**Analyze text with Azure AI Language**

**Introduction**

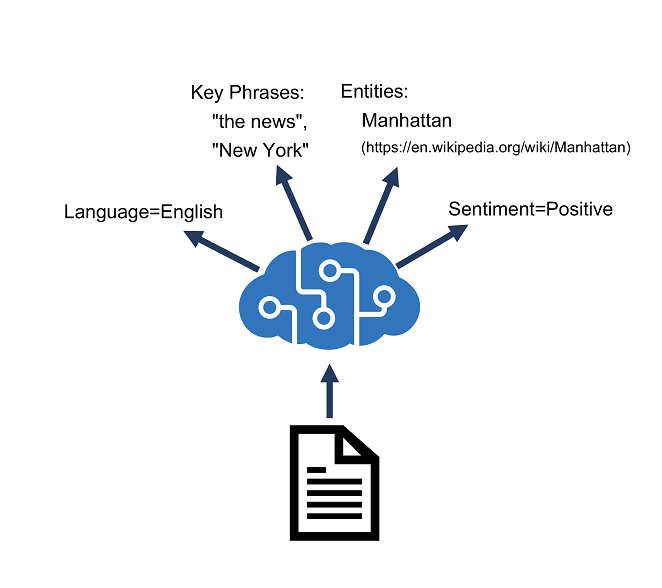
Every day, the world generates a vast quantity of data; much of it text-based in the form of emails, social media posts, online reviews, business documents, and more. Artificial intelligence techniques that apply statistical and semantic models enable you to create applications that extract meaning and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

**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:

* 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.

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**

Completed100 XP

* 5 minutes

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.

{

"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.

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.

{

"documents": [

{

"id": "1",

"text": "Hello, I would like to take a class at your University. ¿Se ofrecen clases en español? Es mi primera lengua y más fácil para escribir. Que diriez-vous des cours en français?"

}

]

}

The following sample shows a response for this multi-language example.

{

"documents": [

{

"id": "1",

"detectedLanguage": {

"name": "Spanish",

"iso6391Name": "es",

"confidenceScore": 0.9375

},

"warnings": []

}

],

"errors": [],

"modelVersion": "2022-10-01"

}

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.

{

"documents": [

{

"id": "1",

"detectedLanguage": {

"name": "(Unknown)",

"iso6391Name": "(Unknown)",

"confidenceScore": 0.0

},

"warnings": []

}

],

"errors": [],

"modelVersion": "2022-10-01"

}

**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.

{

"kind": "KeyPhraseExtraction",

"parameters": {

"modelVersion": "latest"

},

"analysisInput":{

"documents":[

{

"id": "1",

"language": "en",

"text": "You must be the change you wish

to see in the world."

},

{

"id": "2",

"language": "en",

"text": "The journey of a thousand miles

begins with a single step."

}

]

}

}

The response contains a list of key phrases detected in each document:

{

"kind": "KeyPhraseExtractionResults",

"results": {

"documents": [

{

"id": "1",

"keyPhrases": [

"change",

"world"

],

"warnings": []

},

{

"id": "2",

"keyPhrases": [

"miles",

"single step",

"journey"

],

"warnings": []

}

],

"errors": [],

"modelVersion": "2021-06-01"

}

}

**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:

{

"kind": "SentimentAnalysis",

"parameters": {

"modelVersion": "latest"

},

"analysisInput": {

"documents": [

{

"id": "1",

"language": "en",

"text": "Good morning!"

}

]

}

}

The response from the service might look like this:

{

"kind": "SentimentAnalysisResults",

"results": {

"documents": [

{

"id": "1",

"sentiment": "positive",

"confidenceScores": {

"positive": 0.89,

"neutral": 0.1,

"negative": 0.01

},

"sentences": [

{

"sentiment": "positive",

"confidenceScores": {

"positive": 0.89,

"neutral": 0.1,

"negative": 0.01

},

"offset": 0,

"length": 13,

"text": "Good morning!"

}

],

"warnings": []

}

],

"errors": [],

"modelVersion": "2022-11-01"

}

}

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**](https://learn.microsoft.com/en-us/azure/ai-services/language-service/named-entity-recognition/concepts/named-entity-categories?tabs=ga-api).

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"

}

}

**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 link <https://en.wikipedia.org/wiki/Venus>; while "Venus, the goddess of beauty" is associated with <https://en.wikipedia.org/wiki/Venus_(mythology)>.

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:

{

"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"

}

}

**Knowledge check**

Top of Form

**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 Language service to extract key phrases.

Use the Azure AI Language service to perform sentiment analysis of the comments.

**Correct. Sentiment analysis helps you determine if text is negative or positive.**

Use the Azure AI Language service to extract named entities from 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 key phrases.

Use the Azure AI Language service to detect the language of the text.

Use the Azure AI Language service to extract linked entities.

**Correct. Linked entities enable you to disambiguate common entities of the same name**

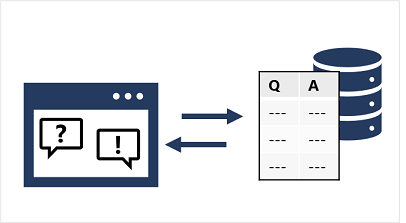
**Create question answering solutions with Azure AI Language**

**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.

**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**](https://learn.microsoft.com/en-us/azure/ai-services/language-service/question-answering/how-to/migrate-qnamaker).

**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:

Bottom of Form

|  |  |  |
| --- | --- | --- |
|  | **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 |
| **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**

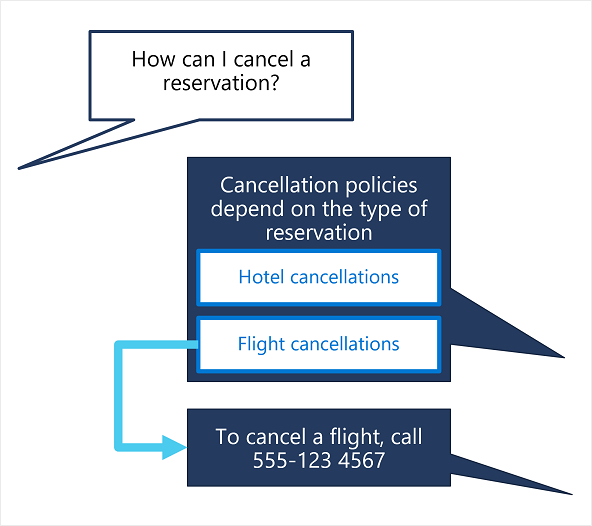
To create a question answering solution, you can use the REST API or SDK to write code that defines, trains, and publishes the knowledge base. However, it's more common to use the [Language Studio](https://language.azure.com/) web interface to define and manage a knowledge base.

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.



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.

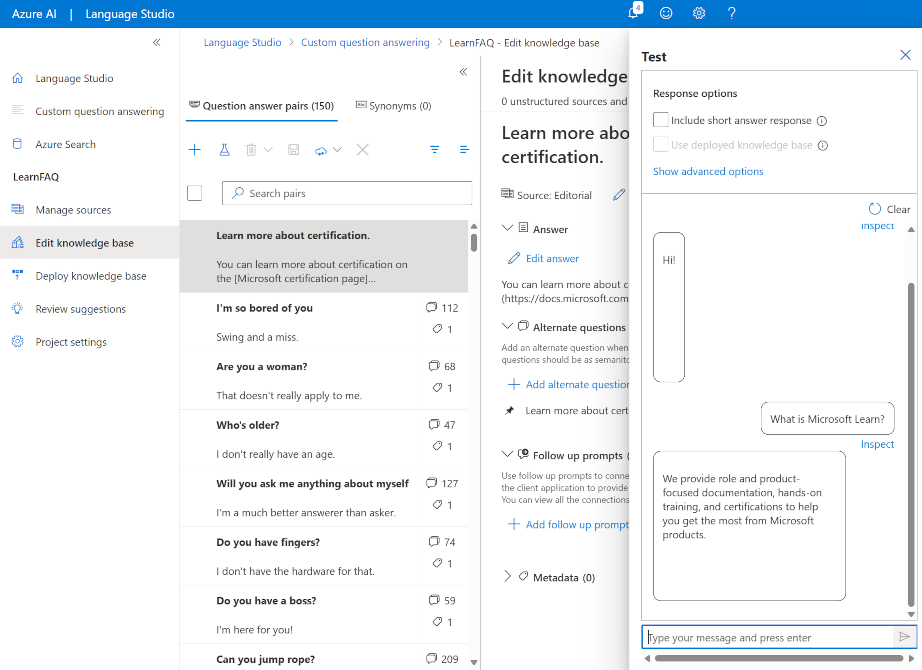
When you define a follow-up prompt for multi-turn conversation, you can link to an existing answer in the knowledge base or define a new answer specifically for the follow-up. You can also restrict the linked answer so that it is only ever displayed in the context of the multi-turn conversation initiated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

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/create-question-answer-solution-ai-language/media/test-new.png#lightbox)

**Deploying a knowledge base**

When you are 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:

{

"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**

Completed100 XP

* 6 minutes

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

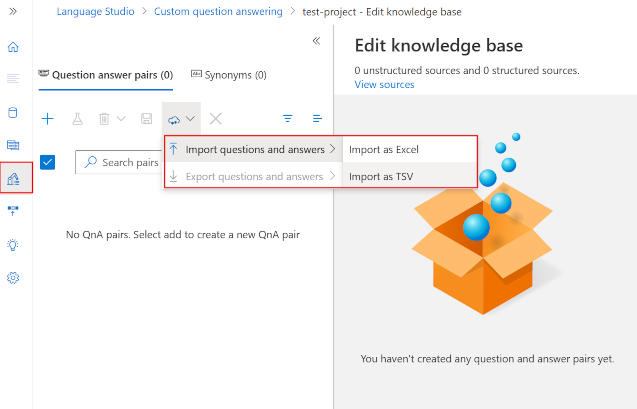
**Use active learning**

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:

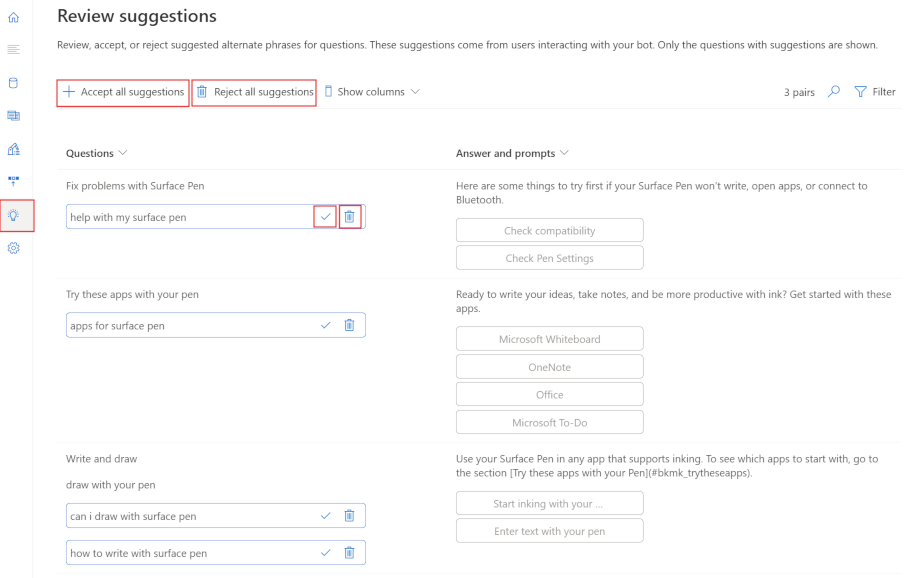
**Create your question and answer pairs**

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.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/create-question-answer-solution-ai-language/media/import-file.png#lightbox)

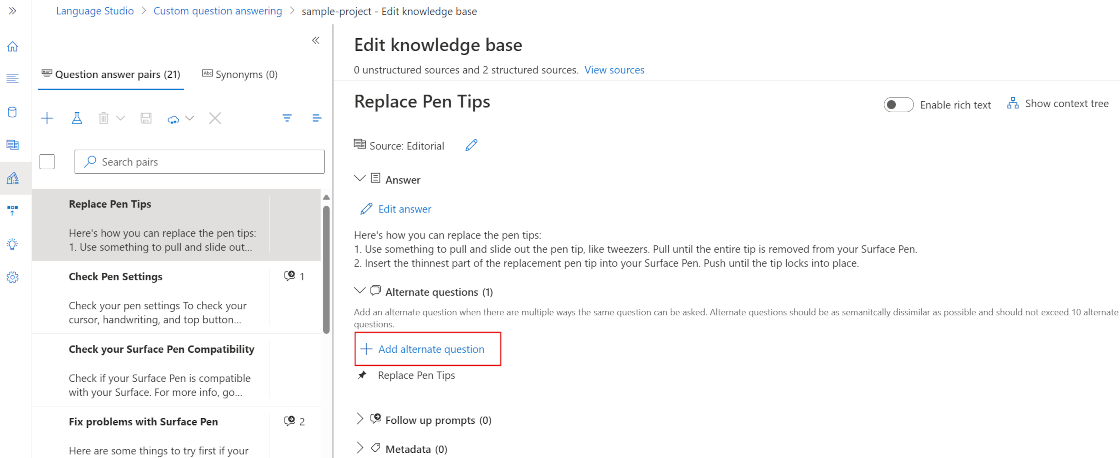
**Review suggestions**

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:

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/create-question-answer-solution-ai-language/media/review-suggestions.png#lightbox)

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:

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/create-question-answer-solution-ai-language/media/add-alternate-questions-manual.png#lightbox)

**Note**

To learn more about active learning, see [**Enrich your project with active learning**](https://learn.microsoft.com/en-us/azure/ai-services/language-service/question-answering/tutorials/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"

]

}

]

}

**Check your knowledge**

Top of Form

**1. You want to create a knowledge base from an existing FAQ document. What should you do?**

Create an empty knowledge base and manually enter the FAQ questions and answers.

Create a new knowledge base, importing the existing FAQ document.

**Correct. You can create a knowledge base from an existing document or web page.**

Create a new knowledge base, selecting only the Professional chit-chat source.

**2. How can you add a multi-turn context for a question in an existing knowledge base?**

Add synonyms to the knowledge base.

Add alternative phrasing to the question.

Add a follow-up prompt to the question.

**Correct. To add a multi-turn context to a question, define a follow-up prompt.**

**3. How can you enable users to use your knowledge base through email?**

Add Friendly Chit-chat to the knowledge base.

Enable Active Learning for the knowledge base and include the user's email address as the userId parameter in responses.

Create a bot based on your knowledge base and configure an email channel.

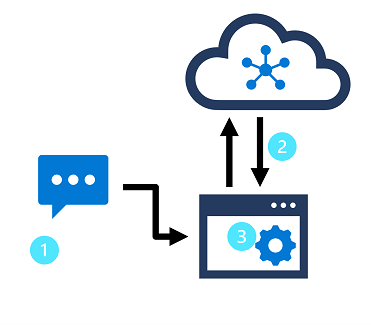
**Correct. You can create a bot for your published knowledge base and configure a channel for email communication.**

**Build a conversational language understanding model**

**Introduction**

*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.

Bottom of Form

**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](https://learn.microsoft.com/en-us/azure/cognitive-services/language-service/overview) 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**](https://learn.microsoft.com/en-us/azure/ai-services/language-service/conversational-language-understanding/how-to/tag-utterances), which includes valuable guidelines for tagging data. Also see the [**evaluation metrics**](https://learn.microsoft.com/en-us/azure/ai-services/language-service/custom-text-classification/concepts/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](https://learn.microsoft.com/en-us/azure/ai-services/language-service/overview).

**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.

**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.

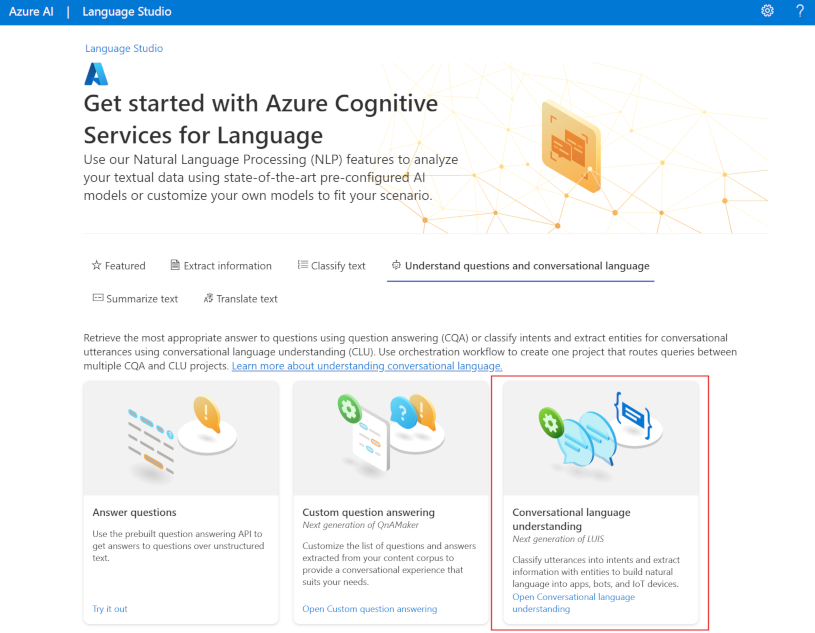
First, you'll need to create your Azure AI Language resource in the [Azure portal](https://portal.azure.com/). Then:

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](https://aka.ms/languageStudio) 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.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/build-language-understanding-model/media/language-studio-conversational.png#lightbox)

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](https://learn.microsoft.com/en-us/azure/ai-services/language-service/language-studio)

**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.

Expand table

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

**Request deployment**

Submit a **POST** request to the following endpoint.

rest

{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMENT-NAME}?api-version={API-VERSION}

| **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.

JSON

{

"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:

rest

{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMENT-NAME}/jobs/{JOB-ID}?api-version={API-VERSION}

**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.

rest

{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMENT-NAME}/jobs/{JOB-ID}?api-version={API-VERSION}

| **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](https://learn.microsoft.com/en-us/azure/ai-services/language-service/conversational-language-understanding/quickstart?pivots=rest-api#create-a-clu-project).

**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.

Python

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.

Python

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

rest

{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.

JSON

{

"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.

rest

{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 calling |

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

JSON

{

"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](https://learn.microsoft.com/en-us/azure/ai-services/language-service/) 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:

Expand table

| **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](https://learn.microsoft.com/en-us/azure/ai-services/language-service/conversational-language-understanding/prebuilt-component-reference).

**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.

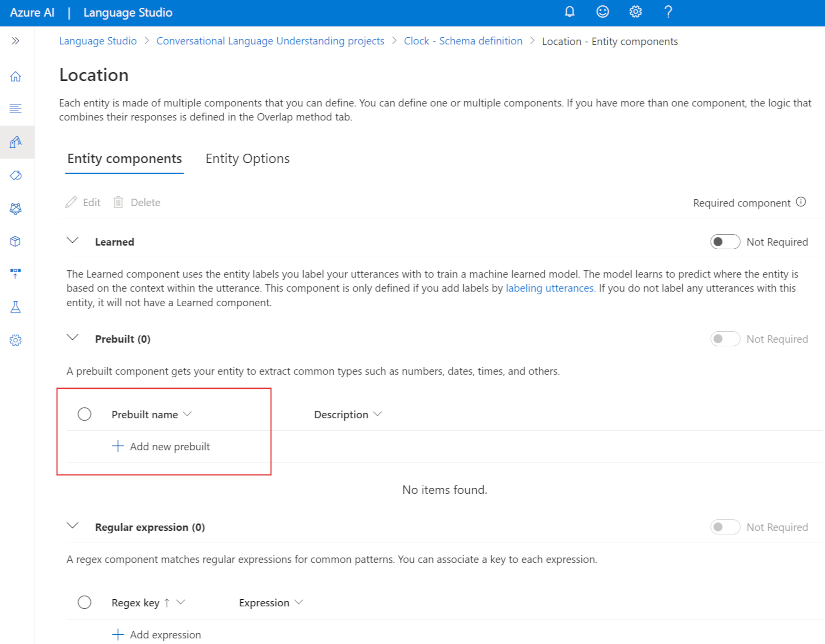
**Use pre-built entity components**

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.](https://learn.microsoft.com/en-us/azure/ai-services/language-service/conversational-language-understanding/prebuilt-component-reference)

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.

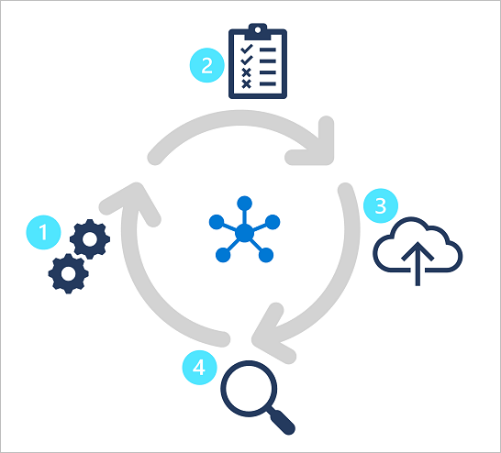
To add a prebuilt component, you can create an entity in your project, then select **Add new prebuilt** to that entity to detect certain entities.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/build-language-understanding-model/media/add-prebuilt-entity.png#lightbox)

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:



1. Train a model to learn intents and entities from sample utterances.
2. Test the model interactively or using a testing dataset with known labels
3. Deploy a trained model to a public endpoint so client apps can use it
4. 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.

**Check your knowledge**

Top of Form

**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?**

Intents.

Utterances.

**Correct. Utterances are example phrases that indicate a specific intent.**

Entities.

**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 intent.

As an utterance.

As an entity.

**Correct. The city is an entity to which the intent (booking a flight) should be applied.**

**3. Your language model needs to detect an email when present in an utterance. What is the simplest way to extract that email?**

Use Regular Expression entities.

Use prebuilt entity components.

**Correct. When a language model needs to detect a common entity, use prebuilt components to have the Azure AI Language service automatically detect the entity.**

Use Learned entity components.

**Create a custom text classification solution**

**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.

**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.

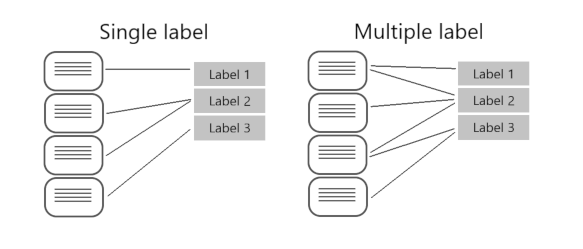
**Single vs. multiple label projects**

Beyond the ability to put files into multiple classifications, the key differences with multiple label classification projects are labeling, considerations for improving your model, and 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.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/custom-text-classification/media/single-multiple-graphic.png#lightbox)

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

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:

{

"projectFileVersion": "<API-VERSION>",

"stringIndexType": "Utf16CodeUnit",

"metadata": {

"projectName": "<PROJECT-NAME>",

"storageInputContainerName": "<CONTAINER-NAME>",

"projectKind": "customSingleLabelClassification",

"description": "Trying out custom multi 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

{

"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"

}

]

}

]

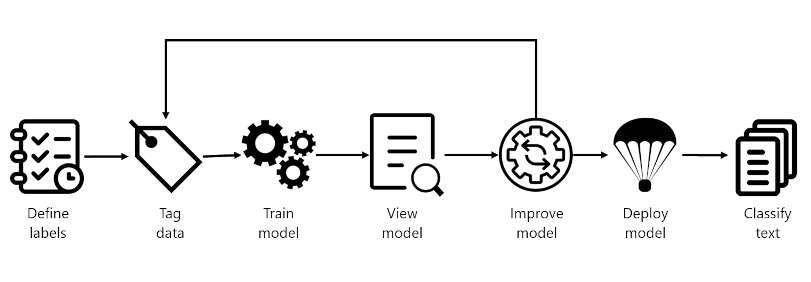
}

}

**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**

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/custom-text-classification/media/classify-development-lifecycle.png#lightbox)

* **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](https://aka.ms/ct-runtime-swagger)

**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:

{

"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-VERSION>

| **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-REQUIRED> | Which task you're requesting. The task is CustomMultiLabelClassification for multiple label projects, or CustomSingleLabelClassification for single 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:

{

"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.

**Check your knowledge**

Top of Form

**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 single label classification project

A multiple label classification project

**That answer's correct. Use a multiple label classification project to label books as multiple genres.**

A varied label classification project

**2. You just got notification your training job is complete. What is your next step?**

Label more data

Deploy your model

View your model details

**That answer's correct. First view your model details to see how it scored, the classification distribution, and where it needs improvement.**

**3. You want to submit a classification task via the API. How do you get the results of the classification?**

The result is in the response of the classification request.

Call an endpoint with your deployment name to get the most recent classification.

Call the URL provided in the header of the request response.

**That answer's correct. Get the value from the operation-location header in the request response, and use that to retrieve the results of the classification request.**

**Custom named entity recognition**

**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.

**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://<your-resource>.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

}

]

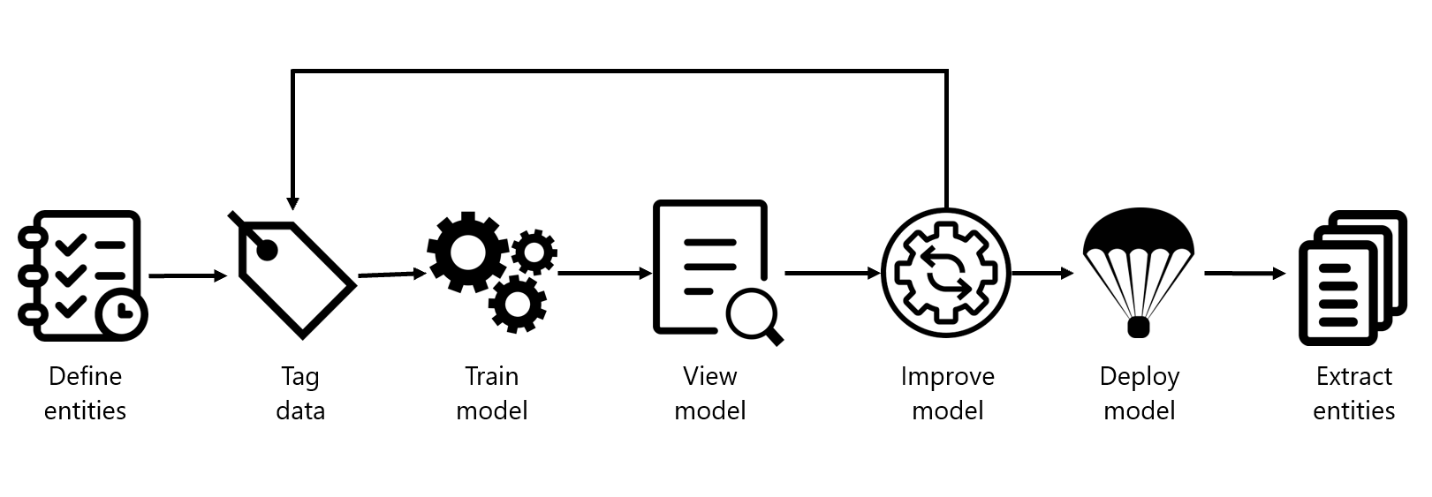
<...> Bottom of Form

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

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**

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/custom-name-entity-recognition/media/extraction-development-lifecycle.png#lightbox) Bottom of Form

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

1. **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.
2. **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.
3. **Train model**: Train your model once your entities are labeled. Training teaches your model how to recognize the entities you label.
4. **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).
5. **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).
6. **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.
7. **Extract entities**: Use your model for extracting entities. The lab covers how to use the API, and you can view the [API reference](https://aka.ms/ct-runtime-swagger) 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](https://learn.microsoft.com/en-us/azure/ai-services/language-service/concepts/data-limits) page for detailed information.

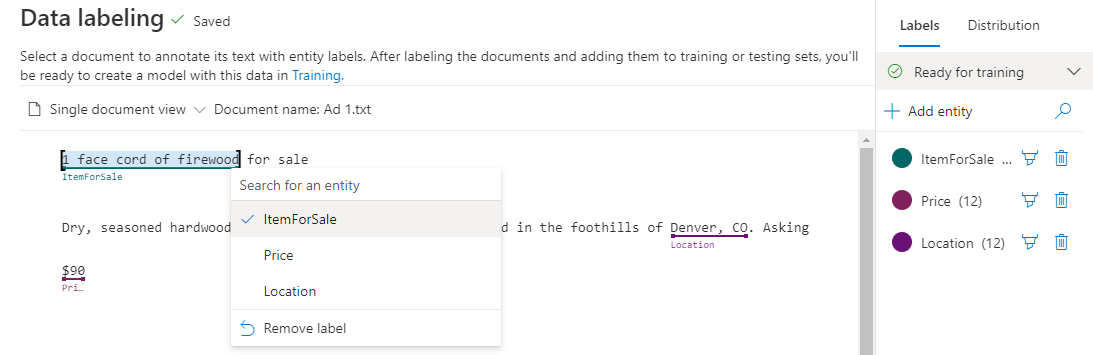
**Label your data**

Completed100 XP

* 4 minutes

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.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/custom-name-entity-recognition/media/tag-entity-screenshot.png#lightbox)

**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](https://learn.microsoft.com/en-us/azure/ai-services/language-service/custom-named-entity-recognition/concepts/data-formats). For example:

{

"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

}

]

}

]

}

]

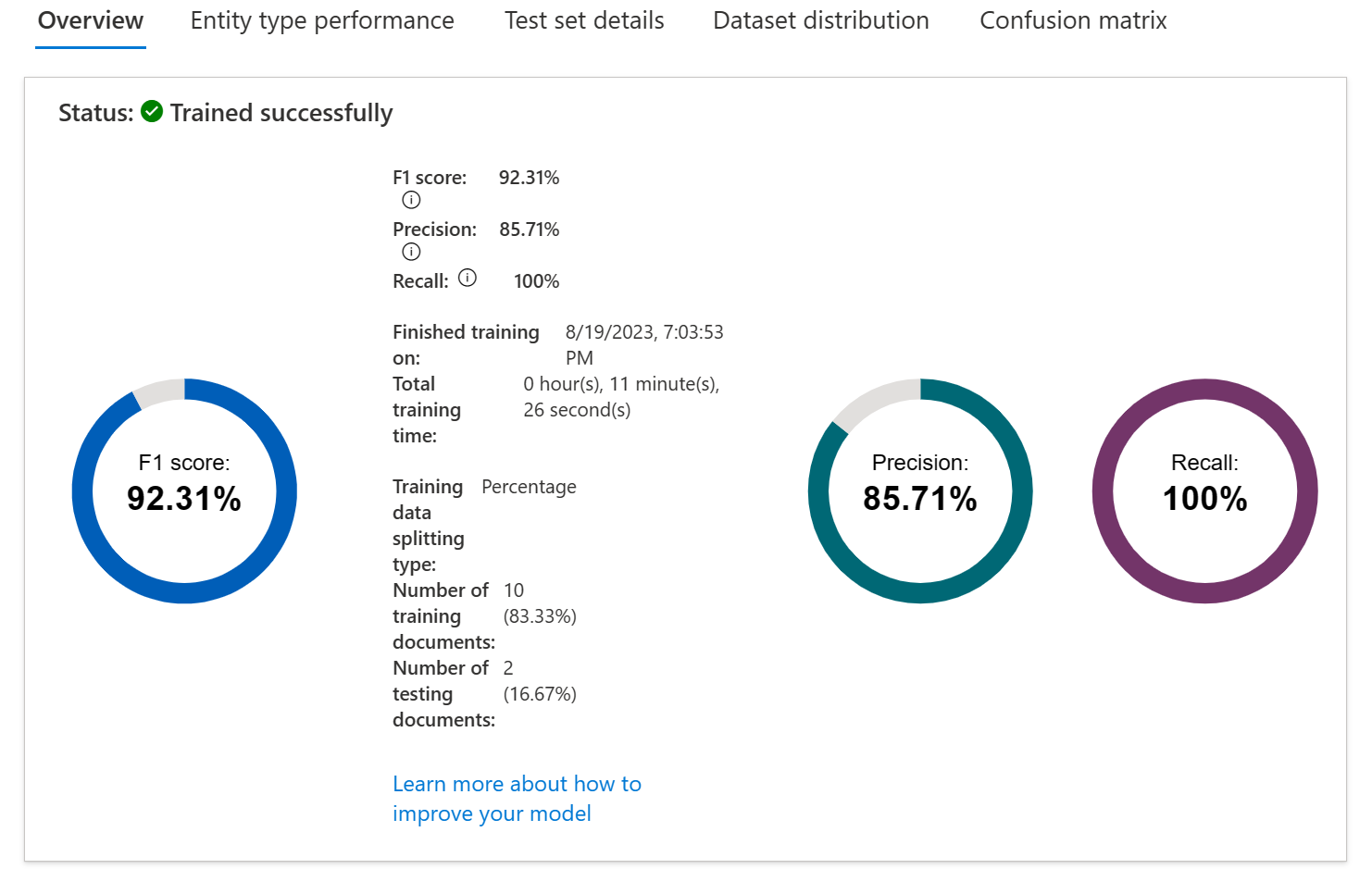
}

}

| **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 |
| 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.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/custom-name-entity-recognition/media/model-scoring-new.png#lightbox)

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. |
| 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 |
| F1 score | Combination of precision and recall providing a single scoring metric |

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**

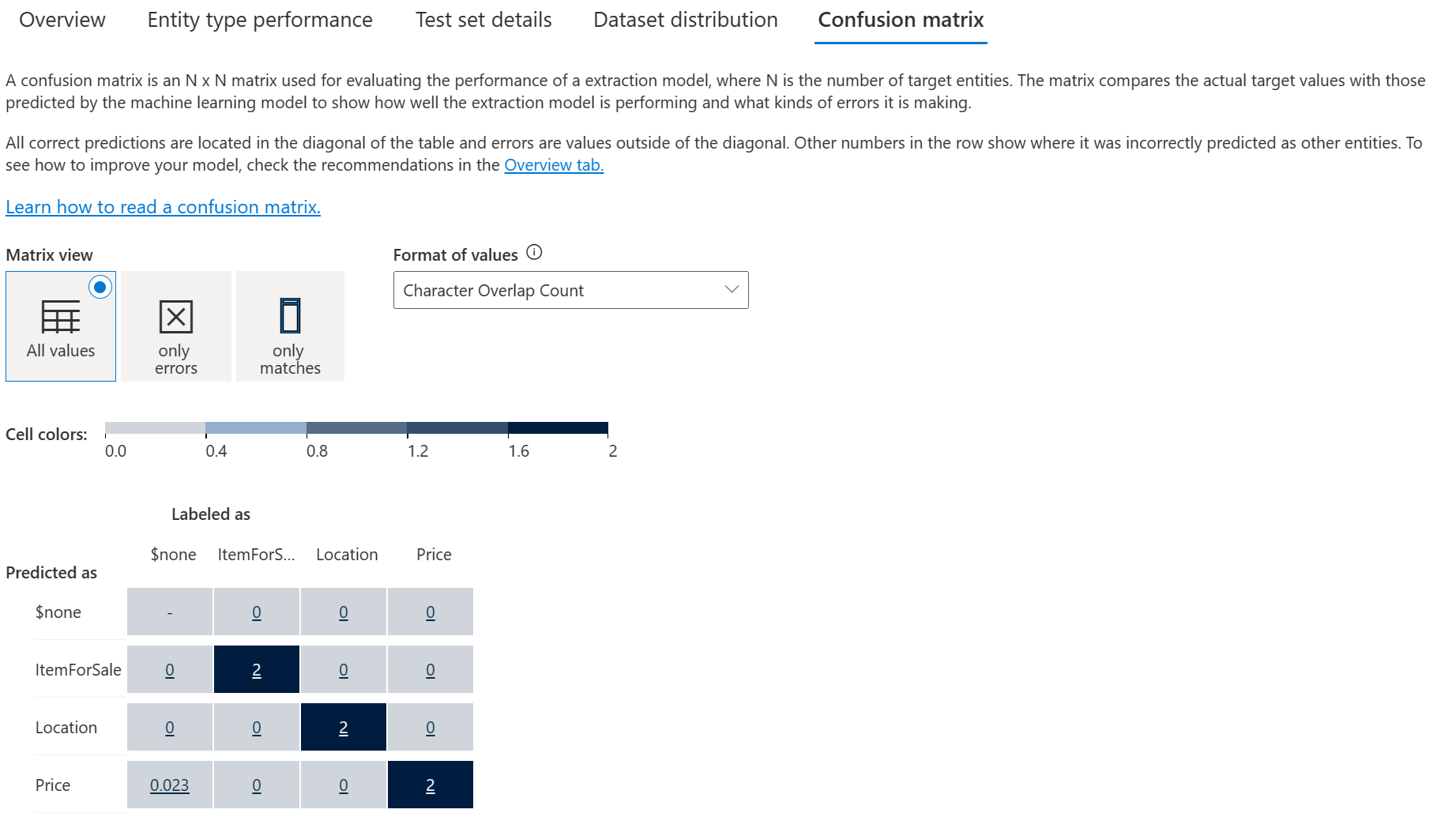
Ideally we want our model to score well in both precision and recall, which means the entity recognition works well. If both metrics have a low score, it means the model is both struggling to recognize entities in the document, and when it does extract that entity, it doesn't assign it the correct label 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.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/custom-name-entity-recognition/media/model-confusion-matrix-new.png#lightbox)

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

**Check your knowledge**

Top of Form

**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

**That answer's correct. Recall indicates how well the model extracts entities, regardless of which entity that is.**

Precision

F1 score

**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

**That answer's correct. The JSON file lives next to the dataset in your container for the model to use during training.**

XML file, in my local project folder

YAML file, anywhere in my Azure account

**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?**

Distribution

Accuracy

Diversity

**That answer's correct. Having the right data diversity will lead to better extraction performance**

**Translate text with Azure AI Translator service**

**Introduction**

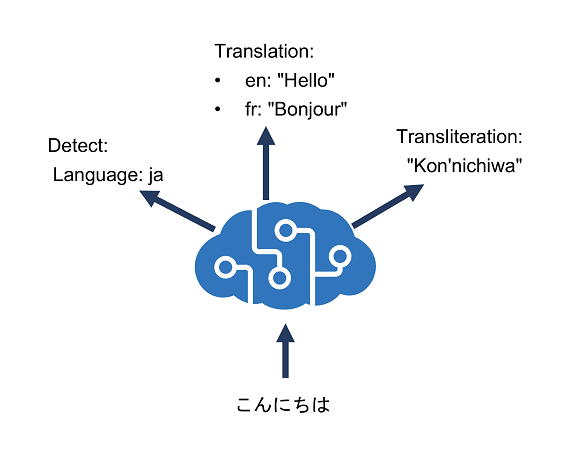
There are many commonly used languages throughout the world, and the ability to exchange information between speakers of different languages is often a critical requirement for global solutions.

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 Analytics API in a multi-service Azure AI Services resource.

After you have provisioned 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.

Bottom of Form

**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 https://api.cognitive.microsofttranslator.com/detect?api-version=3.0 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-Apim-Subscription-Region: <your-service-region>" -H "Ocp-Apim-Subscription-Key: <your-key>" -H "Content-Type: application/json" -d "[{ 'Text' : 'こんにちは' }]

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&to=fr&to=en" -H "Ocp-Apim-Subscription-Key: <your-key>" -H "Ocp-Apim-Subscription-Region: <your-service-region>" -H "Content-Type: application/json; charset=UTF-8" -d "[{ 'Text' : 'こんにちは' }]"

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&fromScript=Jpan&toScript=Latn" -H "Ocp-Apim-Subscription-Key: <your-key>" -H "Ocp-Apim-Subscription-Region: <your-service-region>" -H "Content-Type: application/json" -d "[{ 'Text' : 'こんにちは' }]"

[

{

"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 ▇▇▇▇ 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"

}

]

}

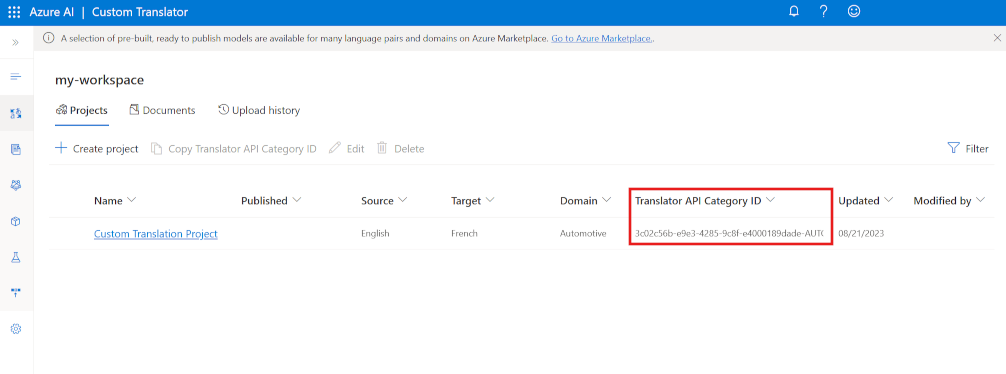
]

**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:

1. [Create a workspace](https://learn.microsoft.com/en-us/azure/ai-services/translator/custom-translator/quickstart) linked to your Azure AI Translator resource.
2. [Create a project](https://learn.microsoft.com/en-us/azure/ai-services/translator/custom-translator/quickstart).
3. [Upload training data files](https://learn.microsoft.com/en-us/azure/ai-services/translator/custom-translator/quickstart) and [train a model](https://learn.microsoft.com/en-us/azure/ai-services/translator/custom-translator/quickstart).
4. [Test your model](https://learn.microsoft.com/en-us/azure/ai-services/translator/custom-translator/quickstart) and [publish your model](https://learn.microsoft.com/en-us/azure/ai-services/translator/custom-translator/quickstart).
5. Make translation calls to the API.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/translate-text-with-translator-service/media/custom-translator-new.png#lightbox)

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&to=nl&category=<category-id>" -H "Ocp-Apim-Subscription-Key: <your-key" -H "Content-Type: application/json; charset=UTF-8" -d "[{'Text':'Where can I find my employee details?'}]"

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](https://learn.microsoft.com/en-us/azure/ai-services/translator/reference/v3-0-translate?tabs=curl) for a full list of codes and their explanation.

**Knowledge check**

Top of Form

**1. What function of Azure AI Translator should you use to convert the Chinese word "你好" to the English word "Hello"?**

Detect

Translate

**Correct. Translation converts text from one language to another.**

Transliterate

**2. What function of Azure AI Translator should you use to convert the Russian word "спасибо" in Cyrillic characters to "spasibo" in Latin characters?**

Detect

Translate

Transliterate

**Correct. Transliteration converts text from one script to another.**

**Create speech-enabled apps with Azure AI services**

**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.
* **Speaker Recognition**: An API that enables your application to recognize individual speakers based on their voice.
* **Intent Recognition**: An API that uses conversational language understanding to determine the semantic meaning of spoken input.Bottom of Form

**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 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.

**Use the Azure AI Speech to Text API**

The Azure AI Speech service supports speech recognition through two REST APIs:

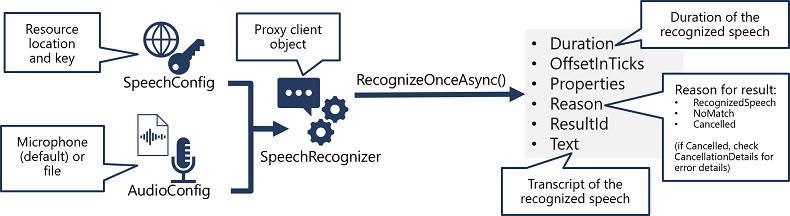
* The **Speech to text** API, which is the primary way to perform speech recognition.
* The **Speech to text Short Audio** API, which is optimized for short streams of audio (up to 60 seconds).

You can use either API for interactive speech recognition, depending on the expected length of the spoken input. You can also use the **Speech to text** API for *batch transcription*, transcribing multiple audio files to text as a batch operation.

You can learn more about the REST APIs in the [Speech to text REST API documentation](https://learn.microsoft.com/en-us/azure/ai-services/speech-service/rest-speech-to-text). In practice, most interactive speech-enabled applications use the Speech service through a (programming) language-specific SDK.

**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:



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

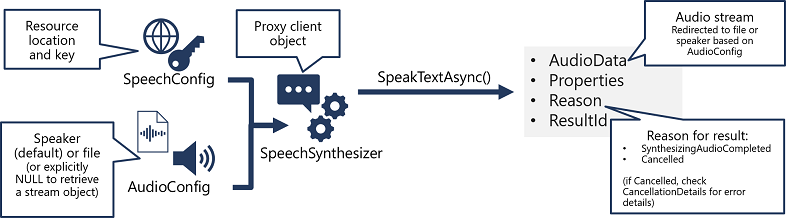
* The **Text to speech** API, which is the primary way to perform speech synthesis.
* The **Batch synthesis** API, which is designed to support batch operations that convert large volumes of text to audio - for example to generate an audio-book from the source text.

You can learn more about the REST APIs in the [Text to speech REST API documentation](https://learn.microsoft.com/en-us/azure/ai-services/speech-service/batch-synthesis). 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:



1. Use a **SpeechConfig** object to encapsulate the information required to connect to your Azure AI Speech resource. Specifically, its **location** and **key**.
2. 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.
3. Use the **SpeechConfig** and **AudioConfig** to create a **SpeechSynthesizer** object. This object is a proxy client for the **Text to speech** API.
4. 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.
5. 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

The supported formats are indicated in the SDK using the **SpeechSynthesisOutputFormat** enumeration. For example, SpeechSynthesisOutputFormat.Riff24Khz16BitMonoPcm.

To specify the required output format, use the **SetSpeechSynthesisOutputFormat** method of the **SpeechConfig** object:

C#Copy

speechConfig.SetSpeechSynthesisOutputFormat(SpeechSynthesisOutputFormat.Riff24Khz16BitMonoPcm);

For a full list of supported formats and their enumeration values, see the [Azure AI Speech SDK documentation](https://learn.microsoft.com/en-us/dotnet/api/microsoft.cognitiveservices.speech.speechsynthesisoutputformat).

**Voices**

The Azure AI Speech service provides multiple voices that you can use to personalize your speech-enabled applications. There are two kinds of voice that you can use:

* *Standard voices* - synthetic voices created from audio samples.
* *Neural voices* - more natural sounding voices created using deep neural networks.

Voices are identified by names that indicate a locale and a person's name - for example en-GB-George.

To specify a voice for speech synthesis in the **SpeechConfig**, set its **SpeechSynthesisVoiceName** property to the voice you want to use:

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

**Use Speech Synthesis Markup Language**

While the Azure AI Speech SDK enables you to submit plain text to be synthesized into speech (for example, by using the **SpeakTextAsync()** method), 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:

XML

<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 the **SpeakSsmlAsync()** method, like this:

speechSynthesizer.SpeakSsmlAsync(ssml\_string);

**Knowledge check**

Top of Form

**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

**Correct. The Azure AI Speech SDK requires the location and a key to connect to the Azure AI Speech service.**

The primary and secondary keys

The endpoint and one of the keys

**2. Which object should you use to specify that the speech input to be transcribed to text is in an audio file?**

SpeechConfig

AudioConfig

**Correct. Use an AudioConfig to specify the input source for speech.**

SpeechRecognizer

**3. How can you change the voice used in speech synthesis?**

Specify a SpeechSynthesisOutputFormat enumeration in the SpeechConfig object.

Set the SpeechSynthesisVoiceName property of the SpeechConfig object to the desired voice name.

**Correct. To set a voice, set the SpeechSynthesisVoiceName property of the SpeechConfig to a voice name, such as "en-GB-George".**

Specify a filename in the AudioConfig object.

**Translate speech with the Azure AI Speech service**

**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.

**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.

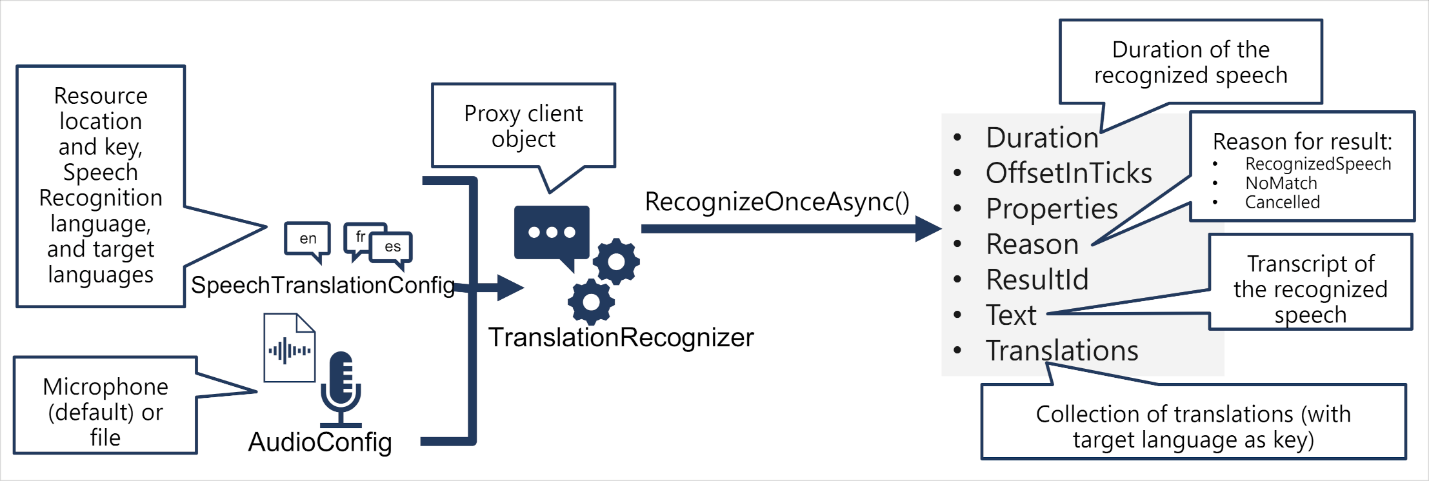
After creating your Azure resource, you'll need the following information to use it from a client application through one of the supported SDKs:

* 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:

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/translate-speech-speech-service/media/translate-speech.png#lightbox)

1. Use a **SpeechTranslationConfig** object to encapsulate the information required to connect to your Azure AI Speech resource. Specifically, its location and key.
2. 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.
3. 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.
4. Use the **SpeechTranslationConfig**, and **AudioConfig** to create a **TranslationRecognizer** object. This object is a proxy client for the Azure AI Speech translation API.
5. 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.
6. 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#](https://learn.microsoft.com/en-us/azure/ai-services/speech-service/get-started-speech-translation?pivots=programming-language-csharp) and [Python](https://learn.microsoft.com/en-us/azure/ai-services/speech-service/get-started-speech-translation?pivots=programming-language-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:

1. Use a **TranslationRecognizer** to translate spoken input into text transcriptions in one or more target languages.
2. Iterate through the **Translations** dictionary in the result of the translation operation, using a **SpeechSynthesizer** to synthesize an audio stream for each language.Bottom of Form

**Knowledge check**

Completed200 XP

* **Module assessment**
* 3 minutes

 Great job! You passed the module assessment.

Dismiss alert

Top of Form

**1. Which SDK object should you use to specify the language(s) into which you want speech translated?**

SpeechConfig

SpeechTranslationConfig

**Correct. Specify target languages in the SpeechTranslationConfig object.**

AudioConfig

**2. Which SDK object should you use as a proxy for the Translation API of Azure AI Speech service?**

TranslationRecognizer

**Correct. Use a TranslationRecognizer to call the Translation API of the Azure AI Speech service.**

SpeechRecognizer

SpeechSynthesizer

**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.

**Correct. You can only use event-based synthesis when translating to a single target language.**

Only when translating to multiple target languages.

When translating to one or more target languages.

Bottom of Form