want your model to understand, so spend some time considering the **domain** your model must support and the kinds of **actions or information** that users might request. In addition to the intents that you define, every model includes a `None` intent that you should use to explicitly identify utterances that a user might submit, but for which there is *no specific action required* (for example, conversational greetings like "hello") or that *fall outside of the scope of the domain for this model*.

After you've identified the intents your model must support, it's important to capture various different example **utterances** for each intent. Collect utterances that you think users will enter; including utterances meaning the same thing but that are constructed in different ways. Keep these guidelines in mind:

- Capture multiple different examples, or alternative ways of saying the same thing
- Vary the length of the utterances from short, to medium, to long
- Vary the location of the noun or subject of the utterance. Place it at the beginning, the end, or somewhere in between
- Use correct grammar and incorrect grammar in different utterances to offer good training data examples

- The precision, consistency and completeness of your labeled data are key factors to determining model performance.
    - Label **precisely**: Label each entity to its right type always. Only include what you want extracted, avoid unnecessary data in your labels.
    - Label **consistently**: The same entity should have the same label across all the utterances.
    - Label **completely**: Label all the instances of the entity in all your utterances.

Entities are used to add specific context to intents. For example, you might define a `TurnOnDevice` intent that can be applied to multiple devices, and use entities to define the different devices.

Consider the following utterances, intents, and entities:

Utterance	Intent	Entities
What is the time?	GetTime	
What time is it in *London*?	GetTime	Location (London)
What's the weather forecast for *Paris*?	GetWeather	Location (Paris)
Will I need an umbrella *tonight*?	GetWeather	Time (tonight)
What's the forecast for *Seattle tomorrow*?	GetWeather	Location (Seattle), Time (tomorrow)
Turn the *light* on.	TurnOnDevice	Device (light)
Switch on the *fan*.	TurnOnDevice	Device (fan)

You can **split entities into a few different component types**:

- **Learned** entities are **the most flexible kind of entity, and should be used in most cases**. You define a learned component with a suitable name, and then associate words or phrases with it in training utterances. When you train your model, it learns to match the appropriate elements in the utterances with the entity.
- **List** entities are useful when you need an entity 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.
- **Prebuilt** entities are useful for common types such as **numbers, datetimes, and names**. For example, when prebuilt components are added, you will automatically detect values such as "6" or organizations such as "Microsoft". You can see this article for a list of supported prebuilt entities.

Use patterns to differentiate similar utterances

In some cases, a model might contain **multiple intents** for which utterances are likely to be similar. You can use the pattern of utterances to disambiguate the intents while minimizing the number of sample utterances.

For example, consider the following utterances:

- "Turn on the kitchen light"
- "Is the kitchen light on?"
- "Turn off the kitchen light"

These utterances are syntactically similar, with only a few differences in words or punctuation. However, they represent three different intents (which could be named `TurnOnDevice`, `GetDeviceStatus`, and `TurnOffDevice`). Additionally, the intents could apply to a wide range of entity values. In addition to "kitchen light", the intent could apply to "living room light", "television", or any other device that the model might need to support.

To correctly train your model, provide a handful of examples of each intent that specify the different formats of utterances.

- TurnOnDevice:
  - "Turn on the {DeviceName}"
  - "Switch on the {DeviceName}"
  - "Turn the {DeviceName} on"
- GetDeviceStatus:
  - "Is the {DeviceName} on[?]"
- TurnOffDevice:
  - "Turn the {DeviceName} off"
  - "Switch off the {DeviceName}"
  - "Turn off the {DeviceName}"

When you teach your model with each different type of utterance, the Azure AI Language service can learn how to categorize intents correctly based off format and punctuation.

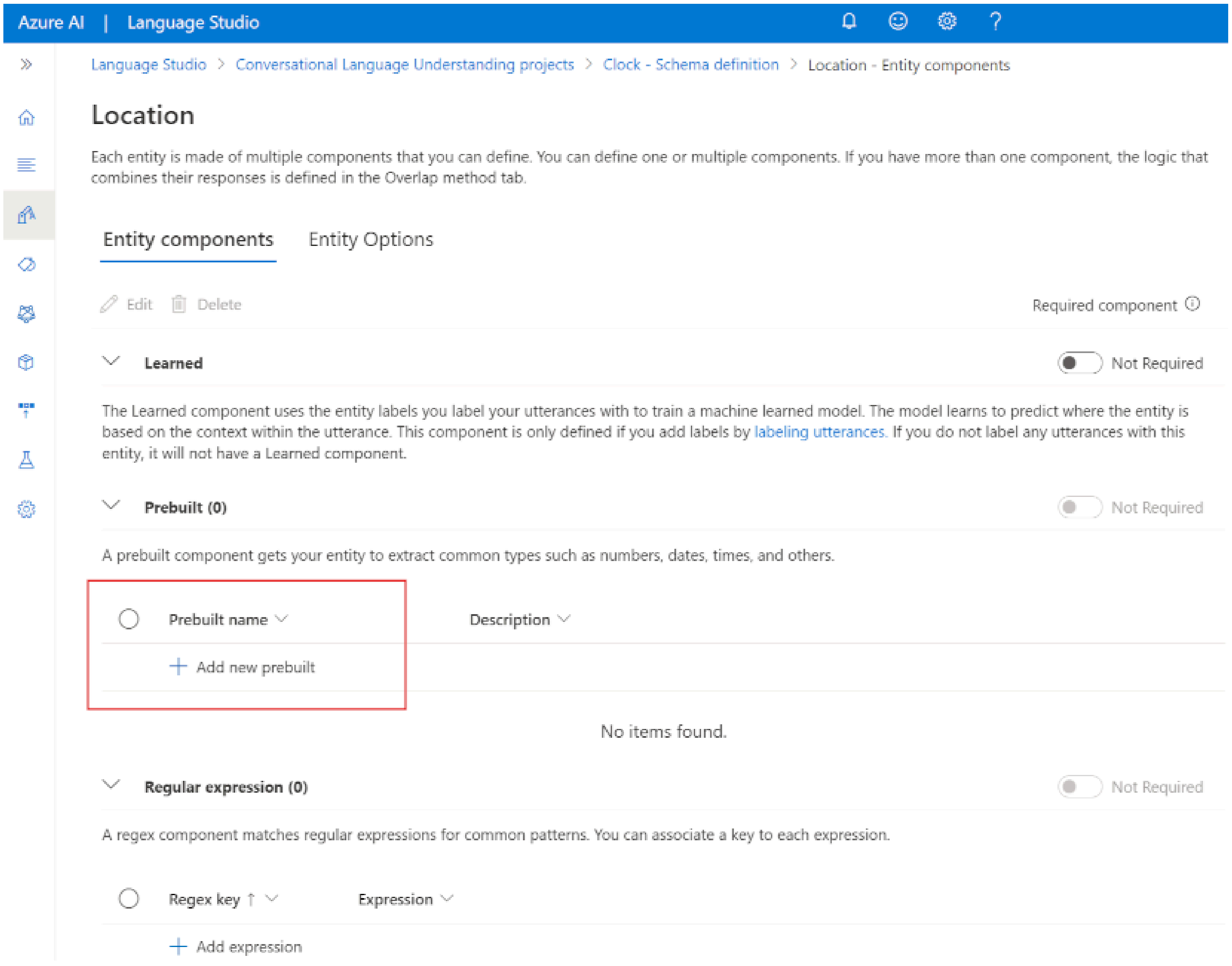
Use pre-built entity components

You can **create your own language models by defining all the intents and utterances it requires**, but often you can use **prebuilt components** to **detect common entities** such as **numbers, emails, URLs, or choices**.

For a full list of prebuilt entities the Azure AI Language service can detect, see the list of supported prebuilt entity components.

Using prebuilt components allows you to let the Azure AI Language service automatically detect the specified type of entity, and not have to train your model with examples of that entity.

To add a prebuilt component, you can create an entity in your project, then select **Add new prebuilt** to that entity to detect certain entities.


Azure AI | Language Studio
Language Studio > Conversational Language Understanding projects > Clock - Schema definition > Location - Entity components
Entity components
Entity Options
Edit
Delete
Required component
Not Required
Prebuilt name
Description
Add new prebuilt
Regex key
Expression
Add expression

Location

Each entity is made of multiple components that you can define. You can define one or multiple components. If you have more than one component, the logic that combines their responses is defined in the Overlap method tab.

Learned

The Learned component uses the entity labels you label your utterances with to train a machine learned model. The model learns to predict where the entity is based on the context within the utterance. This component is only defined if you add labels by labeling utterances. If you do not label any utterances with this entity, it will not have a Learned component.

Prebuilt (0)

A prebuilt component gets your entity to extract common types such as numbers, dates, times, and others.

No items found.

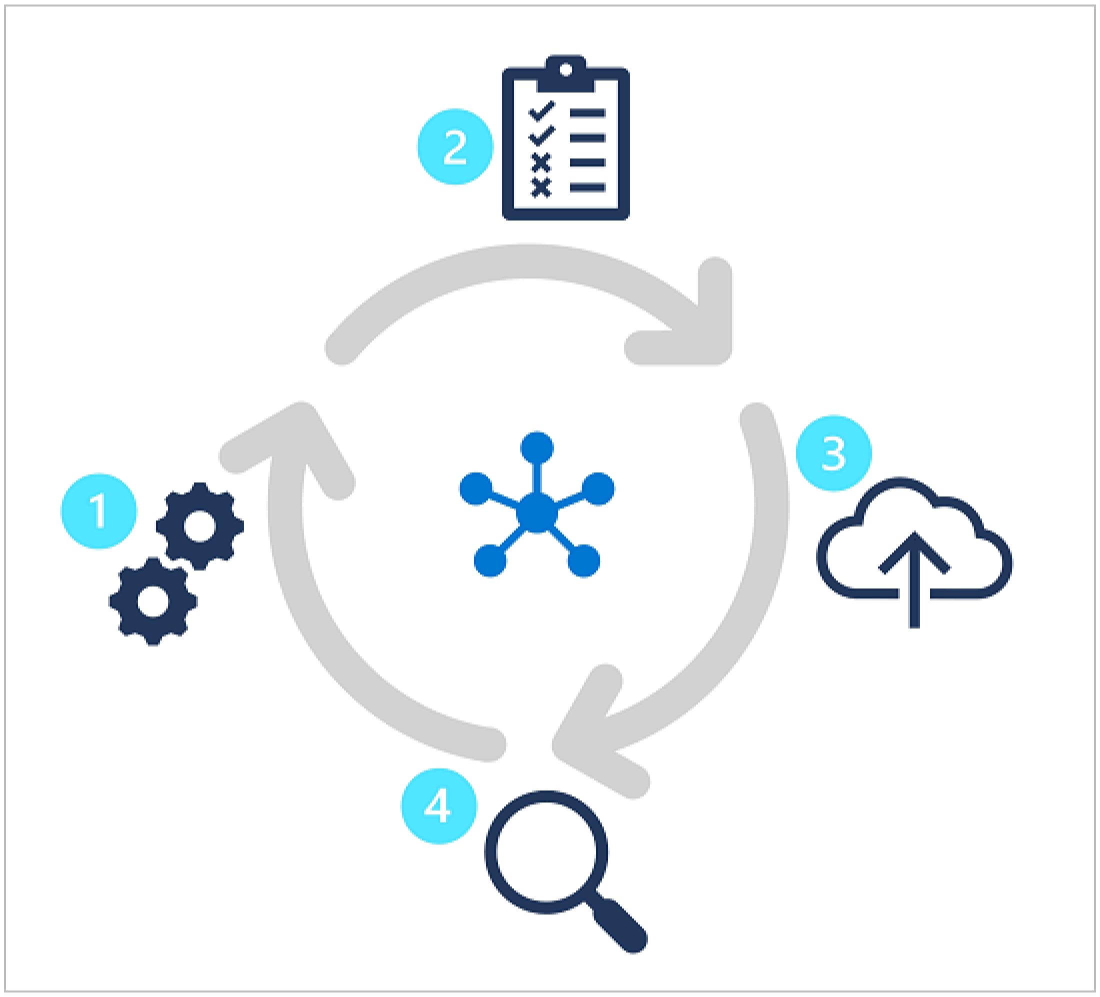
Regular expression (0)

A regex component matches regular expressions for common patterns. You can associate a key to each expression.

You can have **up to five prebuilt components per entity**. Using prebuilt model elements can significantly reduce the time it takes to develop a conversational language understanding solution.

Train, test, publish, and review a conversational language understanding model

Creating a model is an iterative process with the following activities:



- **Train a model** to **learn intents and entities from sample utterances**.
- **Test the model interactively** or using a testing dataset with known labels
- **Deploy a trained model** to *a public endpoint* so client apps can use it
- **Review predictions and iterate on utterances** to train your model

By following this iterative approach, you can improve the language model over time based on user input, helping you develop solutions that reflect the way users indicate their intents using natural language.

Exercise - Build an Azure AI services conversational language understanding model

In this exercise, you use Azure AI Language to build a conversational language understanding model.

Create a language understanding model with the Language service

The Azure AI Language service enables you to define a **conversational language understanding model** that applications can use to interpret natural language utterances from users (text or spoken input), predict the users intent (what they want to achieve), and identify any entities to which the intent should be applied.

For example, a conversational language model for a clock application might be expected to process input such as:

What is the time in London?

This kind of input is an example of an **utterance** (something a user might say or type), for which the desired intent is to get the time in a specific location (an entity); in this case, *London*.

> NOTE: The task of a conversational language model is to predict the user's intent and identify any entities to which the intent applies. It is not the job of a conversational language model to actually perform the actions required to satisfy the intent. For example, a clock application can use a conversational language model to discern that the user wants to know the time in London; but the client application itself must then implement the logic to determine the correct time and present it to the user.

In this exercise, you'll use the Azure AI Language service to create a conversational language understand model, and use the Python SDK to implement a client app that uses it.

While this exercise is based on Python, you can develop conversational understanding applications using multiple language-specific SDKs.

Module assessment

1. Your app must interpret a command such as "turn on the light" or "switch the light on". What do these phrases represent in a language model? **Utterances**
2. Your app must interpret a command to book a flight to a specified city, such as "Book a flight to Paris." How should you model the city element of the command? **As an entity**
3. Your language model needs to detect an email when present in an utterance. What is the simplest way to extract that email? **Use prebuilt entity components**

Summary

In this module, you learned how to create a conversational language understanding model.

Now that you've completed this module, you can:

- Provision an Azure AI Language resource
- Define **intents, entities, and utterances**
- Use patterns to differentiate similar utterances
- Use pre-built entity components
- Train, test, publish, and review a model

To learn more about language understanding, see the Azure AI Language documentation.

Create a custom text classification solution

The Azure AI Language service enables processing of natural language to use in your own app. Learn how to build a **custom text classification** project.

Learning objectives

After completing this module, you'll be able to:

- Understand **types of classification projects**
- Build a custom text classification project
- Tag data, train, and deploy a model
- Submit classification tasks from your own app

Introduction

Natural language processing (NLP) is one of the most common AI problems, where software must **interpret text or speech in the natural form that humans use**. Part of NLP is the ability to classify text, and Azure provides ways to **classify text including sentiment, language, and custom categories defined by the user**.

In this module, you'll learn how to use the Azure AI Language service to classify text into custom groups.

Understand types of classification projects

Custom text classification assigns labels, which in the Azure AI Language service is a class that the developer defines, **to text files**. For example, a video game summary might be classified as "Adventure", "Strategy", "Action" or "Sports".

Custom text classification falls into two types of projects:

- **Single label classification** - you can **assign only one class to each file**. Following the above example, a video game summary could only be classified as "Adventure" or "Strategy".
- **Multiple label classification** - you can **assign multiple classes to each file**. This type of project would allow you to classify a video game summary as "Adventure" or "Adventure and Strategy".

When creating your custom text classification project, you can specify which project you want to build.

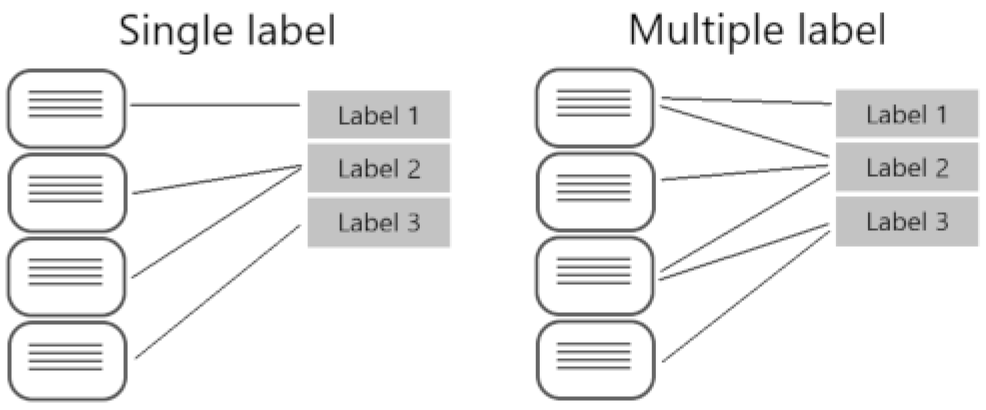
Single vs. multiple label projects

Beyond the ability to put files into multiple classifications, **the key differences with multiple label classification projects are 1) labeling, 2) considerations for improving your model, and 3) the API payload for classification tasks**.

Labeling data

In single label projects, each file is assigned one class during the labeling process; class assignment in Azure AI Language only allows you to select one class.

When labeling multiple label projects, you can assign as many classes that you want per file. The impact of the added complexity means your data has to remain clear and provide a good distribution of possible inputs for your model to learn from.



Labeling data correctly, especially for multiple label projects, is directly correlated with how well your model performs. The higher the quality, clarity, and variation of your data set is, the more accurate your model will be.

Evaluating and improving your model

Measuring predictive performance of your model goes beyond how many predictions were correct. **Correct classifications are when the actual label is x and the model predicts a label x**. In the real world, documents result in different kinds of errors when a classification isn't correct:

- **False positive** - model predicts x, but the file isn't labeled x.
- **False negative** - model doesn't predict label x, but the file in fact is labeled x.

These metrics are translated into three measures provided by Azure AI Language:

- **Recall** - Of all the actual labels, how many were identified; **the ratio of true positives to all that was labeled**.
- **Precision** - How many of the predicted labels are correct; **the ratio of true positives to all identified positives**.
- **F1 Score** - A function of recall and precision, intended to provide a single score to maximize for a balance of each component

> Tip: Learn more about the Azure AI Language evaluation metrics, including exactly how these metrics are calculated.

With a single label project, you can identify which classes aren't classified as well as others and find more quality data to use in training your model. For multiple label projects, figuring out quality data becomes more complex due to the matrix of possible permutations of combined labels.

For example, let's your model is correctly classifying "Action" games and some "Action and Strategy" games, but failing at "Strategy" games. To improve your model, you'll want to find more high quality and varied summaries for both "Action and Strategy" games, as well as "Strategy" games to teach your model how to differentiate the two. This challenge increases exponentially with more possible classes your model is classifying into.

API payload

Azure AI Language provides a REST API to build and interact with your model, **using a JSON body to specify the request**. This API is abstracted into multiple language-specific SDKs, however for this module we'll focus our examples on the base REST API.

To submit a classification task, the API requires the JSON body to specify which task to execute. You'll learn more about the REST API in the next unit, but worth familiarizing yourself with parts of the required body.

Single label classification models specify a project type of `customSingleLabelClassification`

```json
{
  "projectFileVersion": "<API-VERSION>",
  "stringIndexType": "Utf16CodeUnit",
  "metadata": {
    "projectName": "<PROJECT-NAME>",
    "storageInputContainerName": "<CONTAINER-NAME>",
    "projectKind": "customSingleLabelClassification",
    "description": "Trying out custom single label text classification",
    "language": "<LANGUAGE-CODE>",
    "multilingual": true,
    "settings": {}
  },
  "assets": {
    "projectKind": "customSingleLabelClassification",
        "classes": [
            {
                "category": "Class1"
            },
            {
                "category": "Class2"
            }
        ],
        "documents": [
            {
                "location": "<DOCUMENT-NAME>",
                "language": "<LANGUAGE-CODE>",
                "dataset": "<DATASET>",
                "class": {
                    "category": "Class2"
                }
            },
            {
                "location": "<DOCUMENT-NAME>",
                "language": "<LANGUAGE-CODE>",
                "dataset": "<DATASET>",
                "class": {
                    "category": "Class1"
                }
            }
        ]
    }
}
```

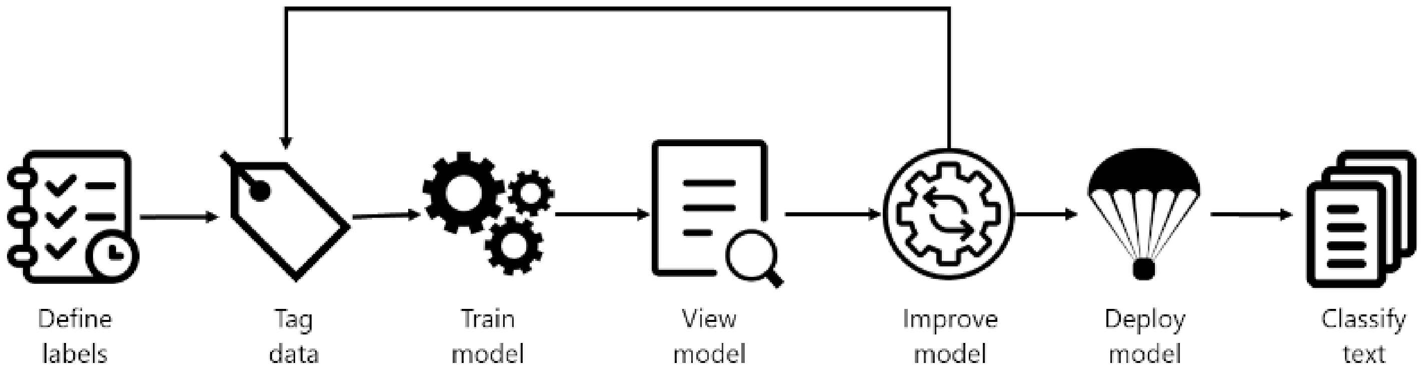
Multiple label classification models specify a project type of `CustomMultiLabelClassification`

```json
{
  "projectFileVersion": "<API-VERSION>",
  "stringIndexType": "Utf16CodeUnit",
  "metadata": {
    "projectName": "<PROJECT-NAME>",
    "storageInputContainerName": "<CONTAINER-NAME>",
    "projectKind": "customMultiLabelClassification",
    "description": "Trying out custom multi label text classification",
    "language": "<LANGUAGE-CODE>",
    "multilingual": true,
    "settings": {}
  },
  "assets": {
    "projectKind": "customMultiLabelClassification",
    "classes": [
      {
        "category": "Class1"
      },
      {
        "category": "Class2"
      }
    ],
    "documents": [
      {
        "location": "<DOCUMENT-NAME>",
        "language": "<LANGUAGE-CODE>",
        "dataset": "<DATASET>",
        "classes": [
          {
            "category": "Class1"
          },
          {
            "category": "Class2"
          }
        ]
      },
      {
        "location": "<DOCUMENT-NAME>",
        "language": "<LANGUAGE-CODE>",
        "dataset": "<DATASET>",
        "classes": [
          {
            "category": "Class2"
          }
        ]
      }
    ]
  }
}
```

Understand how to build text classification projects

Custom text classification projects are your workspace to build, train, improve, and deploy your classification model. **You can work with your project in two ways**: **through Language Studio and via the REST API**. Language Studio is the GUI that will be used in the lab, but the REST API has the same functionality. Regardless of which method you prefer, the steps for developing your model are the same.

Azure AI Language project life cycle



- **Define labels**: Understanding the data you want to classify, identify the possible labels you want to categorize into. In our video game example, our labels would be "Action", "Adventure", "Strategy", and so on.
- **Tag data**: Tag, or label, your existing data, specifying the label or labels each file falls under. Labeling data is important since it's how your model will learn how to classify future files. Best practice is to have clear differences between labels to avoid ambiguity, and provide good examples of each label for the model to learn from. For example, we'd label the game "Quest for the Mine Brush" as "Adventure", and "Flight Trainer" as "Action".

- **Train model**: Train your model with the labeled data. Training will teach our model what types of video game summaries should be labeled which genre.
- **View model**: After your model is trained, view the results of the model. **Your model is scored between 0 and 1, based on the precision and recall of the data tested**. Take note of which genre didn't perform well.
- **Improve model**: Improve your model by seeing which classifications failed to evaluate to the right label, see your label distribution, and find out what data to add to improve performance. For example, you might find your model mixes up "Adventure" and "Strategy" games. Try to find more examples of each label to add to your dataset for retraining your model.
- **Deploy model**: Once your model performs as desired, deploy your model to make it available via the API. Your model might be named "GameGenres", and once deployed can be used to classify game summaries.
- **Classify text**: Use your model for classifying text. The lab covers how to use the API, and you can view the API reference

How to split datasets for training

When labeling your data, you can specify which dataset you want each file to be:

- **Training** - The training dataset is used to actually train the model; the data and labels provided are fed into the machine learning algorithm to teach your model what data should be classified to which label. The training dataset will be the larger of the two datasets, recommended to be about **80% of your labeled data**.
- **Testing** - The testing dataset is labeled data used to verify you model after it's trained. Azure will take the data in the testing dataset, submit it to the model, and compare the output to how you labeled your data to determine how well the model performed. The result of that comparison is how your model gets scored and helps you know how to improve your predictive performance.

During the **Train model** step, there are two options for how to train your model.

- **Automatic split** - Azure takes all of your data, splits it into the specified percentages randomly, and applies them in training the model. **This option is best when you have a larger dataset, data is naturally more consistent, or the distribution of your data extensively covers your classes**.
- **Manual split** - Manually specify which files should be in each dataset. When you submit the training job, the Azure AI Language service will tell you the split of the dataset and the distribution. **This split is best used with smaller datasets to ensure the correct distribution of classes and variation in data are present to correctly train your model**.

To use the automatic split, put all files into the training dataset when labeling your data (this option is the default). To use the manual split, specify which files should be in testing versus training during the labeling of your data.

Deployment options

Azure AI Language allows each project to create both **multiple models** and **multiple deployments**, each with their own unique name. Benefits include ability to:

- Test two models side by side
- Compare how the split of datasets impact performance
- Deploy multiple versions of your model

> Note: Each project has a limit of ten deployment names

During deployment you can choose the name for the deployed model, which can then be selected when submitting a classification task:

```
<...>
  "tasks": [
    {
      "kind": "CustomSingleLabelClassification",
      "taskName": "MyTaskName",
      "parameters": {
        "projectName": "MyProject",
        "deploymentName": "MyDeployment"
      }
    }
  ]
<...>
```

Using the REST API

The REST API available for the Azure AI Language service allows for CLI development of Azure AI Language projects in the same way that Language Studio provides a user interface for building projects. Language Studio is explored further in this module's lab.

Pattern of using the API

The API for the Azure AI Language service operates **asynchronously** for most calls. In each step we **submit a request to the service first**, then **check back with the service via a subsequent call to**

get the status or result.

With each request, a header is required to authenticate your request:

Key	Value
`Ocp-Apim-Subscription-Key`	The key to your Azure AI Language resource

Submit initial request

The URL to submit the request to varies on which step you are on, but all are prefixed with the `endpoint` provided by your Azure AI Language resource.

For example, to train a model, you would create a POST to the URL that would look something like the following:

```
<YOUR-ENDPOINT>/language/analyze-text/projects/<PROJECT-NAME>/:train?api-version=<API-VERSION>
```

Placeholder	Value	Example
`YOUR-ENDPOINT`	The endpoint for your API request	`https://<your-custom-resource>.cognitiveservices.azure.com`
`PROJECT-NAME`	The name for your project (value is case-sensitive)	`myProject`

The following body would be attached to the request:

```json
{
    "modelLabel": "<MODEL-NAME>",
    "trainingConfigVersion": "<CONFIG-VERSION>",
    "evaluationOptions": {
        "kind": "percentage",
        "trainingSplitPercentage": 80,
        "testingSplitPercentage": 20
    }
}
```

Key	Value
`YOUR-MODEL`	Your model name.
`trainingConfigVersion`	The model version to use to train your model.
`runValidation`	Boolean value to run validation on the test set.
`evaluationOptions`	Specifies evaluation options.
`kind`	Specifies data split type. Can be `percentage` if you're using an automatic split, or `set` if you manually split your dataset
`testingSplitPercentage`	Required integer field only if `type` is `percentage` . Specifies testing split.
`trainingSplitPercentage`	Required integer field only if `type` is `percentage` . Specifies training split.

The response to the above request will be a `202` , meaning the request was successful. Grab the location value from the response headers, which will look similar to the following URL:

```
<ENDPOINT>/language/analyze-text/projects/<PROJECT-NAME>/train/jobs/<JOB-ID>?api-version=<API-VE
```

Key	Value
`JOB-ID`	Identifier for your request

This URL is used in the next step to get the training status.

Get training status

To get the training status, use the URL from the header of the request response to submit a `GET` request, with same header that provides our Azure AI Language service key for authentication. The response body will be similar to the following JSON:

```
{
  "result": {
    "modelLabel": "<MODEL-NAME>",
    "trainingConfigVersion": "<CONFIG-VERSION>",
    "estimatedEndDateTime": "2023-05-18T15:47:58.8190649Z",
    "trainingStatus": {
      "percentComplete": 3,
      "startDateTime": "2023-05-18T15:45:06.8190649Z",
      "status": "running"
    },
    "evaluationStatus": {
      "percentComplete": 0,
      "status": "notStarted"
    }
  },
  "jobId": "<JOB-ID>",
  "createdDateTime": "2023-05-18T15:44:44Z",
  "lastUpdatedDateTime": "2023-05-18T15:45:48Z",
  "expirationDateTime": "2023-05-25T15:44:44Z",
  "status": "running"
}
```

Training a model can take some time, so periodically check back at this status URL until the response `status` returns `succeeded`. Once the training has succeeded, you can **view, verify, and deploy your model**.

Consuming a deployed model

Using the model to classify text follows the same pattern as outlined above, with a `POST` request submitting the job and a `GET` request to retrieve the results.

Submit text for classification

To use your model, submit a `POST` to the analyze endpoint at the following URL:

```
<ENDPOINT>/language/analyze-text/jobs?api-version=<API-VERSION>
```

Placeholder	Value	Example
`YOUR-ENDPOINT`	The endpoint for your API request	`https://<your-custom-resource>.cognitiveservices.azure.com`

> Important: Remember to include your resource key in the header for `Ocp-Apim-Subscription-Key`

The following JSON structure would be attached to the request:

```
{
  "displayName": "Classifying documents",
  "analysisInput": {
    "documents": [
      {
        "id": "1",
        "language": "<LANGUAGE-CODE>",
        "text": "Text1"
      },
      {
        "id": "2",
        "language": "<LANGUAGE-CODE>",
        "text": "Text2"
      }
    ]
  },
  "tasks": [
     {
      "kind": "<TASK-REQUIRED>",
      "taskName": "<TASK-NAME>",
      "parameters": {
        "projectName": "<PROJECT-NAME>",
        "deploymentName": "<DEPLOYMENT-NAME>"
      }
    }
  ]
}
```

Key	Value
`TASK-REQUIRE` D	Which task you're requesting. The task is `CustomMultiLabelClassification` for multiple label projects, or `CustomSingleLabelClassification` for single

Key	Value
	label projects
`LANGUAGE-CODE`	The language code such as `en-us`.
`TASK-NAME`	Your task name.
`PROJECT-NAME`	Your project name.
`DEPLOYMENT-NAME`	Your deployment name.

The response to the above request will be a `202`, meaning the request was successful. Look for the `operation-location` value in the response headers, which will look something like the following URL:

```
<ENDPOINT>/language/analyze-text/jobs/<JOB-ID>?api-version=<API-VERSION>
```

Key	Value
`YOUR-ENDPOINT`	The endpoint for your API request
`JOB-ID`	Identifier for your request

This URL is used to get your task results.

Get classification results

Submit a `GET` request to the endpoint from the previous request, with the same header for authentication. The response body will be similar to the following JSON:

```json
{
  "createdDateTime": "2023-05-19T14:32:25.578Z",
  "displayName": "MyJobName",
  "expirationDateTime": "2023-05-19T14:32:25.578Z",
  "jobId": "xxxx-xxxxxx-xxxxx-xxxx",
  "lastUpdateDateTime": "2023-05-19T14:32:25.578Z",
  "status": "succeeded",
  "tasks": {
    "completed": 1,
    "failed": 0,
    "inProgress": 0,
    "total": 1,
    "items": [
      {
        "kind": "customSingleClassificationTasks",
        "taskName": "Classify documents",
        "lastUpdateDateTime": "2022-10-01T15:01:03Z",
        "status": "succeeded",
        "results": {
          "documents": [
            {
              "id": "<DOC-ID>",
              "class": [
                  {
                      "category": "Class_1",
                      "confidenceScore": 0.0551877357
                  }
              ],
              "warnings": []
            }
          ],
          "errors": [],
          "modelVersion": "2022-04-01"
        }
      }
    ]
  }
}
```

The classification result is within the items array's `results` object, for each document submitted.

Exercise - Classify text

In this exercise, you use Azure AI Language to build a custom text classification model.

Custom text classification

Azure AI Language provides several NLP capabilities, including the key phrase identification, text summarization, and sentiment analysis. The Language service also provides custom features like **custom question answering and custom text classification**.

To test the custom text classification of the Azure AI Language service, you'll configure the model using Language Studio then use a Python application to test it.

While this exercise is based on Python, you can develop text classification applications using multiple language-specific SDKs; including:

- Azure AI Text Analytics client library for Python
- Azure AI Text Analytics client library for .NET
- Azure AI Text Analytics client library for JavaScript

Module assessment

1. You want to train a model to classify book summaries by their genre, and some of your favorite books are both *mystery* and *thriller*. Which type of project should you build? **A multiple label classification project**.
2. You just got notification your training job is complete. What is your next step? **View your model details**.
3. You want to submit a classification task via the API. How do you get the results of the classification? **Call the URL provided in the header of the request response**.

Summary

In this module, you learned about custom text classification and how to build a text classification service.

Now that you've completed this module, you can:

- Understand types of classification projects.
- Build a custom text classification project.
- Tag data, train, and deploy a model.
- Submit classification tasks from your own app.

To learn more about the Azure AI Language service, see the Azure AI Language service documentation.

Custom named entity recognition

Build a custom entity recognition solution to **extract entities from unstructured documents**.

Learning objectives

After completing this module, you'll be able to:

- Understand tagging entities in extraction projects
- Understand how to **build entity recognition projects**

Introduction

Custom **named entity recognition (NER)**, otherwise known as **custom entity extraction**, is one of the many features for natural language processing (NLP) offered by Azure AI Language service. Custom NER enables developers to **extract predefined entities from text documents**, without those documents being in a known format - such as legal agreements or online ads.

An entity is a **person, place, thing, event, skill, or value**.

In this module, you'll learn how to use the **Azure AI Language service** to extract entities from unstructured documents.

After completing this module, you'll be able to:

- Understand custom named entities and how they're labeled.
- Build **a custom named entity extraction project**.
- Label data, train, and deploy an **entity extraction model**.
- Submit extraction tasks from your own app.

Understand custom named entity recognition

Custom NER is an Azure API service that looks at documents, *identifies, and extracts user defined entities*. These entities could be anything from *names and addresses from bank statements to knowledge mining to improve search results*.

Custom NER is part of Azure AI Language in Azure AI services.

Custom vs built-in NER

Azure AI Language provides certain built-in entity recognition, to recognize things such as a person, location, organization, or URL. Built-in NER allows you to set up the service with minimal configuration, and extract entities. **To call a built-in NER, create your service and call the endpoint for that NER service** like this:

```
<YOUR-ENDPOINT>/language/analyze-text/jobs?api-version=<API-VERSION>
```

Placeholder	Value	Example
`YOUR-ENDPOINT`	The endpoint for your API request	https://.cognitiveservices.azure.com
`API-VERSION`	The version of the API you are calling	2023-05-01

The body of that call will contain the document(s) the entities are extracted from, and **the headers** contain your service **key**.

The response from the call above contains an array of entities recognized, such as:

```
<...>
"entities":[
    {
        "text":"Seattle",
        "category":"Location",
        "subcategory":"GPE",
        "offset":45,
        "length":7,
        "confidenceScore":0.99
    },
    {
        "text":"next week",
        "category":"DateTime",
        "subcategory":"DateRange",
        "offset":104,
        "length":9,
        "confidenceScore":0.8
    }
]
<...>
```

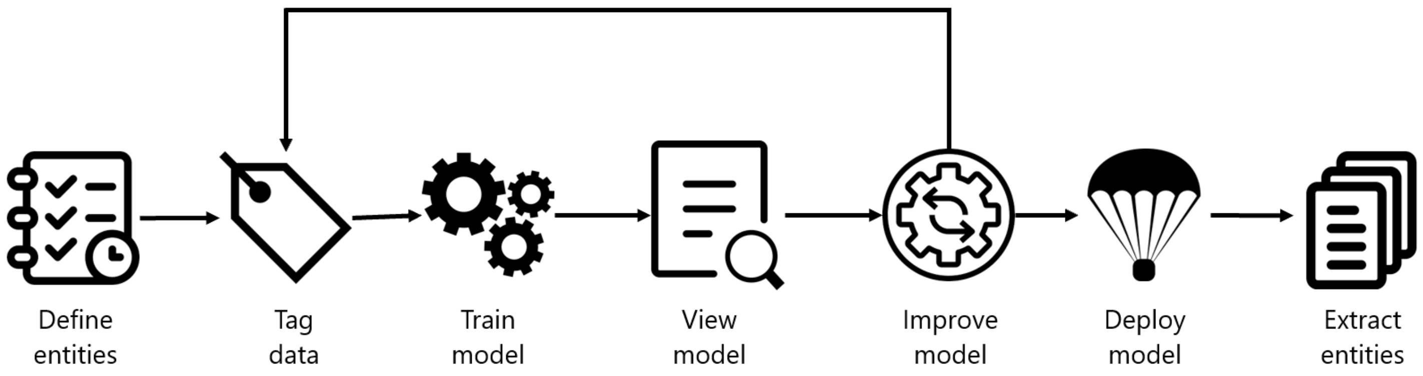
Examples of when to use the **built-in NER** include finding **locations, names, or URLs in long text documents**.

> Tip: A full list of recognized entity categories is available in the NER docs.

Custom NER, which is the focus of the rest of this module, is available when the entities you want to extract aren't part of the built-in service or you only want to extract specific entities. You can make your custom NER model as simple or complex as is required for your app.

Examples of when you'd want custom NER include *specific legal or bank data, knowledge mining to enhance catalog search, or looking for specific text for audit policies*. Each one of these projects requires a specific set of entities and data it needs to extract.

Azure AI Language project life cycle



Creating an entity extraction model typically follows a similar path to most Azure AI Language service features:

- **Define entities**: Understanding the data and entities you want to identify, and try to make them as clear as possible. For example, defining exactly which parts of a bank statement you want to extract.
- **Tag data**: *Label, or tag, your existing data, specifying what text in your dataset corresponds to which entity*. This step is important to do accurately and completely, as any wrong or missed labels will reduce the effectiveness of the trained model. A good variation of possible input documents is useful. For example, label bank name, customer name, customer address, specific loan or account terms, loan or account amount, and account number.
- **Train model**: Train your model once your entities are labeled. Training teaches your model how to recognize the entities you label.
- **View model**: *After your model is trained, view the results of the model*. This page includes **a score of 0 to 1 that is based on the precision and recall of the data tested**. You can see which entities worked well (such as customer name) and which entities need improvement (such as account number).
- **Improve model**: Improve your model by seeing which entities failed to be identified, and which entities were incorrectly extracted. Find out what data needs to be added to your model's training to improve performance. This page shows you how entities failed, and which entities (such as account number) need to be differentiated from other similar entities (such as loan amount).
- **Deploy model**: Once your model performs as desired, deploy your model to make it available via the API. In our example, you can send to requests to the model when it's deployed to extract bank statement entities.
- **Extract entities**: Use your model for extracting entities. The lab covers how to use the API, and you can view the API reference for more details.

Considerations for data selection and refining entities

For the best performance, you'll need to use both **high quality data** to train the model and **clearly defined entity types**.

High quality data will let you spend less time refining and yield better results from your model.

- **Diversity** - use as diverse of a dataset as possible without losing the real-life distribution expected in the real data. You'll want to **use sample data from as many sources as possible**, each with their own formats and number of entities. It's best to have your dataset represent as many different sources as possible.
- **Distribution** - use the appropriate distribution of document types. A more diverse dataset to train your model will help your model avoid learning incorrect relationships in the data.
- **Accuracy** - **use data that is as close to real world data as possible**. Fake data works to start the training process, but it likely will differ from real data in ways that can cause your model to not extract correctly.

Entities *need to also be carefully considered, and defined as distinctly as possible*. Avoid ambiguous entities (such as two names next to each other on a bank statement), as it will make the model struggle to differentiate. If having some ambiguous entities is required, make sure to have more examples for your model to learn from so it can understand the difference.

Keeping your entities distinct will also go a long way in helping your model's performance. For example, trying to extract something like "Contact info" that could be a phone number, social media handle, or email address would require several examples to correctly teach your model. Instead, **try to break them down into more specific entities** such as "Phone", "Email", and "Social media" and let the model label whichever type of contact information it finds.

How to extract entities

To submit an extraction task, the API requires the JSON body to specify which task to execute. For custom NER, the task for the JSON payload is `CustomEntityRecognition`.

Your payload will look similar to the following JSON:

```
{
    "displayName": "string",
    "analysisInput": {
        "documents": [
            {
                "id": "doc1",
                "text": "string"
            },
            {
                "id": "doc2",
                "text": "string"
            }
        ]
    },
    "tasks": [
        {
            "kind": "CustomEntityRecognition",
            "taskName": "MyRecognitionTaskName",
            "parameters": {
            "projectName": "MyProject",
            "deploymentName": "MyDeployment"
            }
        }
    ]
}
```

Project limits

The Azure AI Language service enforces the following restrictions:

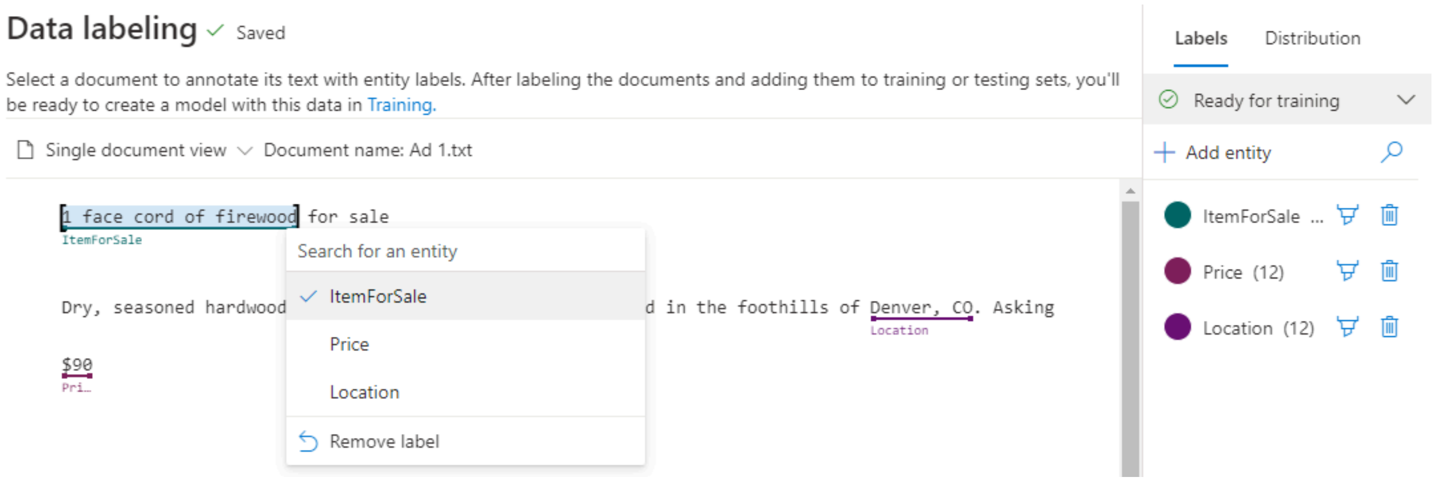
- **Training** - at least 10 files, and not more than 100,000
- **Deployments** - 10 deployment names per project
- **APIs**
  - **Authoring** - this API creates a project, trains, and deploys your model. Limited to 10 POST and 100 GET per minute
  - **Analyze** - this API does the work of actually extracting the entities; it requests a task and retrieves the results. Limited to 20 GET or POST
- **Projects** - only 1 storage account per project, 500 projects per resource, and 50 trained models per project
- **Entities** - each entity can be up to 500 characters. You can have up to 200 entity types.

See the Service limits for Azure AI Language page for detailed information.

Label your data

Labeling, or tagging, your data correctly is an important part of the process to create a custom entity extraction model. Labels identify examples of specific entities in text used to train the model. Three things to focus on are:

- **Consistency** - *Label your data the same way across all files for training*. Consistency allows your model to learn without any conflicting inputs.
- **Precision** - *Label your entities consistently, without unnecessary extra words*. Precision ensures only the correct data is included in your extracted entity.
- **Completeness** - *Label your data completely, and don't miss any entities*. Completeness helps your model always recognize the entities present.


Data labeling ✓ Saved

Select a document to annotate its text with entity labels. After labeling the documents and adding them to training or testing sets, you'll be ready to create a model with this data in Training.

Single document view · Document name: Ad 1.txt

```
1 face cord of firewood for sale
ItemForSale

Dry, seasoned hardwood ... d in the foothills of Denver, CO. Asking
                                                  Location
$90
Pri…
```

How to label your data

Language Studio is the most straight forward method for labeling your data. Language Studio allows you to *see the file, select the beginning and end of your entity, and specify which entity it is*.

Each label that you identify gets saved into a file that lives in your storage account with your dataset, in an auto-generated JSON file. This file then gets used by the model to learn how to extract custom entities. It's possible to provide this file when creating your project (if you're importing the same labels from a different project, for example) however it must be in the Accepted custom NER data formats. For example:

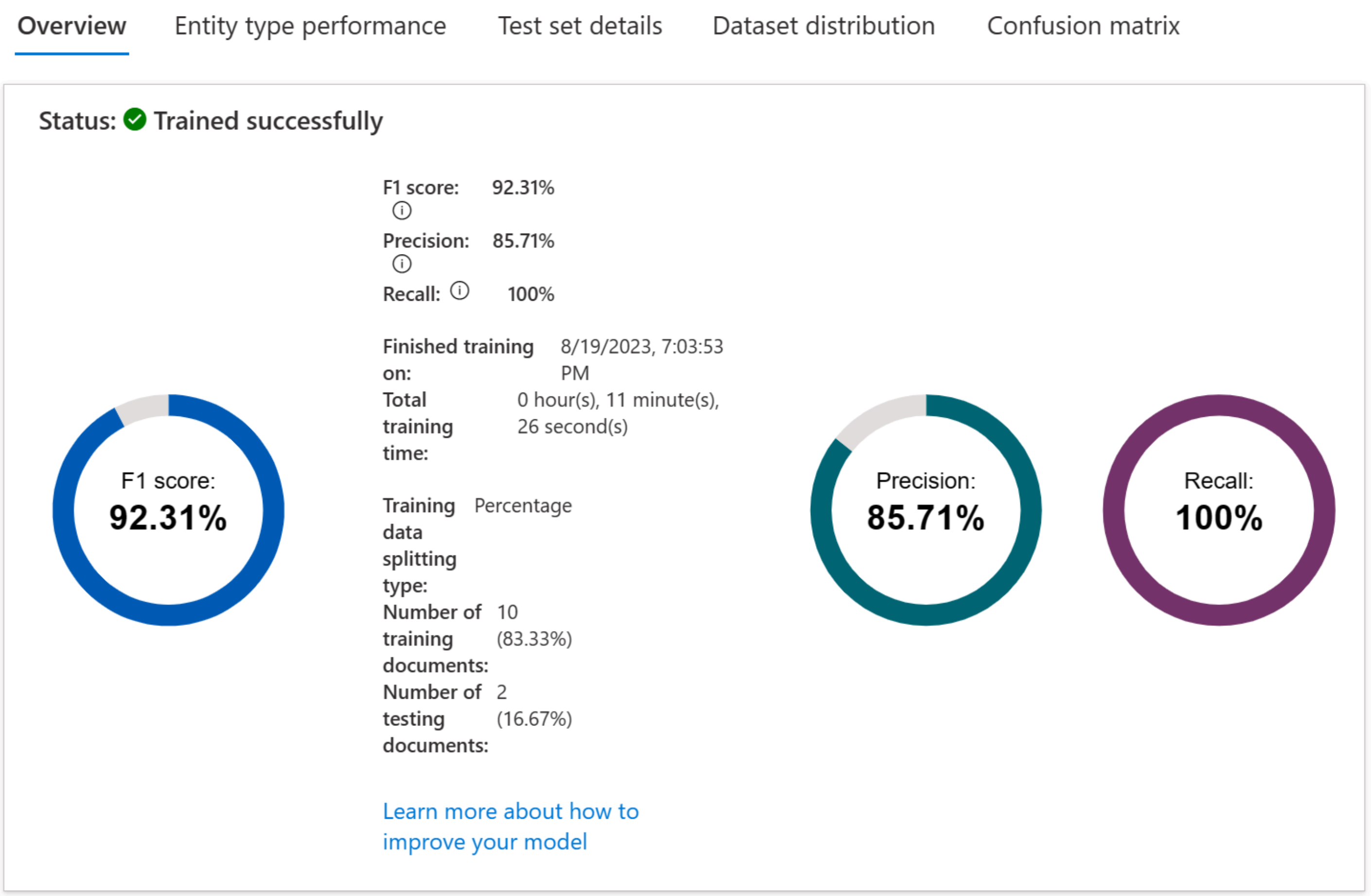
```json
{
  "projectFileVersion": "{DATE}",
  "stringIndexType": "Utf16CodeUnit",
  "metadata": {
    "projectKind": "CustomEntityRecognition",
    "storageInputContainerName": "{CONTAINER-NAME}",
    "projectName": "{PROJECT-NAME}",
    "multilingual": false,
    "description": "Project-description",
    "language": "en-us",
    "settings": {}
  },
  "assets": {
    "projectKind": "CustomEntityRecognition",
    "entities": [
      {
        "category": "Entity1"
      },
      {
        "category": "Entity2"
      }
    ],
    "documents": [
      {
        "location": "{DOCUMENT-NAME}",
        "language": "{LANGUAGE-CODE}",
        "dataset": "{DATASET}",
        "entities": [
          {
            "regionOffset": 0,
            "regionLength": 500,
            "labels": [
              {
                "category": "Entity1",
                "offset": 25,
                "length": 10
              },
              {
                "category": "Entity2",
                "offset": 120,
                "length": 8
              }
            ]
          }
        ]
      },
      {
        "location": "{DOCUMENT-NAME}",
        "language": "{LANGUAGE-CODE}",
        "dataset": "{DATASET}",
        "entities": [
          {
            "regionOffset": 0,
            "regionLength": 100,
            "labels": [
              {
                "category": "Entity2",
                "offset": 20,
                "length": 5
              }
            ]
          }
        ]
      }
    ]
  }
}
```

Field	Description
`documents`	Array of labeled documents
`location`	Path to file within container connected to the project
`language`	Language of the file
`entities`	Array of present entities in the current document
`regionOffset`	Inclusive character position for start of text
`regionLength`	Length in characters of the data used in training
`category`	Name of entity to extract
`labels`	Array of labeled entities in the files
`offset`	Inclusive character position for start of entity

Field	Description
`length`	Length in characters of the entity
`dataset`	Which dataset the file is assigned to

Train and evaluate your model

Training and evaluating your model is *an iterative process of adding data and labels to your training dataset to teach the model more accurately*. To know what types of data and labels need to be improved, Language Studio provides **scoring** in the View model details page on the left hand pane.



Status: Trained successfully

F1 score:	92.31%
Precision:	85.71%
Recall:	100%
Finished training on:	8/19/2023, 7:03:53 PM
Total training time:	0 hour(s), 11 minute(s), 26 second(s)
Training data splitting type:	Percentage
Number of training documents:	10 (83.33%)
Number of testing documents:	2 (16.67%)

F1 score: 92.31%

Precision: 85.71%

Recall: 100%

Individual entities and your overall model score are broken down into three metrics to explain how they're performing and where they need to improve.

Metric	Description
Precision	**The ratio of successful entity recognitions to all attempted recognitions**. A high score means that as long as the entity is recognized, it's labeled correctly. Precision = TP / (TP + FP)
Recall	**The ratio of successful entity recognitions to the actual number of entities in the document**. A high score means it finds the entity or entities well, regardless of if it assigns them the right label. Recall = TP / (TP + FN)
F1 score	**Combination of precision and recall** providing a single scoring metric. F1 score = 2 x P x R / (P + R)

Scores are available both per entity and for the model as a whole. You may find an entity scores well, but the whole model doesn't.

How to interpret metrics

Ideally we want our model to score well in both precision and recall, which means the entity recognition works well.

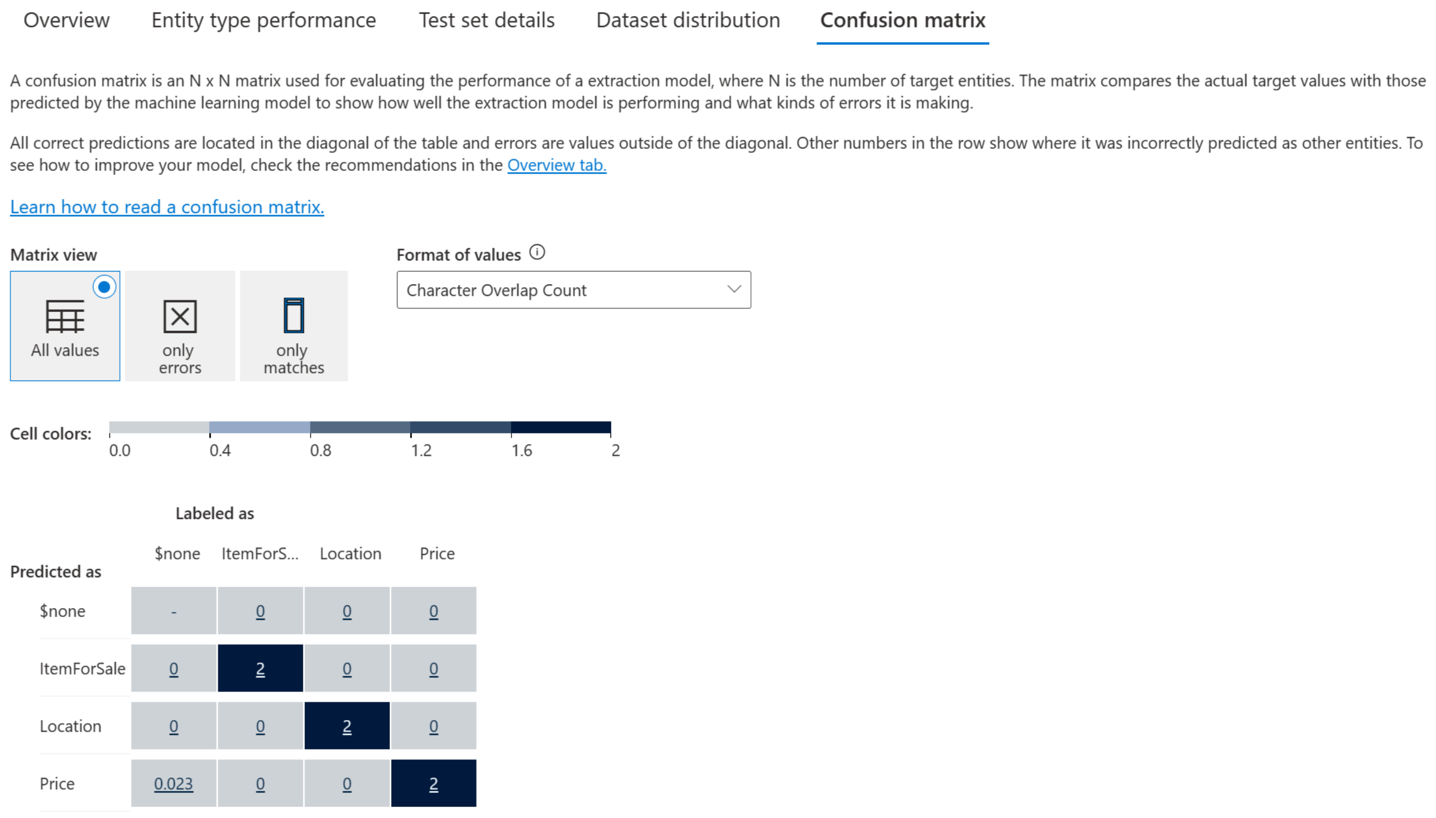
If both metrics have a low score, it means the model is both struggling to recognize entities in the document, and when it does extract that entity, it doesn't assign it the correct label with high confidence.

If precision is low but recall is high, it means that the model recognizes the entity well but doesn't label it as the correct entity type.

If precision is high but recall is low, it means that the model doesn't always recognize the entity, but when the model extracts the entity, the correct label is applied.

Confusion matrix

On the same **View model details** page, there's another tab on the top for the **Confusion matrix**. This view provides a visual table of all the entities and how each performed, giving a complete view of the model and where it's falling short.



A confusion matrix is an N x N matrix used for evaluating the performance of a extraction model, where N is the number of target entities. The matrix compares the actual target values with those predicted by the machine learning model to show how well the extraction model is performing and what kinds of errors it is making.

All correct predictions are located in the diagonal of the table and errors are values outside of the diagonal. Other numbers in the row show where it was incorrectly predicted as other entities. To see how to improve your model, check the recommendations in the Overview tab.

Learn how to read a confusion matrix.

Format of values: Character Overlap Count

Cell colors: 0.0, 0.4, 0.8, 1.2, 1.6, 2

Predicted as \ Labeled as	$none	ItemForS...	Location	Price
$none	-	0	0	0
ItemForSale	0	2	0	0
Location	0	0	2	0
Price	0.023	0	0	2

The confusion matrix allows you to visually identify where to add data to improve your model's performance.

Exercise - Extract custom entities

In this exercise, you use Azure AI Language to build a custom named entity recognition model.

Extract custom entities

In addition to other natural language processing capabilities, Azure AI Language Service enables you to define custom entities, and extract instances of them from text.

To test the custom entity extraction, we’ll create a model and train it through Azure AI Language Studio, then use a command line application to test it.

Module assessment

1. You've trained your model and you're seeing that it doesn't recognize your entities. What metric score is likely low to indicate that issue? **Recall**
2. You just finished labeling your data. How and where is that file stored to train your model? **JSON file, in my storage account container for the project**.
3. You train your model with only one source of documents, even though real extraction tasks will come from several sources. What data quality metric do you need to increase? **Diversity**

Summary

In this module, you learned about **custom named entity recognition (NER)** and how to extract entities.

In this module, you learned how to:

- Understand custom named entities and how they're labeled.
- Build a Language service project.
- Label data, train, and deploy an entity extraction model.
- Submit extraction tasks from your own app.

To learn more about Azure AI Language, see the Azure AI Language documentation.

Translate text with Azure AI Translator service

The Translator service enables you to create intelligent apps and services that can **translate text between languages**.

Learning objectives

After completing this module, you'll be able to:

- Provision a **Translator** resource
- Understand **language detection, translation, and transliteration**
- Specify translation options
- Define custom translations

Introduction

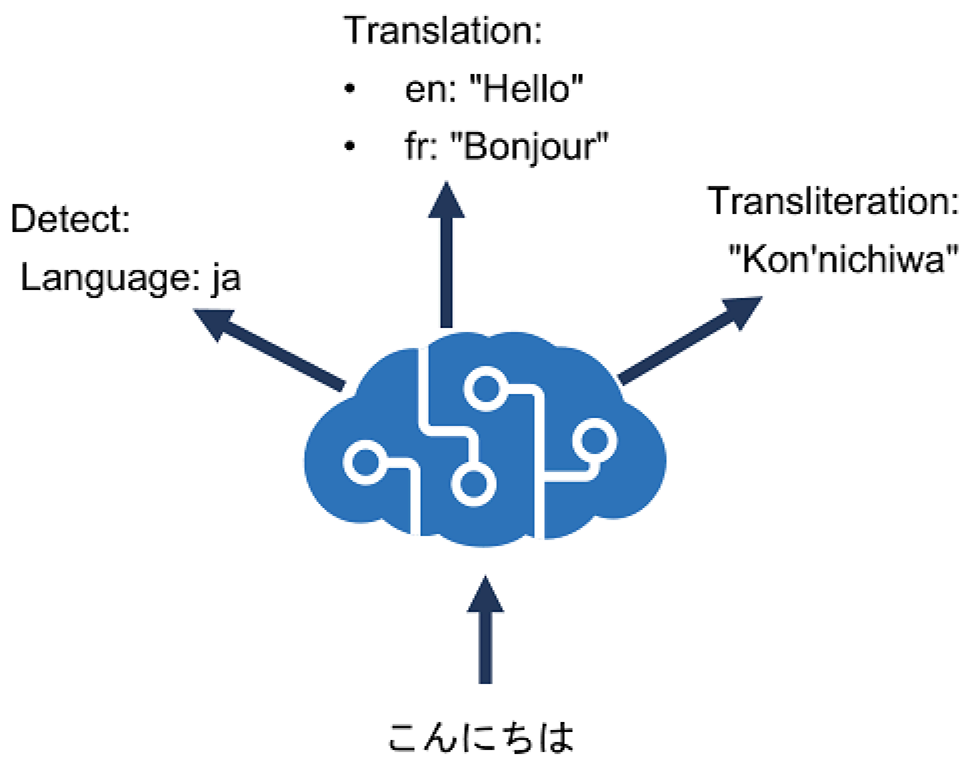
There are many commonly used languages throughout the world, and the ability to exchange information between speakers of different languages is often a critical requirement for global solutions.

The Azure AI Translator provides an API for **translating text between 90 supported languages**.

Provision an Azure AI Translator resource

Azure AI Translator provides a multilingual text translation API that you can use for:

- *Language detection*.
- *One-to-many translation*.
- *Script transliteration* (converting text from its native script to an alternative script).



Azure resource for Azure AI Translator

To use the Azure AI Translator service, you must provision a resource for it in your Azure subscription. You can provision a **single-service Azure AI Translator resource**, or you can use the **Text Translation API in a multi-service Azure AI Services resource**.

After you provision a suitable resource in your Azure subscription, you can use the location where you deployed the resource and one of its subscription keys to call the Azure AI Translator APIs from your code. You can call the APIs by **submitting requests in JSON format to the REST interface**, or by using any of the available **programming language-specific SDKs**.

> Note: The code examples in the subsequent units in this module show the `JSON` requests and responses exchanged with the `REST` interface. When using an SDK, the `JSON` requests are abstracted by appropriate objects and methods that encapsulate the same data values. You'll get a chance to try the SDK for C# or Python for yourself in the exercise later in the module.

Understand language detection, translation, and

transliteration

Let's explore the capabilities of Azure AI Translator. These capabilities include:

Language detection

You can use the **Detect** function of the **REST API** to detect the language in which text is written.

For example, you could submit the following text to the endpoint using curl.

Here's the text we want to translate:

```
{ 'Text' : 'こんにちは' }
```

Here's a call using curl to the endpoint to detect the language of our text:

```
curl -X POST "https://api.cognitive.microsofttranslator.com/detect?api-version=3.0" -H "Ocp-Apir
```

The response to this request looks as follows, indicating that the text is written in Japanese:

```
[
  {
    "language": "ja",
    "score": 1.0,
    "isTranslationSupported": true,
    "isTransliterationSupported": true
   }
]
```

Translation

To translate text from one language to another, use the **Translate** function; specifying *a single from parameter to indicate the source language*, and *one or more to parameters to specify the languages into which you want the text translated*.

For example, you could submit the same JSON we previously used to detect the language, specifying **a from parameter** of ja (Japanese) and **two to parameters** with the values en (English) and fr (French). To do this, you'd call:

```
curl -X POST "https://api.cognitive.microsofttranslator.com/translate?api-version=3.0&from=ja&t
```

This would produce the following result:

```
[
  {"translations":
    [
      {"text": "Hello", "to": "en"},
      {"text": "Bonjour", "to": "fr"}
    ]
  }
]
```

Transliteration

Our Japanese text is written using Hiragana script, so rather than translate it to a different language, you may want to transliterate it to a different script - for example to render the text in Latin script (as used by English language text).

To accomplish this, we can submit the Japanese text to the **Transliterate** function with a **fromScript** parameter of Jpan and a **toScript** parameter of Latn:

```
curl -X POST "https://api.cognitive.microsofttranslator.com/transliterate?api-version=3.0&fromS
```

The response would give you the following result:

```
[
    {
        "script": "Latn",
        "text": "Kon'nichiwa"
    }
]
```

Specify translation options

The **Translate** function of the API supports numerous parameters that affect the output.

Word alignment

In written English (using Latin script), spaces are used to separate words. However, in some other languages (and more specifically, scripts) this is not always the case.

For example, translating "Smart Services" from `en (English)` to `zh (Simplified Chinese)` produces the result "智能服务", and it's difficult to understand the relationship between the characters in the source text and the corresponding characters in the translation. To resolve this problem, you can specify the **includeAlignment** parameter with a value of **true** in your call to produce the following result:

```
[
   {
      "translations":[
         {
            "text":"智能服务",
            "to":"zh-Hans",
            "alignment":{
               "proj":"0:4-0:1 6:13-2:3"
            }
         }
      ]
   }
]
```

These results tell us that characters 0 to 4 in the source correspond to characters 0 to 1 in the translation, while characters 6 to 13 in the source correspond to characters 2 to 3 in the translation.

Sentence length

Sometimes it might be useful to know the **length of a translation**, for example to determine how best to display it in a user interface. You can get this information by setting the **includeSentenceLength** parameter to true.

For example, specifying this parameter when translating the English (en) text "Hello world" to French (fr) produces the following results:

```
[
   {
      "translations":[
         {
            "text":"Salut tout le monde",
            "to":"fr",
            "sentLen":{"srcSentLen":[12],"transSentLen":[20]}
         }
      ]
   }
]
```

Profanity filtering

Sometimes text contains profanities, which you might want to obscure or omit altogether in a translation. You can handle profanities by specifying the **profanityAction** parameter, which can have one of the following values:

- **NoAction**: Profanities are translated along with the rest of the text.
- **Deleted**: Profanities are omitted in the translation.
- **Marked**: Profanities are indicated using the technique indicated in the **profanityMarker** parameter (if supplied). The default value for this parameter is Asterisk, which replaces characters in profanities with "*". As an alternative, you can specify a profanityMarker value of Tag, which causes profanities to be enclosed in XML tags.

For example, translating the English (en) text "JSON is [redacted] great!" (where the blocked out word is a profanity) to German (de) with a profanityAction of Marked and a profanityMarker of Asterisk produces the following result:

```
[
   {
      "translations":[
         {
            "text":"JSON ist *** erstaunlich.",
            "to":"de"
         }
      ]
   }
]
```

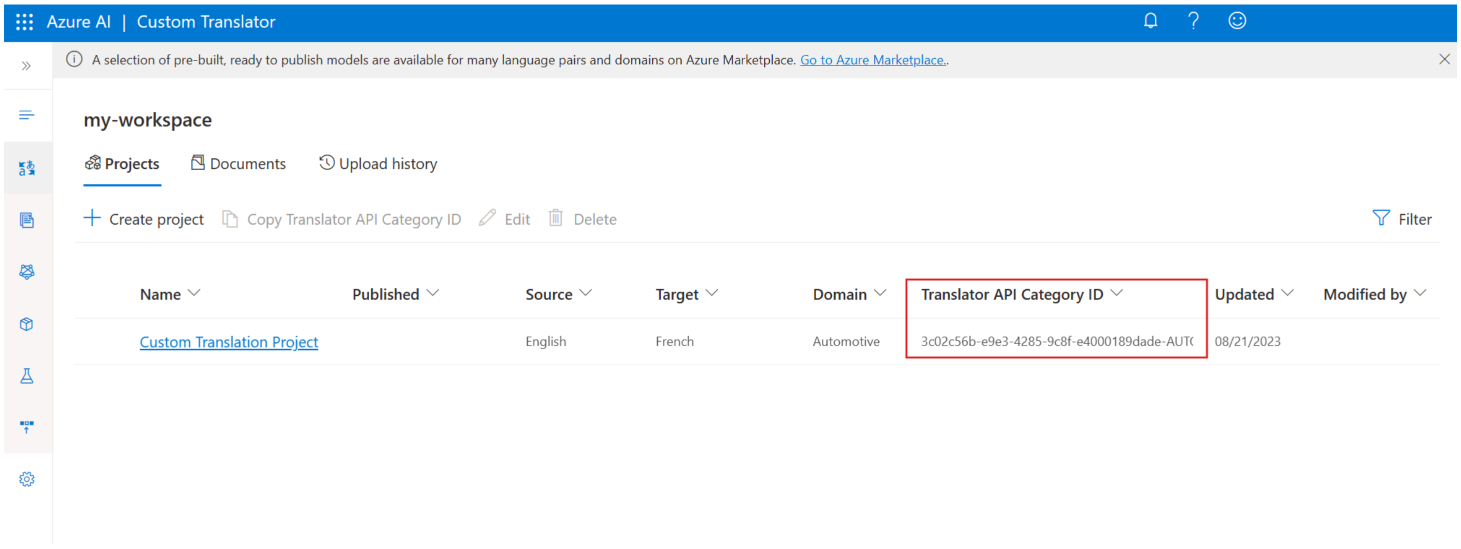
> Note: To learn more about the translation options, including some not described here, see the Azure AI Translator API documentation.

Define custom translations

While the default translation model used by Azure AI Translator is effective for general translation, you may need to develop a **translation solution** for businesses or industries in that have **specific vocabularies of terms that require custom translation**.

To solve this problem, you can **create a custom model that maps your own sets of source and target terms for translation**. To create a custom model, use the Custom Translator portal to:

- Create a workspace linked to your Azure AI Translator resource.
- Create a project.
- Upload training data files and train a model.
- Test your model and publish your model.
- Make translation calls to the API.


Azure AI | Custom Translator
A selection of pre-built, ready to publish models are available for many language pairs and domains on Azure Marketplace. Go to Azure Marketplace.
my-workspace
Projects
Documents
Upload history
Create project
Copy Translator API Category ID
Edit
Delete
Filter

Name	Published	Source	Target	Domain	Translator API Category ID	Updated	Modified by
Custom Translation Project		English	French	Automotive	3c02c56b-e9e3-4285-9c8f-e4000189dade-AUT(	08/21/2023	

Your custom model is assigned **a unique category Id** (highlighted in the screenshot), which you can specify in translate calls to your **Azure AI Translator resource** by using the category parameter, causing translation to be performed by your custom model instead of the default model.

How to call the API

To initiate a translation, you send a POST request to the following request URL:

```
https://api.cognitive.microsofttranslator.com/translate?api-version=3.0
```

Your request needs to include a couple of **parameters**:

- `api-version` : The required version of the API.
- `to` : The **target language to translate to**. For example: `to=fr` for French.
- `category` : Your category Id.

Your request must also include a number of required headers:

- `Ocp-Apim-Subscription-Key` . Header for your client key. For example: `Ocp-Apim-Subscription-Key=<your-client-key>` .
- `Content-Type` . The content type of the payload. The required format is: `Content-Type: application/json; charset=UTF-8` .

The request body should contain an array that includes a JSON object with a `Text` property that specifies the text that you want to translate:

```
[
    {"Text":"Where can I find my employee details?"}
]
```

There are different ways you can send your request to the API, including using the C#, Python, and curl. For instance, to make a quick call, you can send a `POST` request using curl:

```
curl -X POST "https://api.cognitive.microsofttranslator.com/translate?api-version=3.0&from=en&t
```

The request above makes a call to translate a sentence from English to Dutch.

Response returned

The response returns a response code of `200` if the request was successful. It also returns a response body that contains the translated text, like this:

```
[
    {
        "translations":[
            {"text":"Waar vind ik mijn personeelsgegevens?","to":"nl"}
        ]
    }
]
```

If the request wasn't successful, then a number of different status codes may be returned depending on the error type, such as `400` *(missing or invalid query parameters)*. See Response status codes for a full list of codes and their explanation.

> Note: For more information about custom translation, see Quickstart: Build, publish, and translate with custom models.

Exercise - Translate text with the Azure AI Translator service

In this exercise, you build an app that translates text between languages.

Translate Text

Azure AI Translator is a service that enables you to translate text between languages. In this exercise, you'll use it to create a simple app that translates input in any supported language to the target language of your choice.

Module assessment

1. What function of Azure AI Translator should you use to convert the Chinese word "你好" to the English word "Hello"? **Translate**
2. What function of Azure AI Translator should you use to convert the Russian word "спасибо" in Cyrillic characters to "spasibo" in Latin characters? **Transliterate**

Summary

In this module, you learned how to:

- Provision an Azure AI Translator resource
- Understand language detection, translation, and transliteration
- Specify translation options
- Define custom translations

To learn more about Azure AI Translator, see the Azure AI Translator documentation.

Create speech-enabled apps with Azure AI services

The **Azure AI Speech service** enables you to build **speech-enabled applications**. This module focuses on using the **speech-to-text and text to speech APIs**, which enable you to create apps that are capable of **speech recognition** and **speech synthesis**.

Learning objectives

In this module, you'll learn how to:

- Provision an Azure resource for the Azure AI Speech service
- Implement *speech recognition* with the *Azure AI Speech to text API*
- Use the *Text to speech API* to implement *speech synthesis*
- Configure audio format and voices
- Use **Speech Synthesis Markup Language (SSML)**

API	Function
Speech to text	Speech recognition
Text to speech	Speech synthsis

Introduction

Azure AI Speech provides APIs that you can use to build **speech-enabled applications**. This includes:

- **Speech to text**: An API that enables **speech recognition** in which your application can accept **spoken input**.
- **Text to speech**: An API that enables **speech synthesis** in which your application can provide **spoken output**.
- **Speech Translation**: An API that you can use to **translate spoken input into multiple languages**.

- **Keyword Recognition**: An API that enables your application to **recognize keywords or short phrases**.
- **Intent Recognition**: An API that uses **conversational language understanding to determine the semantic meaning of spoken input**.

This module focuses on `speech recognition` and `speech synthesis`, which are core capabilities of any speech-enabled application.

> Note: The code examples in this module are provided in Python, but you can use any of the available Azure AI Speech SDK packages to develop speech-enabled applications in your preferred language. Available SDK packages include:
>
> - azure-cognitiveservices-speech for Python
> - Microsoft.CognitiveServices.Speech for Microsoft .NET
> - microsoft-cognitiveservices-speech-sdk for JavaScript
> - Microsoft Cognitive Services Speech SDK For Java

Provision an Azure resource for speech

Before you can use Azure AI Speech, you need to create an Azure AI Speech resource in your Azure subscription. You can use either **a dedicated Azure AI Speech resource** or **a multi-service Azure AI Services or Azure AI Foundry resource**.

After you create your resource, you'll need the following information to use it from a client application through one of the supported SDKs:

- The location in which the resource is deployed (for example, eastus)
- One of the keys assigned to your resource.

You can view of these values on the **Keys** and **Endpoint** page for your resource in the Azure portal.

While the specific syntax and parameters can vary between language-specific SDKs, most interactions with the Azure AI Speech service start with the creation of a `SpeechConfig` object that encapsulates the connection to your Azure AI Speech resource.

For example, the following Python code instantiates a `SpeechConfig` object based on an Azure AI Speech resource in the East US region:

```python
import azure.cognitiveservices.speech as speech_sdk

speech_config = speech_sdk.SpeechConfig(your_project_key, 'eastus')
```

> Note: This example assumes that the Speech SDK package for python has been installed, like this:
> `pip install azure-cognitiveservices-speech`

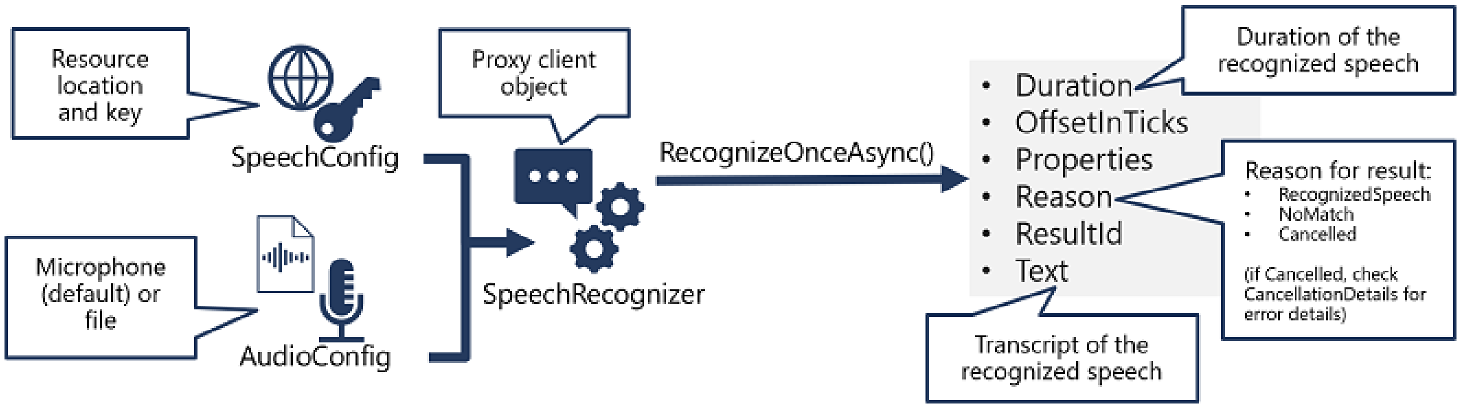
Use the Azure AI Speech to Text API

The Azure AI Speech service supports **speech recognition** through the following features:

- **Real-time transcription**: Instant transcription with intermediate results for live audio inputs.
- **Fast transcription**: Fastest `synchronous` output for situations with predictable latency.
- **Batch transcription**: Efficient processing for large volumes of prerecorded audio.
- **Custom speech**: Models with **enhanced accuracy for specific domains and conditions**.

Using the Azure AI Speech SDK

While the specific details vary, depending on the SDK being used (Python, C#, and so on); there's a consistent pattern for using the **Speech to text** API:


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

- Use a **SpeechConfig** object to encapsulate the information required to connect to your Azure AI Speech resource. Specifically, its **location** and **key**.
- 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.
- Use the **SpeechConfig** and **AudioConfig** to create a **SpeechRecognizer** object. This object is a proxy client for the **Speech to text** API.

- Use the methods of the **SpeechRecognizer** object to call the underlying API functions. For example, the **RecognizeOnceAsync()** method uses the Azure AI Speech service to *asynchronously* transcribe a single spoken utterance.
- Process the response from the Azure AI 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

If the operation was successful, the **Reason** property has the enumerated value **RecognizedSpeech**, and the Text property contains the transcription. Other possible values for **Result** include **NoMatch** (indicating that the audio was successfully parsed but no speech was recognized) or **Canceled**, indicating that an error occurred (in which case, you can check the Properties collection for the **CancellationReason** property to determine what went wrong).

Use the text to speech API

Similarly to its `Speech to text` APIs, the Azure AI Speech service offers other **REST APIs for speech synthesis**:

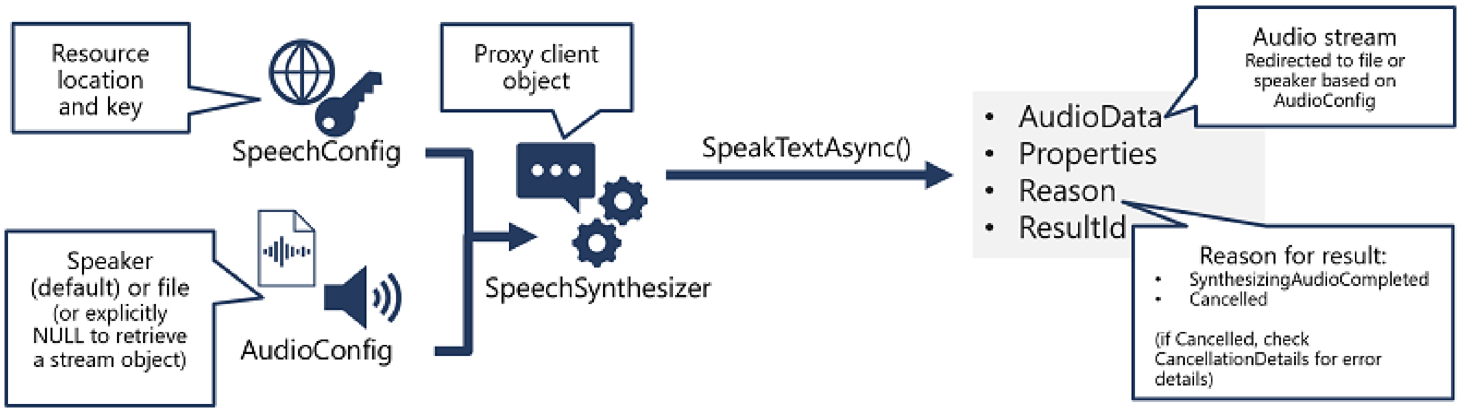
- The **Text to speech API**, which is the primary way to perform speech synthesis.
- The **Batch synthesis API**, which is designed to support batch operations that *convert large volumes of text to audio* - for example to *generate an audio-book from the source text*.

You can learn more about the REST APIs in the Text to speech REST API documentation. In practice, most interactive speech-enabled applications use the Azure AI Speech service through a (programming) language-specific SDK.

Using the Azure AI Speech SDK

As with speech recognition, in practice most interactive speech-enabled applications are built using the Azure AI Speech SDK.

The pattern for implementing speech synthesis is similar to that of speech recognition:


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)

- Use a `SpeechConfig` object to encapsulate the information required to connect to your Azure AI Speech resource. Specifically, its **location** and **key**.
- Optionally, use an `AudioConfig` to define the output device for the speech to be synthesized. By default, this is the default system speaker, but you can also specify an audio file, or by explicitly setting this value to a `null` value, you can process the audio stream object that is returned directly.
- Use the `SpeechConfig` and `AudioConfig` to create a `SpeechSynthesizer` object. This object is a proxy client for the **Text to speech API**.
- Use the methods of the `SpeechSynthesizer` object to call the underlying API functions. For example, the `SpeakTextAsync()` method uses the Azure AI Speech service to convert text to spoken audio.
- Process the response from the Azure AI Speech service. In the case of the `SpeakTextAsync` method, the result is a `SpeechSynthesisResult` object that contains the following properties:
  - AudioData
  - Properties
  - Reason
  - ResultId

When speech has been successfully synthesized, the `Reason` property is set to the `SynthesizingAudioCompleted` enumeration and the `AudioData` property contains the audio stream (which, depending on the `AudioConfig` may have been automatically sent to a speaker or file).

Configure audio format and voices

When synthesizing speech, you can use a `SpeechConfig` object to customize the audio that is returned by the Azure AI Speech service.

Audio format

The Azure AI Speech service supports multiple output formats for the audio stream that is generated by speech synthesis. Depending on your specific needs, you can choose a format based on the required:

- Audio file type
- Sample-rate
- Bit-depth

For example, the following Python code sets the speech output format for a previously defined SpeechConfig object named `speech_config`:

```
speech_config.set_speech_synthesis_output_format(SpeechSynthesisOutputFormat.Riff24Khz16BitMonoP
```

For a full list of supported formats and their enumeration values, see the Azure AI Speech SDK documentation.

Voices

The Azure AI Speech service provides multiple voices that you can use to personalize your speech-enabled applications. Voices are **identified by names** that indicate a locale and a person's name - for example `en-GB-George`.

The following Python example code sets the voice to be used

```
speech_config.speech_synthesis_voice_name = "en-GB-George"
```

For information about voices, see the Azure AI Speech SDK documentation.

Use Speech Synthesis Markup Language

While the Azure AI Speech SDK enables you to submit plain text to be synthesized into speech, the service also supports an XML-based syntax for describing characteristics of the speech you want to generate. This **Speech Synthesis Markup Language (SSML)** syntax offers greater control over how the spoken output sounds, enabling you to:

- Specify a **speaking style**, such as "*excited*" or "*cheerful*" when using a neural voice.

- Insert **pauses or silence**.
- Specify **phonemes** (phonetic pronunciations), for example to pronounce the text "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.

For example, consider the following SSML:

```
<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>
```

This SSML specifies a spoken dialog between two different **neural voices**, like this:

- Ariana (cheerfully): "I say tomato:
- Guy: "I say tomato (pronounced tom-ah-toe) ... Let's call the whole thing off!"

To submit an SSML description to the Speech service, you can use an appropriate method of a SpeechSynthesizer object, like this:

```
speech_synthesizer.speak_ssml('<speak>...');
```

For more information about SSML, see the Azure AI Speech SDK documentation.

Exercise - Create a speech-enabled app

In this exercise, build a speech enabled app for both speech recognition and synthesis.

Recognize and synthesize speech

Azure AI Speech is a service that provides speech-related functionality, including:

- A *speech-to-text API* that enables you to implement *speech recognition* (converting audible spoken words into text).
- A *text-to-speech API* that enables you to implement *speech synthesis* (converting text into audible speech).

In this exercise, you'll use both of these APIs to implement a speaking clock application.

Module assessment

1. What information do you need from your Azure AI Speech service resource to consume it using the Azure AI Speech SDK? **The location and one of the keys.**
   NOTE: Location is not endpoint!! An example of location: eastus
2. Which object should you use to specify that the speech input to be transcribed to text is in an audio file? **AudioConfig** Use the Azure AI Speech to Text API
3. How can you change the voice used in speech synthesis? Set the `SpeechSynthesisVoiceName` property of the `SpeechConfig` object to the desired voice name.

Summary

In this module, you learned how to:

- Provision an Azure resource for the Azure AI Speech service
- Use the **Speech to text API** to implement **speech recognition**
- Use the **Text to speech API** to implement **speech synthesis**
- Configure audio format and voices
- Use **Speech Synthesis Markup Language (SSML)**

To learn more about the Azure AI Speech, refer to the Azure AI Speech service documentation.

Translate speech with the Azure AI Speech service

Translation of speech builds on speech recognition by recognizing and transcribing spoken input in a specified language, and returning translations of the transcription in one or more other languages.

Learning objectives

In this module, you will learn how to:

- Provision Azure resources for **speech translation**.
- Generate text translation from speech.
- Synthesize spoken translations.

Introduction

Translation of speech builds on speech recognition by recognizing and transcribing spoken input in a specified language, and returning translations of the transcription in one or more other languages.

In this module, you'll learn how to:

- Provision Azure resources for speech translation.
- Generate text translation from speech.
- Synthesize spoken translations.

The units in the module include important conceptual information about Azure AI Speech and how to use its API through one of the supported software development kits (SDKs), after which you're able to try Azure AI Speech for yourself in a hands-on exercise. To complete the hands-on exercise, you'll need a Microsoft Azure subscription. If you don't already have one, you can sign up for a free trial.

Provision an Azure resource for speech translation

The Azure AI Speech service provides robust, machine learning and artificial intelligence-based speech translation services, enabling developers to add end-to-end, real-time, speech translations to their applications or services. You can use either **a dedicated Azure AI Speech resource** or **a multi-service Azure AI Services resource**.

Before you can use the service, you need to create an Azure AI Speech resource in your Azure subscription.

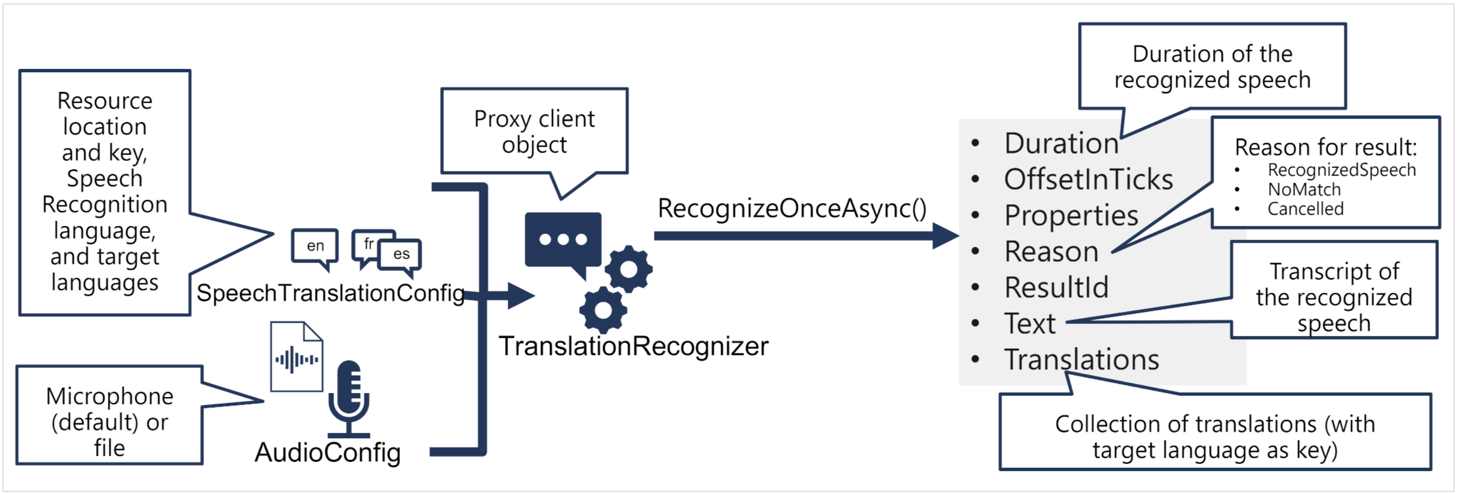
After creating your Azure resource, you'll need the following information to use it from a client application through one of the supported SDKs:

- The **location** in which the resource is deployed (for example, eastus)
- **One of the keys** assigned to your resource.

You can view of these values on the **Keys and Endpoint page** for your resource in the Azure portal.

Translate speech to text

The pattern for **speech translation** using the Azure AI Speech SDK is **similar to speech recognition**, with the addition of information about the source and target languages for translation:


Resource location and key, 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)

- Use a `SpeechTranslationConfig` object to encapsulate the information required to connect to your Azure AI Speech resource. Specifically, its **location and key**.
- The `SpeechTranslationConfig` object is also used to specify the **speech recognition language** (the language in which the input speech is spoken) and the **target languages** into which it should

be translated.

- 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**.
- Use the `SpeechTranslationConfig` , and `AudioConfig` to create a `TranslationRecognizer` object. This object is **a proxy client for the Azure AI Speech translation API**.
- Use the methods of the `TranslationRecognizer` object to call the underlying API functions. For example, the `RecognizeOnceAsync()` method uses the Azure AI Speech service to *asynchronously* translate a single spoken utterance.
- Process the response from Azure AI Speech. In the case of the `RecognizeOnceAsync()` method, the result is a `SpeechRecognitionResult` object that includes the following properties:
  - Duration
  - OffsetInTicks
  - Properties
  - Reason
  - ResultId
  - Text
  - Translations

If the operation was successful, the `Reason` property has the enumerated value `RecognizedSpeech` , the Text property contains the transcription in the original language. You can also access a `Translations` property which contains a dictionary of the translations (using the two-character ISO language code, such as "en" for English, as a key).

Synthesize translations

The `TranslationRecognizer` returns *translated transcriptions of spoken input* - essentially *translating audible speech to text*.

You can also synthesize the translation as speech to create **speech-to-speech translation** solutions. There are two ways you can accomplish this.

Event-based synthesis

When you want to perform **1:1 translation (translating from one source language into a single target language)**, you can use event-based synthesis to capture the translation as an audio stream. To do this, you need to:

Specify the desired voice for the translated speech in the `TranslationConfig`. Create an event handler for the `TranslationRecognizer` object's `Synthesizing` event. In the event handler, use the `GetAudio()` method of the Result parameter to retrieve the byte stream of translated audio. The specific code used to implement an event handler varies depending on the programming language you're using. See the C# and Python examples in the Speech SDK documentation.

Manual synthesis

Manual synthesis is an alternative approach to event-based synthesis that doesn't require you to implement an event handler. You can use manual synthesis to generate audio translations for one or more 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.

Exercise - Translate speech

In this exercise, build an app that recognizes and translates speech into a specific language.

Translate Speech

Azure AI Speech includes a speech translation API that you can use to translate spoken language. For example, suppose you want to develop a translator application that people can use when traveling in places where they don’t speak the local language. They would be able to say phrases such as “Where is the station?” or “I need to find a pharmacy” in their own language, and have it translate them to the local language.

Module assessment

1. Which SDK object should you use to specify the language(s) into which you want speech translated? **SpeechTranslationConfig**
2. Which SDK object should you use as a proxy for the Translation API of Azure AI Speech service? **TranslationRecognizer**
3. When translating speech, in which cases can you use the Synthesizing event to synthesize the translations and speech? **Only when translating to a single target language**.

Summary

Now that you've completed this module, you learned how to:

- Provision Azure resources for speech translation.
- Generate text translation from speech.
- Synthesize spoken translations.

For more information about speech translation, refer to the Azure AI Speech documentation.

Develop an audio-enabled generative AI application

A voice carries meaning beyond words, and audio-enabled generative AI models can interpret spoken input to **understand tone, intent, and language**. Learn how to build **audio-enabled chat apps** that listen and respond to audio.

Learning objectives

After completing this module, you'll be able to:

- Deploy an **audio-enabled generative AI model** in Azure AI Foundry.
- Create a **chat app** that submits audio-based prompts.

Introduction

Generative AI models make it possible to build intelligent chat-based applications that can understand and reason over input. Traditionally, text input is the primary mode of interaction with AI models, but **multimodal models** are increasingly becoming available. These models make it possible for chat applications to respond to audio input as well as text.

In this module, we'll discuss **audio-enabled generative AI** and explore how you can use Azure AI Foundry to create generative AI solutions that respond to prompts that include a mix of text and audio data.

Deploy a multimodal model

To handle prompts that include audio, you need to deploy a **multimodal generative AI model** - in other words, a model that supports not only *text-based input*, but *audio-based input* as well. Multimodal models available in Azure AI Foundry include (among others):

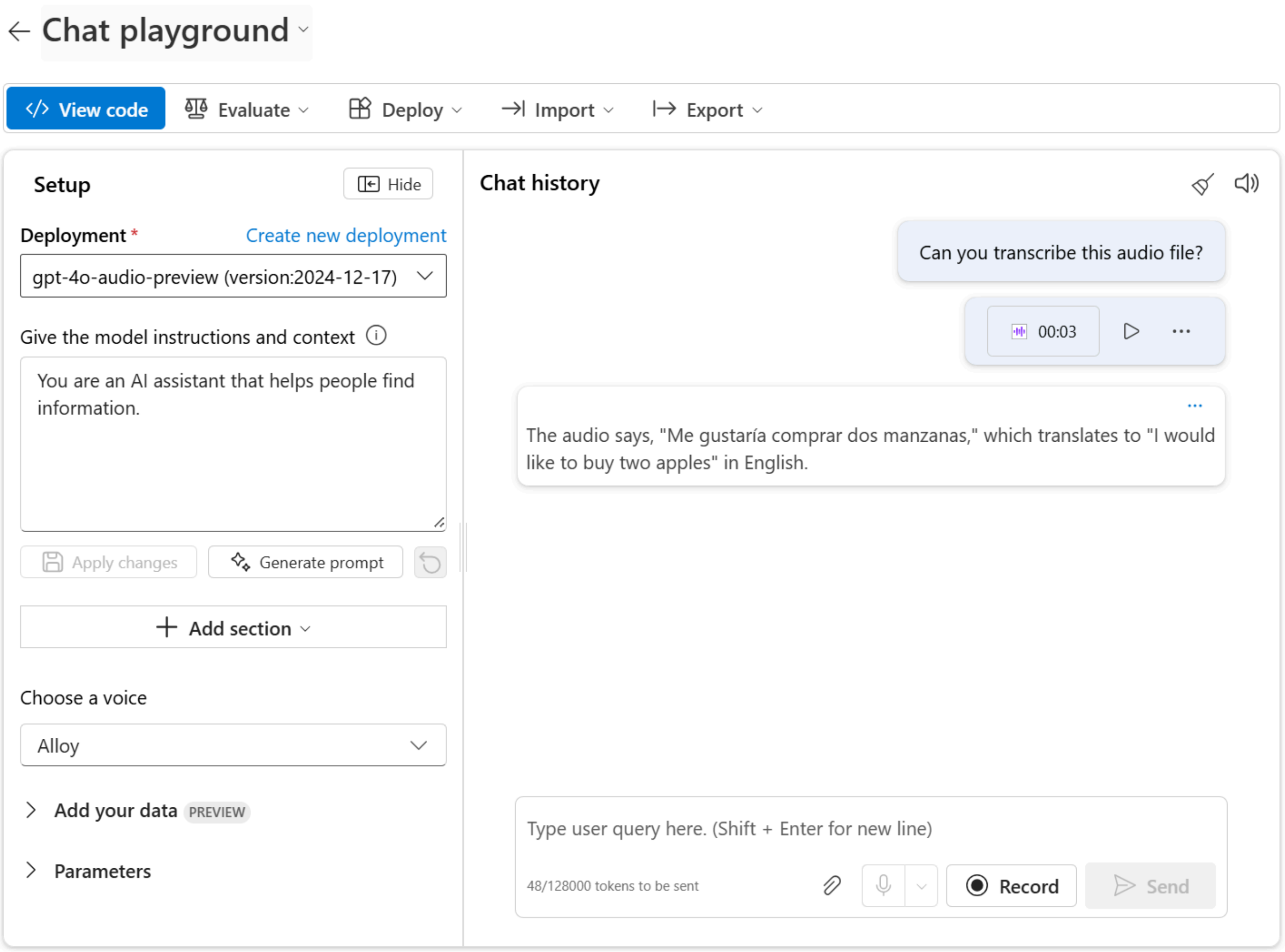
- Microsoft Phi-4-multimodal-instruct

- OpenAI gpt-4o
- OpenAI gpt-4o-mini

> Tip: To learn more about available models in Azure AI Foundry, see the Model catalog and collections in Azure AI Foundry portal article in the Azure AI Foundry documentation.

Testing multimodal models with audio-based prompts

After deploying a multimodal model, you can test it in the chat playground in Azure AI Foundry portal. Some models allow you to include audio attachments in the playground, either by **uploading a file or recording a message**.



Deployment *

gpt-4o-audio-preview (version:2024-12-17)

Give the model instructions and context

You are an AI assistant that helps people find information.

Choose a voice

Alloy

Can you transcribe this audio file?

00:03

The audio says, "Me gustaría comprar dos manzanas," which translates to "I would like to buy two apples" in English.

Type user query here. (Shift + Enter for new line)

48/128000 tokens to be sent

In the chat playground, you can upload a local audio file and add text to the message to elicit a response from a multimodal model.

Develop an audio-based chat app

To develop a client app that engages in audio-based chats with a multimodal model, you can use the same basic techniques used for text-based chats. You require a connection to the **endpoint** where the model is deployed, and you use that endpoint to submit prompts that consists of messages to the model and process the responses.

The key difference is that **prompts for an audio-based chat** include multi-part user messages that contain **both a text content item and an audio content item**.

The JSON representation of a prompt that includes a multi-part user message looks something like this:

```
{
    "messages": [
        { "role": "system", "content": "You are a helpful assistant." },
        { "role": "user", "content": [
            {
                "type": "text",
                "text": "Transcribe this audio:"
            },
            {
                "type": "audio_url",
                "audio_url": {
                    "url": "https://....."
                }
            }
        ] }
    ]
}
```

The audio content item can be:

- **A URL to an audio file in a web site**.
- **Binary audio data**.

When using binary data to submit a local audio file, the `audio_url` content takes the form of a base64 encoded value in a data URL format:

```
{
    "type": "audio_url",
    "audio_url": {
       "url": "data:audio/mp3;base64,<binary_audio_data>"
    }
}
```

Depending on the model type, and where you deployed it, you can **use Microsoft Azure AI Model Inference or OpenAI APIs to submit audio-based prompts**. These libraries also provide language-specific SDKs that abstract the underlying REST APIs.

In the exercise that follows in this module, you can *use the Python or .NET SDK for the Azure AI Model Inference API and the OpenAI API to develop an audio-enabled chat application*.

Exercise - Develop an audio-enabled chat app

If you have an Azure subscription, you can complete this exercise to develop an audio-enabled chat app.

Develop an audio-enabled chat app

In this exercise, you use the `Phi-4-multimodal-instruct` generative AI model to generate responses to prompts that include audio files. You'll develop an app that provides AI assistance for a produce supplier company by using Azure AI Foundry and the Azure AI Model Inference service to summarize voice messages left by customers.

Module assessment

1. Which kind of model can you use to respond to audio input? **Multimodal models**.
2. How can you submit a prompt that asks a model to analyze an audio file? **Submit a prompt that contains a multi-part user message, containing both text content and audio content**.
3. How can you include an audio in a message? **As a URL or as binary data.**

Summary

In this module, you learned about **audio-enabled generative AI models** and how to implement chat solutions that include audio-based input.

Audio-enabled models let you create AI solutions that can understand audio and respond to related questions or instructions. Beyond just identifying spoken words, some models can also use reasoning based on what they hear. For instance, they can **summarize a message or assess the speaker's sentiment**.

> Tip: For more information about working with multimodal models in Azure AI Foundry, see How to use image and audio in chat completions with Azure AI model inference and Quickstart: Use speech and audio in your AI chats.

Develop an Azure AI Voice Live agent

Learn how to develop an Azure AI Voice Live agent using the Voice Live API and SDK. This module covers the fundamentals of the Voice Live platform, including API integration, SDK usage, and building conversational AI agents.

Learning objectives

After completing this module, you'll be able to:

- Implement the Azure AI Voice Live API to enable real-time, bidirectional communication.
- Set up and configure the agent session.
- Develop and manage event handlers to create dynamic and interactive user experiences.
- Build and deploy a Python-based web app with real-time voice interaction capabilities to Azure.

Introduction

Voice-enabled applications are transforming how we interact with technology, and this module guides you through building a real-time, interactive voice solutions using advanced APIs and tools. The Azure AI Voice live API is a solution enabling low-latency, high-quality speech to speech interactions for voice agents. The API is designed for developers seeking scalable and efficient voice-driven experiences as it eliminates the need to manually orchestrate multiple components.

After completing this module, you'll be able to:

- Implement the Azure AI Voice Live API to enable real-time, bidirectional communication.
- Set up and configure the agent session.
- Develop and manage event handlers to create dynamic and interactive user experiences.
- Build and deploy a Python-based web app with real-time voice interaction capabilities to Azure.

Explore the Azure Voice Live API

The Voice live API enables developers to create voice-enabled applications with real-time, bidirectional communication. This unit explores its architecture, configuration, and implementation.

Key features of the Voice Live API

The Voice live API provides real-time communication using WebSocket connections. It supports advanced features such as speech recognition, text-to-speech synthesis, avatar streaming, and audio processing.

- JSON-formatted events manage conversations, audio streams, and responses.
- Events are categorized into client events (sent from client to server) and server events (sent from server to client).

Key features include:

- Real-time audio processing with support for multiple formats like PCM16 and G.711.
- Advanced voice options, including OpenAI voices and Azure custom voices.
- Avatar integration using WebRTC for video and animation.
- Built-in noise reduction and echo cancellation.

> Note: Voice Live API is optimized for Azure AI Foundry resources. We recommend using Azure AI Foundry resources for full feature availability and best Azure AI Foundry integration experience.

For a table of supported models and regions, visit the Voice Live API overview.

Connect to the Voice Live API

The Voice live API supports two authentication methods: Microsoft Entra (keyless) and API key. Microsoft Entra uses token-based authentication for an Azure AI Foundry resource. You apply a retrieved authentication token using a Bearer token with the Authorization header.

For the recommended keyless authentication with Microsoft Entra ID, you need to assign the Cognitive Services User role to your user account or a managed identity. You generate a token using the Azure CLI or Azure SDKs. The token must be generated with the https://ai.azure.com/.default scope, or the legacy https://cognitiveservices.azure.com/.default scope. Use the token in the Authorization header of the WebSocket connection request, with the format `Bearer <token>`.

For key access, an API key can be provided in one of two ways. You can use an api-key connection header on the prehandshake connection. This option isn't available in a browser environment. Or, you can use an api-key query string parameter on the request URI. Query string parameters are encrypted when using https/wss.

> Note: The `api-key` connection header on the prehandshake connection isn't available in a browser environment.

WebSocket endpoint

The endpoint to use varies depending on how you want to access your resources. You can access resources through a connection to the AI Foundry project (Agent), or through a connection to the model.

- Project connection: The endpoint is `wss://<your-ai-foundry-resource-name>.services.ai.azure.com/voice-live/realtime?api-version=2025-10-01`
- Model connection: The endpoint is `wss://<your-ai-foundry-resource-name>.cognitiveservices.azure.com/voice-live/realtime?api-version=2025-10-01`.

The endpoint is the same for all models. The only difference is the required model query parameter, or, when using the Agent service, the agent_id and project_id parameters.

Voice Live API events

Client and server events facilitate communication and control within the Voice live API. Key client events include:

- `session.update` : Modify session configurations.
- `input_audio_buffer.append` : Add audio data to the buffer.
- `response.create` : Generate responses via model inference.

Server events provide feedback and status updates:

- `session.updated` : Confirm session configuration changes.
- `response.done` : Indicate response generation completion.
- `conversation.item.created` : Notify when a new conversation item is added.

For a full list of client/server events, visit Voice live API Reference.

> Note: Proper handling of events ensures seamless interaction between client and server.

Configure session settings for the Voice live API

Often, the first event sent by the caller on a newly established Voice live API session is the `session.update` event. This event controls a wide set of input and output behavior. Session settings can be updated dynamically using the session.update event. Developers can configure voice types, modalities, turn detection, and audio formats.

Example configuration:

```json
{
  "type": "session.update",
  "session": {
    "modalities": ["text", "audio"],
    "voice": {
      "type": "openai",
      "name": "alloy"
    },
    "instructions": "You are a helpful assistant. Be concise and friendly.",
    "input_audio_format": "pcm16",
    "output_audio_format": "pcm16",
    "input_audio_sampling_rate": 24000,
    "turn_detection": {
      "type": "azure_semantic_vad",
      "threshold": 0.5,
      "prefix_padding_ms": 300,
      "silence_duration_ms": 500
    },
    "temperature": 0.8,
    "max_response_output_tokens": "inf"
  }
}
```

> Tip: Use Azure semantic VAD for intelligent turn detection and improved conversational flow.

Implement real-time audio processing with the Voice live API

Real-time audio processing is a core feature of the Voice live API. Developers can append, commit, and clear audio buffers using specific client events.

- Append audio: Add audio bytes to the input buffer.
- Commit audio: Process the audio buffer for transcription or response generation.
- Clear audio: Remove audio data from the buffer.

Noise reduction and echo cancellation can be configured to enhance audio quality. For example:

```
{
  "type": "session.update",
  "session": {
    "input_audio_noise_reduction": {
      "type": "azure_deep_noise_suppression"
    },
    "input_audio_echo_cancellation": {
      "type": "server_echo_cancellation"
    }
  }
}
```

> Note: Noise reduction improves VAD accuracy and model performance by filtering input audio.

Integrate avatar streaming using the Voice live API

The Voice live API supports WebRTC-based avatar streaming for interactive applications. Developers can configure video, animation, and blendshape settings.

- Use the session.avatar.connect event to provide the client's SDP offer.
- Configure video resolution, bitrate, and codec settings.
- Define animation outputs such as blendshapes and visemes.

Example configuration:

```
{
  "type": "session.avatar.connect",
  "client_sdp": "<client_sdp>"
}
```

> Tip: Use high-resolution video settings for enhanced visual quality in avatar interactions.

Explore the AI Voice Live client library for Python

The Azure AI Voice Live client library for Python provides a real-time, speech-to-speech client for Azure AI Voice Live API. It opens a WebSocket session to stream microphone audio to the service and receives server events for responsive conversations.

> Important: As of version 1.0.0, this SDK is async-only. The synchronous API is deprecated to focus exclusively on async patterns. All examples and samples use async/await syntax.

In this unit, you learn how to use the SDK to implement authentication and handle events. You also see a minimal example of creating a session. For a full reference to the Voice Live package, visit the voice live Package reference.

Implement authentication

You can implement authentication with an API key or a Microsoft Entra ID token. The following code sample shows an API key implementation. It assumes environment variables are set in a `.env` file, or directly in your environment.

```python
import asyncio
from azure.core.credentials import AzureKeyCredential
from azure.ai.voicelive import connect

async def main():
    async with connect(
        endpoint="your-endpoint",
        credential=AzureKeyCredential("your-api-key"),
        model="gpt-4o"
    ) as connection:
        # Your async code here
        pass

asyncio.run(main())
```

For production applications, Microsoft Entra authentication is recommended. The following code sample shows implementing the `DefaultAzureCredential` for authentication:

```python
import asyncio
from azure.identity.aio import DefaultAzureCredential
from azure.ai.voicelive import connect

async def main():
    credential = DefaultAzureCredential()

    async with connect(
        endpoint="your-endpoint",
        credential=credential,
        model="gpt-4o"
    ) as connection:
        # Your async code here
        pass

asyncio.run(main())
```

Handling events

Proper handling of events ensures a more seamless interaction between the client and agent. For example, when handling a user interrupting the voice agent you need to cancel agent audio playback immediately in the client. If you don't, the client continues to play the last agent response until the interrupt is processed in the API - resulting in the agent "talking over" the user.

The following code sample shows some basic event handling:

```python
async for event in connection:
    if event.type == ServerEventType.SESSION_UPDATED:
        print(f"Session ready: {event.session.id}")
        # Start audio capture

    elif event.type == ServerEventType.INPUT_AUDIO_BUFFER_SPEECH_STARTED:
        print("User started speaking")
        # Stop playback and cancel any current response

    elif event.type == ServerEventType.RESPONSE_AUDIO_DELTA:
        # Play the audio chunk
        audio_bytes = event.delta

    elif event.type == ServerEventType.ERROR:
        print(f"Error: {event.error.message}")
```

Minimal example

The following code sample shows authenticating to the API and configuring the session.

```python
import asyncio
from azure.core.credentials import AzureKeyCredential
from azure.ai.voicelive.aio import connect
from azure.ai.voicelive.models import (
    RequestSession, Modality, InputAudioFormat, OutputAudioFormat, ServerVad, ServerEventType
)

API_KEY = "your-api-key"
ENDPOINT = "your-endpoint"
MODEL = "gpt-4o"

async def main():
    async with connect(
        endpoint=ENDPOINT,
        credential=AzureKeyCredential(API_KEY),
        model=MODEL,
    ) as conn:
        session = RequestSession(
            modalities=[Modality.TEXT, Modality.AUDIO],
            instructions="You are a helpful assistant.",
            input_audio_format=InputAudioFormat.PCM16,
            output_audio_format=OutputAudioFormat.PCM16,
            turn_detection=ServerVad(
                threshold=0.5,
                prefix_padding_ms=300,
                silence_duration_ms=500
            ),
        )
        await conn.session.update(session=session)

        # Process events
        async for evt in conn:
            print(f"Event: {evt.type}")
            if evt.type == ServerEventType.RESPONSE_DONE:
                break

asyncio.run(main())
```

Exercise - Develop an Azure AI Voice Live agent

In this exercise, you complete a Flask-based Python web app based that enables real-time voice interactions with an agent. You add the code to initialize the session, and handle session events. You use a deployment script that: deploys the AI model; creates an image of the app in Azure Container Registry (ACR) using ACR tasks; and then creates an Azure App Service instance that pulls the image. To test the app, you need an audio device with microphone and speaker capabilities.

While this exercise is based on Python, you can develop similar applications other language-specific SDKs; including:

- Azure VoiceLive client library for .NET

Tasks performed in this exercise:

- Download the base files for the app
- Add code to complete the web app
- Review the overall code base
- Update and run the deployment script
- View and test the application

This exercise takes approximately 30 minutes to complete.

Before you start

To complete the exercise, you need:

- An Azure subscription. If you don't already have one, you can sign up for one https://azure.microsoft.com/.
- An audio device with microphone and speaker capabilities.

Get started

Select the Launch Exercise button to open the exercise instructions in a new browser window. When you're finished with the exercise, return here to:

- Complete the module
- Earn a badge for completing this module

Develop an Azure AI Voice Live voice agent

In this exercise, you complete a Flask-based Python web app based that enables real-time voice interactions with an agent. You add the code to initialize the session, and handle session events. You use a deployment script that: deploys the AI model; creates an image of the app in Azure Container Registry (ACR) using ACR tasks; and then creates an Azure App Service instance that pulls the the image. To test the app you will need an audio device with microphone and speaker capabilities.

Module assessment

1. What are the two authentication methods supported by the Voice Live API? Microsoft Entra (keyless) and API key
2. Which scope is required when generating a token for Microsoft Entra authentication? https://cognitiveservices.azure.com/.default
3. Which protocol is used for avatar streaming integration in Voice Live API? WebRTC

4. Which event should be handled to stop audio playback when a user interrupts the voice agent? ServerEventType.INPUT_AUDIO_BUFFER_SPEECH_STARTED
5. What is the recommended authentication method for production applications using the SDK? Microsoft Entra authentication with DefaultAzureCredential

Summary

In this module, you learned about the Voice live API's features, including WebSocket connections, speech recognition, text-to-speech synthesis, and avatar streaming. You also explored Azure AI Voice Live for creating real-time speech-to-speech applications using Python, including setting up the client library and managing sessions. Additionally, you learned how to implement event handlers in Python for dynamic responses and real-time audio processing. Finally, you developed a Python-based web application using Flask, integrated it with Azure resources, and tested the application.

Additional reading

- What is the Speech service?
- How to customize voice live input and output