

Azure for Python Developers

You can deploy your Python applications to Azure in various environments, including web apps, serverless functions, containers, and machine learning models. Azure offers flexible hosting options to suit different architectural and scalability needs. To interact with Azure services programmatically, use the Azure SDK for Python. This set of libraries streamlines tasks such as authentication, resource management, and service integration —making it easier to build secure, scalable cloud applications with Python.

Azure libraries (SDK)

GET STARTED

[Get started](#)

[Get to know the Azure libraries](#)

[Learn library usage patterns](#)

[Authenticate with Azure services](#)

Web apps

TUTORIAL

[Quickly create and deploy new Django / Flask / FastAPI apps](#)

[Deploy a Django or Flask Web App](#)

[Deploy Web App with PostgreSQL](#)

[Deploy Web App with Managed Identity](#)

[Deploy using GitHub Actions](#)

AI

QUICKSTART

[Develop using Azure AI services](#)

[Python enterprise RAG chat sample](#)

Containers



[Python containers overview](#)

[Deploy to App Service](#)

[Deploy to Container Apps](#)

[Deploy a Kubernetes cluster](#)

Data and storage



[SQL databases](#)

[Tables, blobs, files, NoSQL](#)

[Big data and analytics](#)

Machine learning



[Create an ML experiment](#)

[Train a prediction model](#)

[Create ML pipelines](#)

[Use ready-made AI services \(face, speech, text, image, etc.\)](#)

[Serverless, Cloud ETL](#)

Serverless functions



[Deploy using Visual Studio Code](#)

[Deploy using the command line](#)

[Connect to storage using Visual Studio Code](#)

[Connect to storage using the command line](#)

Developer tools

GET STARTED

[Visual Studio Code \(IDE\) ↗](#)

[Azure Command-Line Interface \(CLI\)](#)

[Windows Subsystem for Linux \(WSL\)](#)

[Visual Studio \(for Python/C++ development\)](#)

Get started with Python on Azure

Article • 01/28/2025

If you're new to developing applications for the cloud, this short series of 8 articles is the best place to start.

- Part 1: [Azure for developers overview](#)
- Part 2: [Key Azure services for developers](#)
- Part 3: [Hosting applications on Azure](#)
- Part 4: [Connect your app to Azure services](#)
- Part 5: [How do I create and manage resources in Azure?](#)
- Part 6: [Key concepts for building Azure apps](#)
- Part 7: [How am I billed?](#)
- Part 8: [Versioning policy for Azure services, SDKs, and CLI tools](#)

Create an Azure Account

To develop Python applications with Azure, you need an Azure account. Your Azure account is the credentials you use to sign-in to Azure with and what you use to create Azure resources.

If you're using Azure at work, talk to your company's cloud administrator to get your credentials used to sign-in to Azure.

Otherwise, you can create an [Azure account for free](#) and receive 12 months of popular services for free and a \$200 credit to explore Azure for 30 days.

[Create an Azure account for free](#)

Create and manage resources

To use Azure resources like databases, message queues, file storage, and so on, you must first create an instance of the resource. Creating resources involves:

- choosing capacity or computing options
- adding the new resource to a resource group
- selecting the region of the world where the service runs
- giving the service a unique name

There are several tools you can use to create and manage Azure resources, depending on your scenario:

- [Azure portal](#) - If you're new to Azure and want a web-based user interface to create and manage a couple of resources.
- [Azure CLI](#) - If you're more comfortable with command line interfaces.
- [Azure PowerShell](#) - If you prefer a PowerShell style syntax in their CLI.
- [Azure Developer CLI](#) - When you want to create repeatable deployments involving many Azure resources with intricate dependencies. Requires learning Bicep templates.
- [Azure Tools extension pack](#) - The extension pack contains extensions for working with some of the most popular Azure services in one convenient package.

You can also use the [Azure Management Libraries for Python](#) to create and manage resources. The management libraries allow you to use Python to implement custom deployment and management functionality. Here are a few articles that can help you get started:

- [Create a resource group](#)
- [List groups and resources](#)
- [Create Azure storage](#)
- [Create and deploy a web app](#)
- [Create and query a database](#)
- [Create a virtual machine](#)

Write your Python app

Developing on Azure requires [Python](#) 3.8 or higher. To verify the version of Python on your workstation, in a console window type the command `python3 --version` for macOS/Linux or `py --version` for Windows.

Use your favorite tools to write your Python app. If you use Visual Studio Code, you should try the [Python extension for Visual Studio Code](#).

Most of the instructions in this set of articles use a virtual environment because it's a best practice. Feel free to use any virtual environment you want, but the article instructions standardize on `venv`.

Use client libraries

As you're getting started, the articles instruct you on which Python on Azure libraries to install and reference using the `pip` utility.

At some point, you might want to [install and reference](#) the [Azure SDK for Python](#) client [libraries](#) without having to follow the instructions in an article. The [Azure SDK](#)

[Overview](#) is a great starting point.

Authenticate your app to Azure

When you use the Azure SDK for Python, you must add authentication logic to your app. How your app authenticates depends on whether you're running your app locally during development and testing, hosting the app on your own servers, or hosting the app in Azure. Read [Authenticate Python apps to Azure services by using the Azure SDK for Python](#) to understand more about authentication on Azure.

You'll also need to set up access policies that control what identities (service principals and/or application IDs) are able to access those resources. Access policies are managed through Azure [Role-Based Access Control \(RBAC\)](#); some services have more specific access controls as well. As a cloud developer working with Azure, make sure to familiarize yourself with Azure RBAC because you use it with just about any resource that has security concerns.

Add cross-cutting concerns

- Manage your application secrets using [Azure Key Vault](#)
- Gain visibility into your app by logging with [Azure Monitor](#)

Host your Python app

If you want your app code to run on Azure, you have several options as described in [Hosting applications on Azure](#).

If you're building web apps or APIs (Django, Flask, FastAPI, and so on), consider:

- [Azure App Service](#)
- [Azure App Service \(already containerized\)](#)
- [Azure Container Apps](#)
- [Azure Kubernetes cluster](#)

If you're building a web application, see [Configure your local environment for deploying Python web apps on Azure](#).

Also, if you're building a web API, you should consider using [Azure API Management](#).

If you're building back-end processes:

- [Azure Functions](#)
- [Azure App Service WebJobs](#)

- Azure Container Apps

Next steps

- Develop a Python web app
 - Develop a container app
 - Learn to use the Azure libraries for Python
-

Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Develop AI apps with Python

06/23/2025

This article offers a curated list of top learning resources for Python developers who are new to building AI applications. It includes links to quickstart guides, sample projects, official documentation, training courses, and other helpful materials.

Resources for Azure OpenAI Service

Azure OpenAI Service provides REST API access to the powerful language models available in OpenAI. Azure OpenAI helps you adapt these models to accomplish specific tasks, such as content generation, summarization, image understanding, semantic search, and natural language to code translation. Access Azure OpenAI by using the REST APIs, the Azure OpenAI SDK for .NET, or the web-based interface in [Azure OpenAI Studio](#).

SDKs and libraries

[] [Expand table](#)

Link	Description
OpenAI SDK for Python	The GitHub source code version of the OpenAI Python library, which provides convenient access to the OpenAI API from applications written in the Python language.
OpenAI Python Package	The PyPi version of the OpenAI Python library.
Switch from OpenAI to Azure OpenAI	A guidance article on the small changes you need to make to your code, so you can swap back and forth between OpenAI and the Azure OpenAI Service.
Streaming chat completions	A notebook example that demonstrates how to get chat completions to work by using the Azure endpoints. The example focuses on chat completions, but also introduces other operations available with the API.
Azure embeddings	A notebook example that demonstrates how to use embeddings with Azure endpoints. The example focuses on embeddings, but also introduces other operations available with the API.
Deploy model and generate text	An article with minimal, straightforward detailed steps to deploy a model that can programmatically chat.
OpenAI with Microsoft Entra ID role-based access control	A look at authentication by using Microsoft Entra ID and Azure role-based access control .

Link	Description
OpenAI with Azure AD-managed identities for Azure resources	An article with more complex security scenarios that require Azure role-based access control. Explore how to authenticate to your OpenAI resource with Microsoft Entra ID.
Azure OpenAI Service samples	A compilation of useful Azure OpenAI Service resources and code samples to help you get started and accelerate your technology adoption journey.

Documentation

 [Expand table](#)

Link	Description
Azure OpenAI Service documentation	The hub page for Azure OpenAI Service documentation.
Quickstart: Get started generating text with Azure OpenAI Service	A quickstart that demonstrates how to set up the services you need and write code to prompt a model by using Python.
Quickstart: Get started using GPT-35-Turbo and GPT-4 with Azure OpenAI Service	A quickstart that demonstrates how to work with system, assistant, and user roles to tailor content in response to certain questions.
Quickstart: Chat with Azure OpenAI models by using your own data	A quickstart that helps you add your own data, such as a PDF or other document.
Quickstart: Get started using Azure OpenAI Assistants (Preview)	A quickstart that demonstrates how to instruct a model to use the built-in Python code interpreter to solve math problems step by step. This example provides a starting point to use your own AI assistants accessed through custom instructions.
Quickstart: Use images in your AI chats	A quickstart that shows how to programmatically ask a model to describe the contents of an image.
Quickstart: Generate images with Azure OpenAI Service	A quickstart that demonstrates how to programmatically generate images by using Dall-E based on a prompt.

Resources for other Azure AI services

In addition to Azure OpenAI Service, there are many other Azure AI services. Developers and organizations can rapidly create intelligent, market-ready, and responsible applications with out-of-the-box and prebuilt customizable APIs and models. Example applications include

natural language processing for conversations, search, monitoring, translation, speech, vision, and decision-making.

Samples

 Expand table

Link	Description
Integrate speech into your apps with Azure AI Speech SDK Samples ↗	Samples for the Azure Cognitive Services Speech SDK. Links to samples for speech recognition, translation, speech synthesis, and more.
Azure AI Document Intelligence SDK	Azure AI Document Intelligence (formerly Form Recognizer) is a cloud service that uses machine learning to analyze text and structured data from documents. The Document Intelligence software development kit (SDK) is a set of libraries and tools that enable you to easily integrate Document Intelligence models and capabilities into your applications.
Extract structured data from forms, receipts, invoices, and cards using Form Recognizer in Python ↗	Samples for the Azure.AI.FormRecognizer client library.
Extract, classify, and understand text within documents using Text Analytics in Python	The client Library for Text Analytics. These APIs are part of the Azure AI Language service, which provides Natural Language Processing (NLP) features for understanding and analyzing text.
Document Translation in Python	A quickstart article that uses Document Translation to translate a source document into a target language while preserving structure and text formatting.
Question answering in Python	A quickstart article with steps to get an answer (and confidence score) from a body of text that you send along with your question.
Conversational Language Understanding in Python	The client library for Conversational Language Understanding (CLU). CLU is a cloud-based conversational AI service that can extract intents and entities in conversations. CLU acts like an orchestrator to select the best candidate to analyze conversations to get the best response from apps like QnA, Luis, and Conversation App.
Analyze images	Sample code and setup documents for the Microsoft Azure AI Image Analysis SDK.
Azure AI Content Safety SDK for Python ↗	The SDK can help detect harmful user-generated and AI-generated content in applications and services. Content Safety includes text and image APIs that allow you to detect material that is harmful.

Documentation

[+] Expand table

AI service	Description	API reference	Quickstart
Content Safety	An AI service that detects unwanted content.	Content Safety API reference	Quickstart
Document Intelligence	Turn documents into intelligent data-driven solutions.	Document Intelligence API reference	Quickstart
Language	Build apps with industry-leading natural language understanding capabilities.	Text Analytics API reference	Quickstart
Search	Bring AI-powered cloud search to your applications.	Search API reference	Quickstart
Speech	Speech to text, text to speech, translation, and speaker recognition.	Speech API reference	Quickstart
Translator	Use AI-powered translation to translate more than 100 in-use, at-risk and endangered languages and dialects.	Translation API reference	Quickstart
Vision	Analyze content in images and videos.	Image Analysis API reference	Quickstart

Training

[+] Expand table

Link	Description
Generative AI for beginners workshop ↗	Learn the fundamentals of building Generative AI apps with our 18-lesson comprehensive course by Microsoft Cloud Advocates.
Get started with Azure AI services	Azure AI services are building blocks of AI functionality you can integrate into your applications. Complete this learning path to explore how to provision, secure, monitor, and deploy Azure AI services resources and use them to build intelligent solutions.
Microsoft Azure AI Fundamentals: Generative AI	Complete this learning path to understand how large language models form the foundation of generative AI. Explore how Azure OpenAI Service provides access to the latest generative AI technology. Learn how Azure OpenAI prompts and responses can be fine-tuned and how Microsoft's responsible AI principles drive ethical AI advancements.

Link	Description
Develop Generative AI solutions with Azure OpenAI Service	Azure OpenAI Service provides access to OpenAI's powerful large language models such as ChatGPT, GPT, Codex, and Embeddings models. Complete this learning path for developers and explore how to generate code, images, and text by using the Azure OpenAI SDK and other Azure services.
Build AI apps with Azure Database for PostgreSQL	Complete this learning path to explore Azure AI and Azure Machine Learning Services integrations provided by the Azure AI extension for Azure Database for PostgreSQL - Flexible Server. Learn how these services can enable you to build AI-powered apps.

AI application templates

AI application templates supply you with well-maintained, easy to deploy reference implementations that provide a high-quality starting point for your AI apps.

There are two categories of AI app templates, **building blocks** and **end-to-end solutions**. Building blocks are smaller-scale samples that focus on specific scenarios and tasks. End-to-end solutions are comprehensive reference samples that include documentation, source code, and deployment features. You can build on the solutions and extend them for your own purposes.

- To review a list of key templates available for each programming language, see [AI app templates](#).
- To browse all available templates, see the AI app templates on the [Azure Developer CLI gallery](#).

Get started: Chat using your own data (Python sample)

06/23/2025

This article shows how to deploy and run the [Chat with your own data sample by using example code for Python](#). This sample chat application is built with Python, Azure OpenAI Service, and [Retrieval Augmented Generation \(RAG\)](#) through Azure AI Search.

The app provides answers to user questions about employee benefits at a fictional company. It uses Retrieval-Augmented Generation (RAG) to reference content from supplied PDF files, which may include:

- An employee handbook
- A benefits overview document
- A list of company roles and expectations

By analyzing these documents, the app can respond to natural language queries with accurate, contextually relevant answers. This approach demonstrates how you can use your own data to power intelligent, domain-specific chat experiences with Azure OpenAI and Azure AI Search.

You also learn how to configure the app's settings to modify its response behavior.

After completing the steps in this article, you can begin customizing the project with your own code. This article is part of a series that guides you through building a chat app with Azure OpenAI Service and Azure AI Search. Other articles in the series include:

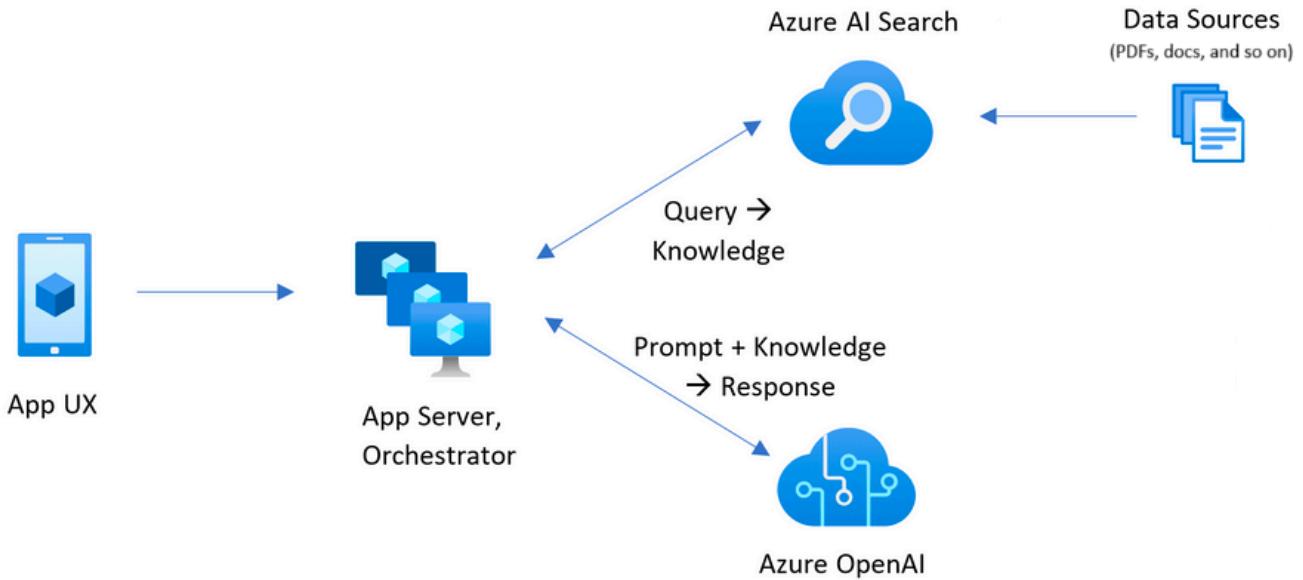
- [.NET](#)
- [Java](#)
- [JavaScript](#)
- [JavaScript frontend with Python backend](#)

ⓘ Note

This article is based on one or more [AI app templates](#), which serve as well-maintained reference implementations. These templates are designed to be easy to deploy and provide a reliable, high-quality starting point for building your own AI applications.

Sample app architecture

The following diagram shows a simple architecture of the chat app.



Key components of the architecture include:

- A web application that hosts the interactive chat interface (usually built with Python Flask or JavaScript/React) and sends user questions to the backend for processing.
- An Azure AI Search resource that performs intelligent search over indexed documents (PDFs, Word files, etc.) and returns relevant document excerpts (chunks) for use in responses.
- An Azure OpenAI Service instance that:
 - Converts documents and user questions into vector representations for semantic similarity search.
 - Extracts important keywords to refine Azure AI Search queries.
 - Synthesizes final responses using the retrieved data and user query.

The typical flow of the chat app is as follows:

- **User submits a question:** A user enters a natural language question through the web app interface.
- **Azure OpenAI processes the question:** The backend uses Azure OpenAI to:
 - Generate an embedding of the question using the text-embedding-ada-002 model.
 - Optionally extract keywords to refine search relevance
- **Azure AI Search retrieves relevant data:** The embedding or keywords are used to perform a semantic search over indexed content (such as, PDFs) in Azure AI Search.
- **Combine results with the question:** The most relevant document excerpts (chunks) are combined with the user's original question.
- **Azure OpenAI generates a response:** The combined input is passed to a GPT model (such as, gpt-35-turbo or gpt-4), which generates a context-aware answer.
- **The response is returned to the user:** The generated answer is displayed in the chat interface.

Prerequisites

A [development container](#) environment is available with all dependencies required to complete this article. You can run the development container in GitHub Codespaces (in a browser) or locally by using Visual Studio Code.

To use this article, you need the following prerequisites:

GitHub Codespaces (recommended)

- An Azure subscription. [Create a free account](#) before you begin.
- Azure account permissions. Your Azure Account must have Microsoft.Authorization/roleAssignments/write permissions. Roles like [User Access Administrator](#) or [Owner](#) satisfy this requirement.
- Access granted to Azure OpenAI in your Azure subscription. In most cases, you can create custom content filters and manage severity levels with general access to Azure OpenAI models. Registration for approval-based access isn't required for general access. For more information, see [Limited Access features for Azure AI services](#).
- Content filter or abuse modifications (optional). To create custom content filters, change severity levels, or support abuse monitoring, you need formal access approval. You can apply for access by completing the necessary registration forms. For more information, see [Registration for modified content filters and/or abuse monitoring](#).
- Support and troubleshooting access. For access to troubleshooting, open a support issue on the GitHub repository.
- A GitHub account. Required to fork the repository and use GitHub Codespaces or clone it locally.

Usage cost for sample resources

Most resources used in this architecture fall under basic or consumption-based pricing tiers. This means you only pay for what you use, and charges are typically minimal during development or testing.

To complete this sample, there may be a small cost incurred from using services like Azure OpenAI, AI Search, and storage. Once you're done evaluating or deploying the app, you can

delete all provisioned resources to avoid ongoing charges.

For a detailed breakdown of expected costs, see the [Cost estimation](#) in the GitHub repository for the sample.

Open development environment

Begin by setting up a development environment that has all the dependencies installed to complete this article.

GitHub Codespaces (recommended)

- An Azure subscription. [Create one for free](#).
- Azure account permissions. Your Azure Account must have Microsoft.Authorization/roleAssignments/write permissions. Roles like [User Access Administrator](#) or [Owner](#) satisfy this requirement.
- A GitHub account. Required to fork the repository and use GitHub Codespaces or clone it locally.

Open a development environment

Use the following instructions to deploy a preconfigured development environment containing all required dependencies to complete this article.

GitHub Codespaces (recommended)

For the simplest and most streamlined setup, use [GitHub Codespaces](#). GitHub Codespaces runs a development container managed by GitHub and provides [Visual Studio Code for the Web](#) as the user interface (UI). This environment includes all required tools, SDKs, extensions, and dependencies preinstalled—so you can start developing immediately without manual configuration.

Using Codespaces ensures:

- Correct developer tools and versions are already installed.
- No need to install Docker, VS Code, or extensions locally.
- Fast onboarding and reproducible environment setup.

 **Important**

All GitHub accounts can use GitHub Codespaces for up to 60 hours free each month with 2 core instances. If you exceed the free quota or use larger compute options, standard GitHub Codespaces billing rates apply. For more information, see [GitHub Codespaces - Monthly included storage and core hours](#).

1. To begin working with the sample project, create a new GitHub codespace on the `main` branch of the [Azure-Samples/azure-search-openai-demo](#) GitHub repository.

Right-click the **GitHub Codespaces - Open** option at the top of the repository page and select **Open link in new window**. This ensures that the development container is launched in a full-screen, dedicated browser tab, giving you access to both the source code and the built-in documentation.



2. On the **Create a new codespace** page, review the codespace configuration settings, and then select **Create codespace**:

A screenshot of the "Create a new codespace" interface. It shows fields for Repository (set to "Azure-Samples/azure-search-openai-demo"), Branch (set to "main"), Dev container configuration (set to "Azure Search OpenAI Demo"), Region (set to "US West"), and Machine type (set to "2-core"). A large green "Create codespace" button is at the bottom right.

Wait for the GitHub codespace to start. The startup process can take a few minutes.

3. After the GitHub codespace opens, sign in to Azure with the Azure Developer CLI by entering the following command in the Terminal pane of the codespace:

```
Bash
```

```
azd auth login
```

GitHub displays a security code in the Terminal pane.

- a. Copy the security code in the Terminal pane and select **Enter**. A browser window opens.
- b. At the prompt, paste the security code into the browser field.
- c. Follow the instructions to authenticate with your Azure account.

You complete the remaining GitHub Codespaces tasks in this article in the context of this development container.

Deploy chat app to Azure

The sample repository includes everything you need to deploy a Chat with your own data application to Azure, including:

- Application source code (Python)
- Infrastructure-as-code files (Bicep)
- Configuration for GitHub integration and CI/CD (optional)

Use the following steps to deploy the app with the Azure Developer CLI (azd).

Important

Azure resources created in this section—especially Azure AI Search—can begin accruing charges immediately upon provisioning, even if the deployment is interrupted before completion. To avoid unexpected charges, monitor your Azure usage and delete unused resources promptly after testing.

1. In the Visual Studio Code Terminal pane, create the Azure resources and deploy the source code by running the following `azd` command:

```
Bash
```

```
azd up
```

2. The process prompts you for one or more of the following settings based on your configuration:

- **Environment name:** This value is used as part of the resource group name. Enter a short name with lowercase letters and dashes (-), such as `myenv`. Uppercase letters, numbers, and special characters aren't supported.
- **Subscription:** Select a subscription to create the resources. If you don't see your desired subscription, use the arrow keys to scroll the full list of available subscriptions.
- **Location:** This region location is used for most resources, including hosting. Select a region location near you geographically.
- **Location for OpenAI model or Document Intelligence resource:** Select the location nearest you geographically. If the region you selected for your **Location** is available for this setting, select the same region.

It take can take some time for the app to deploy. Wait for the deployment to complete before continuing.

3. After the app successfully deploys, the Terminal pane displays an endpoint URL:

```
Deploying services (azd deploy)

(✓) Done: Deploying service backend
- Endpoint: https://app-backend-72xomfpzf3j4o.azurewebsites.net/

SUCCESS: Your Azure app has been deployed!
```

4. Select the endpoint URL to open the chat application in a browser:



Chat with your data

Ask anything or try an example

What is included in my
Northwind Health Plus plan
that is not in standard?

What happens in a
performance review?

What does a Product
Manager do?

Type a new question (e.g. does my plan cover annual eye exams?)



Use chat app to get answers from PDF files

The chat app is preloaded with employee benefits information from [PDF files](#). You can use the chat app to ask questions about the benefits. The following steps walk you through the process of using the chat app. Your answers might vary as the underlying models are updated.

1. In the chat app, select the **What happens in a performance review?** option, or enter the same text in the chat text box. The app returns the initial response:

 Clear chat Developer settings

What happens in a performance review?



During a performance review at Contoso Electronics, your supervisor will discuss your performance over the past year and provide feedback on areas for improvement. They will also provide you with an opportunity to discuss your goals and objectives for the upcoming year. Performance reviews are a two-way dialogue between managers and employees, and employees are encouraged to be honest and open during the review process ¹. The feedback provided during the performance review should be used as an opportunity to help employees develop and grow in their roles. Employees will receive a written summary of their performance review, which will include a rating of their performance, feedback, and goals and objectives for the upcoming year ¹.

Citation: [1. employee_handbook.pdf#page=4](#)

Type a new question (e.g. does my plan cover annual eye exams?)



2. In the answer box, select a citation:

 Clear chat Developer settings

What happens in a performance review?



During a performance review at Contoso Electronics, your supervisor will discuss your performance over the past year and provide feedback on areas for improvement. They will also provide you with an opportunity to discuss your goals and objectives for the upcoming year. Performance reviews are a two-way dialogue between managers and employees, and employees are encouraged to be honest and open during the review process ¹. The feedback provided during the performance review should be used as an opportunity to help employees develop and grow in their roles. Employees will receive a written summary of their performance review, which will include a rating of their performance, feedback, and goals and objectives for the upcoming year ¹.

Citation: [1. employee_handbook.pdf#page=4](#)

Type a new question (e.g. does my plan cover annual eye exams?)



3. GitHub Codespaces opens the right **Citation** pane with three tabbed regions and the focus is on the **Citation** tab:

What happens in a performance review?

During a performance review at Contoso Electronics, your supervisor will discuss your performance over the past year and provide feedback on areas for improvement. They will also provide you with an opportunity to discuss your goals and objectives for the upcoming year. Performance reviews are a two-way dialogue between managers and employees, and employees are encouraged to be honest and open during the review process ¹. The feedback provided during the performance review should be used as an opportunity to help employees develop and grow in their roles. Employees will receive a written summary of their performance review, which will include a rating of their performance, feedback, and goals and objectives for the upcoming year ¹.

Citation: [1. employee_handbook.pdf#page=4](#)

Type a new question (e.g. does my plan cover annual eye exams?)

Thought Process Supporting Content Citation

7. Accountability: We take responsibility for our actions and hold ourselves and others accountable for their performance.
 8. Community: We are committed to making a positive impact in the communities in which we work and live.

Performance Reviews

Performance Reviews at Contoso Electronics

At Contoso Electronics, we strive to ensure our employees are getting the feedback they need to continue growing and developing in their roles. We understand that performance reviews are a key part of this process and it is important to us that they are conducted in an effective and efficient manner.

Performance reviews are conducted annually and are an important part of your career development. During the review, your supervisor will discuss your performance over the past year and provide feedback on areas for improvement. They will also provide you with an opportunity to discuss your goals and objectives for the upcoming year.

Performance reviews are a two-way dialogue between managers and employees. We encourage all employees to be honest and open during the review process, as it is an important opportunity to discuss successes and challenges in the workplace.

We aim to provide positive and constructive feedback during performance reviews. This feedback should be used as an opportunity to help employees develop and grow in their roles.

Employees will receive a written summary of their performance review which will be discussed during the review session. This written summary will include a rating of the employee's performance, feedback, and goals and objectives for the upcoming year.

We understand that performance reviews can be a stressful process. We are committed to making sure that all employees feel supported and empowered during the process. We encourage all employees to reach out to their managers with any questions or concerns they may have.

We look forward to conducting performance reviews with all our employees. They are an important part of our commitment to helping our employees grow and develop in their roles.

GitHub Codespaces provides three tabs of information to help you understand how the chat app generated the answer:

Expand table

Tab	Description
Thought Process	Displays a script of the question/answer interactions in the chat. You can view the content provided by the chat app <code>system</code> , questions entered by the <code>user</code> , and clarifications made by the system <code>assistant</code> .
Supporting Content	Lists the information used to answer your question and the source material. The number of source material citations is specified by the Developer settings . The default number of citations is 3.
Citation	Shows the original source contain for the selected citation.

- When you're done, select the currently selected tab in the right pane. The right pane closes.

Use settings to change response behavior

The specific OpenAI model determines the intelligence of the chat and the settings used to interact with the model. The **Developer settings** option opens the **Configure answer generation** pane where you can change settings for the chat app:

The screenshot shows the Azure OpenAI + AI Search interface. On the left, there's a preview window displaying a performance review text from the employee handbook. A red box highlights the 'Developer settings' button at the top right of this window. A dashed red arrow points from this button to the 'Configure answer generation' pane on the right.

Configure answer generation

Override prompt template [\(i\)](#)
Temperature [\(i\)](#)
0.3
Seed [\(i\)](#)
Minimum search score [\(i\)](#)
0
Minimum reranker score [\(i\)](#)
0
Retrieve this many search results: [\(i\)](#)
3
Include category [\(i\)](#)
All
Exclude category [\(i\)](#)
 Use semantic ranker for retrieval [\(i\)](#)
 Use semantic captions [\(i\)](#)
 Suggest follow-up questions [\(i\)](#)
Retrieval mode [\(i\)](#)
Vectors + Text (Hybrid)
 Stream chat completion responses [\(i\)](#)

[Close](#)

[Expand table](#)

Setting	Description
Override prompt template	Overrides the prompt used to generate the answer based on the question and search results.

Setting	Description
Temperature	Sets the temperature of the request to the large language model (LLM) that generates the answer. Higher temperatures result in more creative responses, but they might be less grounded.
Seed	Sets a seed to improve the reproducibility of the model's responses. The seed can be any integer.
Minimum search score	Sets a minimum score for search results returned from Azure AI Search. The score range depends on whether you use Hybrid (default) , Vectors only , or Text only for the Retrieval mode setting.
Minimum reranker score	Sets a minimum score for search results returned from the semantic reranker. The score always ranges between 0-4. The higher the score, the more semantically relevant the result is to the question.
Retrieve this many search results	Sets the number of search results to retrieve from Azure AI Search. More results can increase the likelihood of finding the correct answer, but might lead to the model getting 'lost in the middle.' You can see the returned sources in the Thought Process and Supporting Content tabs of the Citation pane.
Include category	Specifies the categories to include when generating the search results. Use the dropdown list to make your selection. The default action is to include All categories.
Exclude category	Specifies any categories to exclude from the search results. There are no categories used in the default data set.
Use semantic ranker for retrieval	Enables the Azure AI Search semantic ranker , a model that reranks search results based on semantic similarity to the user's query.
Use semantic captions	Sends semantic captions to the LLM instead of the full search result. A semantic caption is extracted from a search result during the process of semantic ranking.
Suggest follow-up questions	Asks the LLM to suggest follow-up questions based on the user's query.
Retrieval mode	Sets the retrieval mode for the Azure AI Search query. The default action is Vectors + Text (Hybrid) , which uses a combination of vector search and full text search. The Vectors option uses only vector search. The Text option uses only full text search. The Hybrid approach is optimal.
Stream chat completion responses	Continuously streams the response to the chat UI as the content is generated.

The following steps walk you through the process of changing the settings.

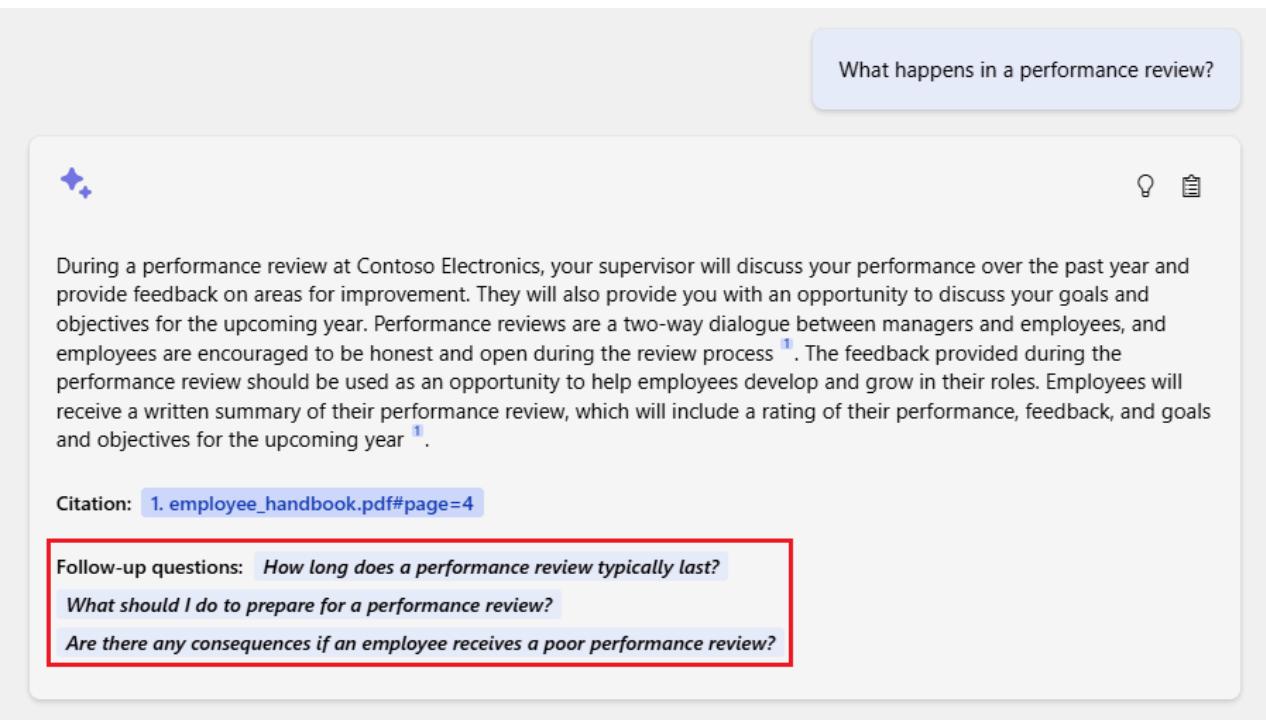
1. In the browser, select the **Developer settings** option.

2. Select the **Suggest follow-up questions** checkbox to enable the option, and select **Close** to apply the setting change.

3. In the chat app, reask the question, this time by entering the text in the question box:



The chat app answer now includes suggested follow-up questions:



4. Select the **Developer settings** option again, and unselect **Use semantic ranker for retrieval** option. Close the settings.

5. Ask the same question again, and notice the difference in the answer from the chat app.

With the Semantic ranker: "During a performance review at Contoso Electronics, your supervisor will discuss your performance over the past year and provide feedback on areas for improvement. You will also have the opportunity to discuss your goals and objectives for the upcoming year. The review is a two-way dialogue between managers and employees, and it is encouraged for employees to be honest and open during the process (1). The feedback provided during the review should be positive and constructive, aimed at helping employees develop and grow in their roles. Employees will receive a written summary of their performance review, which will include a rating of their performance, feedback, and goals and objectives for the upcoming year (1)."."

Without the Semantic ranker: "During a performance review at Contoso Electronics, your supervisor will discuss your performance over the past year and provide feedback on

areas for improvement. It is a two-way dialogue where you are encouraged to be honest and open (1). The feedback provided during the review should be positive and constructive, aimed at helping you develop and grow in your role. You will receive a written summary of the review, including a rating of your performance, feedback, and goals for the upcoming year (1)."

Clean up resources

After you complete the exercise, it's a best practice to remove any resources that are no longer required.

Clean up Azure resources

The Azure resources created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges.

Delete the Azure resources and remove the source code by running the following `azd` command:

```
Bash
```

```
azd down --purge --force
```

The command switches include:

- `purge`: Deleted resources are immediately purged. This option allows you to reuse the Azure OpenAI tokens per minute (TPM) metric.
- `force`: The deletion happens silently, without requiring user consent.

Clean up GitHub Codespaces

GitHub Codespaces

Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement that you get for your account.

 **Important**

For more information about your GitHub account's entitlements, see [GitHub Codespaces - Monthly included storage and core hours](#).

1. Sign in to the [GitHub Codespaces dashboard](#).
2. On the dashboard, locate your currently running codespaces sourced from the [Azure-Samples/azure-search-openai-demo](#) GitHub repository:

The screenshot shows the GitHub Codespaces dashboard. At the top, there's a search bar and various navigation icons. Below that, a sidebar shows 'All' (1) and 'Templates'. The main area is titled 'Your codespaces' and lists one item: 'Owned by developer-bob' (Azure-Samples/azure-search-openai-demo). This item is highlighted with a red box and a magnifying glass icon. Below the list, there are sections for 'Explore quick start templates' (Blank, React, Jupyter Notebook) and a 'See all' button.

3. Open the context menu for the codespace and select **Delete**:

The screenshot shows the GitHub Codespaces dashboard with a context menu open over the selected codespace. The menu options are: 'Open in ...', 'Rename', 'Export changes to a fork', 'Change machine type', 'Keep codespace', and 'Delete'. The 'Delete' option is highlighted with a red box.

Get help

This sample repository offers [troubleshooting information](#).

If your issue isn't addressed, add your issue to the repository's [Issues](#) webpage.

Related content

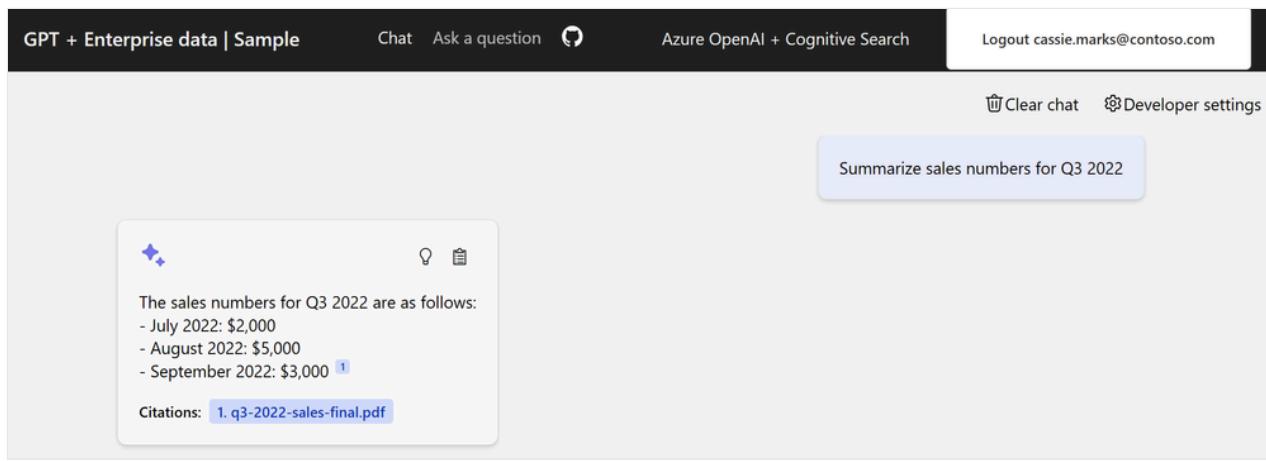
- [Get the source code for the sample used in this article](#)
- [Build a chat app with Azure OpenAI](#) best practice solution architecture
- [Access control in Generative AI Apps with Azure AI Search](#)
- [Build an Enterprise ready OpenAI solution with Azure API Management](#)
- [Outperforming vector search with hybrid retrieval and ranking capabilities](#)

Get started with chat document security for Python

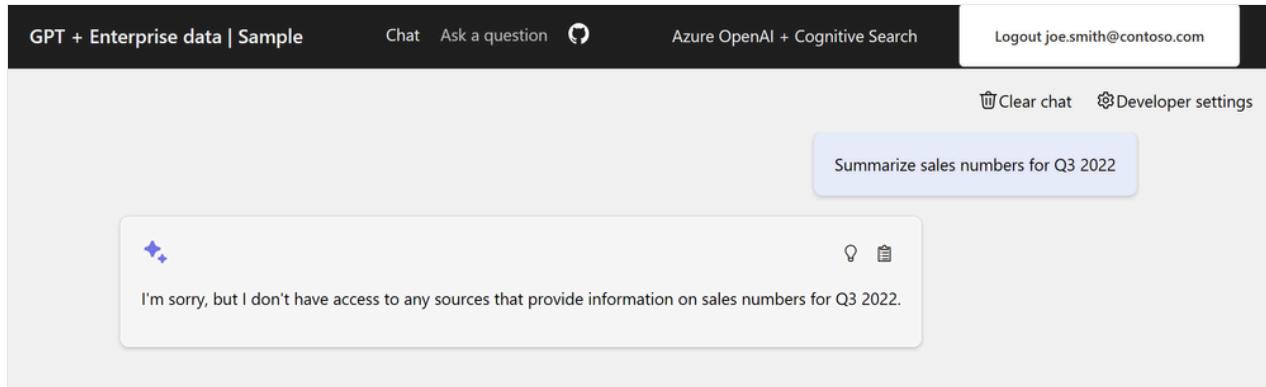
06/26/2025

When you build a [chat application by using the Retrieval Augmented Generation \(RAG\) pattern](#) with your own data, make sure that each user receives an answer based on their permissions. Follow the process in this article to add document access control to your chat app.

- **Authorized user:** This person should have access to answers contained within the documents of the chat app.



- **Unauthorized user:** This person shouldn't have access to answers from secured documents they don't have authorization to see.

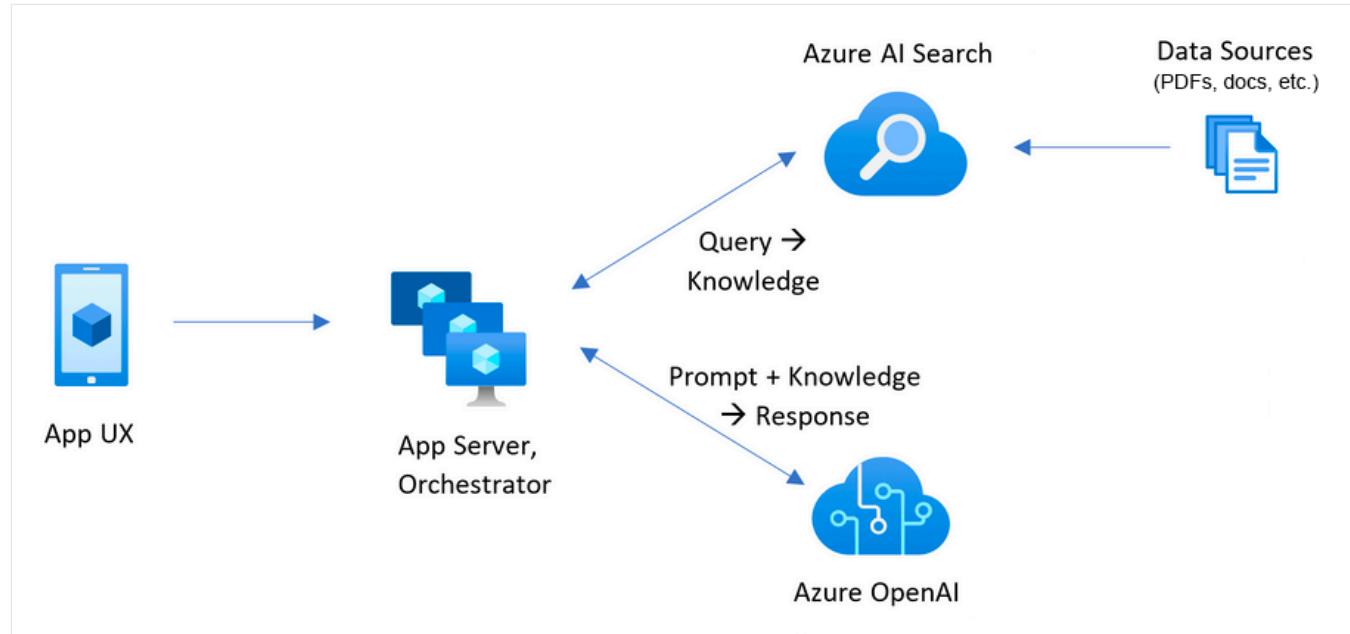


(!) Note

This article uses one or more [AI app templates](#) as the basis for the examples and guidance in the article. AI app templates provide you with well-maintained reference implementations that are easy to deploy. They help to ensure a high-quality starting point for your AI apps.

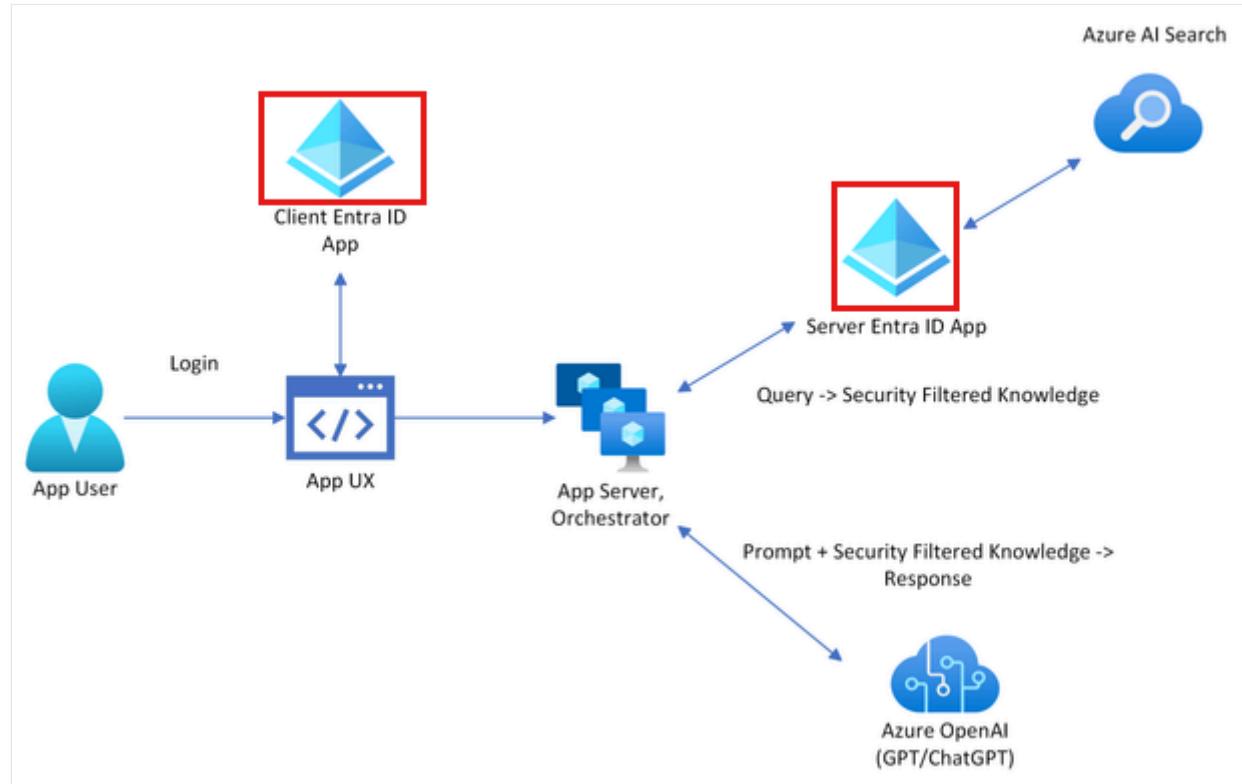
Architectural overview

Without a document security feature, the enterprise chat app has a simple architecture by using Azure AI Search and Azure OpenAI. An answer is determined from queries to Azure AI Search where the documents are stored, in combination with a response from an Azure OpenAI GPT model. No user authentication is used in this simple flow.

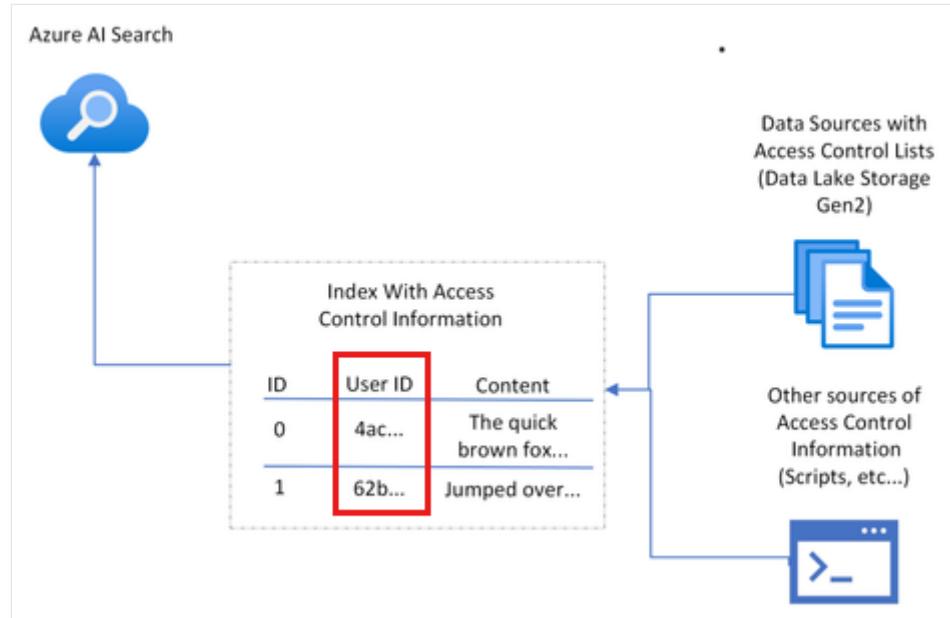


To add security for the documents, you need to update the enterprise chat app:

- Add client authentication to the chat app with Microsoft Entra.
- Add server-side logic to populate a search index with user and group access.



Azure AI Search doesn't provide *native* document-level permissions and can't vary search results from within an index by user permissions. Instead, your application can use search filters to ensure that a document is accessible to a specific user or by a specific group. Within your search index, each document should have a filterable field that stores user or group identity information.



Because the authorization isn't natively contained in Azure AI Search, you need to add a field to hold user or group information, and then [filter](#) any documents that don't match. To implement this technique, you need to:

- Create a document access control field in your index dedicated to storing the details of users or groups with document access.
- Populate the document's access control field with the relevant user or group details.
- Update this access control field whenever there are changes in user or group access permissions.

If your index updates are scheduled with an indexer, changes are picked up on the next indexer run. If you don't use an indexer, you need to manually reindex.

In this article, the process of securing documents in Azure AI Search is made possible with *example* scripts, which you as the search administrator would run. The scripts associate a single document with a single user identity. You can take these [scripts ↗](#) and apply your own security and production requirements to scale to your needs.

Determine security configuration

The solution provides Boolean environment variables to turn on features that are necessary for document security in this sample.

Parameter	Purpose
AZURE_USE_AUTHENTICATION	When set to <code>true</code> , enables user sign-in to the chat app and Azure App Service authentication. Enables <code>Use oid security filter</code> in the chat app Developer settings .
AZURE_ENFORCE_ACCESS_CONTROL	When set to <code>true</code> , requires authentication for any document access. The Developer settings for object ID (OID) and group security are turned on and disabled so that they can't be disabled from the UI.
AZURE_ENABLE_GLOBAL_DOCUMENTS_ACCESS	When set to <code>true</code> , this setting allows authenticated users to search on documents that have no access controls assigned, even when access control is required. This parameter should be used only when <code>AZURE_ENFORCE_ACCESS_CONTROL</code> is enabled.
AZURE_ENABLE_UNAUTHENTICATED_ACCESS	When set to <code>true</code> , this setting allows unauthenticated users to use the app, even when access control is enforced. This parameter should be used only when <code>AZURE_ENFORCE_ACCESS_CONTROL</code> is enabled.

Use the following sections to understand the security profiles supported in this sample. This article configures the *Enterprise profile*.

Enterprise: Required account + document filter

Each user of the site *must* sign in. The site contains content that's public to all users. The document-level security filter is applied to all requests.

Environment variables:

- AZURE_USE_AUTHENTICATION=true
- AZURE_ENABLE_GLOBAL_DOCUMENTS_ACCESS=true
- AZURE_ENFORCE_ACCESS_CONTROL=true

Mixed use: Optional account + document filter

Each user of the site *might* sign in. The site contains content that's public to all users. The document-level security filter is applied to all requests.

Environment variables:

- AZURE_USE_AUTHENTICATION=true
- AZURE_ENABLE_GLOBAL_DOCUMENTS_ACCESS=true

- `AZURE_ENFORCE_ACCESS_CONTROL=true`
- `AZURE_ENABLE_UNAUTHENTICATED_ACCESS=true`

Prerequisites

A [development container](#) environment is available with all the [dependencies](#) that are required to complete this article. You can run the development container in GitHub Codespaces (in a browser) or locally by using Visual Studio Code.

To use this article, you need the following prerequisites:

- An Azure subscription. [Create one for free](#).
- Azure account permissions: Your Azure account must have:
 - Permission to [manage applications in Microsoft Entra ID](#).
 - `Microsoft.Authorization/roleAssignments/write` permissions, such as [User Access Administrator](#) or [Owner](#).

You need more prerequisites depending on your preferred development environment.

GitHub Codespaces (recommended)

- [GitHub account](#)

Open a development environment

Use the following instructions to deploy a preconfigured development environment containing all required dependencies to complete this article.

GitHub Codespaces (recommended)

[GitHub Codespaces](#) runs a development container managed by GitHub with [Visual Studio Code for the Web](#) as the user interface. For the most straightforward development environment, use GitHub Codespaces so that you have the correct developer tools and dependencies preinstalled to complete this article.

 **Important**

All GitHub accounts can use GitHub Codespaces for up to 60 hours free each month with two core instances. For more information, see [GitHub Codespaces monthly included storage and core hours](#).

1. Start the process to create a new GitHub codespace on the `main` branch of the [Azure-Samples/azure-search-openai-demo](#) GitHub repository.
2. Right-click the following button, and select **Open link in new windows** to have the development environment and the documentation available at the same time.



3. On the **Create codespace** page, review the codespace configuration settings, then select **Create new codespace**.

A screenshot of the GitHub Codespaces "Create codespace for" interface. It shows configuration options for a new codespace for the repository "Azure-Samples/azure-search-openai-demo".

The interface includes the following fields:

- Branch:** main (dropdown menu)
- Dev container configuration:** Azure Search OpenAI Demo (dropdown menu)
- Region:** US West (dropdown menu)
- Machine type:** 2-core (dropdown menu)

A green "Create codespace" button is located at the bottom right of the form.

4. Wait for the codespace to start. This startup process can take a few minutes.
5. In the terminal at the bottom of the screen, sign in to Azure with the Azure Developer CLI.

```
Bash
```

```
azd auth login
```

6. Complete the authentication process.

7. The remaining tasks in this article take place in the context of this development container.

Get required information with the Azure CLI

Get your subscription ID and tenant ID with the following Azure CLI command. Copy the value to use as your `AZURE_TENANT_ID` value.

Azure CLI

```
az account list --query "[].{subscription_id:id, name:name, tenantId:tenantId}" -o table
```

If you get an error about your tenant's conditional access policy, you need a second tenant without a conditional access policy.

- Your first tenant, associated with your user account, is used for the `AZURE_TENANT_ID` environment variable.
- Your second tenant, without conditional access, is used for the `AZURE_AUTH_TENANT_ID` environment variable to access Microsoft Graph. For tenants with a conditional access policy, find the ID of a second tenant without a conditional access policy or [create a new tenant](#).

Set environment variables

1. Run the following commands to configure the application for the **Enterprise** profile.

Console

```
azd env set AZURE_USE_AUTHENTICATION true  
azd env set AZURE_ENABLE_GLOBAL_DOCUMENTS_ACCESS true  
azd env set AZURE_ENFORCE_ACCESS_CONTROL true
```

2. Run the following command to set the tenant, which authorizes the user sign-in to the hosted application environment. Replace `<YOUR_TENANT_ID>` with the tenant ID.

Console

```
azd env set AZURE_TENANT_ID <YOUR_TENANT_ID>
```

 Note

If you have a conditional access policy on your user tenant, you need to [specify an authentication tenant](#).

Deploy the chat app to Azure

Deployment consists of the following steps:

- Create the Azure resources.
- Upload the documents.
- Create the Microsoft Entra identity apps (client and server).
- Turn on identity for the hosting resource.

1. Provision the Azure resources and deploy the source code.

Bash

```
azd up
```

2. Use the following table to answer the `AZD` deployment prompts.

 [Expand table](#)

Prompt	Answer
Environment name	Use a short name with identifying information such as your alias and app. An example is <code>tjones-secure-chat</code> .
Subscription	Select a subscription in which to create the resources.
Location for Azure resources	Select a location near you.
Location for <code>documentIntelligentResourceGroupLocation</code>	Select a location near you.
Location for <code>openAIResourceGroupLocation</code>	Select a location near you.

Wait 5 or 10 minutes after the app deploys to allow the app to start up.

3. After the application successfully deploys, a URL appears in the terminal.

4. Select the URL labeled `(✓) Done: Deploying service webapp` to open the chat application in a browser.

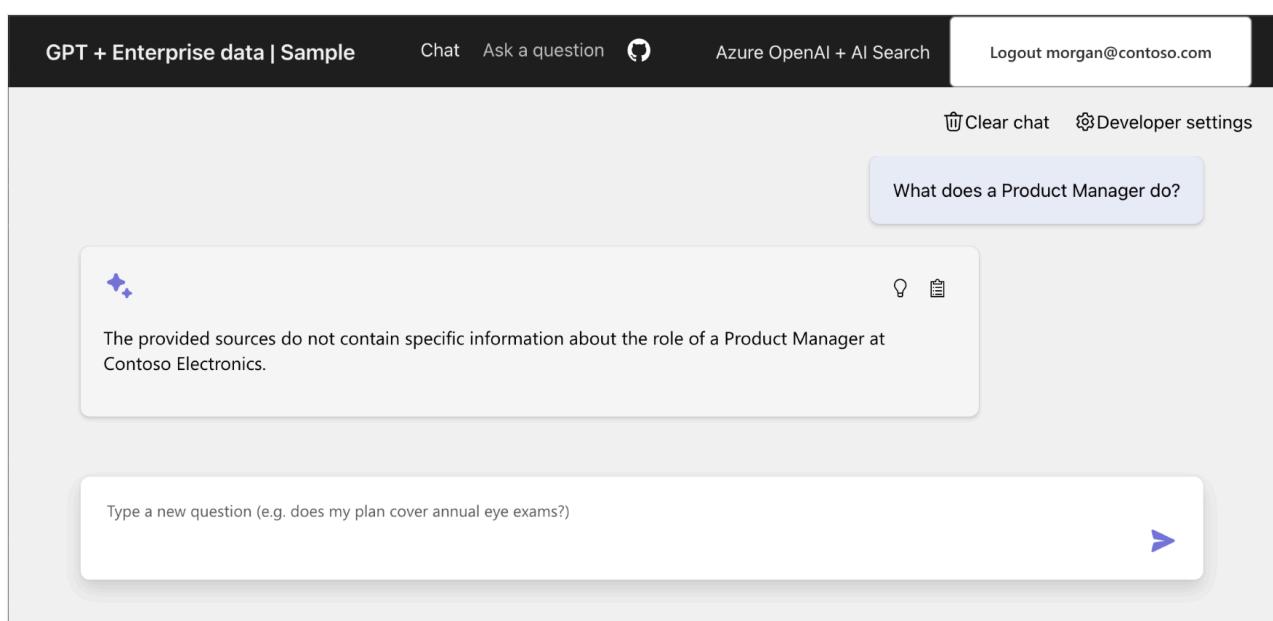
```
Deploying services (azd deploy)

(✓) Done: Deploying service backend
- Endpoint: https://app-backend-72xomfpzf3j4o.azurewebsites.net/

SUCCESS: Your Azure app has been deployed!
```

5. Agree to the app authentication pop-up.
6. When the chat app appears, notice in the upper-right corner that your user is signed in.
7. Open **Developer settings** and notice that both of the following options are selected and disabled for change:
 - Use oid security filter
 - Use groups security filter
8. Select the card with **What does a product manager do?**.

9. You get an answer like: **The provided sources do not contain specific information about the role of a Product Manager at Contoso Electronics.**



Open access to a document for a user

Turn on your permissions for the exact document so that you *can* get the answer. You need several pieces of information:

- Azure Storage

- Account name
- Container name
- Blob/document URL for `role_library.pdf`
- User's ID in Microsoft Entra ID

When this information is known, update the Azure AI Search index `oids` field for the `role_library.pdf` document.

Get the URL for a document in storage

1. In the `.azure` folder at the root of the project, find the environment directory, and open the `.env` file with that directory.
2. Search for the `AZURE_STORAGE_ACCOUNT` entry and copy its value.
3. Use the following Azure CLI commands to get the URL of the `role_library.pdf` blob in the `content` container.

Azure CLI

```
az storage blob url \
--account-name <REPLACE_WITH_AZURE_STORAGE_ACCOUNT> \
--container-name 'content' \
--name 'role_library.pdf'
```

[] Expand table

Parameter	Purpose
--account-name	Azure Storage account name.
--container-name	The container name in this sample is <code>content</code> .
--name	The blob name in this step is <code>role_library.pdf</code> .

4. Copy the blob URL to use later.

Get your user ID

1. In the chat app, select **Developer settings**.
2. In the **ID Token claims** section, copy your `objectidentifier` parameter. This parameter is known in the next section as `USER_OBJECT_ID`.

Provide user access to a document in Azure Search

1. Use the following script to change the `oids` field in Azure AI Search for `role_library.pdf` so that you have access to it.

Python

```
python ./scripts/manageacl.py --acl-type oids --acl-action add --acl <REPLACE_WITH_YOUR_USER_OBJECT_ID> --url <REPLACE_WITH_YOUR_DOCUMENT_URL>
```

[+] Expand table

Parameter	Purpose
<code>-v</code>	Verbose output.
<code>--acl-type</code>	Group or user OIDs: <code>oids</code> .
<code>--acl-action</code>	Add to a Search index field. Other options include <code>enable_acls</code> , <code>remove</code> , <code>remove_all</code> , and <code>view</code> .
<code>--acl</code>	Group or user <code>USER_OBJECT_ID</code> .
<code>--url</code>	The file's location in Azure Storage, such as https://MYSTORAGENAME.blob.core.windows.net/content/role_library.pdf . Don't surround the URL with quotation marks in the CLI command.

2. The console output for this command looks like:

console.

```
Loading azd .env file from current environment...
Creating Python virtual environment "app/backend/.venv"...
Installing dependencies from "requirements.txt" into virtual environment (in quiet mode)...
Running manageacl.py. Arguments to script: -v --acl-type oids --acl-action add --acl 00000000-0000-0000-000000000000 --url https://mystorage.blob.core.windows.net/content/role_library.pdf
Found 58 search documents with storageUrl
https://mystorage.blob.core.windows.net/content/role_library.pdf
Adding acl 00000000-0000-0000-000000000000 to 58 search documents
```

3. Optionally, use the following command to verify that your permission is listed for the file in Azure AI Search.

Python

```
python ./scripts/manageacl.py -v --acl-type groups --acl-action view --url <REPLACE_WITH_YOUR_DOCUMENT_URL>
```

[+] Expand table

Parameter	Purpose
-v	Verbose output.
--acl-type	Group or user OIDs: <code>oids</code> .
--acl-action	View a Search index field <code>oids</code> . Other options include <code>enable_acls</code> , <code>remove</code> , <code>remove_all</code> , and <code>add</code> .
--url	The file's location in that shows, such as <code>https://MYSTORAGENAME.blob.core.windows.net/content/role_library.pdf</code> . Don't surround the URL with quotation marks in the CLI command.

4. The console output for this command looks like:

```
console.
```

```
Loading azd .env file from current environment...
Creating Python virtual environment "app/backend/.venv"...
Installing dependencies from "requirements.txt" into virtual environment (in
quiet mode)...
Running manageacl.py. Arguments to script: -v --acl-type oids --acl-action
view --acl 00000000-0000-0000-000000000000 --url
https://mystorage.blob.core.windows.net/content/role_library.pdf
Found 58 search documents with storageUrl
https://mystorage.blob.core.windows.net/content/role_library.pdf
[00000000-0000-0000-000000000000]
```

The array at the end of the output includes your `USER_OBJECT_ID` parameter and is used to determine if the document is used in the answer with Azure OpenAI.

Verify that Azure AI Search contains your `USER_OBJECT_ID`

1. Open the [Azure portal](#) and search for `AI Search`.
2. Select your search resource from the list.
3. Select **Search management > Indexes**.
4. Select **gptkbindex**.

5. Select View > JSON view.

6. Replace the JSON with the following JSON:

JSON

```
{  
  "search": "*",  
  "select": "sourcefile, oids",  
  "filter": "oids/any()"  
}
```

This JSON searches all documents where the `oids` field has any value and returns the `sourcefile` and `oids` fields.

7. If the `role_library.pdf` doesn't have your OID, return to the [Provide user access to a document in Azure Search](#) section and complete the steps.

Verify user access to the document

If you completed the steps but didn't see the correct answer, verify that your `USER_OBJECT_ID` parameter is set correctly in Azure AI Search for `role_library.pdf`.

1. Return to the chat app. You might need to sign in again.
2. Enter the same query so that the `role_library` content is used in the Azure OpenAI answer: `What does a product manager do?`.
3. View the result, which now includes the appropriate answer from the role library document.

 Clear chat  Developer settings

What does a Product Manager do?



A Product Manager is responsible for leading the product management team and providing guidance on product strategy, design, development, and launch. They collaborate with internal teams and external partners to ensure successful product execution. They also develop and implement product life-cycle management processes, monitor industry trends, develop product marketing plans, and research customer needs to develop customer-centric product roadmaps. Additionally, they oversee the product portfolio, analyze product performance and customer feedback, and identify areas for improvement [\[1\]](#) [\[2\]](#) [\[3\]](#).

Citations: [1. role_library.pdf#page=29](#) [2. role_library.pdf#page=12](#) [3. role_library.pdf#page=23](#)

Type a new question (e.g. does my plan cover annual eye exams?)



Clean up resources

The following steps walk you through the process of cleaning up the resources you used.

Clean up Azure resources

The Azure resources created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges.

Run the following Azure Developer CLI command to delete the Azure resources and remove the source code.

Bash

```
azd down --purge
```

Clean up GitHub Codespaces and Visual Studio Code

The following steps walk you through the process of cleaning up the resources you used.

GitHub Codespaces

Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement that you get for your account.

ⓘ Important

For more information about your GitHub account's entitlements, see [GitHub Codespaces monthly included storage and core hours](#).

1. Sign in to the [GitHub Codespaces dashboard](#).
2. Locate your currently running codespaces that are sourced from the [Azure-Samples/azure-search-openai-demo](#) GitHub repository.

The screenshot shows the GitHub Codespaces dashboard. At the top, there is a search bar and several icons. Below the search bar, there are two tabs: 'All' (selected) and 'Templates'. On the left, there is a sidebar titled 'By repository' with one item: 'Azure-Samples/azure-search-openai-demo' (1). The main area is titled 'Your codespaces' and contains a section titled 'Explore quick start templates' with three options: 'Blank', 'React', and 'Jupyter Notebook'. Below this, there is a list of currently running codespaces. One specific codespace is highlighted with a red box: 'effective orbit' (Owned by developer-bob). This codespace was created from the 'Azure-Samples/azure-search-openai-demo' repository and is currently retrieving data. It has 2-core + 8GB RAM + 32GB storage and was last used 7 minutes ago.

3. Open the context menu for the codespace and then select **Delete**.

The screenshot shows the GitHub Codespaces interface. At the top, there's a search bar and a 'New codespace' button. Below that, a sidebar lists 'All' (1 template) and 'By repository' (1 item: 'Azure-Samples/azure-search-openai-demo'). The main area displays 'Your codespaces' with three quick start templates: 'Blank', 'React', and 'Jupyter Notebook'. Under 'Owned by developer-bob', the repository 'Azure-Samples/azure-search-openai-demo' is listed, showing it's a 'main' branch with 'No changes'. To the right of the repository card is a context menu with options like 'Open in ...', 'Rename', 'Export changes to a fork', 'Change machine type', 'Keep codespace', and 'Delete'. The 'Delete' button is highlighted with a red box.

Get help

The sample repository offers [troubleshooting information ↗](#).

Troubleshooting

This section helps you troubleshoot issues specific to this article.

Provide authentication tenant

When your authentication is in a separate tenant from your hosting application, you need to set that authentication tenant with the following process.

1. Run the following command to configure the sample to use a second tenant for the authentication tenant.

```
Console
azd env set AZURE_AUTH_TENANT_ID <REPLACE-WITH-YOUR-TENANT-ID>
```

[Expand table](#)

Parameter	Purpose
AZURE_AUTH_TENANT_ID	If <code>AZURE_AUTH_TENANT_ID</code> is set, it's the tenant that hosts the app.

2. Redeploy the solution with the following command:

Console

```
azd up
```

Related content

- Build a [chat app with Azure OpenAI](#) ↗ best-practices solution architecture.
- Learn about [access control in generative AI apps with Azure AI Search](#) ↗ .
- Build an [enterprise-ready Azure OpenAI solution with Azure API Management](#) ↗ .
- See [Azure AI Search: Outperforming vector search with hybrid retrieval and ranking capabilities](#) ↗ .

Get started with chat private endpoints for Python

Article • 12/17/2024

This article shows you how to deploy and run the [enterprise chat app sample for Python](#) that's accessible by private endpoints.

This sample implements a chat app by using Python, Azure OpenAI Service, and [Retrieval Augmented Generation \(RAG\)](#) in Azure AI Search to get answers about employee benefits at a fictitious company. The app is seeded with PDF files that include the employee handbook, a benefits document, and a list of company roles and expectations.

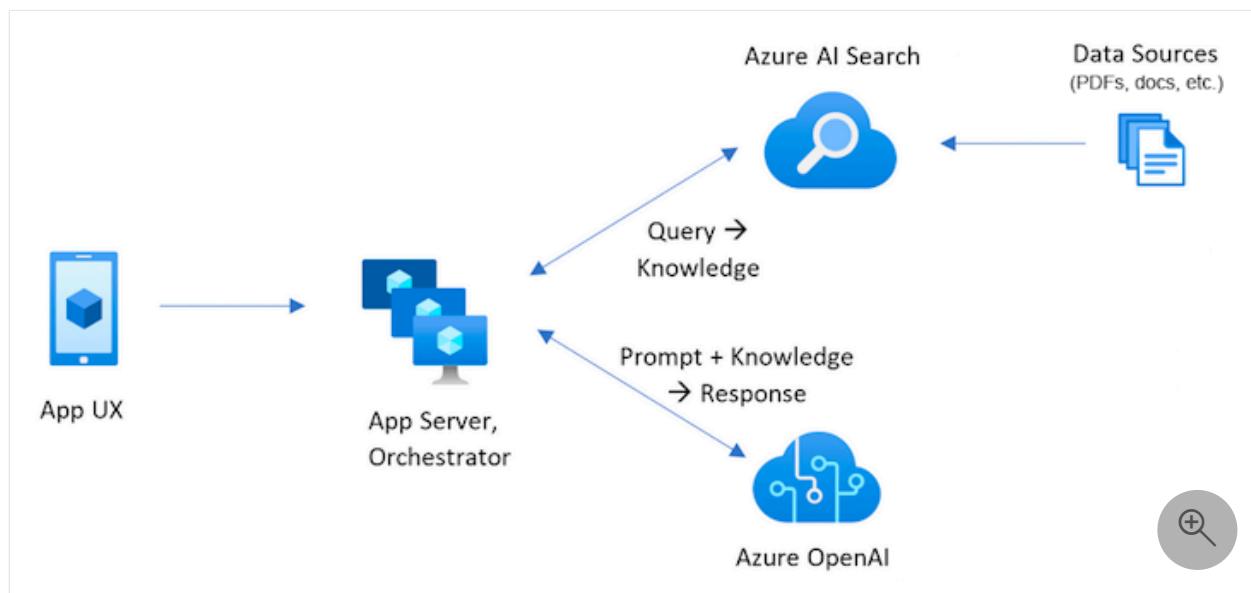
By following the instructions in this article, you:

- Deploy a chat app to Azure for public access in a web browser.
- Redeploy a chat app with private endpoints.

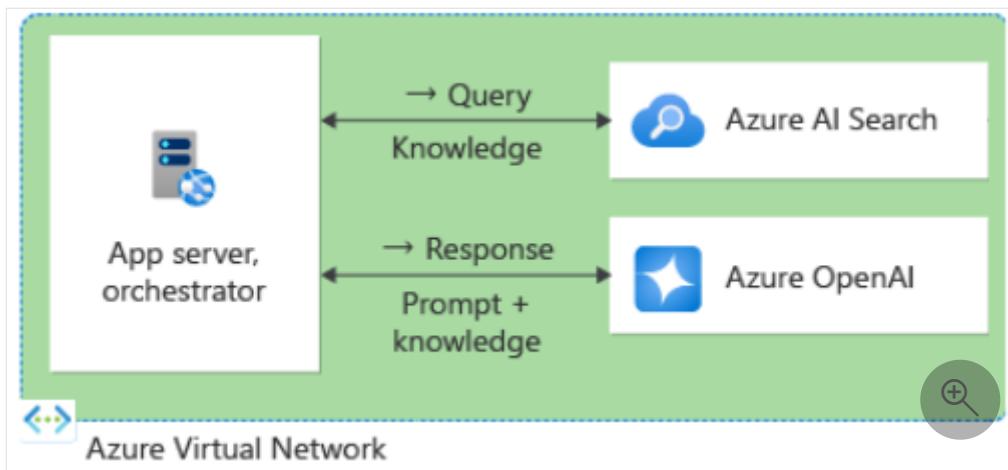
After you finish this procedure, you can start modifying the new project with your custom code and redeploy, knowing that your chat app is accessible only through the private network.

Architectural overview

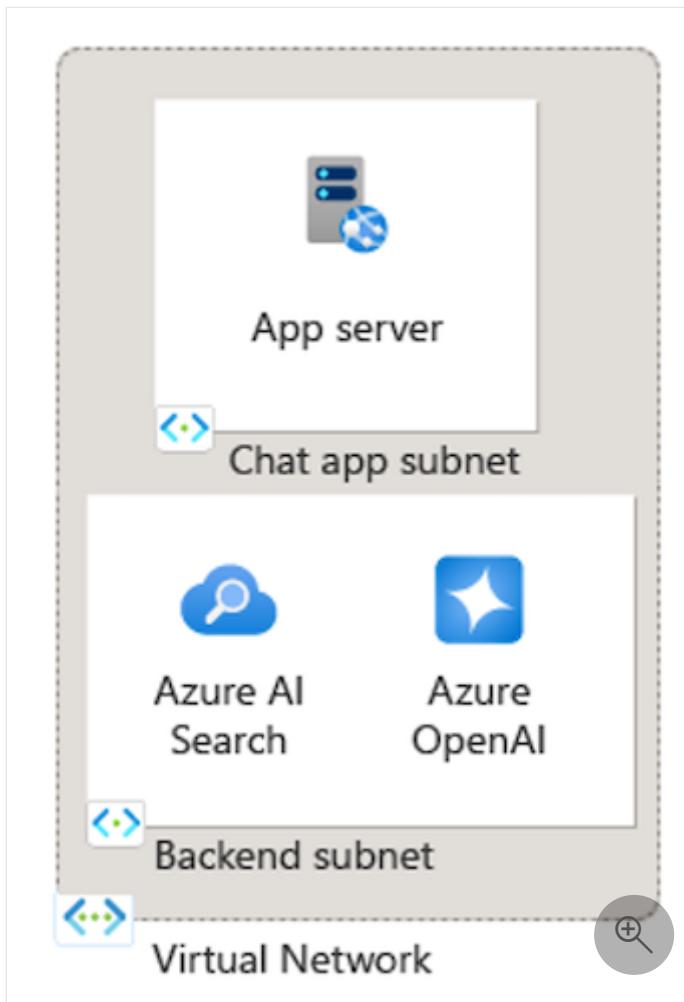
The default deployment creates a chat app with public endpoints.



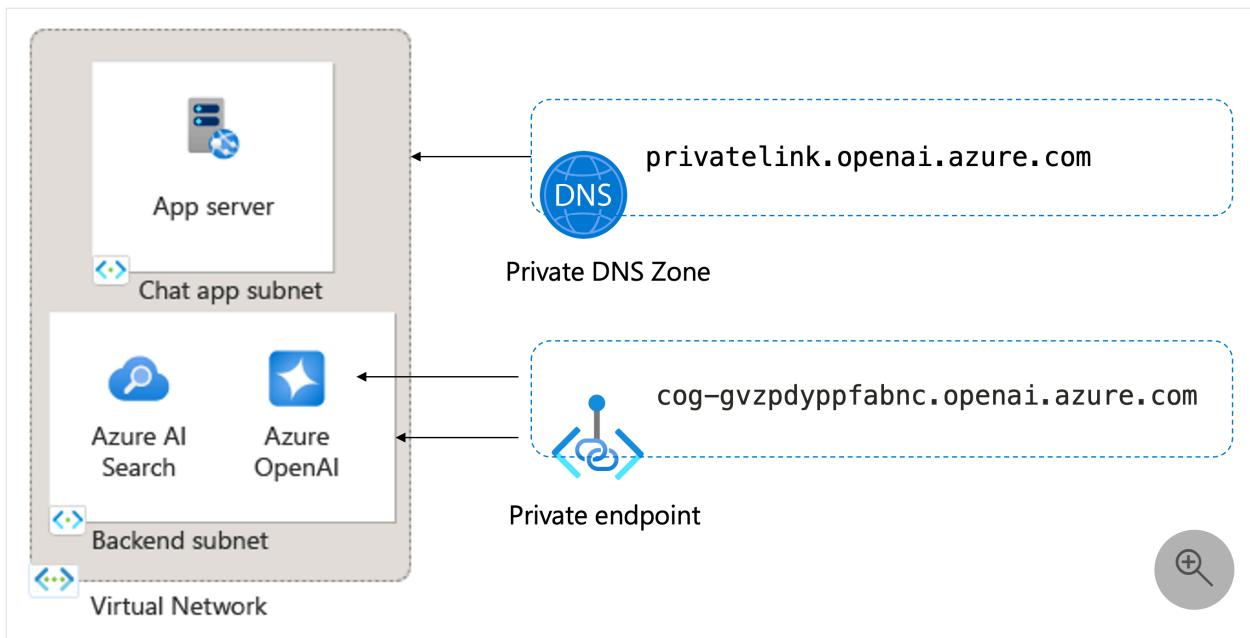
For chat apps enriched with private data, securing access to your chat app is crucial. This article presents a solution by using a virtual network.



Within the virtual network, there's a separate subnet for the Azure App Service app versus the other back-end Azure services. This structure makes it easy to apply different network security group rules to each subnet.



Within the virtual network, the services use private endpoints to communicate with each other. Each private endpoint is associated with a private Domain Name System (DNS) zone to resolve the private endpoint's name to an IP address within the virtual network.



Deployment steps

We recommend that you deploy the solution twice. Deploy once with public access to validate that the chat app is working correctly. Deploy again with private access to secure your chat app by using a virtual network.

Prerequisites

A [development container](#) environment is available with all dependencies that are required to finish this article. You can run the development container in GitHub Codespaces (in a browser) or locally by using Visual Studio Code.

To use this article, you need the following prerequisites.

Codespaces (recommended)

- An Azure subscription. [Create one for free](#).
- Azure account permissions. Your Azure account must have `Microsoft.Authorization/roleAssignments/write` permissions, such as [User Access Administrator](#) or [Owner](#).
- A GitHub account.

Open development environment

Begin now with a development environment that has all the dependencies installed to complete this article.

GitHub Codespaces (recommended)

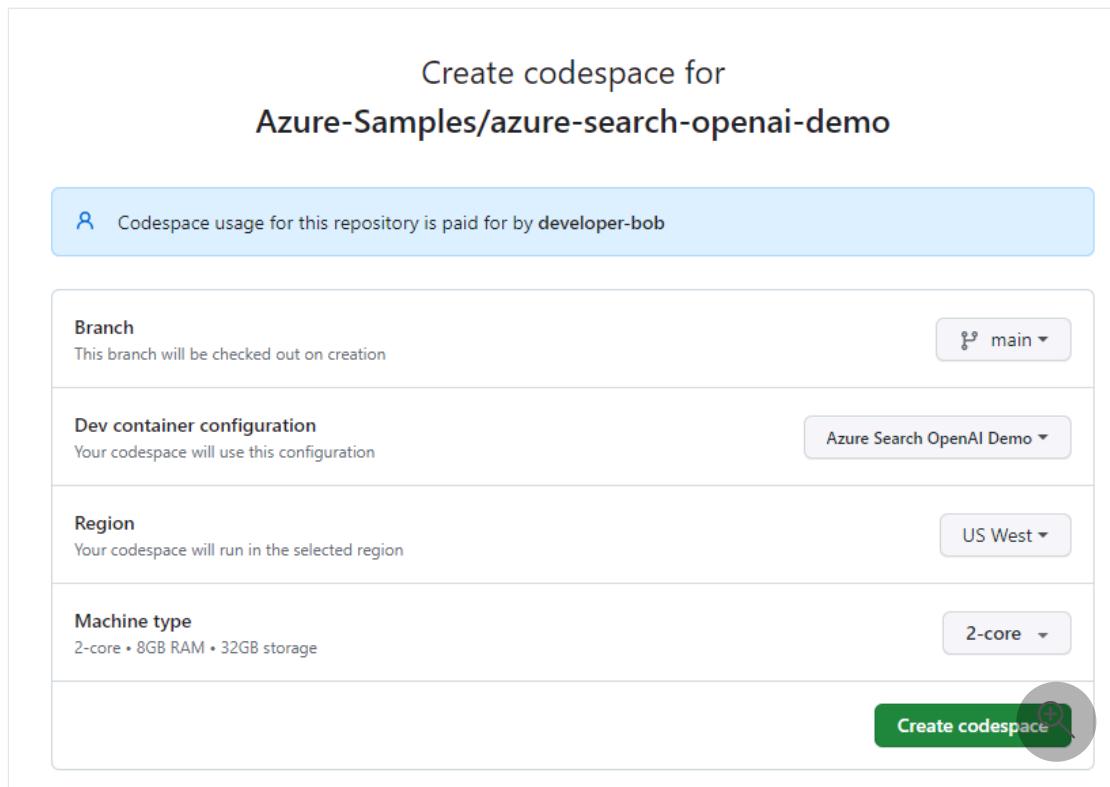
[GitHub Codespaces](#) runs a development container managed by GitHub with [Visual Studio Code for the Web](#) as the user interface. For the most straightforward development environment, use GitHub Codespaces so that you have the correct developer tools and dependencies preinstalled to complete this article.

Important

All GitHub accounts can use GitHub Codespaces for up to 60 hours free each month with two core instances. For more information, see [GitHub Codespaces monthly included storage and core hours](#).

1. Start the process to create a new GitHub codespace on the `main` branch of the [Azure-Samples/azure-search-openai-demo](#) GitHub repository.
2. Right-click the following button, and select **Open link in new windows** to have the development environment and the documentation available at the same time.

3. On the **Create codespace** page, review the codespace configuration settings, and then select **Create codespace**.



4. Wait for the codespace to start. This startup process can take a few minutes.
5. In the terminal at the bottom of the screen, sign in to Azure with the Azure Developer CLI:

```
Bash
azd auth login
```

6. Copy the code from the terminal and then paste it into a browser. Follow the instructions to authenticate with your Azure account.

The remaining tasks in this article take place in the context of this development container.

Custom settings

This solution configures and deploys the infrastructure based on custom settings configured with the Azure Developer CLI. The following table explains the custom settings for this solution.

[] Expand table

Setting	Description
AZURE_PUBLIC_NETWORK_ACCESS	Controls the value of public network access on supported Azure resources. Valid values are <code>Enabled</code> or <code>Disabled</code> .
AZURE_USE_PRIVATE_ENDPOINT	Controls deployment of private endpoints, which connect Azure resources to the virtual network. The <code>TRUE</code> value means that private endpoints are deployed for connectivity.

Deploy the chat app

The first deployment creates the resources and provides a publicly accessible endpoint.

1. Run the following command to configure this solution for public access:

Console

```
azd env set AZURE_PUBLIC_NETWORK_ACCESS Enabled
```

When you're asked for an environment name, remember that the environment name is used to create the resource group. Enter a meaningful name. If you're on a team or in an organization, include your name, as in `morgan-chat-private-endpoints`. Make note of the environment name. You need it later to find the resources in the Azure portal.

2. Run the following command to include provisioning the virtual network resources. Remember that the deployment doesn't restrict access until the second deployment.

Console

```
azd env set AZURE_USE_PRIVATE_ENDPOINT true
```

3. Deploy the solution with the following command:

Console

```
azd up
```

Provisioning resources is the most time-consuming part of the deployment process. Wait for the deployment to finish before you continue.

- At the end of the deployment process, the app endpoint appears. Copy that endpoint into a browser to open the chat app. Select one of the questions on the cards and then wait for the answer.

Make note of the endpoint URL because you need it again later in the article.

Deploy the chat app to Azure with private access

Change the deployment configuration to secure the chat app for private access.

- Run the following command to turn off public access:

```
Console  
azd env set AZURE_PUBLIC_NETWORK_ACCESS Disabled
```

- Run the following command to change the resource configuration. This command doesn't redeploy the application code because that code hasn't changed.

```
Console  
azd provision
```

- After the provisioning finishes, open the chat app in a browser again. The chat app is no longer accessible because the public endpoint is disabled.

Access the chat app

To access the chat app, use a tool such as [Azure VPN Gateway](#) or [Azure Virtual Desktop](#). Remember that any tool you use to access the app must be secure and compliant with your organization's security policies.

Clean up resources

The following steps walk you through the process of cleaning up the resources you used.

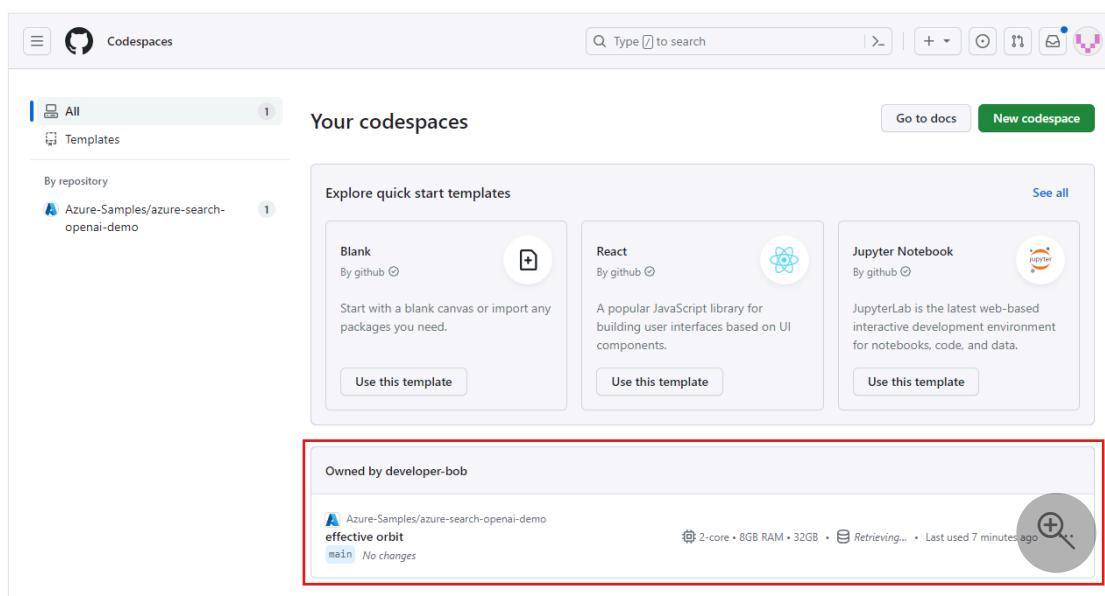
GitHub Codespaces

Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement that you get for your account.

ⓘ Important

For more information about your GitHub account's entitlements, see [GitHub Codespaces monthly included storage and core hours](#).

1. Sign in to the [GitHub Codespaces dashboard](#).
2. Locate your currently running codespaces that are sourced from the [Azure-Samples/azure-search-openai-demo](#) GitHub repository.



3. Open the context menu for the codespace and then select **Delete**.

The screenshot shows the GitHub Codespaces interface. At the top, there's a search bar and various navigation icons. Below that, a sidebar on the left lists 'All' (1), 'Templates', and a specific repository 'Azure-Samples/azure-search-openai-demo' (1). The main area is titled 'Your codespaces' and features a section for 'Explore quick start templates' with options for 'Blank', 'React', and 'Jupyter Notebook'. Below this is a section for repositories owned by 'developer-bob', specifically 'Azure-Samples/azure-search-openai-demo'. This card shows details like 'main', 'No changes', and machine specs ('2-core + 8GB RAM + 32GB'). A context menu is open over this card, with the 'Delete' option highlighted by a red box. The bottom of the interface has a navigation bar with links like Terms, Privacy, Security, Status, Docs, Contact GitHub, Pricing, API, Training, Blog, and About.

Get help

This sample repository offers [troubleshooting information](#).

If your issue isn't addressed, add your issue to the repository's [Issues](#) webpage.

Related content

- See the [enterprise chat app GitHub repository](#).
- Build a [chat app with Azure OpenAI](#) best-practices solution architecture.
- Learn about [access control in generative AI apps with Azure AI Search](#).

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Get started with evaluating answers in a chat app in Python

06/23/2025

This article shows you how to evaluate a chat app's answers against a set of correct or ideal answers (known as ground truth). Whenever you change your chat application in a way that affects the answers, run an evaluation to compare the changes. This demo application offers tools that you can use today to make it easier to run evaluations.

By following the instructions in this article, you:

- Use provided sample prompts tailored to the subject domain. These prompts are already in the repository.
- Generate sample user questions and ground truth answers from your own documents.
- Run evaluations by using a sample prompt with the generated user questions.
- Review analysis of answers.

(!) Note

This article uses one or more [AI app templates](#) as the basis for the examples and guidance in the article. AI app templates provide you with well-maintained reference implementations that are easy to deploy. They help to ensure a high-quality starting point for your AI apps.

Architectural overview

Key components of the architecture include:

- **Azure-hosted chat app:** The chat app runs in Azure App Service.
- **Microsoft AI Chat Protocol:** The protocol provides standardized API contracts across AI solutions and languages. The chat app conforms to the [Microsoft AI Chat Protocol](#), which allows the evaluations app to run against any chat app that conforms to the protocol.
- **Azure AI Search:** The chat app uses Azure AI Search to store the data from your own documents.
- **Sample questions generator:** The tool can generate many questions for each document along with the ground truth answer. The more questions there are, the longer the evaluations.

- **Evaluator:** The tool runs sample questions and prompts against the chat app and returns the results.
- **Review tool:** The tool reviews the results of the evaluations.
- **Diff tool:** The tool compares the answers between evaluations.

When you deploy this evaluation to Azure, the Azure OpenAI Service endpoint is created for the GPT-4 model with its own [capacity](#). When you evaluate chat applications, it's important that the evaluator has its own Azure OpenAI resource by using GPT-4 with its own capacity.

Prerequisites

- An Azure subscription. [Create one for free](#).
- Complete the [previous chat app procedure](#) to deploy the chat app to Azure. This resource is required for the evaluations app to work. Don't complete the "Clean up resources" section of the previous procedure.

You need the following Azure resource information from that deployment, which is referred to as the *chat app* in this article:

- Chat API URI. The service backend endpoint shown at the end of the `azd up` process.
- Azure AI Search. The following values are required:
 - **Resource name:** The name of the Azure AI Search resource name, reported as `Search service` during the `azd up` process.
 - **Index name:** The name of the Azure AI Search index where your documents are stored. You can find it in the Azure portal for the Search service.

The Chat API URL allows the evaluations to make requests through your backend application. The Azure AI Search information allows the evaluation scripts to use the same deployment as your backend, loaded with the documents.

After you have this information collected, you shouldn't need to use the chat app development environment again. This article refers to it later, several times, to indicate how the evaluations app uses the chat app. Don't delete the chat app resources until you finish the entire procedure in this article.

- A [development container](#) environment is available with all the dependencies that are required to complete this article. You can run the development container in GitHub Codespaces (in a browser) or locally by using Visual Studio Code.

GitHub Codespaces (recommended)

- GitHub account

Open a development environment

Follow these instructions to set up a preconfigured development environment with all the required dependencies to complete this article. Arrange your monitor workspace so that you can see this documentation and the development environment at the same time.

This article was tested with the `switzerlandnorth` region for the evaluation deployment.

GitHub Codespaces (recommended)

[GitHub Codespaces](#) runs a development container managed by GitHub with [Visual Studio Code for the Web](#) as the user interface. Use GitHub Codespaces for the easiest development environment. It comes with the right developer tools and dependencies preinstalled to complete this article.

Important

All GitHub accounts can use GitHub Codespaces for up to 60 hours free each month with two core instances. For more information, see [GitHub Codespaces monthly included storage and core hours](#).

1. Start the process to create a new GitHub codespace on the `main` branch of the [Azure-Samples/ai-rag-chat-evaluator](#) GitHub repository.
2. To display the development environment and the documentation available at the same time, right-click the following button, and select **Open link in new window**.



3. On the **Create codespace** page, review the codespace configuration settings, and then select **Create new codespace**.

Create codespace for Azure-Samples/ai-rag-chat-evaluator

The screenshot shows the 'Create codespace' dialog for the repository 'Azure-Samples/ai-rag-chat-evaluator'. It includes fields for Branch (main), Dev container configuration (AI RAG Chat Evaluator), Region (US West), Machine type (2-core), and a 'Create codespace' button.

Branch	main
Dev container configuration	AI RAG Chat Evaluator
Region	US West
Machine type	2-core
Create codespace	

4. Wait for the codespace to start. This startup process can take a few minutes.

5. In the terminal at the bottom of the screen, sign in to Azure with the Azure Developer CLI:

```
Bash
azd auth login --use-device-code
```

6. Copy the code from the terminal and then paste it into a browser. Follow the instructions to authenticate with your Azure account.

7. Provision the required Azure resource, Azure OpenAI Service, for the evaluations app:

```
Bash
azd up
```

This `AZD` command doesn't deploy the evaluations app, but it does create the Azure OpenAI resource with a required `GPT-4` deployment to run the evaluations in the local development environment.

The remaining tasks in this article take place in the context of this development container.

The name of the GitHub repository appears in the search bar. This visual indicator helps you distinguish the evaluations app from the chat app. This `ai-rag-chat-evaluator` repo is referred to as the *evaluations app* in this article.

Prepare environment values and configuration information

Update the environment values and configuration information with the information you gathered during [Prerequisites](#) for the evaluations app.

1. Create a `.env` file based on `.env.sample`.

Bash

```
cp .env.sample .env
```

2. Run this command to get the required values for `AZURE_OPENAI_EVAL_DEPLOYMENT` and `AZURE_OPENAI_SERVICE` from your deployed resource group. Paste those values into the `.env` file.

shell

```
azd env get-value AZURE_OPENAI_EVAL_DEPLOYMENT  
azd env get-value AZURE_OPENAI_SERVICE
```

3. Add the following values from the chat app for its Azure AI Search instance to the `.env` file, which you gathered in the [Prerequisites](#) section.

Bash

```
AZURE_SEARCH_SERVICE=<service-name>  
AZURE_SEARCH_INDEX=<index-name>
```

Use the Microsoft AI Chat Protocol for configuration information

The chat app and the evaluations app both implement the Microsoft AI Chat Protocol specification, an open-source, cloud, and language-agnostic AI endpoint API contract that's used for consumption and evaluation. When your client and middle-tier endpoints adhere to this API specification, you can consistently consume and run evaluations on your AI backends.

1. Create a new file named `my_config.json` and copy the following content into it:

JSON

```
{  
    "testdata_path": "my_input/qa.jsonl",  
    "results_dir": "my_results/experiment<TIMESTAMP>",  
    "target_url": "http://localhost:50505/chat",  
    "target_parameters": {  
        "overrides": {  
            "top": 3,  
            "temperature": 0.3,  
            "retrieval_mode": "hybrid",  
            "semantic_ranker": false,  
            "prompt_template": "<READFILE>my_input/prompt_refined.txt",  
            "seed": 1  
        }  
    }  
}
```

The evaluation script creates the `my_results` folder.

The `overrides` object contains any configuration settings that are needed for the application. Each application defines its own set of settings properties.

2. Use the following table to understand the meaning of the settings properties that are sent to the chat app.

 Expand table

Settings property	Description
<code>semantic_ranker</code>	Whether to use <code>semantic ranker</code> , a model that reranks search results based on semantic similarity to the user's query. We disable it for this tutorial to reduce costs.
<code>retrieval_mode</code>	The retrieval mode to use. The default is <code>hybrid</code> .
<code>temperature</code>	The temperature setting for the model. The default is <code>0.3</code> .
<code>top</code>	The number of search results to return. The default is <code>3</code> .
<code>prompt_template</code>	An override of the prompt used to generate the answer based on the question and search results.
<code>seed</code>	The seed value for any calls to GPT models. Setting a seed results in more consistent results across evaluations.

3. Change the `target_url` value to the URI value of your chat app, which you gathered in the [Prerequisites](#) section. The chat app must conform to the chat protocol. The URI has the following format: `https://CHAT-APP-URL/chat`. Make sure the protocol and the `chat` route are part of the URI.

Generate sample data

To evaluate new answers, they must be compared to a *ground truth* answer, which is the ideal answer for a particular question. Generate questions and answers from documents that are stored in Azure AI Search for the chat app.

1. Copy the `example_input` folder into a new folder named `my_input`.
2. In a terminal, run the following command to generate the sample data:

```
Bash
```

```
python -m evaltools generate --output=my_input/qa.jsonl --persource=2 --numquestions=14
```

The question-and-answer pairs are generated and stored in `my_input/qa.jsonl` (in [JSONL format](#)) as input to the evaluator that's used in the next step. For a production evaluation, you would generate more question-and-answer pairs. More than 200 are generated for this dataset.

 **Note**

Only a few questions and answers are generated per source so that you can quickly complete this procedure. It isn't meant to be a production evaluation, which should have more questions and answers per source.

Run the first evaluation with a refined prompt

1. Edit the `my_config.json` configuration file properties.

 [Expand table](#)

Property	New value
<code>results_dir</code>	<code>my_results/experiment_refined</code>

Property	New value
prompt_template	<READFILE>my_input/prompt_refined.txt

The refined prompt is specific about the subject domain.

txt
If there isn't enough information below, say you don't know. Do not generate answers that don't use the sources below. If asking a clarifying question to the user would help, ask the question.
Use clear and concise language and write in a confident yet friendly tone. In your answers, ensure the employee understands how your response connects to the information in the sources and include all citations necessary to help the employee validate the answer provided.
For tabular information, return it as an html table. Do not return markdown format. If the question is not in English, answer in the language used in the question.
Each source has a name followed by a colon and the actual information. Always include the source name for each fact you use in the response. Use square brackets to reference the source, e.g. [info1.txt]. Don't combine sources, list each source separately, e.g. [info1.txt][info2.pdf].

2. In a terminal, run the following command to run the evaluation:

Bash
<code>python -m evaltools evaluate --config=my_config.json --numquestions=14</code>

This script created a new experiment folder in `my_results/` with the evaluation. The folder contains the results of the evaluation.

[Expand table](#)

File name	Description
config.json	A copy of the configuration file used for the evaluation.
evaluate_parameters.json	The parameters used for the evaluation. Similar to <code>config.json</code> but includes other metadata like time stamp.
eval_results.jsonl	Each question and answer, along with the GPT metrics for each question-and-answer pair.
summary.json	The overall results, like the average GPT metrics.

Run the second evaluation with a weak prompt

1. Edit the `my_config.json` configuration file properties.

[] Expand table

Property	New value
<code>results_dir</code>	<code>my_results/experiment_weak</code>
<code>prompt_template</code>	<code><READFILE>my_input/prompt_weak.txt</code>

That weak prompt has no context about the subject domain.

```
txt
You are a helpful assistant.
```

2. In a terminal, run the following command to run the evaluation:

```
Bash
python -m evaltools evaluate --config=my_config.json --numquestions=14
```

Run the third evaluation with a specific temperature

Use a prompt that allows for more creativity.

1. Edit the `my_config.json` configuration file properties.

[] Expand table

Existing	Property	New value
Existing	<code>results_dir</code>	<code>my_results/experiment_ignoreresources_temp09</code>
Existing	<code>prompt_template</code>	<code><READFILE>my_input/prompt_ignoreresources.txt</code>
New	<code>temperature</code>	<code>0.9</code>

The default `temperature` is 0.7. The higher the temperature, the more creative the answers.

The `ignore` prompt is short.

text

```
Your job is to answer questions to the best of your ability. You will be given sources but you should IGNORE them. Be creative!
```

2. The configuration object should look like the following example, except that you replaced `results_dir` with your path:

JSON

```
{  
    "testdata_path": "my_input/qa.jsonl",  
    "results_dir": "my_results/prompt_ignoresources_temp09",  
    "target_url": "https://YOUR-CHAT-APP/chat",  
    "target_parameters": {  
        "overrides": {  
            "temperature": 0.9,  
            "semantic_ranker": false,  
            "prompt_template": "<READFILE>my_input/prompt_ignoresources.txt"  
        }  
    }  
}
```

3. In a terminal, run the following command to run the evaluation:

Bash

```
python -m evaltools evaluate --config=my_config.json --numquestions=14
```

Review the evaluation results

You performed three evaluations based on different prompts and app settings. The results are stored in the `my_results` folder. Review how the results differ based on the settings.

1. Use the review tool to see the results of the evaluations.

Bash

```
python -m evaltools summary my_results
```

2. The results look *something* like:

folder	groundedness	%	relevance	%	coherence	%	citation	%	length
experiment_ignoresources_temp09	5.00	1.00	4.71	0.93	4.86	0.93	0.00		1063.14
experiment_refined	5.00	1.00	5.00	1.00	5.00	1.00	1.00		1404.79
experiment_weak	5.00	1.00	5.00	1.00	5.00	1.00	0.00		1331.57

Each value is returned as a number and a percentage.

- Use the following table to understand the meaning of the values.

[Expand table](#)

Value	Description
Groundedness	Checks how well the model's responses are based on factual, verifiable information. A response is considered grounded if it's factually accurate and reflects reality.
Relevance	Measures how closely the model's responses align with the context or the prompt. A relevant response directly addresses the user's query or statement.
Coherence	Checks how logically consistent the model's responses are. A coherent response maintains a logical flow and doesn't contradict itself.
Citation	Indicates if the answer was returned in the format requested in the prompt.
Length	Measures the length of the response.

- The results should indicate that all three evaluations had high relevance while the `experiment_ignoresources_temp09` had the lowest relevance.

- Select the folder to see the configuration for the evaluation.

- Enter `Ctrl + C` to exit the app and return to the terminal.

Compare the answers

Compare the returned answers from the evaluations.

- Select two of the evaluations to compare, and then use the same review tool to compare the answers.

Bash

```
python -m evaltools diff my_results/experiment_refined
my_results/experiment_ignoresources_temp09
```

- Review the results. Your results might vary.

What should one expect when choosing an out-of-network provider or services not covered under the Northwind Standard plan?

experiment_refined

When choosing an out-of-network provider or services not covered under the Northwind Standard plan, there are a few things you should expect:

1. Limited or no coverage: The Northwind Standard plan does not provide coverage for services received from health care providers who are not contracted with Northwind Health [Northwind_Standard_Benefits_Details.pdf#page=89].
2. Out-of-pocket expenses: If you choose to receive services from an out-of-network provider or

groundedness	relevance	coherence
5	5	5

experiment_ignoresources_temp09

When choosing an out-of-network provider or services not covered under the Northwind Standard plan, you should expect to incur additional out-of-pocket costs. These costs can vary depending on the specific provider or service you choose. It is important to note that the Northwind Standard plan does not provide coverage for any out-of-network services, so you will likely be responsible for paying the full cost.

To minimize your out-of-pocket costs and ensure that you are receiving the best care possible, you should

groundedness	relevance	coherence
5	5	5

3. Enter `Ctrl + C` to exit the app and return to the terminal.

Suggestions for further evaluations

- Edit the prompts in `my_input` to tailor the answers such as subject domain, length, and other factors.
- Edit the `my_config.json` file to change the parameters such as `temperature`, and `semantic_ranker` and rerun experiments.
- Compare different answers to understand how the prompt and question affect the answer quality.
- Generate a separate set of questions and ground truth answers for each document in the Azure AI Search index. Then rerun the evaluations to see how the answers differ.
- Alter the prompts to indicate shorter or longer answers by adding the requirement to the end of the prompt. An example is `Please answer in about 3 sentences.`

Clean up resources and dependencies

The following steps walk you through the process of cleaning up the resources you used.

Clean up Azure resources

The Azure resources created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges.

To delete the Azure resources and remove the source code, run the following Azure Developer CLI command:

```
Bash
```

```
azd down --purge
```

Clean up GitHub Codespaces and Visual Studio Code

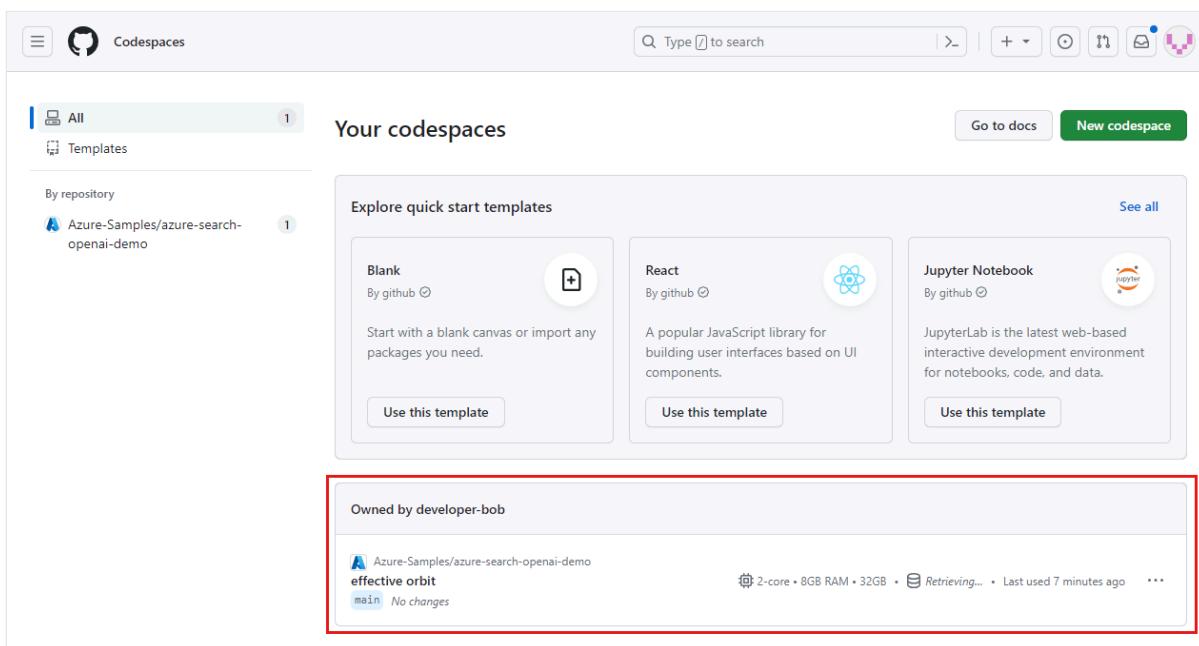
GitHub Codespaces

Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement that you get for your account.

ⓘ Important

For more information about your GitHub account's entitlements, see [GitHub Codespaces monthly included storage and core hours ↗](#).

1. Sign in to the [GitHub Codespaces dashboard ↗](#).
2. Locate your currently running codespaces that are sourced from the [Azure-Samples/ai-rag-chat-evaluator ↗](#) GitHub repository.



3. Open the context menu for the codespace, and then select **Delete**.

The screenshot shows the GitHub Codespaces interface. At the top, there's a search bar and several navigation icons. Below that, a sidebar on the left lists 'All' (1), 'Templates', and a section 'By repository' containing 'Azure-Samples/azure-search-openai-demo' (1). The main area is titled 'Your codespaces' and features a section 'Explore quick start templates' with three cards: 'Blank', 'React', and 'Jupyter Notebook'. Each card has a 'Use this template' button. Below this is a section 'Owned by developer-bob' which lists the repository 'Azure-Samples/azure-search-openai-demo' with the branch 'effective orbit'. The card shows 'main' and 'No changes'. To the right of the card is a context menu with options: 'Open in ...', 'Rename', 'Export changes to a fork', 'Change machine type', 'Keep codespace', and 'Delete'. The 'Delete' button is highlighted with a red box. At the bottom of the page, there's a footer with links to GitHub's Terms, Privacy, Security, Status, Docs, Contact GitHub, Pricing, API, Training, Blog, and About.

Return to the chat app article to clean up those resources.

- [JavaScript](#)
- [Python](#)

Related content

- See the [evaluations repository](#).
- See the [enterprise chat app GitHub repository](#).
- Build a [chat app with Azure OpenAI](#) best-practices solution architecture.
- Learn about [access control in generative AI apps with Azure AI Search](#).
- Build an [enterprise-ready Azure OpenAI solution with Azure API Management](#).
- See [Azure AI Search: Outperforming vector search with hybrid retrieval and ranking capabilities](#).

Scale Azure OpenAI for Python chat by using RAG with Azure Container Apps

06/26/2025

Learn how to add load balancing to your application to extend the chat app beyond the Azure OpenAI Service token and model quota limits. This approach uses Azure Container Apps to create three Azure OpenAI endpoints and a primary container to direct incoming traffic to one of the three endpoints.

This article requires you to deploy two separate samples:

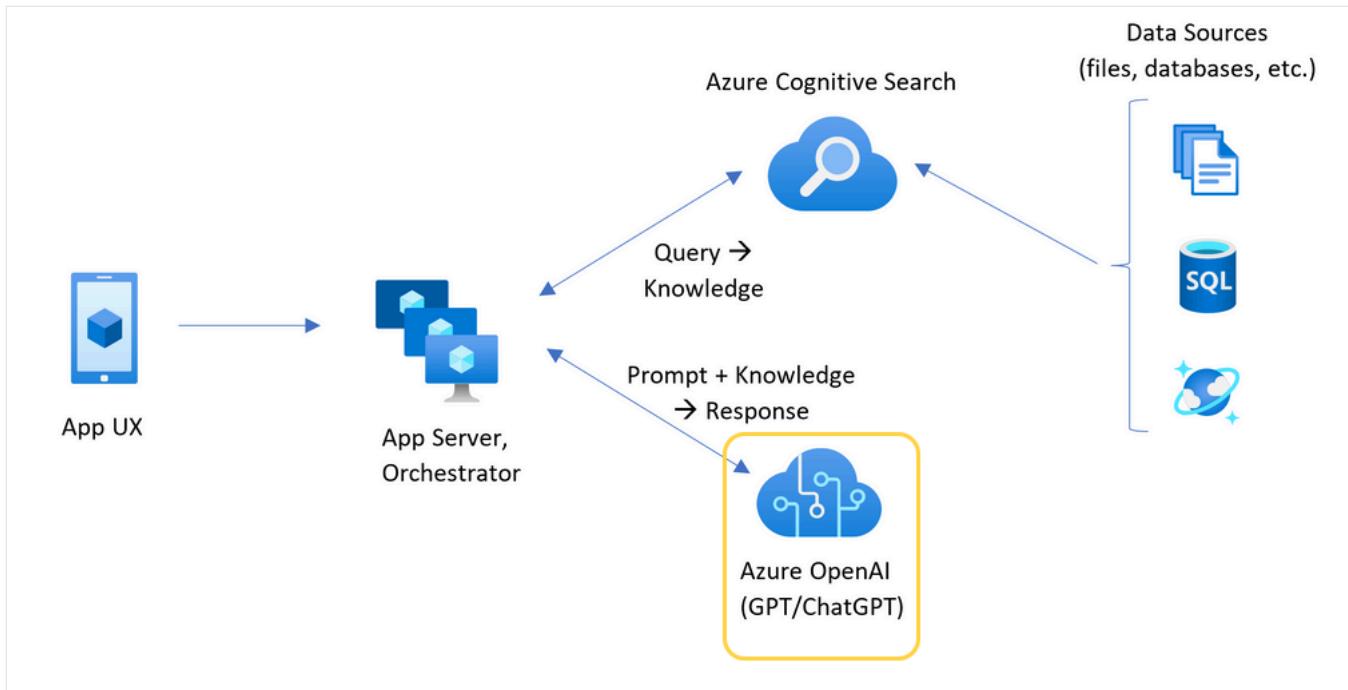
- Chat app
 - If you haven't deployed the chat app yet, wait until after the load balancer sample is deployed.
 - If you already deployed the chat app once, change the environment variable to support a custom endpoint for the load balancer and redeploy it again.
 - The chat app is available in these languages:
 - [.NET](#)
 - [JavaScript](#)
 - [Python](#)
- Load balancer app

(!) Note

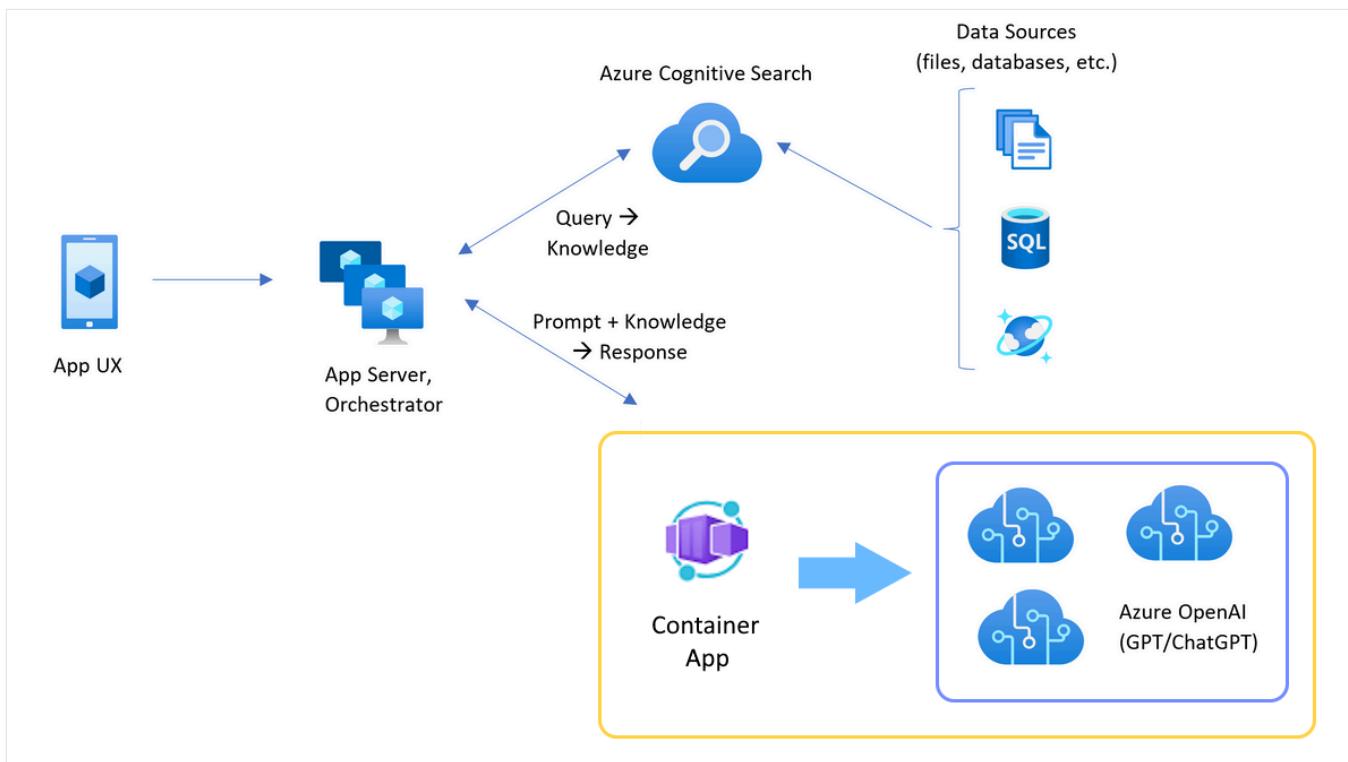
This article uses one or more [AI app templates](#) as the basis for the examples and guidance in the article. AI app templates provide you with well-maintained reference implementations that are easy to deploy. They help to ensure a high-quality starting point for your AI apps.

Architecture for load balancing Azure OpenAI with Azure Container Apps

Because the Azure OpenAI resource has specific token and model quota limits, a chat app that uses a single Azure OpenAI resource is prone to have conversation failures because of those limits.

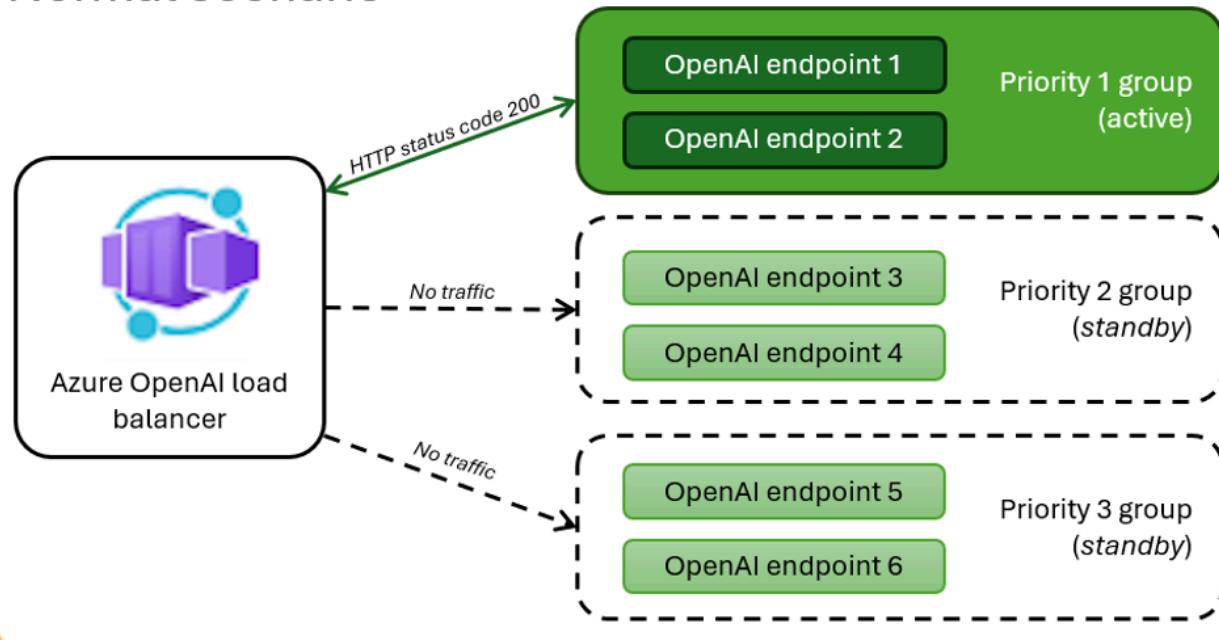


To use the chat app without hitting those limits, use a load-balanced solution with Container Apps. This solution seamlessly exposes a single endpoint from Container Apps to your chat app server.



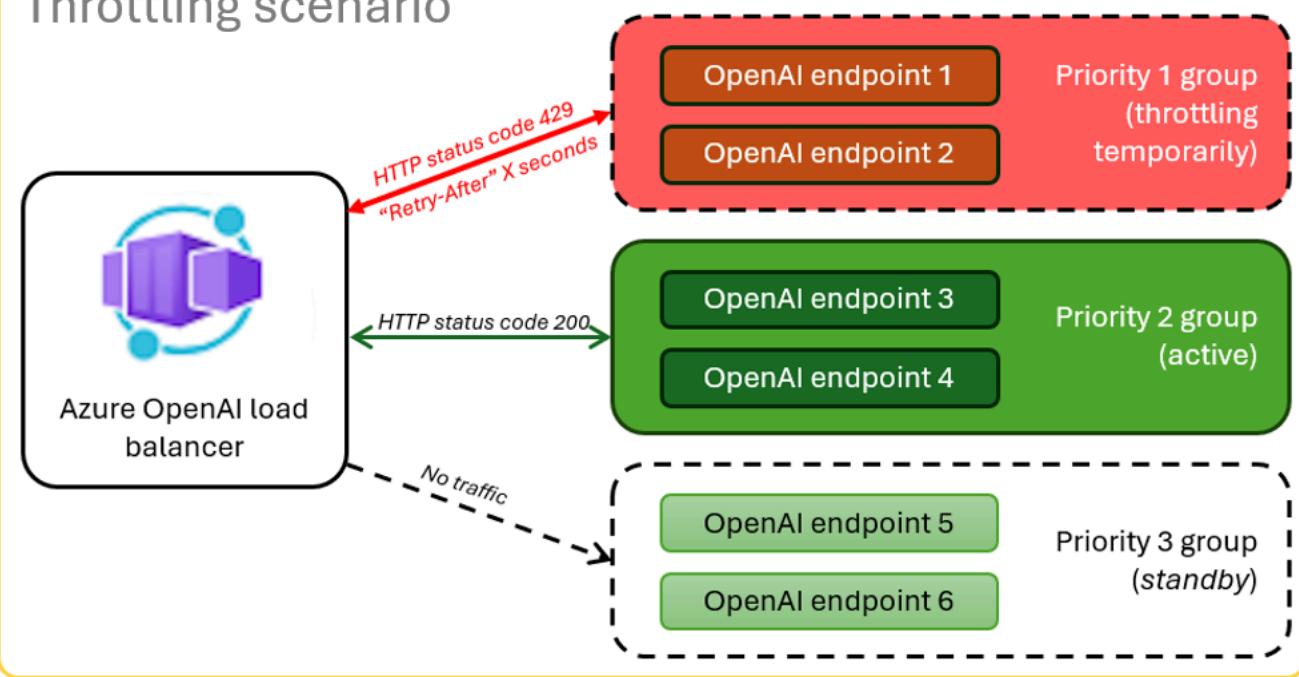
The container app sits in front of a set of Azure OpenAI resources. The container app solves two scenarios: normal and throttled. During a *normal scenario* where token and model quota is available, the Azure OpenAI resource returns a 200 back through the container app and app server.

Normal scenario



When a resource is in a *throttled scenario* because of quota limits, the container app can retry a different Azure OpenAI resource immediately to fulfill the original chat app request.

Throttling scenario



Prerequisites

- An Azure subscription. [Create one for free ↗](#).
- [Dev containers ↗](#) are available for both samples with all the dependencies that are required to complete this article. You can run the dev containers in GitHub Codespaces (in

a browser) or locally by using Visual Studio Code.

GitHub Codespaces (recommended)

- o GitHub account

Open the Container Apps load balancer sample app

GitHub Codespaces (recommended)

[GitHub Codespaces](#) runs a development container managed by GitHub with [Visual Studio Code for the Web](#) as the user interface. For the most straightforward development environment, use GitHub Codespaces so that you have the correct developer tools and dependencies preinstalled to complete this article.



[Open in GitHub Codespaces](#)

Important

All GitHub accounts can use GitHub Codespaces for up to 60 hours free each month with two core instances. For more information, see [GitHub Codespaces monthly included storage and core hours](#).

Deploy the Azure Container Apps load balancer

1. Sign in to the Azure Developer CLI to provide authentication to the provisioning and deployment steps:

Bash

```
azd auth login --use-device-code
```

2. Set an environment variable to use Azure CLI authentication to the post provision step:

Bash

```
azd config set auth.useAzCliAuth "true"
```

3. Deploy the load balancer app:

```
Bash
```

```
azd up
```

Select a subscription and region for the deployment. They don't have to be the same subscription and region as the chat app.

4. Wait for the deployment to finish before you continue.

Get the deployment endpoint

1. Use the following command to display the deployed endpoint for the container app:

```
Bash
```

```
azd env get-values
```

2. Copy the `CONTAINER_APP_URL` value. You use it in the next section.

Redeploy the chat app with the load balancer endpoint

These examples are completed on the chat app sample.

Initial deployment

1. Open the chat app sample's dev container by using one of the following choices.

[Expand table](#)

Language	GitHub Codespaces	Visual Studio Code
.NET	 Open in GitHub Codespaces 	Dev Containers Open 
JavaScript	 Open in GitHub Codespaces 	Dev Containers Open 
Python	 Open in GitHub Codespaces 	Dev Containers Open 

2. Sign in to the Azure Developer CLI (azd):

```
Bash
```

```
azd auth login
```

Finish the sign-in instructions.

3. Create an azd environment with a name such as chat-app:

```
Bash
```

```
azd env new <name>
```

4. Add the following environment variable, which tells the chat app's backend to use a custom URL for the Azure OpenAI requests:

```
Bash
```

```
azd env set OPENAI_HOST azure_custom
```

5. Add the following environment variable. Substitute <CONTAINER_APP_URL> for the URL from the previous section. This action tells the chat app's backend what the value is of the custom URL for the Azure OpenAI request.

```
Bash
```

```
azd env set AZURE_OPENAI_CUSTOM_URL <CONTAINER_APP_URL>
```

6. Deploy the chat app:

```
Bash
```

```
azd up
```

Use the chat app with the confidence that it scales across many users without running out of quota.

Stream logs to see the load balancer results

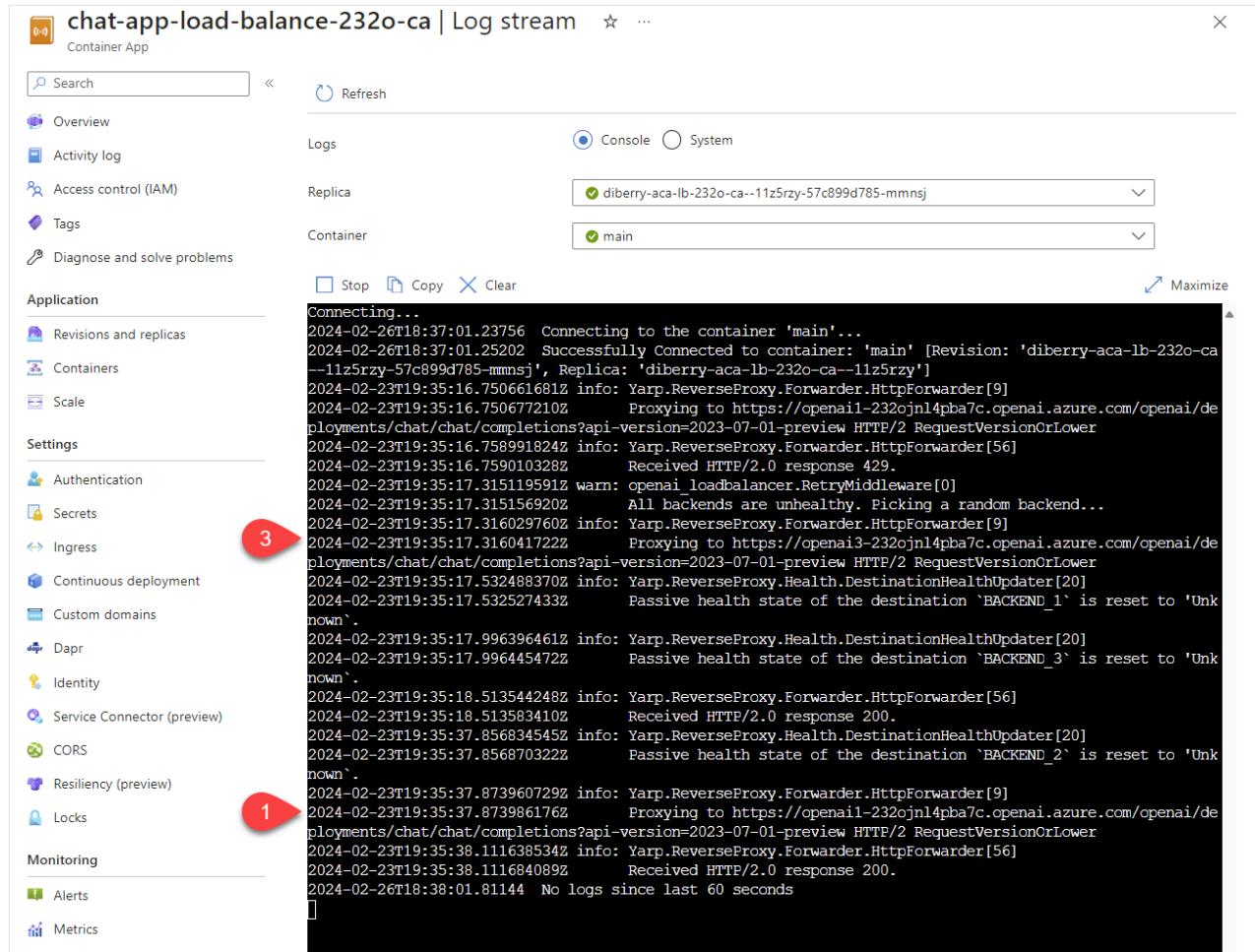
1. In the [Azure portal](#), search your resource group.

2. From the list of resources in the group, select the Azure Container Apps resource.

3. Select **Monitoring > Log stream** to view the log.

4. Use the chat app to generate traffic in the log.

5. Look for the logs, which reference the Azure OpenAI resources. Each of the three resources has its numeric identity in the log comment that begins with `Proxying to https://openai3`, where `3` indicates the third Azure OpenAI resource.



The screenshot shows the Azure Container Apps Log stream interface. The left sidebar includes sections for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Application (Revisions and replicas, Containers, Scale), Settings (Authentication, Secrets, Ingress, Continuous deployment, Custom domains, Dapr, Identity, Service Connector (preview), CORS, Resiliency (preview), Locks), Monitoring (Alerts, Metrics), and a search bar. The main area displays logs for a Replica ('diberry-aca-lb-232o-ca--11z5rzy-57c899d785-mmnsj') and Container ('main'). The log output shows the application connecting to the container and proxying requests to Azure OpenAI instances. Red arrows labeled '1' and '3' highlight the 'Monitoring' section in the sidebar and the proxying logs in the main pane respectively.

```
Connecting...
2024-02-26T18:37:01.23756 Connecting to the container 'main'...
2024-02-26T18:37:01.25202 Successfully Connected to container: 'main' [Revision: 'diberry-aca-lb-232o-ca--11z5rzy-57c899d785-mmnsj', Replica: 'diberry-aca-lb-232o-ca--11z5rzy']
2024-02-23T19:35:16.750661681Z info: Yarp.ReverseProxy.Forwarder.HttpForwarder[9]
2024-02-23T19:35:16.750677210Z Proxying to https://openai3-232ojnl4pba7c.openai.azure.com/openai/deployments/chat/chat/completions?api-version=2023-07-01-preview HTTP/2 RequestVersionOrLower
2024-02-23T19:35:16.758991824Z info: Yarp.ReverseProxy.Forwarder.HttpForwarder[56]
2024-02-23T19:35:16.759010328Z Received HTTP/2.0 response 429.
2024-02-23T19:35:17.315119591Z warn: openai.loadbalancer.RetryMiddleware[0]
2024-02-23T19:35:17.315156920Z All backends are unhealthy. Picking a random backend...
2024-02-23T19:35:17.316029760Z info: Yarp.ReverseProxy.Forwarder.HttpForwarder[9]
2024-02-23T19:35:17.316041722Z Proxying to https://openai3-232ojnl4pba7c.openai.azure.com/openai/deployments/chat/chat/completions?api-version=2023-07-01-preview HTTP/2 RequestVersionOrLower
2024-02-23T19:35:17.532488370Z info: Yarp.ReverseProxy.Health.DestinationHealthUpdater[20]
2024-02-23T19:35:17.532527433Z Passive health state of the destination 'BACKEND_1' is reset to 'Unknown'.
2024-02-23T19:35:17.996396461Z info: Yarp.ReverseProxy.Health.DestinationHealthUpdater[20]
2024-02-23T19:35:17.996445472Z Passive health state of the destination 'BACKEND_3' is reset to 'Unknown'.
2024-02-23T19:35:18.513544248Z info: Yarp.ReverseProxy.Forwarder.HttpForwarder[56]
2024-02-23T19:35:18.513583410Z Received HTTP/2.0 response 200.
2024-02-23T19:35:37.8568634545Z info: Yarp.ReverseProxy.Health.DestinationHealthUpdater[20]
2024-02-23T19:35:37.856870322Z Passive health state of the destination 'BACKEND_2' is reset to 'Unknown'.
2024-02-23T19:35:37.873960729Z info: Yarp.ReverseProxy.Forwarder.HttpForwarder[9]
2024-02-23T19:35:37.873986176Z Proxying to https://openai1-232ojnl4pba7c.openai.azure.com/openai/deployments/chat/chat/completions?api-version=2023-07-01-preview HTTP/2 RequestVersionOrLower
2024-02-23T19:35:38.111638534Z info: Yarp.ReverseProxy.Forwarder.HttpForwarder[56]
2024-02-23T19:35:38.111684089Z Received HTTP/2.0 response 200.
2024-02-26T18:38:01.81144 No logs since last 60 seconds
[]
```

When the load balancer receives status that the request exceeds quota, the load balancer automatically rotates to another resource.

Configure the TPM quota

By default, each of the Azure OpenAI instances in the load balancer is deployed with a capacity of 30,000 tokens per minute (TPM). You can use the chat app with the confidence that it scales across many users without running out of quota. Change this value when:

- You get deployment capacity errors: Lower the value.
- You need higher capacity: Raise the value.

1. Use the following command to change the value:

```
Bash
```

```
azd env set OPENAI_CAPACITY 50
```

2. Redeploy the load balancer:

```
Bash
```

```
azd up
```

Clean up resources

When you're finished with the chat app and the load balancer, clean up the resources. The Azure resources created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges.

Clean up chat app resources

Return to the chat app article to clean up the resources:

- [.NET](#)
- [JavaScript](#)
- [Python](#)

Clean upload balancer resources

Delete the Azure resources and remove the source code:

```
Bash
```

```
azd down --purge --force
```

The switches provide:

- `purge`: Deleted resources are immediately purged so that you can reuse the Azure OpenAI Service tokens per minute.
- `force`: The deletion happens silently, without requiring user consent.

Clean up GitHub Codespaces and Visual Studio Code

Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement that you get for your account.

ⓘ Important

For more information about your GitHub account's entitlements, see [GitHub Codespaces monthly included storage and core hours](#).

1. Sign in to the [GitHub Codespaces dashboard](#).
2. Locate your currently running codespaces that are sourced from the [azure-samples/openai-aca-lb](#) GitHub repository.

The screenshot shows the 'Your codespaces' page. On the left, there are filters for 'All' (selected), 'Templates', and 'By repository'. A repository filter is set to 'Azure-Samples/openai-aca-lb'. The main area displays a single codespace entry:

- Owned by Azure-Samples**
- A Azure-Samples/openai-aca-lb**
- stunning orbit**
- main**
- 2-core • Retrieving... • Active**
- ...**

3. Open the context menu for the codespace, and then select **Delete**.

The screenshot shows the context menu for the 'stunning orbit' codespace. The menu items include:

- Rename
- Export changes to a fork
- Change machine type
- Stop codespace
- Auto-delete codespace (with a checked checkbox)
- Open in Browser
- Open in Visual Studio Code
- Open in JetBrains Gateway (Beta)
- Open in JupyterLab (Beta)
- Delete (highlighted with a red box and a red arrow)

At the bottom of the menu, there is a red button labeled '1'.

Get help

If you have trouble deploying the Azure Container Apps load balancer, add your issue to the repository's [Issues ↗](#) webpage.

Sample code

Samples used in this article include:

- [Python chat app with Retrieval Augmented Generation \(RAG\) ↗](#)
- [Azure Load Balancer with Azure Container Apps ↗](#)

Related content

Use [Azure Load Testing](#) to load test your chat app.

Scale Azure OpenAI for Python with Azure API Management

06/27/2025

Learn how to add enterprise-grade load balancing to your application to extend the chat app beyond the Azure OpenAI Service token and model quota limits. This approach uses Azure API Management to intelligently direct traffic between three Azure OpenAI resources.

This article requires you to deploy two separate samples:

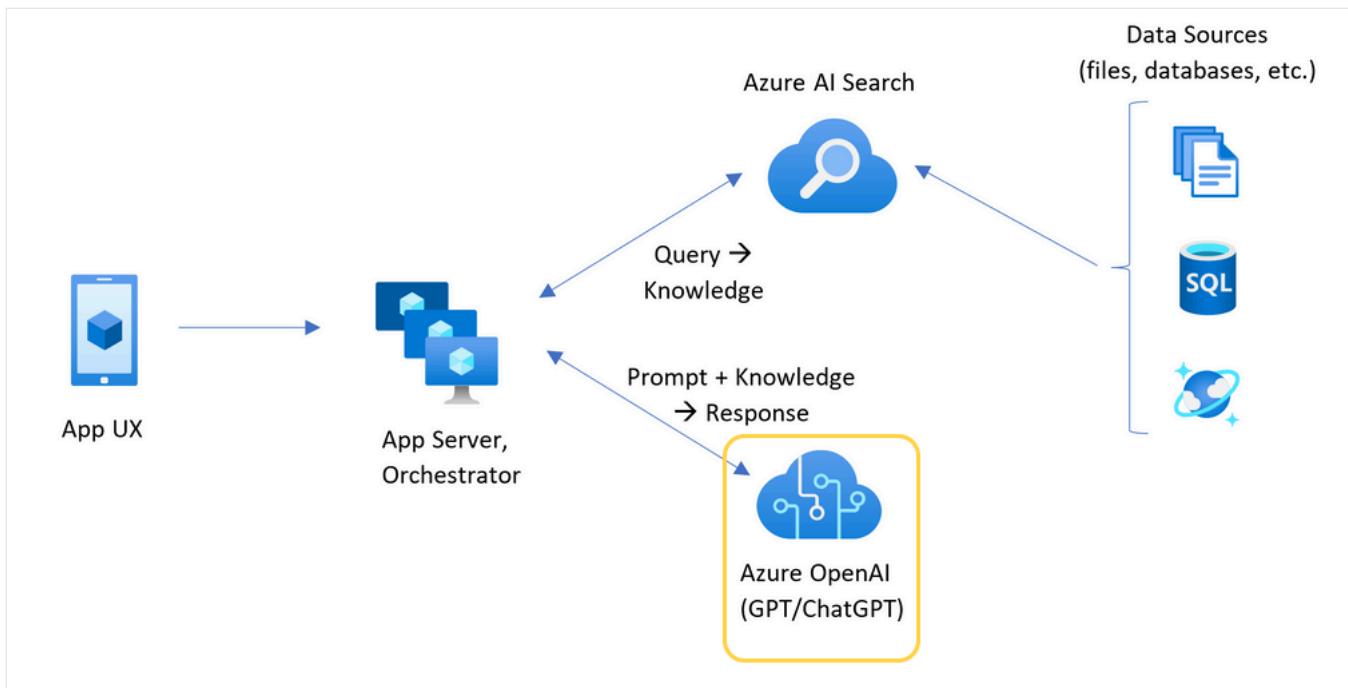
- Chat app:
 - Wait to deploy the chat app until after the load balancer sample is deployed.
 - If you already deployed the chat app once, change the environment variable to support a custom endpoint for the load balancer and redeploy it again.
- Load balancer with Azure API Management.

(!) Note

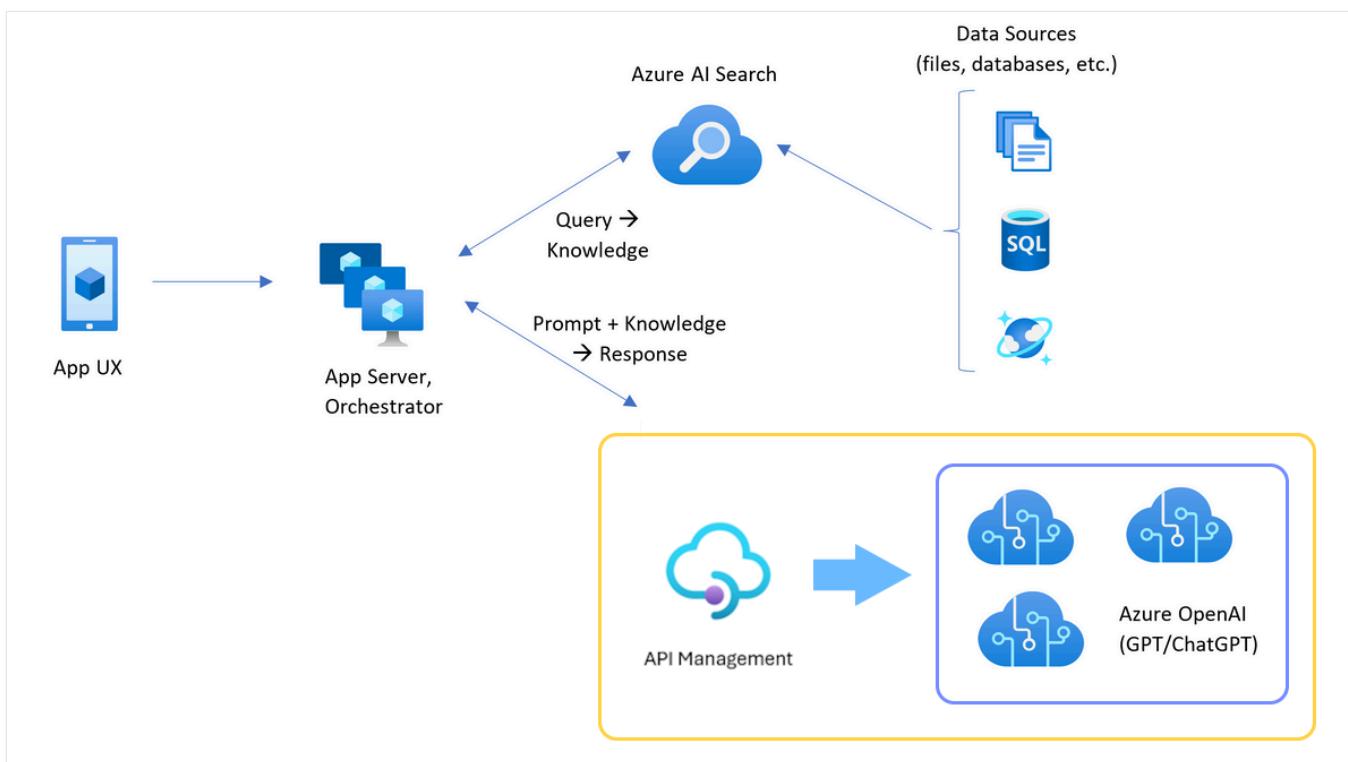
This article uses one or more [AI app templates](#) as the basis for the examples and guidance in the article. AI app templates provide you with well-maintained reference implementations that are easy to deploy. They help to ensure a high-quality starting point for your AI apps.

Architecture for load balancing Azure OpenAI with Azure API Management

Because the Azure OpenAI resource has specific token and model quota limits, a chat app that uses a single Azure OpenAI resource is prone to have conversation failures because of those limits.

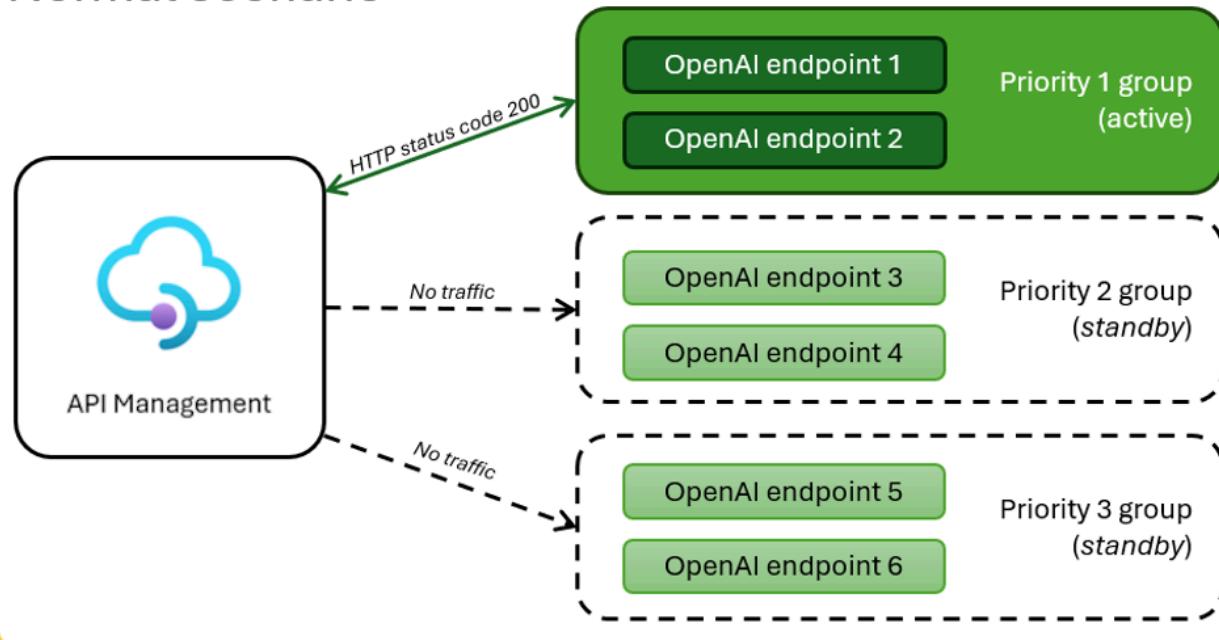


To use the chat app without hitting those limits, use a load-balanced solution with API Management. This solution seamlessly exposes a single endpoint from API Management to your chat app server.



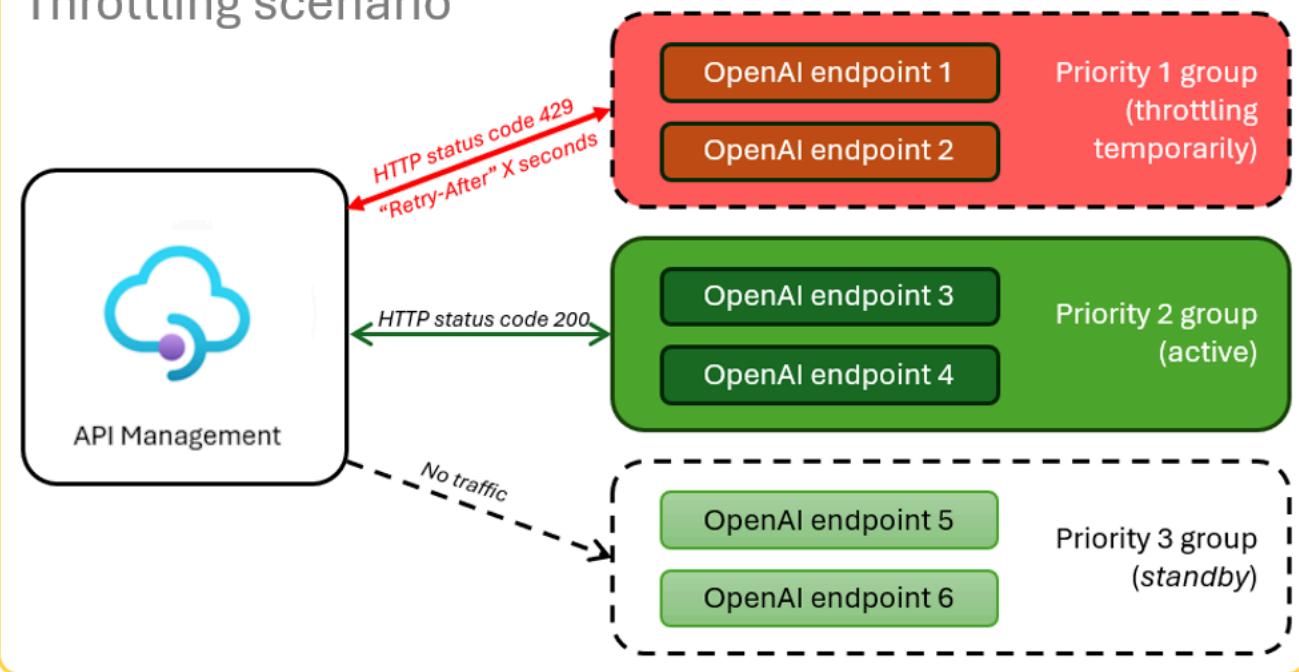
The API Management resource, as an API layer, sits in front of a set of Azure OpenAI resources. The API layer applies to two scenarios: normal and throttled. During a *normal scenario* where token and model quota is available, the Azure OpenAI resource returns a 200 back through the API layer and backend app server.

Normal scenario



When a resource is *throttled* because of quota limits, the API layer can retry a different Azure OpenAI resource immediately to fulfill the original chat app request.

Throttling scenario



Prerequisites

- An Azure subscription. [Create one for free ↗](#).
- [Dev containers ↗](#) are available for both samples, with all the dependencies that are required to complete this article. You can run the dev containers in GitHub Codespaces (in

a browser) or locally by using Visual Studio Code.

GitHub Codespaces (recommended)

- Only a [GitHub account](#) is required to use GitHub Codespaces.

Open the Azure API Management local balancer sample app

GitHub Codespaces (recommended)

[GitHub Codespaces](#) runs a development container managed by GitHub with [Visual Studio Code for the Web](#) as the user interface. For the most straightforward development environment, use GitHub Codespaces so that you have the correct developer tools and dependencies preinstalled to complete this article.



[Open in GitHub Codespaces](#)

Important

All GitHub accounts can use GitHub Codespaces for up to 60 hours free each month with two core instances. For more information, see [GitHub Codespaces monthly included storage and core hours](#).

Deploy the Azure API Management load balancer

- To deploy the load balancer to Azure, sign in to the Azure Developer CLI (`AZD`):

Bash

```
azd auth login
```

- Finish the sign-in instructions.

- Deploy the load balancer app:

Bash

```
azd up
```

Select a subscription and region for the deployment. They don't have to be the same subscription and region as the chat app.

4. Wait for the deployment to finish before you continue. This process might take up to 30 minutes.

Get the load balancer endpoint

Run the following Bash command to see the environment variables from the deployment. You need this information later.

```
Bash
```

```
azd env get-values | grep APIM_GATEWAY_URL
```

Redeploy the chat app with the load balancer endpoint

These examples are completed on the chat app sample.

Initial deployment

1. Open the chat app sample's dev container by using one of the following choices.

[Expand table](#)

Language	GitHub Codespaces	Visual Studio Code
.NET	 Open in GitHub Codespaces 	Dev Containers 
JavaScript	 Open in GitHub Codespaces 	Dev Containers 
Python	 Open in GitHub Codespaces 	Dev Containers 

2. Sign in to the Azure Developer CLI (`AZD`):

```
Bash
```

```
azd auth login
```

Finish the sign-in instructions.

3. Create an AZD environment with a name such as `chat-app`:

```
Bash
```

```
azd env new <name>
```

4. Add the following environment variable, which tells the chat app's backend to use a custom URL for the Azure OpenAI requests:

```
Bash
```

```
azd env set OPENAI_HOST azure_custom
```

5. Add this environment variable to tell the chat app's backend the custom URL for the Azure OpenAI request:

```
Bash
```

```
azd env set AZURE_OPENAI_CUSTOM_URL <APIM_GATEWAY_URL>
```

6. Deploy the chat app:

```
Bash
```

```
azd up
```

Configure the TPM quota

By default, each of the Azure OpenAI instances in the load balancer is deployed with a capacity of 30,000 tokens per minute (TPM). You can use the chat app with the confidence that it scales across many users without running out of quota. Change this value when:

- You get deployment capacity errors: Lower the value.
- You need higher capacity: Raise the value.

1. Use the following command to change the value:

```
Bash
```

```
azd env set OPENAI_CAPACITY 50
```

2. Redeploy the load balancer:

```
Bash
```

```
azd up
```

Clean up resources

When you're finished with the chat app and the load balancer, clean up the resources. The Azure resources created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges.

Clean up the chat app resources

Return to the chat app article to clean up those resources.

- [.NET](#)
- [JavaScript](#)
- [Python](#)

Clean up the load balancer resources

Delete the Azure resources and remove the source code:

```
Bash
```

```
azd down --purge --force
```

The switches provide:

- `purge`: Deleted resources are immediately purged. You can reuse the Azure OpenAI tokens per minute.
- `force`: The deletion happens silently, without requiring user consent.

Clean up resources

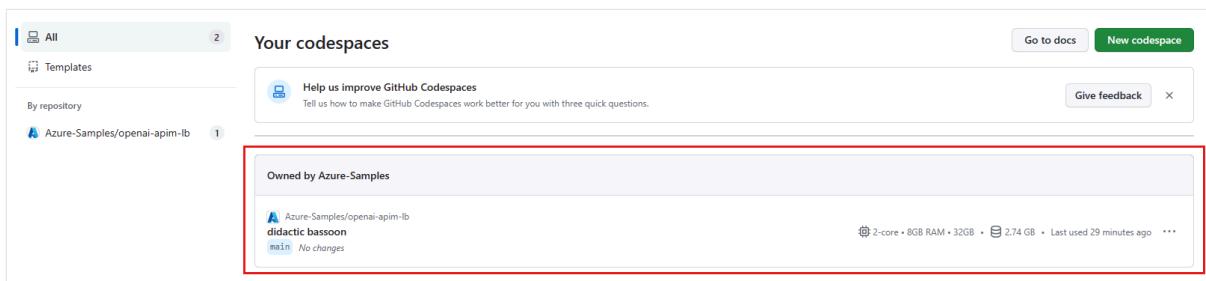
GitHub Codespaces

Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement that you get for your account.

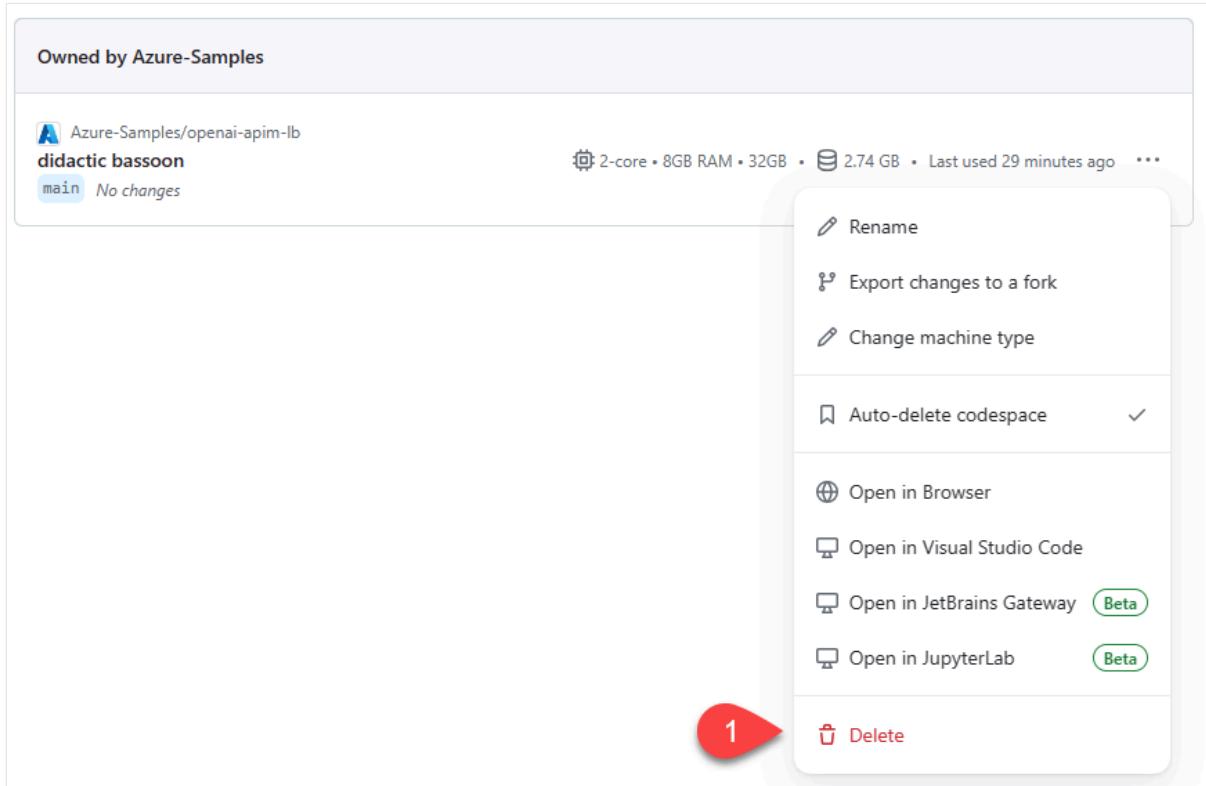
ⓘ Important

For more information about your GitHub account's entitlements, see [GitHub Codespaces monthly included storage and core hours](#).

1. Sign in to the [GitHub Codespaces dashboard](#).
2. Locate your currently running codespaces that are sourced from the [azure-samples/openai-apim-lb](#) GitHub repository.



3. Open the context menu for the GitHub Codespaces item, and then select **Delete**.



Get help

If you have trouble deploying the Azure API Management load balancer, add your issue to the repository's [Issues ↗](#) webpage.

Sample code

Samples used in this article include:

- [Python chat app with RAG ↗](#)
- [Azure Load Balancer with Azure API Management ↗](#)

Related content

- View [Azure API Management diagnostic data in Azure Monitor](#).
- Use [Azure Load Testing](#) to load test your chat app.

Load testing a Python chat app by using RAG with Locust

06/27/2025

This article provides the process to perform load testing on a Python chat application by using the Retrieval Augmented Generation (RAG) pattern with Locust, a popular open-source load testing tool. The primary objective of load testing is to ensure that the expected load on your chat application doesn't exceed the current Azure OpenAI Service transactions per minute (TPM) quota. By simulating user behavior under heavy load, you can identify potential bottlenecks and scalability issues in your application. This process is crucial for ensuring that your chat application remains responsive and reliable, even when faced with a high volume of user requests.

! Note

This article uses one or more [AI app templates](#) as the basis for the examples and guidance in the article. AI app templates provide you with well-maintained reference implementations that are easy to deploy. They help to ensure a high-quality starting point for your AI apps.

Prerequisites

- An Azure subscription. [Create one for free](#).
- [Dev containers](#) are available for both samples with all the dependencies that are required to complete this article. You can run the dev containers in GitHub Codespaces (in a browser) or locally by using Visual Studio Code.

GitHub Codespaces (recommended)

- You need only a [GitHub account](#).

- [Python chat app with RAG](#). If you configured your chat app to use one of the load balancing solutions, this article helps you test the load balancing. The load balancing solutions include [Azure Container Apps](#).

Open the load test sample app

The load test is in the [Python chat app solution](#) as a [Locust test](#). Return to that article, deploy the solution, and then use that dev container development environment to complete the following steps.

Run the test

1. Install the locust package for the load test:

```
Bash
```

```
python -m pip install locust
```

2. Start Locust, which uses the Locust test file [locustfile.py](#). You can find it at the root of the repository. The sample has a `ChatUser` class that simulates a user asking questions and receiving answers from the chat app.

```
Bash
```

```
locust ChatUser
```

3. Open the running Locust website, such as <http://localhost:8089>.

4. Enter the following values in the Locust website.

[\[+\] Expand table](#)

Property	Value
Number of users	20
Ramp up	1
Host	<a href="https://<YOUR-CHAT-APP-URL>.azurewebsites.net">https://<YOUR-CHAT-APP-URL>.azurewebsites.net



Locust

HOST STATUS RPS FAILURES 

Start new load test

Number of users (peak concurrency)

Ramp Up (users started/second)

Host

Advanced options 

START SWARM

5. Select **Start Swarm** to start the test.

6. Select **Charts** to watch the test progress.



Clean up resources

When you're finished with load testing, clean up the resources. The Azure resources created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges. After you delete resources specific to this article, remember to return to the other chat app tutorial and follow the clean-up steps.

Return to the chat app article to [clean up](#) those resources.

Get help

If you have trouble using this load tester, add your issue to the repository's [Issues](#)  webpage.

Configure your local environment for deploying Python web apps on Azure

Article • 02/04/2025

This article walks you through setting up your local environment to develop Python *web apps* and deploy them to Azure. Your web app can be pure Python or use one of the common Python-based web frameworks like [Django](#), [Flask](#), or [FastAPI](#).

Python web apps developed locally can be deployed to services such as [Azure App Service](#), [Azure Container Apps](#), or [Azure Static Web Apps](#). There are many options for deployment. For example, for App Service deployment, you can choose to deploy from code, a Docker container, or a Static Web App. If you deploy from code, you can deploy with Visual Studio Code, with the Azure CLI, from a local Git repository, or with GitHub actions. If you deploy in a Docker Container, you can do so from Azure Container Registry, Docker Hub, or any private registry.

Before continuing with this article, we suggest you review the [Set up your dev environment](#) for guidance on setting up your dev environment for Python and Azure. Below, we'll discuss setup and configuration specific to Python web app development.

After you get your local environment setup for Python web app development, you'll be ready to tackle these articles:

- Quickstart: Create a Python (Django or Flask) web app in Azure App Service.
- Tutorial: Deploy a Python (Django or Flask) web app with PostgreSQL in Azure
- Create and deploy a Flask web app to Azure with a system-assigned managed identity

Working with Visual Studio Code

The [Visual Studio Code](#) integrated development environment (IDE) is an easy way to develop Python web apps and work with Azure resources that web apps use.

💡 Tip

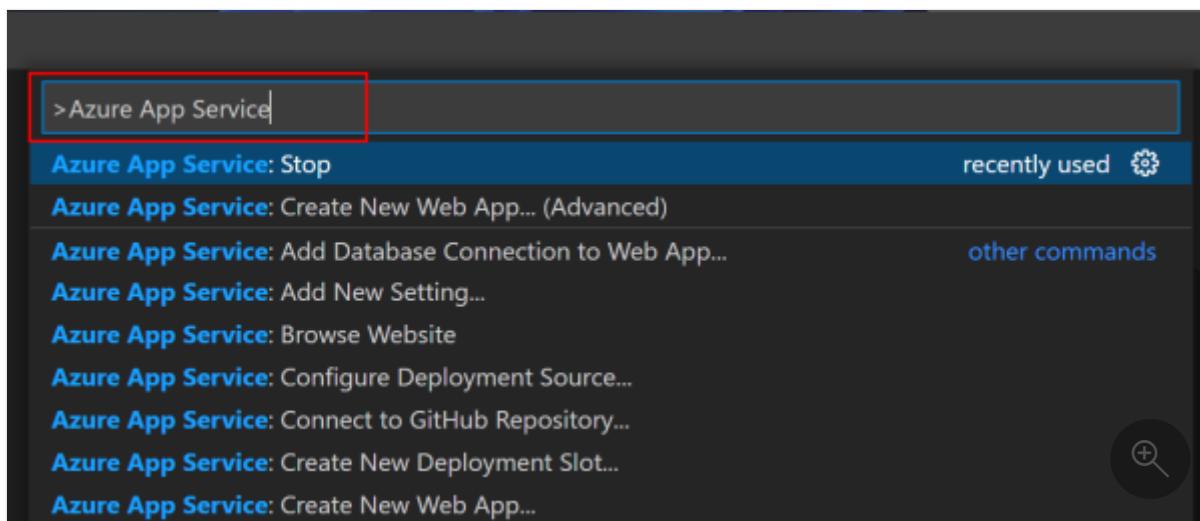
Make sure you have the [Python](#) extension installed. For an overview of working with Python in VS Code, see [Getting Started with Python in VS Code](#).

In VS Code, you work with Azure resources through [VS Code extensions](#). You can install extensions from the [Extensions](#) View or the key combination Ctrl+Shift+X. For

Python web apps, you'll likely be working with one or more of the following extensions:

- The [Azure App Service](#) extension enables you to interact with Azure App Service from within Visual Studio Code. App Service provides fully managed hosting for web applications including websites and web APIs.
- The [Azure Static Web Apps](#) extension enables you to create Azure Static Web Apps directly from VS Code. Static Web Apps is serverless and a good choice for static content hosting.
- If you plan on working with containers, then install:
 - The [Docker](#) extension to build and work with containers locally. For example, you can run a containerized Python web app on Azure App Service using [Web Apps for Containers](#).
 - The [Azure Container Apps](#) extension to create and deploy containerized apps directly from Visual Studio Code.
- There are other extensions such as the [Azure Storage](#), [Azure Databases](#), and [Azure Resources](#) extensions. You can always add these and other extensions as needed.

Extensions in Visual Studio Code are accessible as you would expect in a typical IDE interface and with rich keyword support using the [VS Code command palette](#). To access the command palette, use the key combination Ctrl+Shift+P. The command palette is a good way to see all the possible actions you can take on an Azure resource. The screenshot below shows some of the actions for App Service.



Working with Dev Containers in Visual Studio Code

Python developers often rely on virtual environments to create an isolated and self-contained environment for a specific project. Virtual environments allow developers to manage dependencies, packages, and Python versions separately for each project, avoiding conflicts between different projects that might require different package versions.

While there are popular options available in Python for managing environments like `virtualenv` or `venv`, the [Visual Studio Code Dev Container](#) extension (based on the [open Dev Container specification](#)) lets you use a [Docker container](#) as a full-featured containerized environment. It enables developers to define a consistent and easily reproducible toolchain with all the necessary tools, dependencies, and extensions pre-configured. This means if you have system requirements, shell configurations, or use other languages entirely, you can use a Dev Container to explicitly configure all of those parts of your project that might live outside of a basic Python environment.

For example, a developer can configure a single Dev Container to include everything needed to work on a project, including a PostgreSQL database server along with the project database and sample data, a Redis server, Nginx, front-end code, client libraries like React, and so on. In addition, the container would contain the project code, the Python runtime, and all the Python project dependencies with the correct versions. Finally, the container can specify Visual Studio Code extensions to be installed so the entire team has the same tooling available. So when a new developer joins the team, the whole environment, including tooling, dependencies, and data, is ready to be cloned to their local machine, and they can begin working immediately.

See [Developing inside a Container](#).

Working with Visual Studio 2022

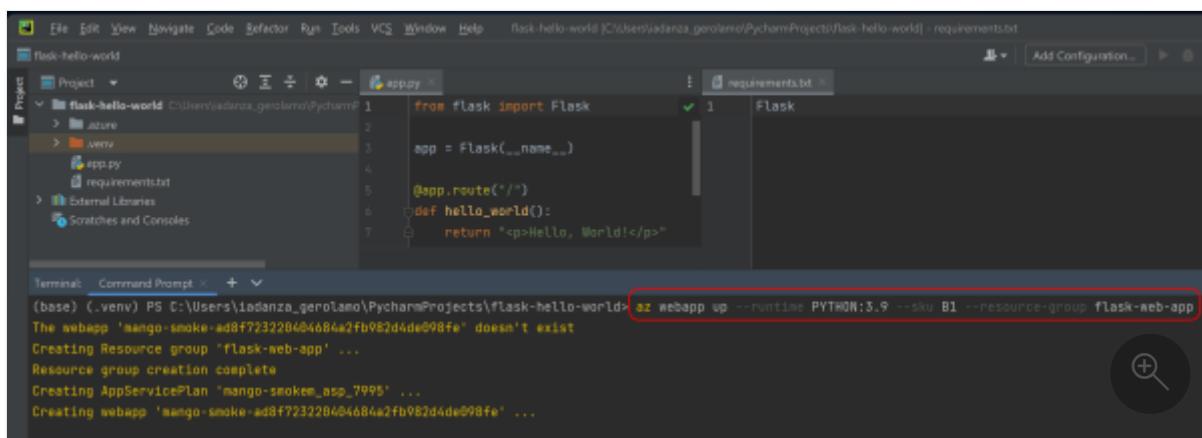
[Visual Studio 2022](#) is a full-featured integrated development environment (IDE) with support for Python application development and many built-in tools and extensions to access and deploy to Azure resources. While most documentation for building Python web apps on Azure focuses on using Visual Studio Code, Visual Studio 2022 is a great option if you already have it installed, you're comfortable with using it, and are using it for .NET or C++ projects.

- In general, see [Visual Studio | Python documentation](#) for all documentation related to using Python on Visual Studio 2022.
- For setup steps, see [Install Python support in Visual Studio](#) which walks you through the steps of installing the Python workload into Visual Studio 2022.

- For general workflow of using Python for web development, see [Quickstart: Create your first Python web app using Visual Studio](#). This article is useful for understanding how to build a Python web application from scratch (but does not include deployment to Azure).
- For using Visual Studio 2022 to manage Azure resources and deploy to Azure, see [Azure Development with Visual Studio](#). While much of the documentation here specifically mentions .NET, the tooling for managing Azure resources and deploying to Azure works the same regardless of the programming language.
- When there's no built-in tool available in Visual Studio 2022 for a given Azure management or deployment task, you can always use [Azure CLI commands](#).

Working with other IDEs

If you're working in another IDE that doesn't have explicit support for Azure, then you can use the Azure CLI to manage Azure resources. In the screenshot below, a simple Flask web app is open in the [PyCharm](#) IDE. The web app can be deployed to an Azure App Service using the `az webapp up` command. In the screenshot, the CLI command runs within the PyCharm embedded terminal emulator. If your IDE doesn't have an embedded emulator, you can use any terminal and the same command. The Azure CLI must be installed on your computer and be accessible in either case.



The screenshot shows the PyCharm interface with a project named "flask-hello-world". The code editor displays a file named "app.py" containing a simple Flask application. Below the code editor is a terminal window showing the execution of the Azure CLI command `az webapp up`. The command is highlighted with a red box, indicating it is the focus of the discussion. The terminal output shows the creation of a resource group and an App Service Plan, followed by the deployment of the web app.

```
(base) (.venv) PS C:\Users\iadanza_gerolamo\PycharmProjects\flask-hello-world> az webapp up --runtime PYTHON:3.9 --sku B1 --resource-group flask-web-app
The webapp 'mango-smoke-ad8f723220404684a2fb982d4de098fe' doesn't exist
Creating Resource group 'flask-web-app' ...
Resource group creation complete
Creating AppServicePlan 'mango-smokem_asp_7995' ...
Creating webapp 'mango-smoke-ad8f723220404684a2fb982d4de098fe' ...
```

Azure CLI commands

When working locally with web apps using the [Azure CLI](#) commands, you'll typically work with the following commands:

[Expand table](#)

Command	Description
az webapp	Manages web apps. Includes the subcommands <code>create</code> and <code>up</code> to create a web app or to create and deploy from a local workspace, respectively.
az container app	Manages Azure Container Apps.
az staticwebapp	Manages Azure Static Web Apps.
az group	Manages resource groups and template deployments. Use the subcommand <code>create</code> to make a resource group to put your Azure resources in.
az appservice	Manages App Service plans.
az config	Manages Azure CLI configuration. To save keystrokes, you can define a default location or resource group that other commands use automatically.

Here's an example Azure CLI command to create a web app and associated resources, and deploy it to Azure in one command using [az webapp up](#). Run the command in the root directory of your web app.

```
bash
Azure CLI
az webapp up \
--runtime PYTHON:3.9 \
--sku B1 \
--logs
```

For more about this example, see [Quickstart: Deploy a Python \(Django or Flask\) web app to Azure App Service](#).

Keep in mind that for some of your Azure workflow you can also use the Azure CLI from an [Azure Cloud Shell](#). Azure Cloud Shell is an interactive, authenticated, browser-accessible shell for managing Azure resources.

Azure SDK key packages

In your Python web apps, you can refer programmatically to Azure services using the [Azure SDK for Python](#). This SDK is discussed extensively in the section [Use the Azure libraries \(SDK\) for Python](#). In this section, we'll briefly mention some key packages of the SDK that you'll use in web development. And, we'll show an example around the best practices for authenticating your code with Azure resources.

Below are some of the packages commonly used in web app development. You can install packages in your virtual environment directly with `pip`. Or put the Python package index (Pypi) name in your `requirements.txt` file.

[+] Expand table

SDK docs	Install	Python package index
Azure Identity	<code>pip install azure-identity</code>	azure-identity
Azure Storage Blobs	<code>pip install azure-storage-blob</code>	azure-storage-blob
Azure Cosmos DB	<code>pip install azure-cosmos</code>	azure-cosmos
Azure Key Vault Secrets	<code>pip install azure-keyvault-secrets</code>	azure-keyvault-secrets

The [azure-identity](#) package allows your web app to authenticate with Microsoft Entra ID. For authentication in your web app code, it's recommended that you use the `DefaultAzureCredential` in the `azure-identity` package. Here's an example of how to access Azure Storage. The pattern is similar for other Azure resources.

Python

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

azure_credential = DefaultAzureCredential()
blob_service_client = BlobServiceClient(
    account_url=account_url,
    credential=azure_credential)
```

The `DefaultAzureCredential` will look in predefined locations for account information, for example, in environment variables or from the Azure CLI sign-in. For in-depth information on the `DefaultAzureCredential` logic, see [Authenticate Python apps to Azure services by using the Azure SDK for Python](#).

Python-based web frameworks

In Python web app development, you often work with Python-based web frameworks. These frameworks provide functionality, such as page templates, session management, database access, and easy access to HTTP request and response objects. Frameworks enable you to avoid the need for you to have to reinvent the wheel for common functionality.

Three common Python web frameworks are [Django](#), [Flask](#), or [FastAPI](#). These and other web frameworks can be used with Azure.

Below is an example of how you might get started quickly with these frameworks locally. Running these commands, you'll end up with an application, albeit a simple one that could be deployed to Azure. Run these commands inside a [virtual environment](#).

Step 1: Download the frameworks with [pip](#).

Django

```
pip install Django
```

Step 2: Create a hello world app.

Django

Create a sample project using the [django-admin startproject](#) command. The project includes a *manage.py* file that is the entry point for running the app.

```
django-admin startproject hello_world
```

Step 3: Run the code locally.

Django

Django uses WSGI to run the app.

```
python hello_world\manage.py runserver
```

Step 4: Browse the hello world app.

Django

<http://127.0.0.1:8000/>

At this point, add a *requirements.txt* file and then you can deploy the web app to Azure or containerize it with Docker and then deploy it.

Next steps

- Quickstart: Create a Python (Django or Flask) web app in Azure App Service.
- Tutorial: Deploy a Python (Django or Flask) web app with PostgreSQL in Azure
- Create and deploy a Flask web app to Azure with a system-assigned managed identity

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Overview of the Python web azd templates

06/23/2025

Python web Azure Developer CLI (`azd`) templates are the fastest and easiest way to build, configure, and deploy Python web applications to Azure. This article provides contextual background information to help you understand the components involved and how the templates simplify deployment.

The best way to begin is to [follow the quickstart](#) to create your first Python web app and deploy it to Azure in minutes with `azd` templates. If you prefer not to set up a local development environment, you can follow the [quickstart by using GitHub Codespaces](#) for a fully cloud-based experience with all tools preconfigured.

What are the Python web azd templates?

The `azd` templates are designed for experienced Python web developers who want to deploy scalable, cloud-ready applications on Azure with minimal setup time.

These templates offer the easiest possible starting point for building and deploying Python web applications by:

- Quickly setting up a complete local development and hosting environment.
- Automating the creation of a matching Azure deployment environment.
- Using a simple and memorable CLI workflow.

Once your environments are set up, the templates provide the fastest way to start building your Python web app. You can:

- Modify the provided code files to match your app's requirements.
- Deploy updates with minimal effort using `azd` commands.
- Extend the template to fit your architecture.

These templates reflect proven design patterns and best practices, enabling you to:

- Build with confidence on a solid architectural foundation.
- Follow guidance developed by industry experts with deep experience in Python and Azure.
- Ensure maintainability, scalability, and security from the start.

What tasks can I do with the templates?

When you run a Python web `azd` template, you quickly complete several tasks:

- **Create starter application.** You build a website for a fictitious company named Relecloud. This starter project includes:
 - Well-organized, production-ready code
 - Best practices for Python web frameworks (such as Flask, Django).
 - Proper use of dependencies, configuration, and structure.

The template is designed to be a starting point—you can freely customize the logic and expand or remove Azure resources to fit your project.

- **Provision Azure resources.** Using [Bicep](#), a modern infrastructure-as-code (IaC) language, the template provisions all necessary Azure resources for:
 - Hosting your web app (such as App Service, Container Apps)
 - Connecting to databases (such as PostgreSQL, Cosmos DB)

The Bicep files are fully editable—you can add or customize Azure services as your app evolves. Similar to the previous task, you can [modify the Bicep templates](#) to add more Azure services, as needed.

- **Deploy starter app to provisioned Azure resources.** Once resources are provisioned, your application is automatically deployed to the Azure environment. You can now:
 - See your app running in the cloud within minutes.
 - Test its behavior.
 - Decide what functionality or configuration to update next.
- **(Optional) Set up GitHub repository and CI/CD pipeline.** You can optionally initialize a GitHub repository with a GitHub Actions [continuous integration/continuous delivery \(CI/CD\) pipeline](#) to:
 - Automate deployments on code changes.
 - Collaborate with team members.
 - Push updates to Azure by merging into the main branch.

This integration helps you adopt DevOps best practices from the start.

Where can I access the templates?

Many `azd` templates are available on the [Awesome Azure Developer CLI Templates gallery](#). These templates provide ready-to-use Python web app projects with feature parity across popular combinations of Azure services and Python web frameworks.

Each template includes:

- A sample application with clean, maintainable code.
- Pre-configured infrastructure-as-code using Bicep.

- Seamless deployment workflows using the Azure Developer CLI.
- Optional CI/CD integration via GitHub Actions

The following tables list the Python web `azd` template monikers that are available for use with the `azd init` command. The tables identify the technologies implemented in each template and provide a link to the corresponding GitHub repository, where you can contribute changes.

Django

The following `azd` templates are available for the [Django web framework](#).

[Expand table](#)

Template	Database	Hosting platform	GitHub repository
<code>azure-django-postgres-flexible-aca</code>	Azure Database for PostgreSQL Flexible Server	Azure Container Apps	https://github.com/Azure-Samples/azure-django-postgres-flexible-aca
<code>azure-django-postgres-flexible-appservice</code>	Azure Database for PostgreSQL Flexible Server	Azure App Service	https://github.com/Azure-Samples/azure-django-postgres-flexible-appservice
<code>azure-django-cosmos-postgres-aca</code>	Azure Cosmos DB for Azure Database for PostgreSQL	Azure Container Apps	https://github.com/Azure-Samples/azure-django-cosmos-postgres-aca
<code>azure-django-cosmos-postgres-appservice</code>	Azure Cosmos DB for Azure Database for PostgreSQL	Azure App Service	https://github.com/Azure-Samples/azure-django-cosmos-postgres-appservice
<code>azure-django-postgres-addon-aca</code>	Azure Container Apps with Azure Database for PostgreSQL	Azure Container Apps	https://github.com/Azure-Samples/azure-django-postgres-addon-aca

How should I use the templates?

Each `azd` template comprises a GitHub repository that contains the application code (Python code that utilizes a popular web framework) and the infrastructure-as-code (namely, [Bicep](#)) files to create the Azure resources. The template also contains the configuration required to set up a GitHub repository with a CI/CD pipeline.

Key components of each template include:

- **Application Code:** Written in Python and built using a popular web framework (such as Flask, Django, FastAPI). The sample app demonstrates best practices in routing, data access, and configuration.
- **Infrastructure-as-Code (IaC):** Provided via Bicep files to define and provision the required Azure resources, such as:
 - App Service or Container Apps
 - Azure Databases (such as PostgreSQL, Cosmos DB)
 - Azure AI services, Storage, and more
- **CI/CD Configuration (Optional):** Includes files to set up a GitHub repository with a GitHub Actions CI/CD pipeline, enabling:
 - Automatic deployment to Azure on every push or pull request to the main branch.
 - Seamless integration into your DevOps workflow

These templates are fully customizable, giving you a strong foundation to build on and adapt to your project's specific needs.

To perform the tasks defined by an `azd` web template, you use various Python `azd` commands. For detailed descriptions of these commands, see [Quickstart: Deploy an Azure Developer CLI template](#). The quickstart walks you through the steps to use a specific `azd` template. You only need to run five essential command-line instructions to the production-hosting environment and the local-development environment.

The following table summarizes the five essential commands:

 [Expand table](#)

Command	Task description
<code>azd init --template <template name></code>	Create a new project from a template and create a copy of the application code on your local computer. The command prompts you to provide an environment name (like "myapp") that's used as a prefix in the naming of the deployed resources.
<code>azd auth login</code>	Sign in to Azure. The command opens a browser window where you can sign in to Azure. After you sign in, the browser window closes and the command completes. The <code>azd auth login</code> command is required only the first time you use the Azure Developer CLI (<code>azd</code>) per session.
<code>azd up</code>	Provision the cloud resources and deploy the app to those resources.
<code>azd deploy</code>	Deploy changes to the application source code to resources already provisioned by the <code>azd up</code> command.
<code>azd down</code>	Delete the Azure resources and the CI/CD pipeline, if it was used.

💡 Tip

When you work with the `azd` commands, watch for prompts to enter more information. After you execute the `azd up` command, you might be prompted to select a subscription, if you have more than one. You might also be prompted to specify your region. You can change the answers to prompts by editing the environment variables stored in the `./azure/` folder of the template.

After completing the essential tasks provided by the `azd` template, you have a personal copy of the original template where you can modify any file, as needed.

- **Application Code:** Customize the Python project code to implement your own design, routes, and business logic.
- **Infrastructure-as-Code (Bicep):** Update the Bicep files to provision additional Azure services, change configurations, or remove unneeded resources.

This flexible starting point allows you to build on top of a well-structured foundation while tailoring the app to your real-world use case.

You can also [modify the infrastructure-as-code configuration](#) if you need to change the Azure resources. For more information, see the [What can I edit or delete](#) section later in this article.

Optional template tasks

In addition to the five essential commands, there are optional tasks you can complete with the `azd` templates.

Reprovision and modify Azure resources

After you provision Azure resources with an `azd` template, you can modify and reprovision a resource.

- To modify a provisioned resource, you [edit the appropriate Bicep files](#) in the template.
- To initiate the reprovisioning task, use the `azd provision` command.

Set up CI/CD pipeline

The Azure Developer CLI (`azd`) provides an easy way to set up a CI/CD pipeline for your new Python web app. When you merge commits or pull requests into your main branch, the pipeline automatically builds and publishes the changes to your Azure resources.

- To set up the CI/CD pipeline, you designate the GitHub repository and desired settings to enable the pipeline.
- To create the pipeline, use the `azd pipeline config` command.

After you configure the pipeline, each time code changes are merged to the *main* branch of the repository, the pipeline deploys the changes to your provisioned Azure services.

Alternatives to the templates

If you prefer to not use the Python web `azd` templates, there are alternate methods for deploying Python web apps to Azure and provisioning Azure resources.

You can create many resources and complete the deployment steps by using several tools:

- [Azure portal](#)
- The [Azure CLI](#)
- Visual Studio Code with the [Azure Tools extension](#)

You can also follow an end-to-end tutorial that features Python web development frameworks:

- [Deploy a Flask or FastAPI web app on Azure App Service](#)
- [Containerized Python web app on Azure with MongoDB](#)

Frequently asked questions

The following sections summarize answers to frequently asked questions about working with the Python web `azd` templates.

Do I have to use Dev Containers?

No. The Python web `azd` templates use [Visual Studio Code Dev Containers](#) by default. Dev Containers provide many benefits, but they require some prerequisite knowledge and software. If you prefer to not use Dev Containers, and instead use your local development environment, see the *README.md* file in the root directory of the sample app for environment setup instructions.

What can I edit or delete?

The contents of each Python web `azd` template can vary depending on the type of project and the underlying technology stack employed. The templates identified in this article follow a common folder and file convention, as described in the following table.

Folder/file(s)	Purpose	Description
/	Root directory	The root folder for each template contains many different kinds of files and folders for different purposes.
.azure	azd configuration files	The <code>.azure</code> folder is created after you run the <code>azd init</code> command. The folder stores configuration files for the environment variables used by the <code>azd</code> commands. You can change the values of the environment variables to customize the app and the Azure resources. For more information, see Environment-specific .env file .
.devcontainer	Dev Container configuration files	Dev Containers allow you to create a container-based development environment complete with all of the resources you need for software development inside of Visual Studio Code. The <code>.devcontainer</code> folder is created after Visual Studio Code generates a Dev Container configuration file in response to a template command.
.github	GitHub Actions configuration files	This folder contains configuration settings for the optional GitHub Actions CI/CD pipeline, linting, and tests. If you don't want to set up the GitHub Actions pipeline by using <code>azd pipeline config</code> command, you can modify or deleted the <code>azure-dev.yaml</code> file.
/infra	Bicep files	The <code>infra</code> folder holds the Bicep configuration files. Bicep allows you to declare the Azure resources you want deployed to your environment. You should only modify the <code>main.bicep</code> and <code>web.bicep</code> files. For more information, see Quickstart: Scaling services deployed with the azd Python web templates by using Bicep .
/src	Starter project code files	The <code>src</code> folder contains various code files required to prepare the starter project. Examples of the files include templates required by the web framework, static files, Python (.py) files for the code logic and data models, a <code>requirements.txt</code> file, and more. The specific files depend on the web framework, the data access framework, and so on. You can modify these files to suit your project requirements.
.cruft.json	Template generation file	The <code>.cruft</code> JSON file is used internally to generate the Python web <code>azd</code> templates. You can safely delete this file, as needed.
.gitattributes	File with attribute settings for git	This file provides git with important configuration settings for handling files and folders. You can modify this file, as needed.
.gitignore	File with ignored items for git	The <code>.gitignore</code> file informs git about the files and folders to exclude (ignore) when writing to the GitHub repository for the template. You can modify this file, as needed.
/azure.yaml	azd up configuration	This configuration file contains the configuration settings for the <code>azd up</code> command. It specifies the services and project folders to

Folder/file(s)	Purpose	Description
	file	deploy. Important: This file must not be deleted.
/*.md	Markdown format files	A template can include various Markdown (.md) format files for different purposes. You can safely delete Markdown files.
/docker-compose.yml	Docker compose settings	This YML file creates the container package for the Python web application before the app deploys to Azure.
/pyproject.toml	Python build settings file	The TOML file contains the build system requirements of Python projects. You can modify this file to identify your tool preferences, such as a specific linter or unit testing framework.
/requirements-dev.in	pip requirements file	This file is used to create a development environment version of the requirements by using the <code>pip install -r</code> command. You can modify this file to include other packages, as needed.

💡 Tip

As you modify template files for your program, be sure to practice good version control. This approach can help you restore your repository to a previous working version, if new changes cause program issues.

How can I handle template errors?

If you receive an error when you use an `azd` template, review the options described in the [Troubleshoot Azure Developer CLI](#) article. You can also report issues on the GitHub repository associated with the `azd` template.

Related content

- [Create and deploy Python web apps to Azure with azd templates](#)
- [Create and deploy Python web apps from GitHub Codespaces to Azure with azd templates](#)

Quickstart: Create and deploy a Python web app to Azure using an azd template

Article • 12/16/2024

This quickstart guides you through the easiest and fastest way to create and deploy a Python web and database solution to Azure. By following the instructions in this quickstart, you will:

- Choose an `azd` template based on the Python web framework, Azure database platform, and Azure web hosting platform you want to build on.
- Use CLI commands to run an `azd` template to create a sample web app and database, and create and configure the necessary Azure resources, then deploy the sample web app to Azure.
- Edit the web app on your local computer and use an `azd` command to redeploy.
- Use an `azd` command to clean up Azure resources.

It should take less than 15 minutes to complete this tutorial. Upon completion, you can start modifying the new project with your custom code.

To learn more about these `azd` templates for Python web app development:

- [What are these templates?](#)
- [How do the templates work?](#)
- [Why would I want to do this?](#)
- [What are my other options?](#)

Prerequisites

An Azure subscription - [Create one for free ↗](#)

You must have the following installed on your local computer:

- [Azure Developer CLI](#)
- [Docker Desktop ↗](#)
- [Visual Studio Code ↗](#)
- [Dev Container Extension ↗](#)

Choose a template

Choose an `azd` template based on the Python web framework, Azure web hosting platform, and Azure database platform you want to build on.

1. Select a template name (first column) from the following list of templates in the following tables. You'll use the template name during the `azd init` step in the next section.

Django

[Expand table](#)

Template	Web Framework	Database	Hosting Platform	GitHub Repo
azure-django-postgres-flexible-aca	Django	PostgreSQL Flexible Server	Azure Container Apps	repo ↗
azure-django-postgres-flexible-appservice	Django	PostgreSQL Flexible Server	Azure App Service	repo ↗
azure-django-cosmos-postgres-aca	Django	Cosmos DB (PostgreSQL Adapter)	Azure Container Apps	repo ↗
azure-django-cosmos-postgres-appservice	Django	Cosmos DB (PostgreSQL Adapter)	Azure App Service	repo ↗
azure-django-postgres-addon-aca	Django	Azure Container Apps PostgreSQL Add-on	Azure Container Apps	repo ↗

The GitHub repository (last column) is only provided for reference purposes. You should only clone the repository directly if you want to contribute changes to the template. Otherwise, follow the instructions in this quickstart to use the `azd` CLI to interact with the template in a normal workflow.

Run the template

Running an `azd` template is the same across languages and frameworks. And, the same basic steps apply to all templates. The steps are:

1. At a terminal, navigate to a folder on your local computer where you typically store your local git repositories, then create a new folder named `azdtest`. Then, change into that directory using the `cd` command.

```
shell  
  
mkdir azdtest  
cd azdtest
```

Don't use Visual Studio Code's Terminal for this quickstart.

2. To set up the local development environment, enter the following commands in your terminal and answer any prompts:

```
shell  
  
azd init --template <template name>
```

Substitute `<template name>` with one of the templates from the [tables](#) you selected in a previous step, such as `azure-django-postgres-aca` for example.

When prompted for an environment name, use `azdtest` or any other name. The environment name is used when naming Azure resource groups and resources. For best results, use a short name, lower case letters, no special characters.

3. To authenticate `azd` to your Azure account, enter the following commands in your terminal and follow the prompt:

```
shell  
  
azd auth login
```

Follow the instructions when prompted to "Pick an account" or log into your Azure account. Once you have successfully authenticated, the following message is displayed in a web page: "Authentication complete. You can return to the application. Feel free to close this browser tab."

When you close the tab, the shell displays the message:

```
Output
```

Logged in to Azure.

4. Ensure that Docker Desktop is open and running in the background before attempting the next step.
5. To create the necessary Azure resources, enter the following commands in your terminal and answer any prompts:

shell

```
azd up
```

ⓘ Important

Once `azd up` completes successfully, the sample web app will be available on the public internet and your Azure Subscription will begin accruing charges for all resources that are created. The creators of the `azd` templates intentionally chose inexpensive tiers but not necessarily *free* tiers since free tiers often have restricted availability.

Follow the instructions when prompted to choose Azure Subscription to use for payment, then select an Azure location to use. Choose a region that is close to you geographically.

Executing `azd up` could take several minutes since it's provisioning and deploying multiple Azure services. As progress is displayed, watch for errors. If you see errors, try the following to fix the problem:

- Delete the `azd-quickstart` folder and the quickstart instructions from the beginning.
- When prompted, choose a simpler name for your environment. Only use lower-case letters and dashes. No numbers, upper-case letters, or special characters.
- Choose a different location.

If you still have problems, see the [Troubleshooting](#) section at the bottom of this document.

ⓘ Important

Once you have finished working with the sample web app, use `azd down` to remove all of the services that were created by `azd up`.

6. When `azd up` completes successfully, the following output is displayed:

```
Deploying services (azd deploy)
|==      |      Deploying service web (Fetching endpoints for container a
(✓) Done: Deploying service web
- Endpoint: https://azdtest-cpz2yvly5xa-ca.delightfulflower-54a525cc.eastus2.azurecontainerapps.io/
SUCCESS: Your application was provisioned and deployed to Azure in 10 minutes 45 seconds.
You can view the resources created under the resource group azdtest-rg in Azure Portal:
https://portal.azure.com/#@/resource/subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-e
eeeeee4e4e4e/resourceGroups/azdtest-rg/overview
c:\source\azdtest>
```

Copy the first URL after the word - `Endpoint:` and paste it into the location bar of a web browser to see the sample web app project running live in Azure.

7. Open a new tab in your web browser, copy the second URL from the previous step and paste it into the location bar. The Azure portal displays all of the services in your new resource group that have been deployed to host the sample web app project.

Edit and redeploy

The next step is to make a small change to the web app and then redeploy.

1. Open Visual Studio Code and open the `azdtest` folder created earlier.
2. This template is configured to optionally use Dev Containers. When you see the Dev Container notification appear in Visual Studio Code, select the "Reopen in Container" button.
3. Use Visual Studio Code's Explorer view to navigate to `src/templates` folder, and open the `index.html` file. Locate the following line of code:

HTML

```
<h1 id="page-title">Welcome to ReleCloud</h1>
```

Change the text inside of the H1:

HTML

```
<h1 id="page-title">Welcome to ReleCloud - UPDATED</h1>
```

Save your changes.

4. To redeploy the app with your change, in your terminal run the following command:

Shell

```
azd deploy
```

Since you're using Dev Containers and are connected remotely into the container's shell, don't use Visual Studio Code's Terminal pane to run `azd` commands.

5. Once the command completes, refresh your web browser to see the update. Depending on the web hosting platform being used, it could take several minutes before your changes are visible.

You're now ready to edit and delete files in the template. For more information, see [What can I edit or delete in the template?](#)

Clean up resources

1. Clean up the resources created by the template by running the `azd down` command.

Shell

```
azd down
```

The `azd down` command deletes the Azure resources and the GitHub Actions workflow. When prompted, agree to deleting all resources associated with the resource group.

You may also delete the `azdtest` folder, or use it as the basis for your own application by modifying the files of the project.

Troubleshooting

If you see errors during `azd up`, try the following steps:

- Run `azd down` to remove any resources that may have been created. Alternatively, you can delete the resource group that was created in the Azure portal.
- Delete the `azdtest` folder on your local computer.
- In the Azure portal, search for Key Vaults. Select to *Manage deleted vaults*, choose your subscription, select all key vaults that contain the name `azdtest` or whatever you named your environment, and select *Purge*.
- Retry the steps in this quickstart again. This time when prompted, choose a simpler name for your environment. Try a short name, lower-case letters, no numbers, no upper-case letters, no special characters.
- When retrying the quickstart steps, choose a different location.

See the [FAQ](#) for a more comprehensive list of possible issues and solutions.

Related Content

- [Learn more about the Python web azd templates](#)
- [Learn more about the azd commands.](#)
- Learn what each of the folders and files in the project do and [what you can edit or delete?](#)
- [Learn more about Dev Containers ↗](#).
- [Update the Bicep templates to add or remove Azure services](#). Don't know Bicep? Try this [Learning Path: Fundamentals of Bicep](#)
- [Use azd to set up a GitHub Actions CI/CD pipeline to redeploy on merge to main branch](#)
- Set up monitoring so that you can [Monitor your app using the Azure Developer CLI](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Create and deploy a Python web app from GitHub Codespaces to Azure using an Azure Developer CLI template

Article • 12/16/2024

This quickstart guides you through the easiest and fastest way to create and deploy a Python web and database solution to Azure. By following the instructions in this quickstart, you will:

- Choose an [Azure Developer CLI](#) (`azd`) template based on the Python web framework, Azure database platform, and Azure web hosting platform you want to build on.
- Create a new GitHub Codespace containing code generated from the `azd` template you selected.
- Use GitHub Codespaces and the online Visual Studio Code's bash terminal. The terminal allows you to use Azure Developer CLI commands to run an `azd` template to create a sample web app and database, and create and configure the necessary Azure resources, then deploy the sample web app to Azure.
- Edit the web app in a GitHub Codespace and use an `azd` command to redeploy.
- Use an `azd` command to clean up Azure resources.
- Close and reopen your GitHub Codespace.
- Publish your new code to a GitHub repository.

It should take less than 25 minutes to complete this tutorial. Upon completion, you can start modifying the new project with your custom code.

To learn more about these `azd` templates for Python web app development:

- [What are these templates?](#)
- [How do the templates work?](#)
- [Why would I want to do this?](#)
- [What are my other options?](#)

Prerequisites

- An Azure subscription - [Create one for free ↗](#)
- A GitHub Account - [Create one for free ↗](#)

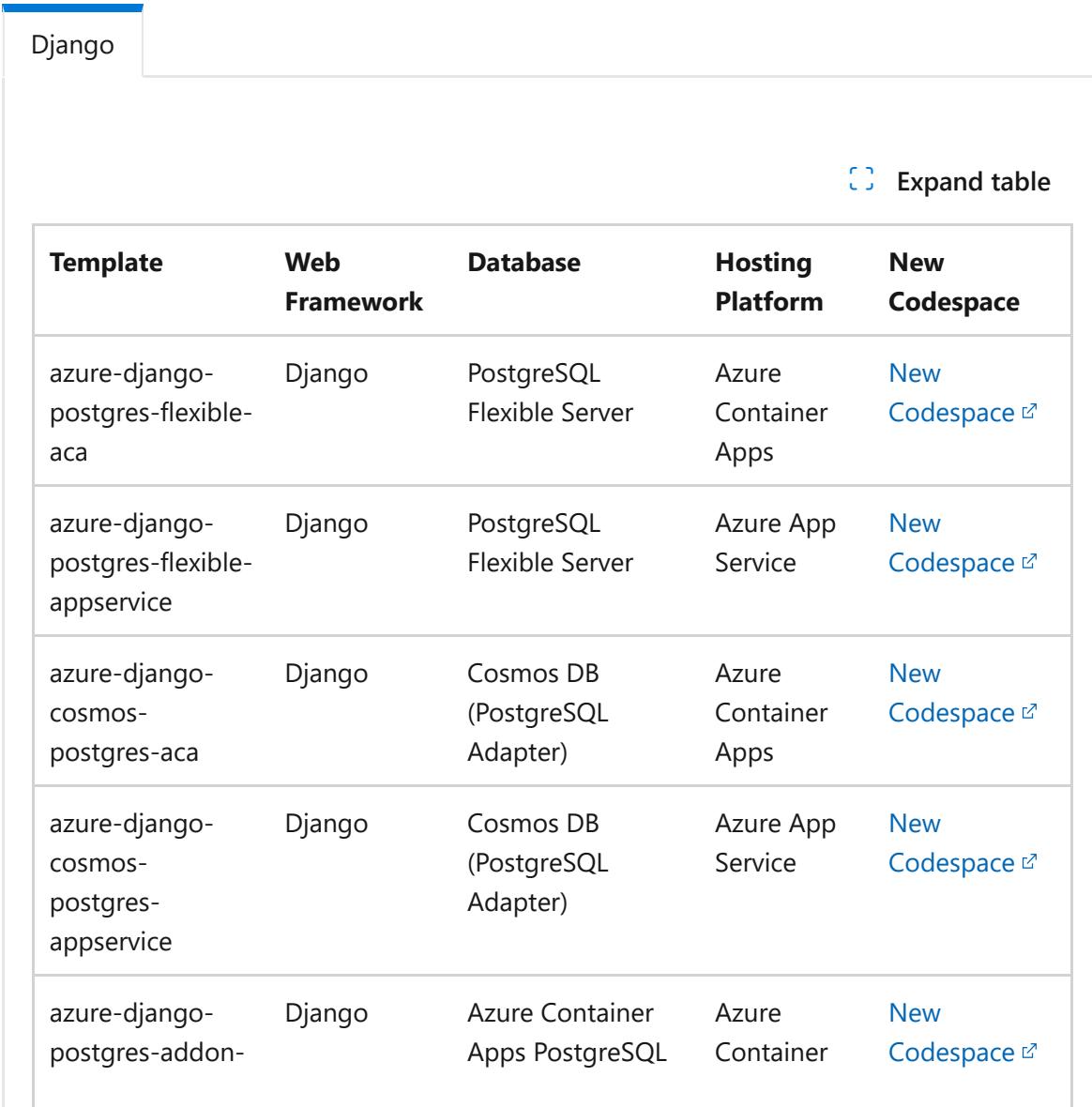
ⓘ Important

Both GitHub Codespaces and Azure are paid subscription based services. After some free allotments, you may be charged for using these services. Following this quickstart could affect these allotments or billing. When possible, the azd templates were built using the least expensive tier of options, but some may not be free. Use the [Azure Pricing calculator](#) to better understand the costs. For more information, see [GitHub Codespaces pricing](#) for more details.

Choose a template and create a codespace

Choose an azd template based on the Python web framework, Azure web hosting platform, and Azure database platform you want to build on.

1. From the following list of templates, choose one that uses the technologies that you want to use in your new web application.



Django

Expand table

Template	Web Framework	Database	Hosting Platform	New Codespace
azure-django-postgres-flexible-aca	Django	PostgreSQL Flexible Server	Azure Container Apps	New Codespace
azure-django-postgres-flexible-appservice	Django	PostgreSQL Flexible Server	Azure App Service	New Codespace
azure-django-cosmos-postgres-aca	Django	Cosmos DB (PostgreSQL Adapter)	Azure Container Apps	New Codespace
azure-django-cosmos-postgres-appservice	Django	Cosmos DB (PostgreSQL Adapter)	Azure App Service	New Codespace
azure-django-postgres-addon-	Django	Azure Container Apps PostgreSQL	Azure Container	New Codespace

Template	Web Framework	Database	Hosting Platform	New Codespace
aca		Add-on	Apps	

2. For your convenience, the last column of each table contains a link that creates a new Codespace and initializes the `azd` template in your GitHub account. Right-click and select "Open in new tab" on the "New Codespace" link next to the template name you selected to initiate the setup process.

During this process, you may be prompted to log into your GitHub account, and you're asked to confirm that you want to create the Codespace. Select the "Create Codespace" button to see the "Setting up your codespace" page.

3. After a few minutes, a web-based version of Visual Studio Code is loaded in a new browser tab with the Python web template loaded as a workspace in the Explorer view.

Authenticate to Azure and deploy the azd template

Now that you have a GitHub Codespace containing the newly generated code, you use the `azd` utility from within the Codespace to publish the code to Azure.

- In the web-based Visual Studio Code, the terminal should be open by default. If it isn't, use the tilde `~` key to open the terminal. Furthermore, by default, the terminal should be a bash terminal. If it isn't, change to bash in the upper right hand area of the terminal window.
- In the bash terminal, enter the following command:

Bash

```
azd auth login
```

`azd auth login` begins the process of authenticating your Codespace to your Azure account.

Output

```
Start by copying the next code: XXXXXXXXX  
Then press enter and continue to log in from your browser...
```

```
Waiting for you to complete authentication in the browser...
```

3. Follow the instructions, which include:

- Copying a generated code
- Selecting enter to open a new browser tab and pasting the code into the text box
- Choosing your Azure account from a list
- Confirming that you're trying to sign in to Microsoft Azure CLI

4. When successful, the following message is displayed back in the Codespaces tab at the terminal:

Output

```
Device code authentication completed.  
Logged in to Azure.
```

5. Deploy your new application to Azure by entering the following command:

Bash

```
azd up
```

During this process, you're asked to:

- Enter a new environment name
- Select an Azure Subscription to use [Use arrows to move, type to filter]
- Select an Azure location to use: [Use arrows to move, type to filter]

Once you answer those questions, the output from `azd` indicates the deployment is progressing.

 **Important**

Once `azd up` completes successfully, the sample web app will be available on the public internet and your Azure Subscription will begin accruing charges for all resources that are created. The creators of the `azd` templates intentionally chose inexpensive tiers but not necessarily *free* tiers since free tiers often have restricted availability. Once you have finished working with

the sample web app, use `azd down` to remove all of the services that were created by `azd up`.

Follow the instructions when prompted to choose Azure Subscription to use for payment, then select an Azure location to use. Choose a region that is close to you geographically.

Executing `azd up` could take several minutes since it's provisioning and deploying multiple Azure services. As progress is displayed, watch for errors. If you see errors, see the [Troubleshooting](#) section at the bottom of this document.

6. When `azd up` completes successfully, similar output is displayed:

```
Output

(✓) Done: Deploying service web
- Endpoint: https://xxxxx-xxxxxxxxxxxx-ca.example-
xxxxxxxx.westus.azurecontainerapps.io/

SUCCESS: Your application was provisioned and deployed to Azure in 11
minutes 44 seconds.
You can view the resources created under the resource group xxxx-rg in
Azure Portal:
https://portal.azure.com/#@/resource/subscriptions/xxxxxxxx-xxxx-xxxx-
xxxx-xxxxxxxxxxxx/resourceGroups/xxxx-rg/overview
```

If you see a default screen or error screen, the app may be starting up. Please wait 5-10 minutes to see if the issue resolves itself before troubleshooting.

Ctrl + click the first URL after the word `- Endpoint:` to see the sample web app project running live in Azure.

7. Ctrl + click the second URL from the previous step to view the provisioned resources in the Azure portal.

Edit and redeploy

The next step is to make a small change to the web app and then redeploy.

1. Return to the browser tab containing Visual Studio Code, and use Visual Studio Code's Explorer view to navigate to `src/templates` folder, and open the `index.html` file. Locate the following line of code:

HTML

```
<h1 id="page-title">Welcome to ReleCloud</h1>
```

Change the text inside of the H1:

HTML

```
<h1 id="page-title">Welcome to ReleCloud - UPDATED</h1>
```

Your code is saved as you type.

2. To redeploy the app with your change, run the following command in the terminal:

Bash

```
azd deploy
```

3. Once the command completes, refresh the browser tab with the ReleCloud website to see the update. Depending on the web hosting platform being used, it could take several minutes before your changes are visible.

You're now ready to edit and delete files in the template. For more information, see [What can I edit or delete in the template?](#)

Clean up resources

Clean up the resources created by the template by running the `azd down` command.

Bash

```
azd down
```

The `azd down` command deletes the Azure resources and the GitHub Actions workflow. When prompted, agree to deleting all resources associated with the resource group.

Optional: Find your codespace

This section demonstrates how your code is (temporarily) running and persisted short-term in a Codespace. If you plan on continuing to work on the code, you should publish the code to a new repository.

1. Close all tabs related to this Quickstart article, or shut down your web browser entirely.
2. Open your web browser and a new tab, and navigate to:
<https://github.com/codespaces>
3. Near the bottom, you'll see a list of recent Codespaces. Look for the one you created in a section titled "Owned by Azure-Samples".
4. Select the ellipsis to the right of this Codespace to view a context menu. From here you can rename the codespace, publish to a new repository, change machine type, stop the codespace, and more.

Optional: Publish a GitHub repository from Codespaces

At this point, you have a Codespace, which is a container hosted by GitHub running your Visual Studio Code development environment with your new code generated from an `azd` template. However, the code isn't stored in a GitHub repository. If you plan on continuing to work on the code, you should make that a priority.

1. From the context menu for the codespace, select "Publish to a new repository".
2. In the "Publish to a new repository" dialog, rename your new repo and choose whether you want it to be a public or private repo. Select "Create repository".
3. After a few moments, the repository will be created and the code you generated earlier in this Quickstart will be pushed to the new repository. Select the "See repository" button to navigate to the new repo.
4. To reopen and continue editing code, select the green "< > Code" drop-down, switch to the Codespaces tab, and select the name of the Codespace you were working on previously. You should now be returned to your Codespace Visual Studio Code development environment.
5. Use the Source Control pane to create new branches and stage and commit new changes to your code.

Troubleshooting

If you see errors during `azd up`, try the following:

- Run `azd down` to remove any resources that may have been created. Alternatively, you can delete the resource group that was created in the Azure portal.

- Go to the Codespaces page for your GitHub account, find the Codespace created during this Quickstart, select the ellipsis at the right and choose "Delete" from the context menu.
- In the Azure portal, search for Key Vaults. Select to *Manage deleted vaults*, choose your subscription, select all key vaults that contain the name *azdtest* or whatever you named your environment, and select *Purge*.
- Retry the steps in this quickstart again. This time when prompted, choose a simpler name for your environment. Try a short name, lower-case letters, no numbers, no upper-case letters, no special characters.
- When retrying the quickstart steps, choose a different location.

See the [FAQ](#) for a more comprehensive list of possible issues and solutions.

Related content

- [Learn more about the Python web azd templates](#)
- [Learn more about the azd commands.](#)
- Learn what each of the folders and files in the project do and [what you can edit or delete?](#)
- [Learn more about GitHub Codespaces ↗](#)
- [Update the Bicep templates to add or remove Azure services.](#) Don't know Bicep? Try this [Learning Path: Fundamentals of Bicep](#)
- [Use azd to set up a GitHub Actions CI/CD pipeline to redeploy on merge to main branch](#)
- Set up monitoring so that you can [Monitor your app using the Azure Developer CLI](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Scaling services deployed with the azd Python web templates using Bicep

Article • 03/20/2025

The [Python web azd templates](#) allow you to quickly create a new web application and deploy it to Azure. The `azd` templates were designed to use low-cost Azure service options. Undoubtedly, you'll want to adjust the service levels (or skus) for each of the services defined in the template for your scenario.

In this Quickstart, you'll update the appropriate bicep template files to scale up existing services and add new services to your deployment. Then, you'll run the `azd provision` command and view the change you made to the Azure deployment.

Prerequisites

An Azure subscription - [Create one for free](#)

You must have the following installed on your local computer:

- [Azure Developer CLI](#)
- [Docker Desktop](#)
- [Visual Studio Code](#)
- [Dev Container Extension](#)
- [Visual Studio Code Bicep](#) This extension helps you author Bicep syntax.

Deploy a template

To begin, you need a working `azd` deployment. Once you have that in place, you're able to modify the Bicep files generated by the `azd` template.

1. Follow steps 1 through 7 in the [Quickstart article](#). In step 2, use the `azure-django-postgres-flexible-appservice` template. For your convenience, here's the entire sequence of commands to issue from the command line:

```
shell
```

```
mkdir azdtest
cd azdtest
azd init --template azure-django-postgres-flexible-appservice
```

```
azd auth login  
azd up
```

Once `azd up` finishes, open the Azure portal, navigate to the Azure App Service that was deployed in your new Resource Group and take note of the App Service pricing plan (see the App Service plan's Overview page, Essentials section, "Pricing plan" value).

2. In step 1 of the Quickstart article, you were instructed to create the `azdtest` folder. Open that folder in Visual Studio Code.
3. In the Explorer pane, navigate to the `infra` folder. Observe the subfolders and files in the `infra` folder.

The `main.bicep` file orchestrates the creation of all the services deployed when performing an `azd up` or `azd provision`. It calls into other files, like `db.bicep` and `web.bicep`, which in turn call into files contained in the `\core` subfolder.

The `\core` subfolder is a deeply nested folder structure containing bicep templates for many Azure services. Some of the files in the `\core` subfolder are referenced by the three top level bicep files (`main.bicep`, `db.bicep` and `web.bicep`) and some aren't used at all in this project.

Scale a service by modifying its Bicep properties

You can scale an existing resource in your deployment by changing its SKU. To demonstrate this, you'll change the App Service plan from the "Basic Service plan" (which is designed for apps with lower traffic requirements and don't need advanced auto scale and traffic management features) to the "Standard Service plan", which is designed for running production workloads.

ⓘ Note

Not all SKU changes can be made after the fact. Some research may be necessary to better understand your scaling options.

1. Open the `web.bicep` file and locate the `appService` module definition. In particular, look for the property setting:

Bicep

```
sku: {
    name: 'B1'
}
```

Change the value from `B1` to `S1` as follows:

Bicep

```
sku: {
    name: 'S1'
}
```

ⓘ Important

As a result of this change, the price per hour will increase slightly. Details about the different service plans and their associated costs can be found on the [App Service pricing page](#).

- Assuming you already have the application deployed in Azure, use the following command to deploy changes to the infrastructure while not redeploying the application code itself.

shell

```
azd provision
```

You shouldn't be prompted for a location and subscription. Those values are saved in the `.azure<environment-name>.env` file where `<environment-name>` is the environment name you provided during `azd init`.

- When `azd provision` is complete, confirm your web application still works. Also find the App Service Plan for your Resource Group and confirm that the Pricing Plan is set to the Standard Service Plan (S1).

This concludes the Quickstart, however there are many Azure services that can help you build more scalable and production-ready applications. A great place to start would be to learn about [Azure API Management](#), [Azure Front Door](#), [Azure CDN](#), and [Azure Virtual Network](#), to name a few.

Clean up resources

Clean up the resources created by the template by running the `azd down` command.

```
shell
```

```
azd down
```

The `azd down` command deletes the Azure resources and the GitHub Actions workflow. When prompted, agree to deleting all resources associated with the resource group.

You can also delete the `azdtest` folder, or use it as the basis for your own application by modifying the files of the project.

Related Content

- [Learn more about the Python web azd templates](#)
- [Learn more about the azd commands.](#)
- Learn what each of the folders and files in the project do and [what you can edit or delete?](#)
- Update the Bicep templates to add or remove Azure services. Don't know Bicep? Try this [Learning Path: Fundamentals of Bicep](#)
- [Use azd to set up a GitHub Actions CI/CD pipeline to redeploy on merge to main branch](#)
- Set up monitoring so that you can [Monitor your app using the Azure Developer CLI](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Deploy a Python (Django, Flask, or FastAPI) web app to Azure App Service

06/04/2025

In this quickstart, you deploy a Python web app (Django, Flask, or FastAPI) to [Azure App Service](#). Azure App Service is a fully managed web hosting service that supports Python apps hosted in a Linux server environment.

To complete this quickstart, you need:

- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Python 3.9 or higher ↗](#) installed locally.

! Note

This article contains current instructions on deploying a Python web app using Azure App Service. Python on Windows is no longer supported.

Skip to the end

You can quickly deploy the sample app in this tutorial using Azure Developer CLI and see it running in Azure. Just run the following commands in the [Azure Cloud Shell ↗](#) want, and follow the prompt:

Flask

Bash

```
mkdir flask-quickstart
cd flask-quickstart
azd init --template https://github.com/Azure-Samples/msdocs-python-flask-
webapp-quickstart
azd up
```

And, to delete the resources:

Bash

```
azd down
```

Sample application

This quickstart can be completed using either Flask, Django, or FastAPI. A sample application in each framework is provided to help you follow along with this quickstart. Download or clone the sample application to your local workstation.

Flask

Console

```
git clone https://github.com/Azure-Samples/msdocs-python-flask-webapp-quickstart
```

To run the application locally:

Flask

1. Go to the application folder:

Console

```
cd msdocs-python-flask-webapp-quickstart
```

2. Create a virtual environment for the app:

Windows

Console

```
py -m venv .venv  
.venv\scripts\activate
```

3. Install the dependencies:

Console

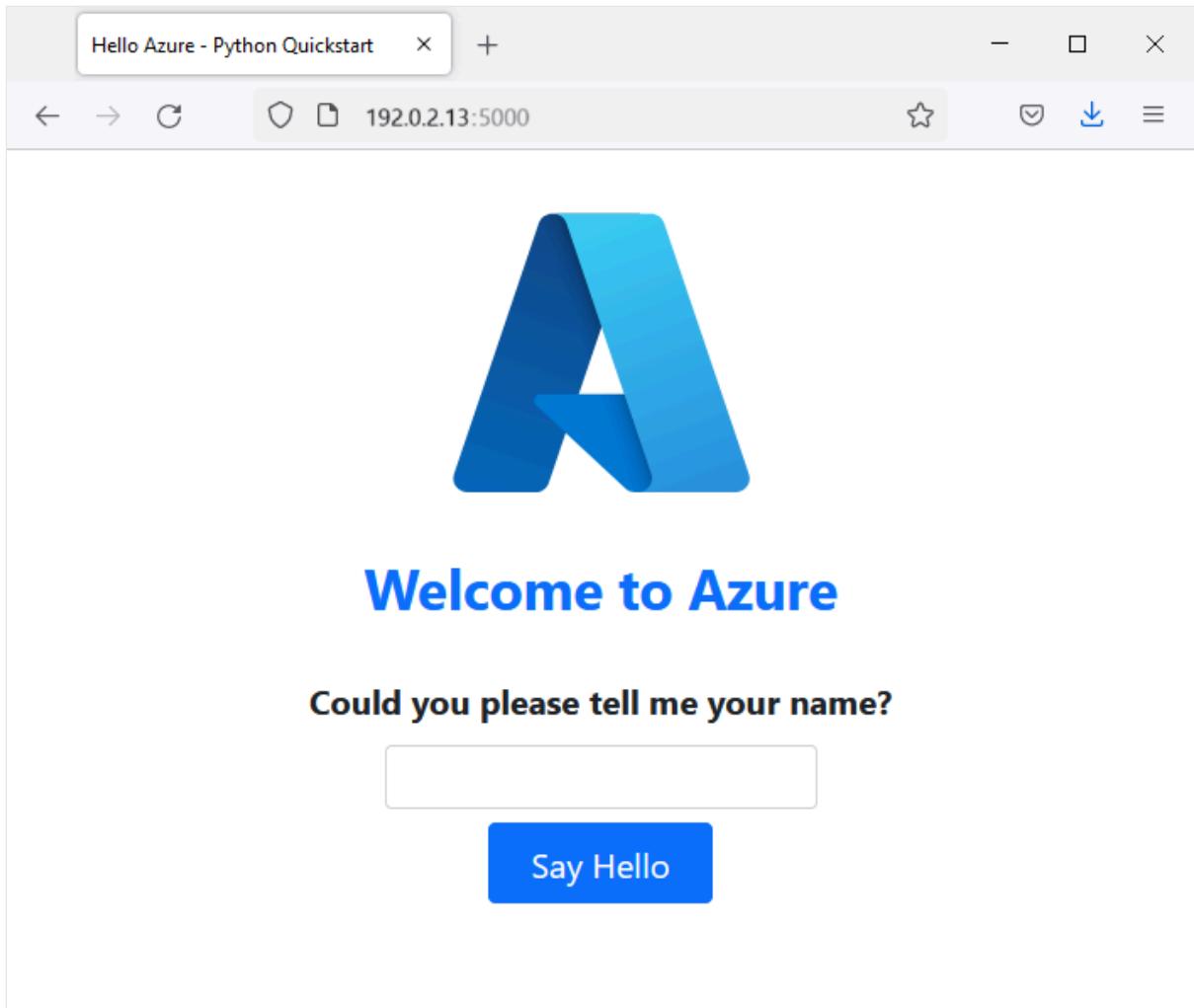
```
pip install -r requirements.txt
```

4. Run the app:

Console

```
flask run
```

5. Browse to the sample application at `http://localhost:5000` in a web browser.



Having issues? [Let us know ↗](#).

Create a web app in Azure

To host your application in Azure, you need to create an Azure App Service web app in Azure. You can create a web app using the [Azure CLI](#), [VS Code ↗](#), [Azure Tools extension pack ↗](#), or the [Azure portal ↗](#).

Azure CLI

Azure CLI commands can be run on a computer with the [Azure CLI installed](#).

Azure CLI has a command `az webapp up` that will create the necessary resources and deploy your application in a single step.

If necessary, log in to Azure using [az login](#).

```
Azure CLI
```

```
az login
```

Create the webapp and other resources, then deploy your code to Azure using [az webapp up](#).

```
Azure CLI
```

```
az webapp up --runtime PYTHON:3.13 --sku B1 --logs
```

- The `--runtime` parameter specifies what version of Python your app is running. This example uses Python 3.13. To list all available runtimes, use the command `az webapp list-runtimes --os linux --output table`.
- The `--sku` parameter defines the size (CPU, memory) and cost of the app service plan. This example uses the B1 (Basic) service plan, which will incur a small cost in your Azure subscription. For a full list of App Service plans, view the [App Service pricing](#) page.
- The `--logs` flag configures default logging required to enable viewing the log stream immediately after launching the webapp.
- You can optionally specify a name with the argument `--name <app-name>`. If you don't provide one, then a name will be automatically generated.
- You can optionally include the argument `--location <location-name>` where `<location_name>` is an available Azure region. You can retrieve a list of allowable regions for your Azure account by running the [az appservice list-locations](#) command.

The command may take a few minutes to complete. While the command is running, it provides messages about creating the resource group, the App Service plan, and the app resource, configuring logging, and doing ZIP deployment. It then returns a message that includes the app's URL, which is the app's URL on Azure.

```
The webapp '<app-name>' doesn't exist
Creating Resource group '<group-name>' ...
Resource group creation complete
Creating AppServicePlan '<app-service-plan-name>' ...
Creating webapp '<app-name>' ...
Configuring default logging for the app, if not already enabled
Creating zip with contents of dir /home/cephas/myExpressApp ...
Getting scm site credentials for zip deployment
Starting zip deployment. This operation can take a while to complete ...
Deployment endpoint responded with status code 202
You can launch the app at <URL>
```

```
{  
    "URL": "<URL>",  
    "appserviceplan": "<app-service-plan-name>",  
    "location": "centralus",  
    "name": "<app-name>",  
    "os": "<os-type>",  
    "resourcegroup": "<group-name>",  
    "runtime_version": "python|3.13",  
    "runtime_version_detected": "0.0",  
    "sku": "FREE",  
    "src_path": "<your-folder-location>"  
}
```

⚠ Note

The `az webapp up` command does the following actions:

- Create a default [resource group](#).
- Create a default [App Service plan](#).
- [Create an app](#) with the specified name.
- [Zip deploy](#) all files from the current working directory, [with build automation enabled](#).
- Cache the parameters locally in the `.azure/config` file so that you don't need to specify them again when deploying later with `az webapp up` or other `az webapp` commands from the project folder. The cached values are used automatically by default.

Having issues? [Let us know ↗](#).

Deploy your application code to Azure

Azure App Service supports multiple methods to deploy your application code to Azure, including GitHub Actions and all major CI/CD tools. This article focuses on how to deploy your code from your local workstation to Azure.

[Deploy using Azure CLI](#)

Since the `az webapp up` command created the necessary resources and deployed your application in a single step, you can move on to the next step.

Having issues? Refer first to the [Troubleshooting guide](#). If that doesn't help, [let us know ↗](#).

Configure startup script

Based on the presence of certain files in a deployment, App Service automatically detects whether an app is a Django or Flask app and performs default steps to run your app. For apps based on other web frameworks like FastAPI, you need to configure a startup script for App Service to run your app; otherwise, App Service runs a default read-only app located in the `opt/defaultsite` folder.

To learn more about how App Service runs Python apps and how you can configure and customize its behavior with your app, see [Configure a Linux Python app for Azure App Service](#).

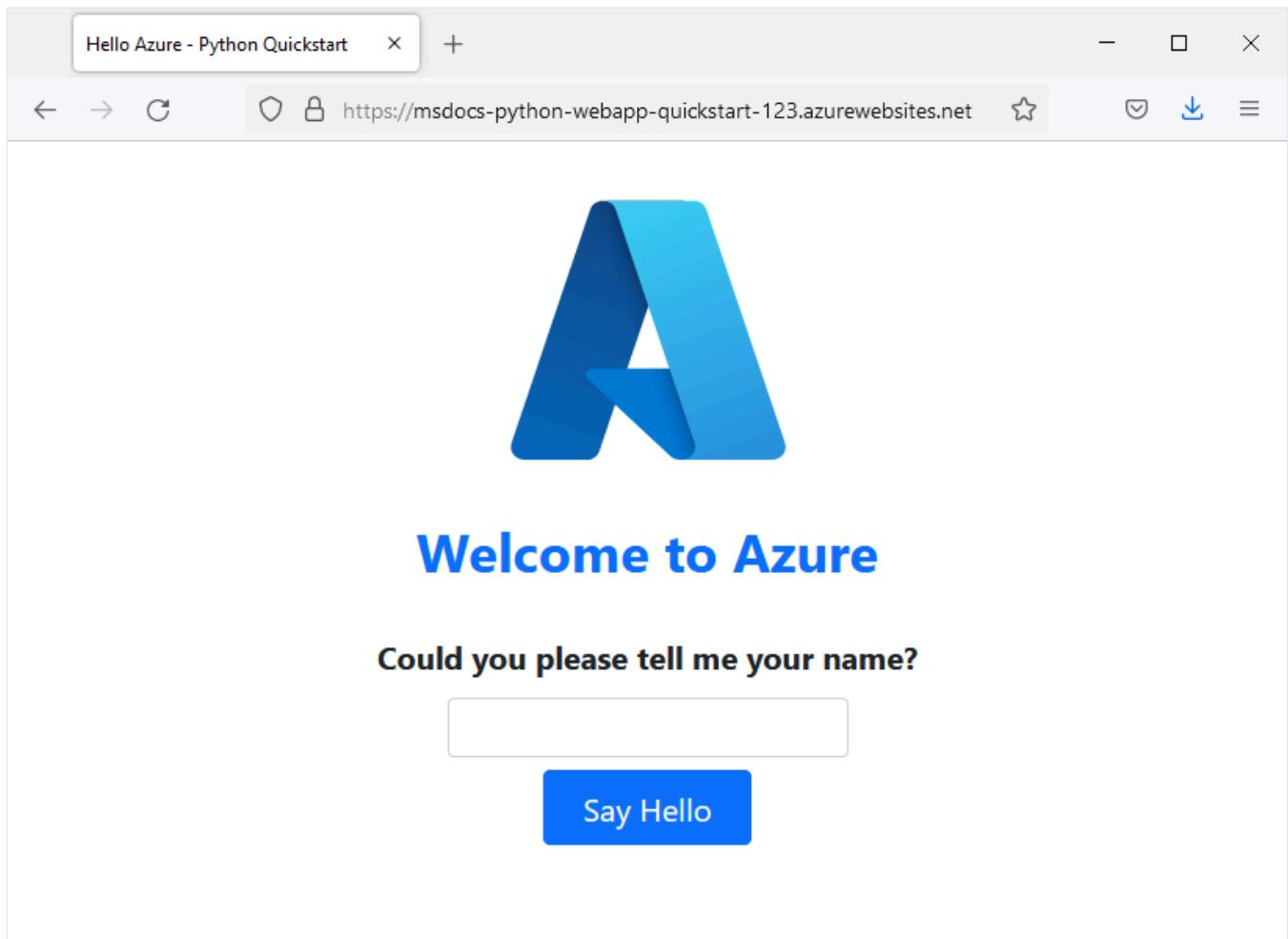
Azure CLI

App Service automatically detects the presence of a Flask app. No additional configuration is needed for this quickstart.

Browse to the app

Browse to the deployed application in your web browser. You can follow a link from the Azure portal. Go to the [Overview](#) page and select **Default Domain**. If you see a default app page, wait a minute and refresh the browser.

The Python sample code is running a Linux container in App Service using a built-in image.



Congratulations! You've deployed your Python app to App Service.

Having issues? Refer first to the [Troubleshooting guide](#). If that doesn't help, [let us know ↗](#).

Stream logs

Azure App Service captures all message output to the console to assist you in diagnosing issues with your application. The sample apps include `print()` statements to demonstrate this capability.

Flask

Python

```
@app.route('/')
def index():
    print('Request for index page received')
    return render_template('index.html')

@app.route('/favicon.ico')
def favicon():
    return send_from_directory(os.path.join(app.root_path, 'static'),
                             'favicon.ico',
```

```
mimetype='image/vnd.microsoft.icon')

@app.route('/hello', methods=['POST'])
def hello():
    name = request.form.get('name')

    if name:
        print('Request for hello page received with name=%s' % name)
        return render_template('hello.html', name = name)
    else:
        print('Request for hello page received with no name or blank name --'
              'redirecting')
        return redirect(url_for('index'))
```

You can review the contents of the App Service diagnostic logs by using the Azure CLI, VS Code, or the Azure portal.

Azure CLI

First, you need to configure Azure App Service to output logs to the App Service filesystem by using the [az webapp log config](#) command.

bash

Azure CLI

```
az webapp log config \
    --web-server-logging filesystem \
    --name $APP_SERVICE_NAME \
    --resource-group $RESOURCE_GROUP_NAME
```

To stream logs, use the [az webapp log tail](#) command.

bash

Azure CLI

```
az webapp log tail \
    --name $APP_SERVICE_NAME \
    --resource-group $RESOURCE_GROUP_NAME
```

Refresh the home page in the app or attempt other requests to generate some log messages. The output should look similar to the following.

Output

Starting Live Log Stream ---

```
2021-12-23T02:15:52.740703322Z Request for index page received
2021-12-23T02:15:52.740740222Z 169.254.130.1 - - [23/Dec/2021:02:15:52 +0000]
"GET / HTTP/1.1" 200 1360 "https://msdocs-python-webapp-quickstart-
123.azurewebsites.net/hello" "Mozilla/5.0 (Windows NT 10.0; Win64; x64;
rv:95.0) Gecko/20100101 Firefox/95.0"
2021-12-23T02:15:52.841043070Z 169.254.130.1 - - [23/Dec/2021:02:15:52 +0000]
"GET /static/bootstrap/css/bootstrap.min.css HTTP/1.1" 200 0 "https://msdocs-
python-webapp-quickstart-123.azurewebsites.net/" "Mozilla/5.0 (Windows NT
10.0; Win64; x64; rv:95.0) Gecko/20100101 Firefox/95.0"
2021-12-23T02:15:52.884541951Z 169.254.130.1 - - [23/Dec/2021:02:15:52 +0000]
"GET /static/images/azure-icon.svg HTTP/1.1" 200 0 "https://msdocs-python-
webapp-quickstart-123.azurewebsites.net/" "Mozilla/5.0 (Windows NT 10.0;
Win64; x64; rv:95.0) Gecko/20100101 Firefox/95.0"
2021-12-23T02:15:53.043211176Z 169.254.130.1 - - [23/Dec/2021:02:15:53 +0000]
"GET /favicon.ico HTTP/1.1" 404 232 "https://msdocs-python-webapp-quickstart-
123.azurewebsites.net/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:95.0)
Gecko/20100101 Firefox/95.0"

2021-12-23T02:16:01.304306845Z Request for hello page received with name=David
2021-12-23T02:16:01.304335945Z 169.254.130.1 - - [23/Dec/2021:02:16:01 +0000]
"POST /hello HTTP/1.1" 200 695 "https://msdocs-python-webapp-quickstart-
123.azurewebsites.net/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:95.0)
Gecko/20100101 Firefox/95.0"
2021-12-23T02:16:01.398399251Z 169.254.130.1 - - [23/Dec/2021:02:16:01 +0000]
"GET /static/bootstrap/css/bootstrap.min.css HTTP/1.1" 304 0 "https://msdocs-
python-webapp-quickstart-123.azurewebsites.net/hello" "Mozilla/5.0 (Windows NT
10.0; Win64; x64; rv:95.0) Gecko/20100101 Firefox/95.0"
2021-12-23T02:16:01.430740060Z 169.254.130.1 - - [23/Dec/2021:02:16:01 +0000]
"GET /static/images/azure-icon.svg HTTP/1.1" 304 0 "https://msdocs-python-
webapp-quickstart-123.azurewebsites.net/hello" "Mozilla/5.0 (Windows NT 10.0;
Win64; x64; rv:95.0) Gecko/20100101 Firefox/95.0"
```

Having issues? Refer first to the [Troubleshooting guide](#). If that doesn't help, [let us know](#).

Clean up resources

When you're finished with the sample app, you can remove all of the resources for the app from Azure. Removing the resource group ensures that you don't incur extra charges and helps keep your Azure subscription uncluttered. Removing the resource group also removes all resources in the resource group and is the fastest way to remove all Azure resources for your app.

Delete the resource group by using the [az group delete](#) command.

Azure CLI

```
az group delete \
  --name msdocs-python-webapp-quickstart \
  --no-wait
```

The `--no-wait` argument allows the command to return before the operation is complete.

Having issues? [Let us know ↗](#).

Next steps

[Tutorial: Python \(Flask\) web app with PostgreSQL](#)

[Tutorial: Python \(Django\) web app with PostgreSQL](#)

[Configure a Python app](#)

[Add user sign-in to a Python web app](#)

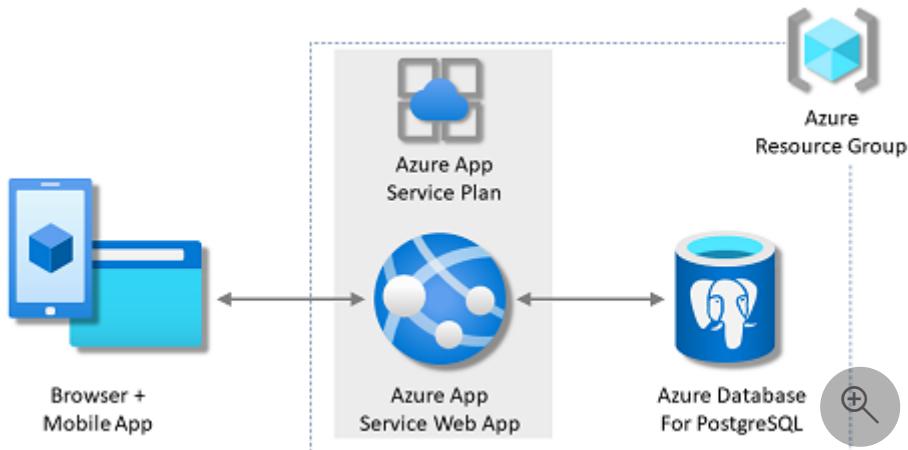
[Tutorial: Run a Python app in a custom container](#)

[Secure an app with a custom domain and certificate](#)

Deploy a Python (Flask) web app with PostgreSQL in Azure

04/17/2025

In this tutorial, you'll deploy a data-driven Python web app ([Flask](#)) to [Azure App Service](#) with the [Azure Database for PostgreSQL](#) relational database service. Azure App Service supports [Python](#) in a Linux server environment. If you want, see the [Django tutorial](#) or the [FastAPI tutorial](#) instead.



In this tutorial, you learn how to:

- ✓ Create a secure-by-default App Service, PostgreSQL, and Redis cache architecture.
- ✓ Secure connection secrets using a managed identity and Key Vault references.
- ✓ Deploy a sample Python app to App Service from a GitHub repository.
- ✓ Access App Service connection strings and app settings in the application code.
- ✓ Make updates and redeploy the application code.
- ✓ Generate database schema by running database migrations.
- ✓ Stream diagnostic logs from Azure.
- ✓ Manage the app in the Azure portal.
- ✓ Provision the same architecture and deploy by using Azure Developer CLI.
- ✓ Optimize your development workflow with GitHub Codespaces and GitHub Copilot.

Prerequisites

- An Azure account with an active subscription. If you don't have an Azure account, you [can create one for free](#).
- A GitHub account. you can also [get one for free](#).
- Knowledge of Python with Flask development.

- (Optional) To try GitHub Copilot, a [GitHub Copilot account](#). A 30-day free trial is available.

Skip to the end

If you just want to see the sample app in this tutorial running in Azure, just run the following commands in the [Azure Cloud Shell](#), and follow the prompt:

Bash

```
mkdir msdocs-flask-postgresql-sample-app
cd msdocs-flask-postgresql-sample-app
azd init --template msdocs-flask-postgresql-sample-app
azd up
```

1. Run the sample

First, you set up a sample data-driven app as a starting point. For your convenience, the [sample repository](#), includes a [dev container](#) configuration. The dev container has everything you need to develop an application, including the database, cache, and all environment variables needed by the sample application. The dev container can run in a [GitHub codespace](#), which means you can run the sample on any computer with a web browser.

! Note

If you are following along with this tutorial with your own app, look at the *requirements.txt* file description in [README.md](#) to see what packages you'll need.

Step 1: In a new browser window:

1. Sign in to your GitHub account.
2. Navigate to <https://github.com/Azure-Samples/msdocs-flask-postgresql-sample-app/fork>.
3. Unselect **Copy the main branch only**. You want all the branches.
4. Select **Create fork**.

The screenshot shows the GitHub fork creation interface for the repository `msdocs-flask-postgresql-sample-app`. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Security, and Insights. The 'Code' tab is selected.

Create a new fork

A **fork** is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

Required fields are marked with an asterisk ()*.

Owner * **Repository name ***

/ `msdocs-flask-postgresql-san`

`msdocs-flask-postgresql-sample-app` is available.

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

Copy the `main` branch only
Contribute back to Azure-Samples/msdocs-flask-postgresql-sample-app by adding your own branch. [Learn more.](#)

You are creating a fork in your personal account.

Create fork

Step 2: In the GitHub fork:

1. Select `main > starter-no-infra` for the starter branch. This branch contains just the sample project and no Azure-related files or configuration.
2. Select **Code > Create codespace on starter-no-infra**. The codespace takes a few minutes to set up, and it runs `pip install -r requirements.txt` for your repository at the end.

This branch is up to date
Azure-Samples/msdocs-1

Contribute

add copilot ext

.devcontainer

.github

azureproject

migrations

static

templates

Remove spurious line. 3 years ago

.env

convert to service connecto... 20 hours ago

starter-no-infra

Go to file

Code spaces

No codespaces

You don't have any codespaces with this repository checked out

Create codespace on starter-no-infra

Learn more about codespaces...

About

No description, website, or topics provided.

Readme

MIT license

Code of conduct

Activity

0 stars

0 watching

0 forks

Releases

No releases published

Create a new release

Packages

No packages published

Step 3: In the codespace terminal:

1. Run database migrations with `flask db upgrade`.
2. Run the app with `flask run`.
3. When you see the notification `Your application running on port 5000 is available.`, select **Open in Browser**. You should see the sample application in a new browser tab. To stop the application, type `Ctrl+C`.

This screenshot shows the Microsoft Visual Studio Code interface with a Python project named "MSDOCS-FLASK-POSTGRESQL-SAMPLE-APP". The terminal tab is active, displaying the output of running `flask db upgrade` and `flask run`. A tooltip in the bottom right corner of the terminal area says "Your application running on port 5000 is available. See all forwarded ports". There are buttons for "Open in Browser" and "Make Public".

💡 Tip

You can ask [GitHub Copilot](#) about this repository. For example:

- @workspace *What does this project do?*
- @workspace *What does the .devcontainer folder do?*

Having issues? Check the [Troubleshooting section](#).

2. Create App Service and PostgreSQL

In this step, you create the Azure resources. The steps used in this tutorial create a set of secure-by-default resources that include App Service and Azure Database for PostgreSQL. For the creation process, you specify:

- The **Name** for the web app. It's used as part of the DNS name for your app.
- The **Region** to run the app physically in the world. It's also used as part of the DNS name for your app.
- The **Runtime stack** for the app. It's where you select the version of Python to use for your app.
- The **Hosting plan** for the app. It's the pricing tier that includes the set of features and scaling capacity for your app.

- The **Resource Group** for the app. A resource group lets you group (in a logical container) all the Azure resources needed for the application.

Sign in to the [Azure portal](#) and follow these steps to create your Azure App Service resources.

Step 1: In the Azure portal:

1. Enter "web app database" in the search bar at the top of the Azure portal.
2. Select the item labeled **Web App + Database** under the **Marketplace** heading. You can also navigate to the [creation wizard](#) directly.

Step 2: In the **Create Web App + Database** page, fill out the form as follows.

1. **Resource Group:** Select **Create new** and use a name of **msdocs-flask-postgres-tutorial**.
2. **Region:** Any Azure region near you.
3. **Name:** **msdocs-python-postgres-XYZ**.
4. **Runtime stack:** **Python 3.12**.
5. **Database:** **PostgreSQL - Flexible Server** is selected by default as the database engine. The server name and database name are also set by default to appropriate values.
6. **Add Azure Cache for Redis?** **No**.
7. **Hosting plan:** **Basic**. When you're ready, you can [scale up](#) to a production pricing tier.
8. Select **Review + create**.
9. After validation completes, select **Create**.

Home >

Create Web App + Database

X

Basics Tags Review + create

This template will create a secure by default configuration where the only publicly accessible endpoint will be your app following the recommended security best practices. [Learn more](#)

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

Resource Group * ⓘ

(New) msdocs-flask-postgres-tutorial

[Create new](#)

Region *

Central US

Web App Details

Name

msdocs-python-postgres-234

-epb5abc3augnh4b2.centralus-01.azurewebsites.net

Secure unique default hostname on. [More about this update](#)

Runtime stack *

Python 3.12

Database



Database access will be locked down and not exposed to the public internet. This is in compliance with recommended best practices for security.

Engine * ⓘ

PostgreSQL - Flexible Server (recommended)

Server name *

msdocs-python-postgres-234-server

Database name *

msdocs-python-postgres-234-database

Azure Cache for Redis

Add Azure Cache for Redis?

Yes

No

Hosting

Hosting plan *

Basic - For hobby or research purposes

Standard - General purpose production apps

[Review + create](#)

< Previous

Next : Tags >



Step 3: The deployment takes a few minutes to complete. Once deployment completes, select the **Go to resource** button. You're taken directly to the App Service app, but the following

resources are created:

- **Resource group:** The container for all the created resources.
- **App Service plan:** Defines the compute resources for App Service. A Linux plan in the *Basic* tier is created.
- **App Service:** Represents your app and runs in the App Service plan.
- **Virtual network:** Integrated with the App Service app and isolates back-end network traffic.
- **Network interfaces:** Represents private IP addresses, one for each of the private endpoints.
- **Azure Database for PostgreSQL flexible server:** Accessible only from within the virtual network. A database and a user are created for you on the server.
- **Private DNS zones:** Enables DNS resolution of the key vault and the database server in the virtual network.

 Your deployment is complete

 Deployment name : Microsoft.Web-WebAppDatabase-Portal-afa69d9d-97bc
Subscription :
Resource group : [msdocs-python-postgres-tutorial](#)
Start time : 11/29/2023, 10:17:05 AM
Correlation ID :

> Deployment details
▼ Next steps

[Go to resource](#)

Give feedback 

 [Tell us about your experience with deployment](#)

3. Secure connection secrets

The creation wizard generated the connectivity variables for you already as [app settings](#). However, the security best practice is to keep secrets out of App Service completely. You'll move your secrets to a key vault and change your app setting to [Key Vault references](#) with the help of Service Connectors.

Step 1: Retrieve the existing connection string

1. In the left menu of the App Service page, select **Settings > Environment variables**.
2. Select **AZURE_POSTGRESQL_CONNECTIONSTRING**.

3. In Add/Edit application setting, in the Value field, find the *password=* part at the end of the string.
4. Copy the password string after *Password=* for use later. This app setting lets you connect to the Postgres database secured behind a private endpoint. However, the secret is saved directly in the App Service app, which isn't the best. You'll change this.

The screenshot shows the 'Add/Edit application setting' dialog in the Azure portal. The 'Name' field is set to 'AZURE_POSTGRESQL_CONNECTIONSTRING'. The 'Value' field contains the connection string: 'host=msdocs-python-postgres-3-server.postgres.database.azure.com port=5432 sslmode=require user=gqlcbtig password=xxxxxx'. The 'Deployment slot setting' checkbox is checked. The 'Apply' and 'Discard' buttons are visible at the bottom.

Step 2: Create a key vault for secure management of secrets

1. In the top search bar, type "key vault", then select Marketplace > Key Vault.
2. In Resource Group, select msdocs-python-postgres-tutorial.
3. In Key vault name, type a name that consists of only letters and numbers.
4. In Region, set it to the same location as the resource group.

Azure Key Vault is a cloud service used to manage keys, secrets, and certificates. Key Vault eliminates the need for developers to store security information in their code. It allows you to centralize the storage of your application secrets which greatly reduces the chances that secrets may be leaked. Key Vault also allows you to securely store secrets and keys backed by Hardware Security Modules or HSMs. The HSMs used are Federal Information Processing Standards (FIPS) 140-2 Level 2 validated. In addition, key vault provides logs of all access and usage attempts of your secrets so you have a complete audit trail for compliance.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	Dev Compute and Platform - FY25Q3
Resource group *	msdocs-python-postgres-123_group
	Create new

Instance details

Key vault name *	vault091979
Region *	Canada Central
Pricing tier *	Standard

Recovery options

Soft delete protection will automatically be enabled on this key vault. This feature allows you to recover or permanently delete a key vault and secrets for the duration of the retention period. This protection applies to the key vault and the secrets stored within the key vault.

To enforce a mandatory retention period and prevent the permanent deletion of key vaults or secrets prior to the retention period elapsing, you can turn on purge protection. When purge protection is enabled, secrets cannot be purged by users or

[Previous](#)

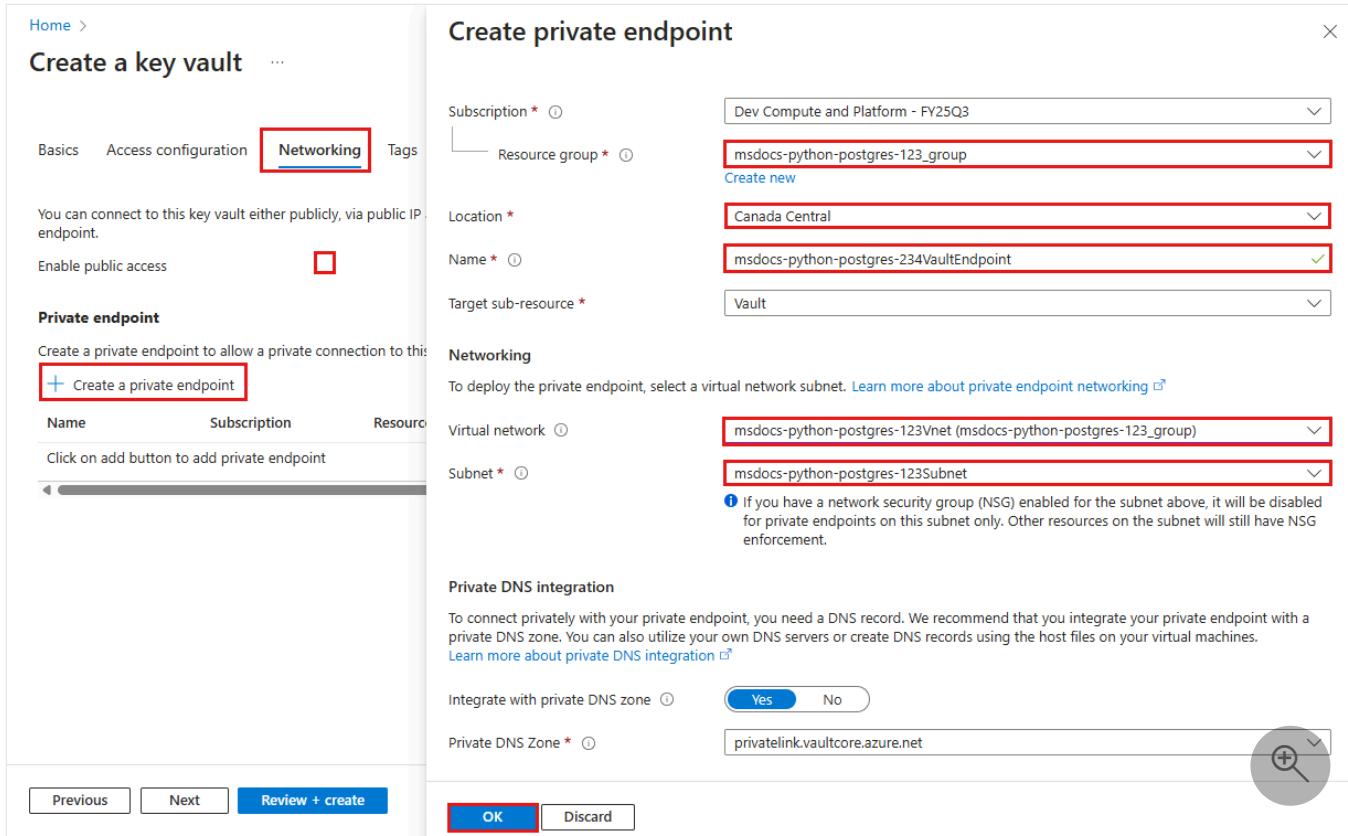
[Next](#)

[Review + create](#)



Step 3: Secure the key vault with a Private Endpoint

1. Select the **Networking** tab.
2. Unselect **Enable public access**.
3. Select **Create a private endpoint**.
4. In **Resource Group**, select **msdocs-python-postgres-tutorial**.
5. In the dialog, in **Location**, select the same location as your App Service app.
6. In **Name**, type **msdocs-python-postgres-XYZVaultEndpoint**.
7. In **Virtual network**, select **msdocs-python-postgres-XYZVnet**.
8. In **Subnet**, **msdocs-python-postgres-XYZSubnet**.
9. Select **OK**.
10. Select **Review + create**, then select **Create**. Wait for the key vault deployment to finish.
You should see "Your deployment is complete."



Step 4: Configure the PostgreSQL connector

1. In the top search bar, type *msdocs-python-postgres*, then select the App Service resource called **msdocs-python-postgres-XYZ**.
2. In the App Service page, in the left menu, select **Settings > Service Connector**. There's already a connector, which the app creation wizard created for you.
3. Select checkbox next to the PostgreSQL connector, then select **Edit**.
4. In **Client type**, select **Django**. Even though you have a Flask app, the [Django client type in the PostgreSQL service connector](#) gives you database variables in separate settings instead of one connection string. The separate variables are easier for you to use in your application code, which uses [SQLAlchemy](#) to connect to the database.
5. Select the **Authentication** tab.
6. In **Password**, paste the password you copied earlier.
7. Select **Store Secret in Key Vault**.
8. Under **Key Vault Connection**, select **Create new**. A **Create connection** dialog is opened on top of the edit dialog.

The screenshot shows the Azure portal interface for managing a Service Connector. On the left, the sidebar is open with various service options like Environment variables, Configuration, Authentication, Identity, Backups, Custom domains, Certificates, Networking, Scale up (App Service plan), Scale out (App Service plan), WebJobs (preview), and Service Connector. The 'Service Connector' item is highlighted with a red box. In the main pane, a 'defaultConnector' dialog is displayed under the 'Authentication' tab. The dialog title is 'defaultConnector'. It has tabs for Basics, Authentication (which is selected and highlighted with a red box), and Networking. The Basics tab contains a note about using Service Connector to register a resource provider. The Authentication tab lists four authentication methods: System assigned managed identity, User assigned managed identity, Connection string (which is selected and highlighted with a red circle), and Service principal. Below these are sections for 'Continue with...' (Database credentials and Key Vault) and connection details (Username: 'gcibcbtiq', Password: masked, Store Secret In Key Vault checked, Key Vault Connection dropdown set to 'mangesh-key-vault-python (keyvault_362d5)', and a 'Create new' button highlighted with a red box). At the bottom are 'Next : Networking', 'Previous', and 'Cancel' buttons.

Step 5: Establish the Key Vault connection

1. In the **Create connection** dialog for the Key Vault connection, in **Key Vault**, select the key vault you created earlier.
2. Select **Review + Create**.
3. When validation completes, select **Create**.

Create connection

Basics Networking Review + Create

Select the service instance and client type.

Service type * ⓘ

Key Vault

Connection name * ⓘ

keyvault_cc22c

Subscription * ⓘ

Dev Compute and Platform - FY25Q3

Key vault * ⓘ

mangesh-key-vault-python

Create new

Client type * ⓘ

Python

Next : Networking Cancel Report an issue.

The screenshot shows the 'Create connection' dialog box. The 'Review + Create' tab is selected. The 'Service type' dropdown contains 'Key Vault'. The 'Connection name' input field contains 'keyvault_cc22c'. The 'Subscription' dropdown contains 'Dev Compute and Platform - FY25Q3'. The 'Key vault' dropdown is highlighted with a red box and contains 'mangesh-key-vault-python'. There is a 'Create new' link below it. The 'Client type' dropdown contains 'Python'. At the bottom, there are 'Next : Networking' (blue), 'Cancel', and 'Report an issue.' buttons.

Step 6: Finalize the PostgreSQL connector settings

1. You're back in the edit dialog for **defaultConnector**. In the **Authentication** tab, wait for the key vault connector to be created. When it's finished, the **Key Vault Connection** dropdown automatically selects it.
2. Select **Next: Networking**.
3. Select **Save**. Wait until the **Update succeeded** notification appears.

defaultConnector

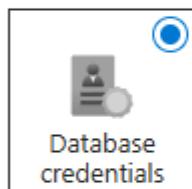
X

Basics **Authentication** Networking

Select the authentication type you'd like to use between your compute service and target service. [Learn more about authentication.](#)

- System assigned managed identity (Supported via Azure CLI. [Learn more](#)) ⓘ
- User assigned managed identity (Supported via Azure CLI. [Learn more](#)) ⓘ
- Connection string ⓘ
- Service principal (Supported via Azure CLI. [Learn more](#)) ⓘ

Continue with...



Username *

gcibcbtiq

Password *

.....

[Forgot password?](#)

Store Secret In Key Vault ⓘ

Key Vault Connection * ⓘ

mangesh-key-vault-python (keyvault_362d5) ⏺

[Create new](#)

Store Configuration in App Configuration ⓘ

Next : Networking

Previous

Cancel



Step 7: Verify the Key Vault integration

1. From the left menu, select **Settings > Environment variables** again.
2. Next to **AZURE_POSTGRESQL_PASSWORD**, select **Show value**. The value should be `@Microsoft.KeyVault(...)`, which means that it's a [key vault reference](#) because the secret is now managed in the key vault.

The screenshot shows the 'App settings' blade in the Azure portal. On the left, there's a sidebar with 'Recommended services (preview)', 'Deployment' (with 'Deployment slots' and 'Deployment Center'), and 'Settings' (with 'Environment variables' selected). The main area has tabs for 'App settings' (selected) and 'Connection strings'. It includes a search bar, 'Add', 'Refresh', 'Show values', and 'Advanced edit' buttons. A table lists environment variables:

Name	Value	Deployment slot
AZURE_KEYVAULT_RESOURCE...	Show value	✓
AZURE_KEYVAULT_SCOPE	Show value	✓
AZURE_POSTGRESQL_CONNE...	Show value	✓
AZURE_REDIS_CONNECTIONS...	Show value	✓

At the bottom are 'Apply' and 'Discard' buttons, and a 'Send us your feedback' link.

To summarize, the process for securing your connection secrets involved:

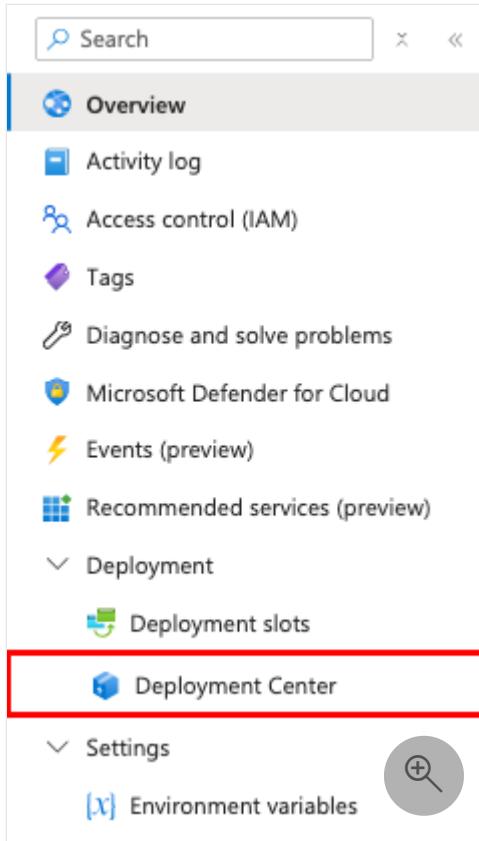
- Retrieving the connection secrets from the App Service app's environment variables.
- Creating a key vault.
- Creating a Key Vault connection with the system-assigned managed identity.
- Updating the service connectors to store the secrets in the key vault.

Having issues? Check the [Troubleshooting section](#).

4. Deploy sample code

In this step, you configure GitHub deployment using GitHub Actions. It's just one of many ways to deploy to App Service, but also a great way to have continuous integration in your deployment process. By default, every `git push` to your GitHub repository kicks off the build and deploy action.

Step 1: In the left menu, select Deployment > Deployment Center.



Step 2: In the Deployment Center page:

1. In **Source**, select **GitHub**. By default, **GitHub Actions** is selected as the build provider.
2. Sign in to your GitHub account and follow the prompt to authorize Azure.
3. In **Organization**, select your account.
4. In **Repository**, select **msdocs-flask-postgresql-sample-app**.
5. In **Branch**, select **starter-no-infra**. This is the same branch that you worked in with your sample app, without any Azure-related files or configuration.
6. For **Authentication type**, select **User-assigned identity**.
7. In the top menu, select **Save**. App Service commits a workflow file into the chosen GitHub repository, in the `.github/workflows` directory. By default, the deployment center [creates a user-assigned identity](#) for the workflow to authenticate using Microsoft Entra (OIDC authentication). For alternative authentication options, see [Deploy to App Service using GitHub Actions](#).

Save Discard Browse Manage publish profile Sync Leave Feedback

Settings * Logs FTPS credentials

You're now in the production slot, which is not recommended for setting up CI/CD. Learn more X

Deploy and build code from your preferred source and build provider. [Learn more](#)

Source* GitHub

Building with GitHub Actions. [Change provider](#).

GitHub

App Service will place a GitHub Actions workflow in your chosen repository to build and deploy your app whenever there is a commit on the chosen branch. If you can't find an organization or repository, you may need to enable additional permissions on GitHub. You must have write access to your chosen GitHub repository to deploy with GitHub Actions.

[Learn more](#)

Signed in as Icephas [Change Account](#) ⓘ

Organization*

Repository* msdocs-flask-postgresql-sample-app

Branch * starter-no-infra

Workflow Option*

Overwrite the workflow. Overwrite the existing workflow file 'starter-no-infra_msdocs-python-postgres-234.yml' in the selected repository and branch.

Use existing workflow. Use the existing workflow file 'starter-no-infra_msdocs-python-postgres-234.yml' in the selected repository and branch.

Build

Runtime stack Python

Version Python 3.12

Authentication settings

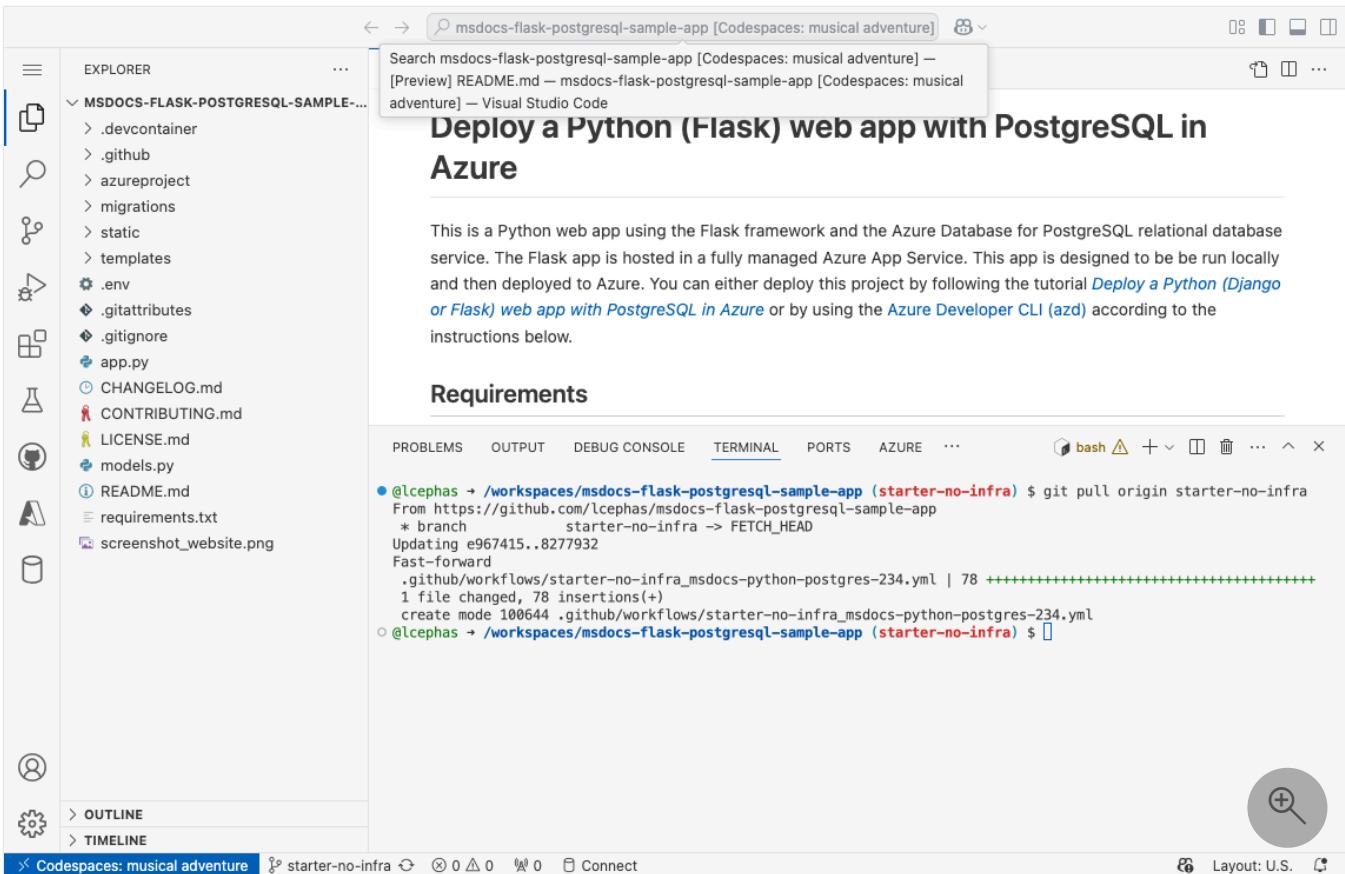
Select how you want your GitHub Action workflow to authenticate to Azure. If you choose user-assigned identity, the identity selected will be federated with GitHub as an authorized client and given write permissions on the app. [Learn more](#)

Authentication type*

User-assigned identity 🔍

Basic authentication

Step 3: Back in the GitHub codespace of your sample fork, run `git pull origin starter-no-infra`. This pulls the newly committed workflow file into your codespace.



Step 4 (Option 1: with GitHub Copilot):

1. Start a new chat session by selecting the **Chat** view, then selecting **+**.
2. Ask, "*@workspace How does the app connect to the database?*" Copilot might give you some explanation about `SQLAlchemy` how its connection URI is configured in `azureproject/development.py` and `azureproject/production.py`.
3. Ask, "*@workspace In production mode, my app is running in an App Service web app, which uses Azure Service Connector to connect to a PostgreSQL flexible server using the Django client type. What are the environment variable names I need to use?*" Copilot might give you a code suggestion similar to the one in the **Option 2: without GitHub Copilot** steps below and even tell you to make the change in the `azureproject/production.py` file.
4. Open `azureproject/production.py` in the explorer and add the code suggestion. GitHub Copilot doesn't give you the same response every time, and it's not always correct. You might need to ask more questions to fine-tune its response. For tips, see [What can I do with GitHub Copilot in my codespace?](#)



Ask Copilot

Copilot is powered by AI, so mistakes are possible.

Review output carefully before use.

As an internal user, additional telemetry is collected. If you work on a project that contains customer content, you must [disable telemetry](#).

⌚ or type # to attach context

@ to chat with extensions

Type / to use commands

[/help What can you do?](#)

@workspace How does the app connect to the database?

@ ⌚

GPT 4o ▾ ▶ ▾



Step 4 (Option 2: without GitHub Copilot):

1. Open `Program.cs` in the explorer.
2. Find the commented code (lines 3-8) and uncomment it. This creates a connection string for SQLAlchemy by using `AZURE_POSTGRESQL_USER`, `AZURE_POSTGRESQL_PASSWORD`, `AZURE_POSTGRESQL_HOST`, and `AZURE_POSTGRESQL_NAME`.

The screenshot shows the Microsoft Visual Studio Code interface. The left sidebar contains a file tree with a folder named 'MSDOCS-FLASK-POSTGRESQL-SAMPLE-' containing files like .devcontainer, .github, azureproject, __init__.py, development.py, and production.py. The 'production.py' file is selected and has a red border around it. The main editor area shows the following Python code:

```
import os

DATABASE_URI = 'postgresql+psycopg2://{}dbuser:{}@{}dbhost/{}dbname'.
dbuser=os.getenv('AZURE_POSTGRESQL_USER'),
dbpass=os.getenv('AZURE_POSTGRESQL_PASSWORD'),
dbhost=os.getenv('AZURE_POSTGRESQL_HOST'),
dbname=os.getenv('AZURE_POSTGRESQL_NAME')
)
```

Below the editor is a terminal window showing a git pull command:

```
@lcephas ~ /workspaces/msdocs-flask-postgresql-sample-app (starter-no-infra) $ git pull origin s
tarter-no-infra
From https://github.com/lcephas/msdocs-flask-postgresql-sample-app
 * branch      starter-no-infra -> FETCH_HEAD
Updating e967415..8277932
Fast-forward
.github/workflows/starter-no-infra_msdocs-python-postgres-234.yml | 78 ++++++
+++++
1 file changed, 78 insertions(+)
create mode 100644 .github/workflows/starter-no-infra_msdocs-python-postgres-234.yml
@lcephas ~ /workspaces/msdocs-flask-postgresql-sample-app (starter-no-infra) $
```

Step 5:

1. Select the Source Control extension.
2. In the textbox, type a commit message like `Configure Azure database connection`. Or, select and let GitHub Copilot generate a commit message for you.
3. Select Commit, then confirm with Yes.
4. Select Sync changes 1, then confirm with OK.

Screenshot of the Azure Deployment Center page showing the Logs tab. The Refresh button is highlighted with a red box. The Build/Deploy Log entry for 01/22/2025, 3:26:04.. is also highlighted with a red box.

Time	Commit ID	Logs	Commit Author	Status	Message
01/22/2025, 3:26:04..	Ea55fb8	Build/Deploy Lo...	lophas	In Progress....	Configure Azure database connection
01/22/2025, 3:08:15..	80c37f1	App Logs	N/A	Failed	OneDeploy
01/22/2025, 3:08:00..	temp-9d	App Logs	N/A	Failed	OneDeploy
01/22/2025, 3:05:24..	8277932	Build/Deploy Lo...	lophas	Failed	Add or update the Azure App Service build and deployment workflow config

Step 6: Back in the Deployment Center page in the Azure portal:

1. Select the **Logs** tab, then select **Refresh** to see the new deployment run.
2. In the log item for the deployment run, select the **Build/Deploy Logs** entry with the latest timestamp.

Screenshot of the Azure Deployment Center page showing the Logs tab. The Refresh button is highlighted with a red box. The Build/Deploy Log entry for 01/22/2025, 3:26:04.. is also highlighted with a red box.

Time	Commit ID	Logs	Commit Author	Status	Message
01/22/2025, 3:26:04..	Ea55fb8	Build/Deploy Lo...	lophas	In Progress....	Configure Azure database connection
01/22/2025, 3:08:15..	80c37f1	App Logs	N/A	Failed	OneDeploy
01/22/2025, 3:08:00..	temp-9d	App Logs	N/A	Failed	OneDeploy
01/22/2025, 3:05:24..	8277932	Build/Deploy Lo...	lophas	Failed	Add or update the Azure App Service build and deployment workflow config

Step 7: You're taken to your GitHub repository and see that the GitHub action is running. The workflow file defines two separate stages, build and deploy. Wait for the GitHub run to show a status of **Success**. It takes about 5 minutes.

The screenshot shows a GitHub Actions run for the workflow file `starter-no-infra_msdocs-python-postgres-234.yml`. The run was triggered via push 5 minutes ago. It consists of two jobs: `build` and `deploy`. The `build` job completed successfully in 25 seconds. The `deploy` job also completed successfully in 3m 27s, resulting in one artifact. The total duration of the run was 4m 12s. A link to the deployment is provided: <http://msdocs-python-postgres-234-e...>.

Having issues? Check the [Troubleshooting guide](#).

5. Generate database schema

With the PostgreSQL database protected by the virtual network, the easiest way to run [Flask database migrations](#) is in an SSH session with the Linux container in App Service.

Step 1: Back in the App Service page, in the left menu,

1. Select **Development Tools > SSH**.
2. Select **Go**.

Microsoft Azure (Preview) ⟳ Search resources, services, and docs (G+/)

Home > msdocs-python-postgres-234

msdocs-python-postgres-234 | SSH

App Service

Search

Quotas

Change App Service plan

Development Tools

Clone App

SSH

Advanced Tools

Extensions

API

API Management

SSH provides a Web SSH console experience for your Linux App code. [Learn more](#)

Go →

A P P S E R V I C E O N L I N U X

Documentation: <http://aka.ms/webapp-linux>

Python 3.9.7

Note: Any data outside '/home' is not persisted

(antenv) root@aa2d84bd54c7:/tmp/8daa8347537426e# flask db upgrade

Loading config.production.

INFO [alembic.runtime.migration] Context impl PostgresqlImpl.

INFO [alembic.runtime.migration] Will assume transactional DDL.

INFO [alembic.runtime.migration] Running upgrade -> 336483b236e3, initial migration

(antenv) root@aa2d84bd54c7:/tmp/8daa8347537426e#

Step 2: In the SSH session, run `flask db upgrade`. If it succeeds, App Service is [connecting successfully to the database](#).

msdocs-python-postgres-234 | SSH

SSH CONNECTION ESTABLISHED

💡 Tip

In the SSH session, only changes to files in `/home` can persist beyond app restarts.

Changes outside of `/home` aren't persisted.

Having issues? Check the [Troubleshooting section](#).

6. Browse to the app

Step 1: In the App Service page:

1. From the left menu, select **Overview**.
2. Select the URL of your app.

The screenshot shows the Azure App Service Overview page. On the left, there's a sidebar with links like Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Deployment slots, and Deployment Center. The 'Overview' tab is selected and highlighted with a red box. At the top right, there are buttons for Browse, Stop, Swap, Restart, Delete, and more. Below the sidebar, under 'Essentials', are the following configuration details:

Resource group (move)	: msdocs-python-postgres-tutorial
Status	: Running
Location (move)	: East US
Subscription (move)	:
Subscription ID	:
Default domain	: msdocs-python-postgres-125.azurewebsites...
App Service Plan	: ASP-msdocspythonpostrestutorial-b709 (B1)
Operating System	: Linux
Health Check	: Error fetching health check data. Please try again later.

Step 2: Add a few restaurants to the list. Congratulations, you're running a web app in Azure App Service, with secure connectivity to Azure Database for PostgreSQL.

Restaurants

Name	Rating	Details
Fourth Coffee	★★	2.0 (2 reviews) Details
Contoso Restaurant	★★★★	4.0 (3 reviews) Details

[Add new restaurant](#)

7. Stream diagnostic logs

Azure App Service captures all console logs to help you diagnose issues with your application. The sample app includes `print()` statements to demonstrate this capability as shown below.

Python

```
@app.route('/', methods=['GET'])
def index():
    print('Request for index page received')
    restaurants = Restaurant.query.all()
    return render_template('index.html', restaurants=restaurants)
```

Step 1: In the App Service page:

1. From the left menu, select **Monitoring > App Service logs**.
2. Under **Application logging**, select **File System**.
3. In the top menu, select **Save**.

Microsoft Azure (Preview) Search resources, services, and docs (G+/)

msdocs-python-postgres-234 | App Service logs

App Service

Search Save Discard Send us your feedback

Metrics

Logs

Advisor recommendations

Health check

Diagnostic settings

App Service logs

Log stream

Log stream (preview)

Process explorer

Automation

Application logging Off File System

Quota (MB) * 35

Retention Period (Days)

Download logs

FTP/deployment username

FTP

Save

File System

35

1

Download logs

FTP/deployment username

FTP

Step 2: From the left menu, select **Log stream**. You see the logs for your app, including platform logs and logs from inside the container.

Microsoft Azure (Preview) Search resources, services, and docs (G+/)

msdocs-python-postgres-234 | Log stream

App Service

Search Reconnect Copy Pause Clear

Logs

Advisor recommendations

Health check

Diagnostic settings

App Service logs

Log stream

Log stream (preview)

Process explorer

Automation

Tasks (preview)

```
details page received
2022-10-05T18:49:34.756098791Z 169.254.130.1 - -
[05/Oct/2022:18:49:34 +0000] "GET /2/ HTTP/1.1" 200 6858
"https://msdocs-python-postgres-234.azurewebsites.net/2/"
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/106.0.5249.30 Safari/537.36"

2022-10-05T18:49:35.087646396Z Request for index page
received

2022-10-05T18:49:35.087686798Z 169.254.130.1 - -
[05/Oct/2022:18:49:35 +0000] "GET / HTTP/1.1" 200 5891
"https://msdocs-python-postgres-234.azurewebsites.net/2/"
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/106.0.5249.30 Safari/537.36"

2022-10-05T18:48:30.785Z INFO - Starting container for
```

Reconnect

Copy

Pause

Clear

details page received

2022-10-05T18:49:34.756098791Z 169.254.130.1 - -

[05/Oct/2022:18:49:34 +0000] "GET /2/ HTTP/1.1" 200 6858

"https://msdocs-python-postgres-234.azurewebsites.net/2/"

Mozilla/5.0 (Windows NT 10.0; Win64; x64)

AppleWebKit/537.36 (KHTML, like Gecko)

Chrome/106.0.5249.30 Safari/537.36"

2022-10-05T18:49:35.087646396Z Request for index page

received

2022-10-05T18:49:35.087686798Z 169.254.130.1 - -

[05/Oct/2022:18:49:35 +0000] "GET / HTTP/1.1" 200 5891

"https://msdocs-python-postgres-234.azurewebsites.net/2/"

Mozilla/5.0 (Windows NT 10.0; Win64; x64)

AppleWebKit/537.36 (KHTML, like Gecko)

Chrome/106.0.5249.30 Safari/537.36"

2022-10-05T18:48:30.785Z INFO - Starting container for

Learn more about logging in Python apps in the series on [setting up Azure Monitor for your Python application](#).

8. Clean up resources

When you're finished, you can delete all of the resources from your Azure subscription by deleting the resource group.

Step 1: In the search bar at the top of the Azure portal:

1. Enter the resource group name.
2. Select the resource group.

The screenshot shows the Microsoft Azure portal interface. At the top, there's a search bar with the text "msdocs-flask-postgres-tutorial". Below the search bar, the "Resource Groups" section is visible, showing a single item: "msdocs-flask-postgres-tutorial", which is highlighted with a red box. To the left, there's a sidebar with sections for "Azure services" (Create a resource, Key vaults) and "Resources" (Recent, Favorite). On the right, there are links for "Virtual networks" and "Documentation". At the bottom, there are buttons for "Continue searching in Microsoft Entra ID" and "Give feedback".

Step 2: In the resource group page, select **Delete resource group**.

[Create](#) [Manage view](#) [Delete resource group](#) ...

[View Cost](#) | [JSON View](#)

Essentials

Subscription ([move](#)) : [Antares-Demo](#)

Subscription ID :

Deployments : [7 Succeeded](#)

Location : West Europe

Tags ([edit](#)) : [Click here to add tags](#)

Resources Recommendations (1)

Filter for any field... [Add filter](#) [More \(2\)](#)

Showing 1 to 5 of 5 records. Show hidden types [?](#)

No grouping [List view](#)

Step 3:

1. Enter the resource group name to confirm your deletion.
2. Select **Delete**.
3. Confirm with **Delete** again.

Are you sure you want to delete "msdoc..." [X](#)

 Warning! Deleting the "msdocs-python-postgres-tutorial" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources in it permanently.

TYPE THE RESOURCE GROUP NAME:
 

AFFECTED RESOURCES
There are 6 resources in this resource group that will be deleted.

Name	Type	Location
ASP-msdocspythonpostgrestut...	App Service plan	West Europe
msdocs-python-postgres-234	App Service	West Europe
msdocs-python-postgres-234-s...	Azure Database for ...	West Europe

[Delete](#) [Cancel](#) 

Troubleshooting

Listed below are issues you might encounter while trying to work through this tutorial and steps to resolve them.

I can't connect to the SSH session

If you can't connect to the SSH session, then the app itself has failed to start. Check the [diagnostic logs](#) for details. For example, if you see an error like `KeyError:`

`'AZURE_POSTGRESQL_HOST'`, it might mean that the environment variable is missing (you might have removed the app setting).

I get an error when running database migrations

If you encounter any errors related to connecting to the database, check if the app settings (`AZURE_POSTGRESQL_USER`, `AZURE_POSTGRESQL_PASSWORD`, `AZURE_POSTGRESQL_HOST`, and `AZURE_POSTGRESQL_NAME`) were changed or deleted. Without that connection string, the `migrate` command can't communicate with the database.

Frequently asked questions

- [How much does this setup cost?](#)
- [How do I connect to the PostgreSQL server that's secured behind the virtual network with other tools?](#)
- [How does local app development work with GitHub Actions?](#)
- [How do I debug errors during the GitHub Actions deployment?](#)
- [I don't have permissions to create a user-assigned identity](#)
- [What can I do with GitHub Copilot in my codespace?](#)

How much does this setup cost?

Pricing for the created resources is as follows:

- The App Service plan is created in **Basic** tier and can be scaled up or down. See [App Service pricing ↗](#).
- The PostgreSQL flexible server is created in the lowest burstable tier **Standard_B1ms**, with the minimum storage size, which can be scaled up or down. See [Azure Database for PostgreSQL pricing ↗](#).
- The virtual network doesn't incur a charge unless you configure extra functionality, such as peering. See [Azure Virtual Network pricing ↗](#).

- The private DNS zone incurs a small charge. See [Azure DNS pricing](#).

How do I connect to the PostgreSQL server that's secured behind the virtual network with other tools?

- For basic access from a command-line tool, you can run `psql` from the app's SSH session.
- To connect from a desktop tool, your machine must be within the virtual network. For example, it could be an Azure VM that's connected to one of the subnets, or a machine in an on-premises network that has a [site-to-site VPN](#) connection with the Azure virtual network.
- You can also [integrate Azure Cloud Shell](#) with the virtual network.

How does local app development work with GitHub Actions?

Using the autogenerated workflow file from App Service as an example, each `git push` kicks off a new build and deployment run. From a local clone of the GitHub repository, you make the desired updates and push to GitHub. For example:

terminal

```
git add .
git commit -m "<some-message>"
git push origin main
```

How do I debug errors during the GitHub Actions deployment?

If a step fails in the autogenerated GitHub workflow file, try modifying the failed command to generate more verbose output. For example, you can get more output from the `python` command by adding the `-d` option. Commit and push your changes to trigger another deployment to App Service.

I don't have permissions to create a user-assigned identity

See [Set up GitHub Actions deployment from the Deployment Center](#).

What can I do with GitHub Copilot in my codespace?

You might have noticed that the GitHub Copilot chat view was already there for you when you created the codespace. For your convenience, we include the GitHub Copilot chat extension in

the container definition (see `.devcontainer/devcontainer.json`). However, you need a [GitHub Copilot account](#) (30-day free trial available).

A few tips for you when you talk to GitHub Copilot:

- In a single chat session, the questions and answers build on each other and you can adjust your questions to fine-tune the answer you get.
- By default, GitHub Copilot doesn't have access to any file in your repository. To ask questions about a file, open the file in the editor first.
- To let GitHub Copilot have access to all of the files in the repository when preparing its answers, begin your question with `@workspace`. For more information, see [Use the @workspace agent](#).
- In the chat session, GitHub Copilot can suggest changes and (with `@workspace`) even where to make the changes, but it's not allowed to make the changes for you. It's up to you to add the suggested changes and test it.

Next steps

Advance to the next tutorial to learn how to secure your app with a custom domain and certificate.

[Secure with custom domain and certificate](#)

Learn how App Service runs a Python app:

[Configure Python app](#)

Create and deploy a Flask Python web app to Azure with system-assigned managed identity

Article • 09/23/2024

In this tutorial, you deploy Python [Flask](#) code to create and deploy a web app running in Azure App Service. The web app uses its system-assigned [managed identity](#) (passwordless connections) with Azure role-based access control to access [Azure Storage](#) and [Azure Database for PostgreSQL - Flexible Server](#) resources. The code uses the `DefaultAzureCredential` class of the [Azure Identity client library for Python](#). The `DefaultAzureCredential` class automatically detects that a managed identity exists for the App Service and uses it to access other Azure resources.

You can configure passwordless connections to Azure services using Service Connector or you can configure them manually. This tutorial shows how to use Service Connector. For more information about passwordless connections, see [Passwordless connections for Azure services](#). For information about Service Connector, see the [Service Connector documentation](#).

This tutorial shows you how to create and deploy a Python web app using the Azure CLI. The commands in this tutorial are written to be run in a Bash shell. You can run the tutorial commands in any Bash environment with the CLI installed, such as your local environment or the [Azure Cloud Shell](#). With some modification -- for example, setting and using environment variables -- you can run these commands in other environments like Windows command shell. For examples of using a user-assigned managed identity, see [Create and deploy a Django web app to Azure with a user-assigned managed identity](#).

Get the sample app

A sample Python application using the Flask framework are available to help you follow along with this tutorial. Download or clone one of the sample applications to your local workstation.

1. Clone the sample in an Azure Cloud Shell session.

Console

```
git clone https://github.com/Azure-Samples/msdocs-flask-web-app-
```

```
managed-identity.git
```

2. Navigate to the application folder.

```
Console
```

```
cd msdocs-flask-web-app-managed-identity
```

Examine authentication code

The sample web app needs to authenticate to two different data stores:

- Azure blob storage server where it stores and retrieves photos submitted by reviewers.
- An Azure Database for PostgreSQL - Flexible Server database where it stores restaurants and reviews.

It uses `DefaultAzureCredential` to authenticate to both data stores. With `DefaultAzureCredential`, the app can be configured to run under the identity of different service principals, depending on the environment it's running in, without making changes to code. For example, in a local development environment, the app can run under the identity of the developer signed in to the Azure CLI, while in Azure, as in this tutorial, it can run under its system-assigned managed identity.

In either case, the security principal that the app runs under must have a role on each Azure resource the app uses that permits it to perform the actions on the resource that the app requires. In this tutorial, you use service connectors to automatically enable the system-assigned managed identity on your app in Azure and to assign that identity appropriate roles on your Azure storage account and Azure Database for PostgreSQL server.

After the system-assigned managed identity is enabled and is assigned appropriate roles on the data stores, you can use `DefaultAzureCredential` to authenticate with the required Azure resources.

The following code is used to create a blob storage client to upload photos in `app.py`. An instance of `DefaultAzureCredential` is supplied to the client, which it uses to acquire access tokens to perform operations against Azure storage.

```
Python
```

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient
```

```
azure_credential = DefaultAzureCredential()
blob_service_client = BlobServiceClient(
    account_url=account_url,
    credential=azure_credential)
```

An instance of `DefaultAzureCredential` is also used to get an access token for Azure Database for PostgreSQL in `./azureproject/get_conn.py`. In this case, the token is acquired directly by calling `get_token` on the credential instance and passing it the appropriate `scope` value. The token is then used in the place of the password in the PostgreSQL connection URI returned to the caller.

Python

```
azure_credential = DefaultAzureCredential()
token = azure_credential.get_token("https://osssrdbms-
aad.database.windows.net")
conn =
str(current_app.config.get('DATABASE_URI')).replace('PASSWORDORTOKEN',
token.token)
```

To learn more about authenticating your apps with Azure services, see [Authenticate Python apps to Azure services by using the Azure SDK for Python](#). To learn more about `DefaultAzureCredential`, including how to customize the credential chain it evaluates for your environment, see [DefaultAzureCredential overview](#).

Create an Azure PostgreSQL server

1. Set up the environment variables needed for the tutorial.

Bash

```
LOCATION="eastus"
RAND_ID=$RANDOM
RESOURCE_GROUP_NAME="msdocs-mi-web-app"
APP_SERVICE_NAME="msdocs-mi-web-$RAND_ID"
DB_SERVER_NAME="msdocs-mi-postgres-$RAND_ID"
ADMIN_USER="demoadmin"
ADMIN_PW="ChAnG33#ThsPssWD$RAND_ID"
```

Important

The `ADMIN_PW` must contain 8 to 128 characters from three of the following categories: English uppercase letters, English lowercase letters, numbers, and

nonalphanumeric characters. When creating usernames or passwords **do not** use the `$` character. Later you create environment variables with these values where the `$` character has special meaning within the Linux container used to run Python apps.

2. Create a resource group with the [az group create](#) command.

```
Azure CLI
```

```
az group create --location $LOCATION --name $RESOURCE_GROUP_NAME
```

3. Create a PostgreSQL server with the [az postgres flexible-server create](#) command.
(This and subsequent commands use the line continuation character for Bash Shell ('`\`). Change the line continuation character for your shell if needed.)

```
Azure CLI
```

```
az postgres flexible-server create \
--resource-group $RESOURCE_GROUP_NAME \
--name $DB_SERVER_NAME \
--location $LOCATION \
--admin-user $ADMIN_USER \
--admin-password $ADMIN_PW \
--sku-name Standard_D2ds_v4
```

The *sku-name* is the name of the pricing tier and compute configuration. For more information, see [Azure Database for PostgreSQL pricing](#). To list available SKUs, use `az postgres flexible-server list-skus --location $LOCATION`.

4. Create a database named `restaurant` using the [az postgres flexible-server execute](#) command.

```
Azure CLI
```

```
az postgres flexible-server execute \
--name $DB_SERVER_NAME \
--admin-user $ADMIN_USER \
--admin-password $ADMIN_PW \
--database-name postgres \
--querytext 'create database restaurant;'
```

Create an Azure App Service and deploy the code

1. Create an app service using the [az webapp up](#) command.

```
Azure CLI
```

```
az webapp up \
--resource-group $RESOURCE_GROUP_NAME \
--name $APP_SERVICE_NAME \
--runtime PYTHON:3.9 \
--sku B1
```

The *sku* defines the size (CPU, memory) and cost of the app service plan. The B1 (Basic) service plan incurs a small cost in your Azure subscription. For a full list of App Service plans, view the [App Service pricing](#) page.

2. Configure App Service to use the *start.sh* in the repo with the [az webapp config set](#) command.

```
Azure CLI
```

```
az webapp config set \
--resource-group $RESOURCE_GROUP_NAME \
--name $APP_SERVICE_NAME \
--startup-file "start.sh"
```

Create passwordless connectors to Azure resources

The Service Connector commands configure Azure Storage and Azure Database for PostgreSQL resources to use managed identity and Azure role-based access control. The commands create app settings in the App Service that connect your web app to these resources. The output from the commands lists the service connector actions taken to enable passwordless capability.

1. Add a PostgreSQL service connector with the [az webapp connection create postgres-flexible](#) command. The system-assigned managed identity is used to authenticate the web app to the target resource, PostgreSQL in this case.

```
Azure CLI
```

```
az webapp connection create postgres-flexible \
--resource-group $RESOURCE_GROUP_NAME \
--name $APP_SERVICE_NAME \
--target-resource-group $RESOURCE_GROUP_NAME \
--server $DB_SERVER_NAME \
--database restaurant \
```

```
--client-type python \
--system-identity
```

2. Add a storage service connector with the [az webapp connection create storage-blob](#) command.

This command also adds a storage account and adds the web app with role *Storage Blob Data Contributor* to the storage account.

Azure CLI

```
STORAGE_ACCOUNT_URL=$(az webapp connection create storage-blob \
    --new true \
    --resource-group $RESOURCE_GROUP_NAME \
    --name $APP_SERVICE_NAME \
    --target-resource-group $RESOURCE_GROUP_NAME \
    --client-type python \
    --system-identity \
    --query configurations[].value \
    --output tsv)
STORAGE_ACCOUNT_NAME=$(cut -d . -f1 <<< $(cut -d / -f3 <<<
$STORAGE_ACCOUNT_URL))
```

Create a container in the storage account

The sample Python app stores photos submitted by reviewers as blobs in a container in your storage account.

- When a user submits a photo with their review, the sample app writes the image to the container using its system-assigned managed identity for authentication and authorization. You configured this functionality in the last section.
- When a user views the reviews for a restaurant, the app returns a link to the photo in blob storage for each review that has one associated with it. For the browser to display the photo, it must be able to access it in your storage account. The blob data must be available for read publicly through anonymous (unauthenticated) access.

To enhance security, storage accounts are created with anonymous access to blob data disabled by default. In this section, you enable anonymous read access on your storage account and then create a container named *photos* that provides public (anonymous) access to its blobs.

1. Update the storage account to allow anonymous read access to blobs with the [az storage account update](#) command.

Azure CLI

```
az storage account update \
--name $STORAGE_ACCOUNT_NAME \
--resource-group $RESOURCE_GROUP_NAME \
--allow-blob-public-access true
```

Enabling anonymous access on the storage account doesn't affect access for individual blobs. You must explicitly enable public access to blobs at the container-level.

2. Create a container called *photos* in the storage account with the [az storage container create](#) command. Allow anonymous read (public) access to blobs in the newly created container.

Azure CLI

```
az storage container create \
--account-name $STORAGE_ACCOUNT_NAME \
--name photos \
--public-access blob \
--account-key $(az storage account keys list --account-name
$STORAGE_ACCOUNT_NAME \
--query [0].value --output tsv)
```

① Note

For brevity, this command uses the storage account key to authorize with the storage account. For most scenarios, Microsoft's recommended approach is to use Microsoft Entra ID and Azure (RBAC) roles. For a quick set of instructions, see [Quickstart: Create, download, and list blobs with Azure CLI](#). Note that several Azure roles permit you to create containers in a storage account, including "Owner", "Contributor", "Storage Blob Data Owner", and "Storage Blob Data Contributor".

To learn more about anonymous read access to blob data, see [Configure anonymous read access for containers and blobs](#).

Test the Python web app in Azure

The sample Python app uses the [azure.identity](#) package and its `DefaultAzureCredential` class. When the app is running in Azure, `DefaultAzureCredential` automatically detects if a managed identity exists for the App

Service and, if so, uses it to access other Azure resources (storage and PostgreSQL in this case). There's no need to provide storage keys, certificates, or credentials to the App Service to access these resources.

1. Browse to the deployed application at the URL

```
http://$APP_SERVICE_NAME.azurewebsites.net.
```

It can take a minute or two for the app to start. If you see a default app page that isn't the default sample app page, wait a minute and refresh the browser.

2. Test the functionality of the sample app by adding a restaurant and some reviews with photos for the restaurant.

The restaurant and review information is stored in Azure Database for PostgreSQL and the photos are stored in Azure Storage. Here's an example screenshot:

The screenshot shows a web application titled "Azure Restaurant Review". At the top, there is a logo and a link to "Azure Docs". Below the title, the name of the restaurant is "Contoso Café". Underneath the name, there are three details: "Street address: 1 Main Street", "Description: Nice coffee house.", and "Rating: ★★★★ 1 4.5 (2 reviews)". A "Reviews" section follows, featuring a green "Add new review" button. Below this, a table lists two reviews:

Date	User	Rating	Review	Photo
May 20, 2022, 10:12 a.m.	Davide Sagese	5	Friendly staff, good coffee.	
May 23, 2022, 5:24 p.m.	Francesca Lombo	4	Good breakfast choice.	

Clean up

In this tutorial, all the Azure resources were created in the same resource group. Removing the resource group removes with the `az group delete` command removes all resources in the resource group and is the fastest way to remove all Azure resources used for your app.

```
Azure CLI
```

```
az group delete --name $RESOURCE_GROUP_NAME
```

You can optionally add the `--no-wait` argument to allow the command to return before the operation is complete.

Next steps

- Create and deploy a Django web app to Azure with a user-assigned managed identity
 - Deploy a Python (Django or Flask) web app with PostgreSQL in Azure App Service
-

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create and deploy a Django web app to Azure with a user-assigned managed identity

Article • 09/23/2024

In this tutorial, you deploy a [Django](#) web app to Azure App Service. The web app uses a user-assigned [managed identity](#) (passwordless connections) with Azure role-based access control to access [Azure Storage](#) and [Azure Database for PostgreSQL - Flexible Server](#) resources. The code uses the `DefaultAzureCredential` class of the [Azure Identity client library](#) for Python. The `DefaultAzureCredential` class automatically detects that a managed identity exists for the App Service and uses it to access other Azure resources.

In this tutorial, you create a user-assigned managed identity and assign it to the App Service so that it can access the database and storage account resources. For an example of using a system-assigned managed identity, see [Create and deploy a Flask Python web app to Azure with system-assigned managed identity](#). User-assigned managed identities are recommended because they can be used by multiple resources, and their life cycles are decoupled from the resource life cycles with which they're associated. For more information about best practices for using managed identities, see [Managed identity best practice recommendations](#).

This tutorial shows you how to deploy the Python web app and create Azure resources using the [Azure CLI](#). The commands in this tutorial are written to be run in a Bash shell. You can run the tutorial commands in any Bash environment with the CLI installed, such as your local environment or the [Azure Cloud Shell](#). With some modification -- for example, setting and using environment variables -- you can run these commands in other environments like Windows command shell.

Get the sample app

Use the sample Django sample application to follow along with this tutorial. Download or clone the sample application to your development environment.

1. Clone the sample.

Console

```
git clone https://github.com/Azure-Samples/msdocs-django-web-app-managed-identity.git
```

2. Navigate to the application folder.

```
Console
```

```
cd msdocs-django-web-app-managed-identity
```

Examine authentication code

The sample web app needs to authenticate to two different data stores:

- Azure blob storage server where it stores and retrieves photos submitted by reviewers.
- An Azure Database for PostgreSQL - Flexible Server database where it stores restaurants and reviews.

It uses `DefaultAzureCredential` to authenticate to both data stores. With `DefaultAzureCredential`, the app can be configured to run under the identity of different service principals, depending on the environment it's running in, without making changes to code. For example, in a local development environment, the app can run under the identity of the developer signed in to the Azure CLI, while in Azure, as in this tutorial, it can run under a user-assigned managed identity.

In either case, the security principal that the app runs under must have a role on each Azure resource the app uses that permits it to perform the actions on the resource that the app requires. In this tutorial, you use Azure CLI commands to create a user-assigned managed identity and assign it to your app in Azure. You then manually assign that identity appropriate roles on your Azure storage account and Azure Database for PostgreSQL server. Finally, you set the `AZURE_CLIENT_ID` environment variable for your app in Azure to configure `DefaultAzureCredential` to use the managed identity.

After the user-assigned managed identity is configured on your app and its runtime environment, and is assigned appropriate roles on the data stores, you can use `DefaultAzureCredential` to authenticate with the required Azure resources.

The following code is used to create a blob storage client to upload photos in `./restaurant_review/views.py`. An instance of `DefaultAzureCredential` is supplied to the client, which it uses to acquire access tokens to perform operations against Azure storage.

```
Python
```

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient
```

```
azure_credential = DefaultAzureCredential()
blob_service_client = BlobServiceClient(
    account_url=account_url,
    credential=azure_credential)
```

An instance of `DefaultAzureCredential` is also used to get an access token for Azure Database for PostgreSQL in `./azureproject/get_conn.py`. In this case, the token is acquired directly by calling `get_token` on the credential instance and passing it the appropriate `scope` value. The token is then used to set the password in the PostgreSQL connection URI.

Python

```
azure_credential = DefaultAzureCredential()
token = azure_credential.get_token("https://osssrdbms-
aad.database.windows.net")
conf.settings.DATABASES['default']['PASSWORD'] = token.token
```

To learn more about authenticating your apps with Azure services, see [Authenticate Python apps to Azure services by using the Azure SDK for Python](#). To learn more about `DefaultAzureCredential`, including how to customize the credential chain it evaluates for your environment, see [DefaultAzureCredential overview](#).

Create an Azure PostgreSQL flexible server

1. Set up the environment variables needed for the tutorial.

Bash

```
LOCATION="eastus"
RAND_ID=$RANDOM
RESOURCE_GROUP_NAME="msdocs-mi-web-app"
APP_SERVICE_NAME="msdocs-mi-web-$RAND_ID"
DB_SERVER_NAME="msdocs-mi-postgres-$RAND_ID"
ADMIN_USER="demoadmin"
ADMIN_PW="ChAnG33#ThsPssWD$RAND_ID"
UA_NAME="UAManageredIdentityPythonTest$RAND_ID"
```

Important

The `ADMIN_PW` must contain 8 to 128 characters from three of the following categories: English uppercase letters, English lowercase letters, numbers, and nonalphanumeric characters. When creating usernames or passwords **do not**

use the `$` character. Later you create environment variables with these values where the `$` character has special meaning within the Linux container used to run Python apps.

2. Create a resource group with the [az group create](#) command.

```
Azure CLI
```

```
az group create --location $LOCATION --name $RESOURCE_GROUP_NAME
```

3. Create a PostgreSQL flexible server with the [az postgres flexible-server create](#) command. (This and subsequent commands use the line continuation character for Bash Shell ('`\`'). Change the line continuation character for other shells.)

```
Azure CLI
```

```
az postgres flexible-server create \
--resource-group $RESOURCE_GROUP_NAME \
--name $DB_SERVER_NAME \
--location $LOCATION \
--admin-user $ADMIN_USER \
--admin-password $ADMIN_PW \
--sku-name Standard_D2ds_v4 \
--active-directory-auth Enabled \
--public-access 0.0.0.0
```

The *sku-name* is the name of the pricing tier and compute configuration. For more information, see [Azure Database for PostgreSQL pricing](#). To list available SKUs, use `az postgres flexible-server list-skus --location $LOCATION`.

4. Add your Azure account as a Microsoft Entra admin for the server with the [az postgres flexible-server ad-admin create](#) command.

```
Azure CLI
```

```
ACCOUNT_EMAIL=$(az ad signed-in-user show --query userPrincipalName --output tsv)
ACCOUNT_ID=$(az ad signed-in-user show --query id --output tsv)
echo $ACCOUNT_EMAIL, $ACCOUNT_ID
az postgres flexible-server ad-admin create \
--resource-group $RESOURCE_GROUP_NAME \
--server-name $DB_SERVER_NAME \
--display-name $ACCOUNT_EMAIL \
--object-id $ACCOUNT_ID \
--type User
```

5. Configure a firewall rule on your server with the [az postgres flexible-server firewall-rule create](#) command. This rule allows your local environment access to connect to the server. (If you're using the Azure Cloud Shell, you can skip this step.)

Azure CLI

```
IP_ADDRESS=<your IP>
az postgres flexible-server firewall-rule create \
--resource-group $RESOURCE_GROUP_NAME \
--name $DB_SERVER_NAME \
--rule-name AllowMyIP \
--start-ip-address $IP_ADDRESS \
--end-ip-address $IP_ADDRESS
```

Use any tool or website that shows your IP address to substitute <your IP> in the command. For example, you can use the [What's My IP Address?](#) website.

6. Create a database named `restaurant` using the [az postgres flexible-server execute](#) command.

Azure CLI

```
az postgres flexible-server execute \
--name $DB_SERVER_NAME \
--admin-user $ADMIN_USER \
--admin-password $ADMIN_PW \
--database-name postgres \
--querytext 'create database restaurant;'
```

Create an Azure App Service and deploy the code

Run these commands in the root folder of the sample app to create an App Service and deploy the code to it.

1. Create an app service using the [az webapp up](#) command.

Azure CLI

```
az webapp up \
--resource-group $RESOURCE_GROUP_NAME \
--location $LOCATION \
--name $APP_SERVICE_NAME \
--runtime PYTHON:3.9 \
--sku B1
```

The *sku* defines the size (CPU, memory) and cost of the App Service plan. The B1 (Basic) service plan incurs a small cost in your Azure subscription. For a full list of App Service plans, view the [App Service pricing](#) page.

2. Configure App Service to use the *start.sh* in the sample repo with the [az webapp config set](#) command.

```
Azure CLI
```

```
az webapp config set \
--resource-group $RESOURCE_GROUP_NAME \
--name $APP_SERVICE_NAME \
--startup-file "start.sh"
```

Create a storage account and container

The sample app stores photos submitted by reviewers as blobs in Azure Storage.

- When a user submits a photo with their review, the sample app writes the image to the container using managed identity and `DefaultAzureCredential` to access the storage account.
- When a user views the reviews for a restaurant, the app returns a link to the photo in blob storage for each review that has one associated with it. For the browser to display the photo, it must be able to access it in your storage account. The blob data must be available for read publicly through anonymous (unauthenticated) access.

In this section, you create a storage account and container that permits public read access to blobs in the container. In later sections, you create a user-assigned managed identity and configure it to write blobs to the storage account.

1. Use the [az storage create](#) command to create a storage account.

```
Azure CLI
```

```
STORAGE_ACCOUNT_NAME="msdocsstorage$RAND_ID"
az storage account create \
--name $STORAGE_ACCOUNT_NAME \
--resource-group $RESOURCE_GROUP_NAME \
--location $LOCATION \
--sku Standard_LRS \
--allow-blob-public-access true
```

2. Create a container called *photos* in the storage account with the [az storage container create](#) command.

```
Azure CLI
```

```
az storage container create \
--account-name $STORAGE_ACCOUNT_NAME \
--name photos \
--public-access blob \
--auth-mode login
```

ⓘ Note

If the command fails, for example, if you get an error indicating that the request may be blocked by network rules of the storage account, enter the following command to make sure that your Azure user account is assigned an Azure role with permission to create a container.

```
Azure CLI
```

```
az role assignment create --role "Storage Blob Data Contributor" --
assignee $ACCOUNT_EMAIL --scope
"/subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP_NAM
E/providers/Microsoft.Storage/storageAccounts/$STORAGE_ACCOUNT_NAME
"
```

For more information, see [Quickstart: Create, download, and list blobs with Azure CLI](#). Note that several Azure roles permit you to create containers in a storage account, including "Owner", "Contributor", "Storage Blob Data Owner", and "Storage Blob Data Contributor".

Create a user-assigned managed identity

Create a user-assigned managed identity and assign it to the App Service. The managed identity is used to access the database and storage account.

1. Use the [az identity create](#) command to create a user-assigned managed identity and output the client ID to a variable for later use.

```
Azure CLI
```

```
UA_CLIENT_ID=$(az identity create --name $UA_NAME --resource-group
$RESOURCE_GROUP_NAME --query clientId --output tsv)
```

```
echo $UA_CLIENT_ID
```

2. Use the [az account show](#) command to get your subscription ID and output it to a variable that can be used to construct the resource ID of the managed identity.

Azure CLI

```
SUBSCRIPTION_ID=$(az account show --query id --output tsv)
RESOURCE_ID="/subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP_NAME/providers/Microsoft.ManagedIdentity/userAssignedIdentities/$UA_NAME"
echo $RESOURCE_ID
```

3. Assign the managed identity to the App Service with the [az webapp identity assign](#) command.

Azure CLI

```
export MSYS_NO_PATHCONV=1
az webapp identity assign \
--resource-group $RESOURCE_GROUP_NAME \
--name $APP_SERVICE_NAME \
--identities $RESOURCE_ID
```

4. Create App Service app settings that contain the client ID of the managed identity and other configuration info with the [az webapp config appsettings set](#) command.

Azure CLI

```
az webapp config appsettings set \
--resource-group $RESOURCE_GROUP_NAME \
--name $APP_SERVICE_NAME \
--settings AZURE_CLIENT_ID=$UA_CLIENT_ID \
STORAGE_ACCOUNT_NAME=$STORAGE_ACCOUNT_NAME \
STORAGE_CONTAINER_NAME=photos \
DBHOST=$DB_SERVER_NAME \
DBNAME=restaurant \
DBUSER=$UA_NAME
```

The sample app uses environment variables (app settings) to define connection information for the database and storage account but these variables don't include passwords. Instead, authentication is done passwordless with `DefaultAzureCredential`.

The sample app code uses the `DefaultAzureCredential` class constructor without passing the user-assigned managed identity client ID to the constructor. In this scenario, the

fallback is to check for the `AZURE_CLIENT_ID` environment variable, which you set as an app setting.

If the `AZURE_CLIENT_ID` environment variable doesn't exist, the system-assigned managed identity is used if it's configured. For more information, see [Introducing DefaultAzureCredential](#).

Create roles for the managed identity

In this section, you create role assignments for the managed identity to enable access to the storage account and database.

1. Create a role assignment for the managed identity to enable access to the storage account with the [az role assignment create](#) command.

Azure CLI

```
export MSYS_NO_PATHCONV=1
az role assignment create \
--assignee $UA_CLIENT_ID \
--role "Storage Blob Data Contributor" \
--scope
"/subscriptions/$SUBSCRIPTION_ID/resourcegroups/$RESOURCE_GROUP_NAME"
```

The command specifies the scope of the role assignment to the resource group. For more information, see [Understand role assignments](#).

2. Use the [az postgres flexible-server execute](#) command to connect to the Postgres database and run the same commands to assign roles to the managed identity.

Azure CLI

```
ACCOUNT_EMAIL_TOKEN=$(az account get-access-token --resource-type oss-
rdbms --output tsv --query accessToken)
az postgres flexible-server execute \
--name $DB_SERVER_NAME \
--admin-user $ACCOUNT_EMAIL \
--admin-password $ACCOUNT_EMAIL_TOKEN \
--database-name postgres \
--querytext "select * from pgaadauth_create_principal(''$UA_NAME'', 
false, false);select * from pgaadauth_list_principals(false);"
```

If you have trouble running the command, make sure you added your user account as Microsoft Entra admin for the PostgreSQL server and that you've allowed access

to your IP address in the firewall rules. For more information, see section [Create an Azure PostgreSQL flexible server](#).

Test the Python web app in Azure

The sample Python app uses the [azure.identity](#) package and its `DefaultAzureCredential` class. When the app is running in Azure, `DefaultAzureCredential` automatically detects if a managed identity exists for the App Service and, if so, uses it to access other Azure resources (storage and PostgreSQL in this case). There's no need to provide storage keys, certificates, or credentials to the App Service to access these resources.

1. Browse to the deployed application at the URL

`http://$APP_SERVICE_NAME.azurewebsites.net.`

It can take a minute or two for the app to start. If you see a default app page that isn't the default sample app page, wait a minute and refresh the browser.

2. Test the functionality of the sample app by adding a restaurant and some reviews with photos for the restaurant.

The restaurant and review information is stored in Azure Database for PostgreSQL and the photos are stored in Azure Storage. Here's an example screenshot:

The screenshot shows a web application interface for a restaurant named "Contoso Café". At the top, there is a header with the Azure logo and the text "Azure Restaurant Review" and "Azure Docs ▾". Below the header, the restaurant's name "Contoso Café" is displayed in a large, bold font. Underneath the name, there are three details: "Street address: 1 Main Street", "Description: Nice coffee house.", and "Rating: ★★★★ 4.5 (2 reviews)". A green button labeled "Add new review" is visible. The main content area is titled "Reviews" and lists two entries. The first entry is for "May 20, 2022, 10:12 a.m." by "Davide Sagese" with a rating of 5 and the review "Friendly staff, good coffee.". The second entry is for "May 23, 2022, 5:24 p.m." by "Francesca Lombo" with a rating of 4 and the review "Good breakfast choice.". Each review entry includes a small thumbnail image under the heading "Photo".

Date	User	Rating	Review	Photo
May 20, 2022, 10:12 a.m.	Davide Sagese	5	Friendly staff, good coffee.	
May 23, 2022, 5:24 p.m.	Francesca Lombo	4	Good breakfast choice.	

Clean up

In this tutorial, all the Azure resources were created in the same resource group. Removing the resource group removes with the [az group delete](#) command removes all

resources in the resource group and is the fastest way to remove all Azure resources used for your app.

Azure CLI

```
az group delete --name $RESOURCE_GROUP_NAME
```

You can optionally add the `--no-wait` argument to allow the command to return before the operation is complete.

Next steps

- [Create and deploy a Flask web app to Azure with a system-assigned managed identity](#)
- [Deploy a Python \(Django or Flask\) web app with PostgreSQL in Azure App Service](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Static website hosting in Azure Storage

Article • 04/14/2025

Azure Blob Storage is ideal for storing large amounts of unstructured data such as text, images, and videos. Because blob storage also provides static website hosting support, it's a great option in cases where you don't require a web server to render content. Although you're limited to hosting static content such as HTML, CSS, JavaScript, and image files, you can use serverless architectures including [Azure Functions](#) and other Platform as a service (PaaS) services.

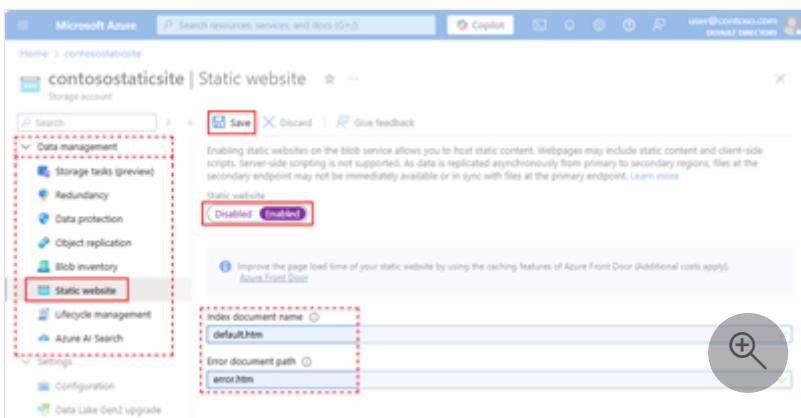
Static websites have some limitations. For example, If you want to configure headers, you'll have to use Azure Content Delivery Network (Azure CDN). There's no way to configure headers as part of the static website feature itself. Also, AuthN and AuthZ are not supported.

If these features are important for your scenario, consider using [Azure Static Web Apps](#). It's a great alternative to static websites and is also appropriate in cases where you don't require a web server to render content. You can configure headers and AuthN / AuthZ is fully supported. Azure Static Web Apps also provides a fully managed continuous integration and continuous delivery (CI/CD) workflow from GitHub source to global deployment.

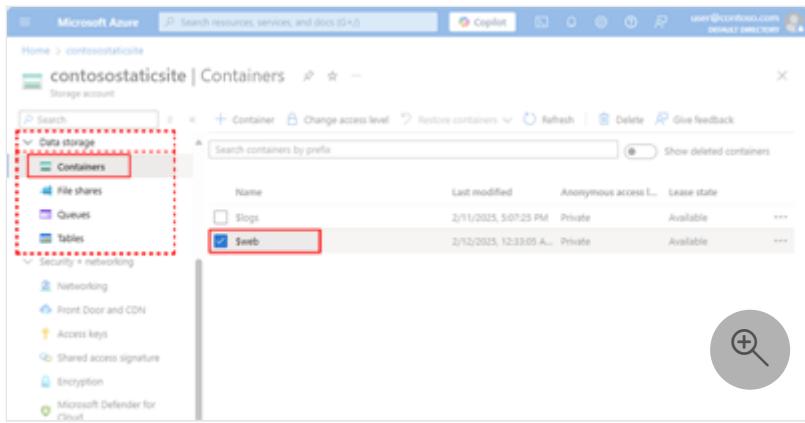
If you need a web server to render content, you can use [Azure App Service](#).

Setting up a static website

Static website hosting functionality is configured within a storage account and isn't enabled by default. To enable static website hosting, select a storage account. In the left navigation pane, select **Static website** from the **Data management** group, and then select **Enabled**. Provide a name for your *Index document name*. You can optionally provide a path to a custom 404 page. Finally, select **Save** to save your configuration changes.



A blob storage container named **\$web** is created for you within the storage account if it doesn't already exist. Add your website's files to the **\$web** container to make them accessible through the static website's primary endpoint.



Files in the **\$web** container are case-sensitive, served through anonymous access requests and are available only through read operations.

For step-by-step guidance, see [Host a static website in Azure Storage](#).

Uploading content

You can use any of these tools to upload content to the **\$web** container:

- ✓ [Azure CLI](#)
- ✓ [Azure PowerShell module](#)
- ✓ [AzCopy](#)
- ✓ [Azure Storage Explorer](#)
- ✓ [Azure portal](#)
- ✓ [Azure Pipelines](#)
- ✓ [Visual Studio Code extension](#) and [Channel 9 video demonstration](#)

Viewing content

Users can view site content from a browser by using the public URL of the website. You can find the URL by using the Azure portal, Azure CLI, or PowerShell. See [Find the website URL](#).

The index document that you specify when you enable static website hosting appears when users open the site and don't specify a specific file (For example:

`https://contosostaticsite.z22.web.core.windows.net`).

If the server returns a 404 error, and you haven't specified an error document when you enabled the website, then a default 404 page is returned to the user.

Note

[Cross-Origin Resource Sharing \(CORS\) support for Azure Storage](#) is not supported with static website.

Secondary endpoints

If you set up [redundancy in a secondary region](#), you can also access website content by using a secondary endpoint. Data is replicated to secondary regions asynchronously. Therefore, the files that are available at the secondary endpoint aren't always in sync with the files that are available on the primary endpoint.

Impact of setting the access level on the web container

You can modify the anonymous access level of the **\$web** container, but making this modification has no impact on the primary static website endpoint because these files are served through anonymous access requests. That means public (read-only) access to all files.

While the primary static website endpoint isn't affected, a change to the anonymous access level does impact the primary blob service endpoint.

For example, if you change the anonymous access level of the **\$web** container from **Private (no anonymous access)** to **Blob (anonymous read access for blobs only)**, then the level of anonymous access to the primary static website endpoint

`https://contosostaticsite.z22.web.core.windows.net/index.html` doesn't change.

However, anonymous access to the primary blob service endpoint

`https://contosostaticsite.blob.core.windows.net/$web/index.html` does change, enabling users to open that file by using either of these two endpoints.

Disabling anonymous access on a storage account by using the [anonymous access setting](#) of the storage account doesn't affect static websites that are hosted in that storage account. For more information, see [Remediate anonymous read access to blob data \(Azure Resource Manager deployments\)](#).

Mapping a custom domain to a static website URL

You can make your static website available via a custom domain.

It's easier to enable HTTP access for your custom domain, because Azure Storage natively supports it. To enable HTTPS, you'll have to use Azure CDN because Azure Storage doesn't yet

natively support HTTPS with custom domains. see [Map a custom domain to an Azure Blob Storage endpoint](#) for step-by-step guidance.

If the storage account is configured to [require secure transfer](#) over HTTPS, then users must use the HTTPS endpoint.

Tip

Consider hosting your domain on Azure. For more information, see [Host your domain in Azure DNS](#).

Adding HTTP headers

There's no way to configure headers as part of the static website feature. However, you can use Azure CDN to add headers and append (or overwrite) header values. See [Standard rules engine reference for Azure CDN](#).

If you want to use headers to control caching, see [Control Azure CDN caching behavior with caching rules](#).

Multi-region website hosting

If you plan to host a website in multiple geographies, we recommend that you use a [Content Delivery Network](#) for regional caching. Use [Azure Front Door](#) if you want to serve different content in each region. It also provides failover capabilities. [Azure Traffic Manager](#) isn't recommended if you plan to use a custom domain. Issues can arise because of how Azure Storage verifies custom domain names.

Permissions

The permission to be able to enable static website is Microsoft.Storage/storageAccounts/blobServices/write or shared key. Built in roles that provide this access include Storage Account Contributor.

Pricing

You can enable static website hosting free of charge. You're billed only for the blob storage that your site utilizes and operations costs. For more details on prices for Azure Blob Storage, check out the [Azure Blob Storage Pricing Page](#).

Metrics

You can enable metrics on static website pages. Once you've enabled metrics, traffic statistics on files in the **\$web** container are reported in the metrics dashboard.

To enable metrics on your static website pages, see [Enable metrics on static website pages](#).

Feature support

Support for this feature might be impacted by enabling Data Lake Storage Gen2, Network File System (NFS) 3.0 protocol, or the SSH File Transfer Protocol (SFTP). If you've enabled any of these capabilities, see [Blob Storage feature support in Azure Storage accounts](#) to assess support for this feature.

Frequently asked questions (FAQ)

Does the Azure Storage firewall work with a static website?

Yes. Storage account [network security rules](#), including IP-based and VNET firewalls, are supported for the static website endpoint, and may be used to protect your website.

Do static websites support Microsoft Entra ID?

No. A static website only supports anonymous read access for files in the **\$web** container.

How do I use a custom domain with a static website?

You can configure a [custom domain](#) with a static website by using [Azure Content Delivery Network \(Azure CDN\)](#). Azure CDN provides consistent low latencies to your website from anywhere in the world.

How do I use a custom Secure Sockets Layer (SSL) certificate with a static website?

You can configure a [custom SSL](#) certificate with a static website by using [Azure CDN](#). Azure CDN provides consistent low latencies to your website from anywhere in the world.

How do I add custom headers and rules with a static website?

You can configure the host header for a static website by using [Azure CDN rules engine](#). We'd be interested to hear your feedback [here ↗](#).

Why am I getting an HTTP 404 error from a static website?

A 404 error can happen if you refer to a file name by using an incorrect case. For example:

`Index.html` instead of `index.html`. File names and extensions in the url of a static website are case-sensitive even though they're served over HTTP. This can also happen if your Azure CDN endpoint isn't yet provisioned. Wait up to 90 minutes after you provision a new Azure CDN for the propagation to complete.

Why isn't the root directory of the website not redirecting to the default index page?

In the Azure portal, open the static website configuration page of your account and locate the name and extension that is set in the **Index document name** field. Ensure that this name is exactly the same as the name of the file located in the **\$web** container of the storage account. File names and extensions in the url of a static website are case-sensitive even though they're served over HTTP.

Why am I unable to access static websites in a storage account when a private endpoint is enabled for the blob in the storage account?

Enabling a private endpoint for blobs in a storage account restricts access to that storage account to only resources within the same virtual network. Consequently, this restriction prevents external access to the static website hosted in the storage account, making the static website content inaccessible. The private endpoint configuration limits access to all storage account resources, including the static website content, to resources within the same virtual network where the private endpoint is enabled. The resolution would be to create a private endpoint specifically for the web. The static website needs a dedicated private end point for the **\$web** domain.

Next steps

- [Host a static website in Azure Storage](#)
- [Map a custom domain to an Azure Blob Storage endpoint](#)
- [Azure Functions](#)
- [Azure App Service](#)
- [Build your first serverless web app](#)
- [Tutorial: Host your domain in Azure DNS](#)

Quickstart: Build your first static site with Azure Static Web Apps

Article • 04/02/2024

Azure Static Web Apps publishes a website by building an app from a code repository. In this quickstart, you deploy an application to Azure Static Web apps using the Visual Studio Code extension.

If you don't have an Azure subscription, [create a free trial account](#).

Prerequisites

- [GitHub](#) account
- [Azure](#) account
- [Visual Studio Code](#)
- [Azure Static Web Apps extension for Visual Studio Code](#)
- [Install Git](#)

Create a repository

This article uses a GitHub template repository to make it easy for you to get started. The template features a starter app to deploy to Azure Static Web Apps.

No Framework

1. Navigate to the following location to create a new repository:
 - a. <https://github.com/staticwebdev/vanilla-basic/generate>
2. Name your repository **my-first-static-web-app**

ⓘ Note

Azure Static Web Apps requires at least one HTML file to create a web app. The repository you create in this step includes a single *index.html* file.

Select **Create repository**.

[Create repository](#)

Clone the repository

With the repository created in your GitHub account, clone the project to your local machine using the following command.

Bash

```
git clone https://github.com/<YOUR_GITHUB_ACCOUNT_NAME>/my-first-static-web-app.git
```

Make sure to replace `<YOUR_GITHUB_ACCOUNT_NAME>` with your GitHub username.

Next, open Visual Studio Code and go to **File > Open Folder** to open the cloned repository in the editor.

Install Azure Static Web Apps extension

If you don't already have the [Azure Static Web Apps extension for Visual Studio Code](#) extension, you can install it in Visual Studio Code.

1. Select **View > Extensions**.
2. In the **Search Extensions in Marketplace**, type **Azure Static Web Apps**.
3. Select **Install** for **Azure Static Web Apps**.

Create a static web app

1. Inside Visual Studio Code, select the Azure logo in the Activity Bar to open the Azure extensions window.



Note

You are required to sign in to Azure and GitHub in Visual Studio Code to continue. If you are not already authenticated, the extension prompts you to sign in to both services during the creation process.

2. Select **F1** to open the Visual Studio Code command palette.

3. Enter **Create static web app** in the command box.

4. Select *Azure Static Web Apps: Create static web app....*

5. Select your Azure subscription.

6. Enter **my-first-static-web-app** for the application name.

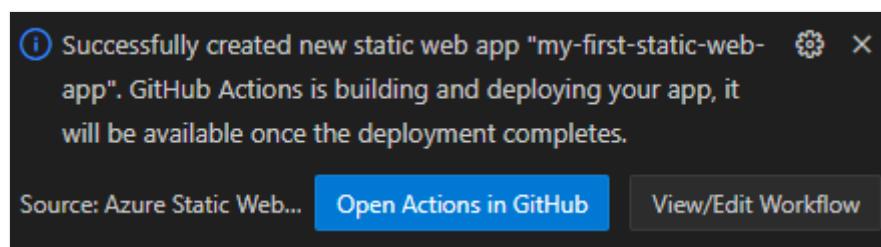
7. Select the region closest to you.

8. Enter the settings values that match your framework choice.

Setting	Value
Framework	Select Custom
Location of application code	Enter <code>/src</code>
Build location	Enter <code>/src</code>

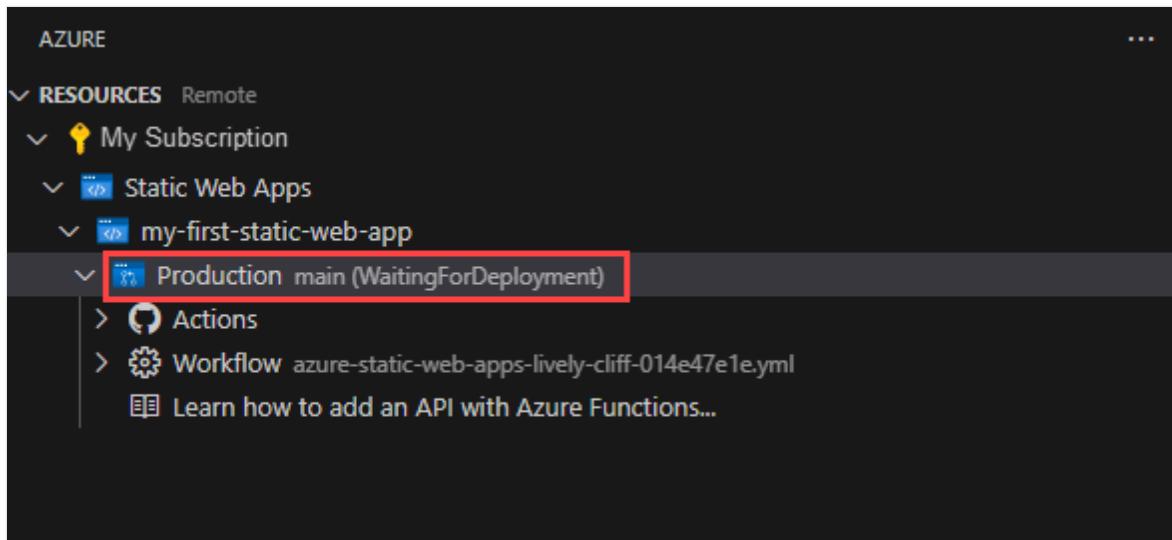
[Expand table](#)

9. Once the app is created, a confirmation notification is shown in Visual Studio Code.



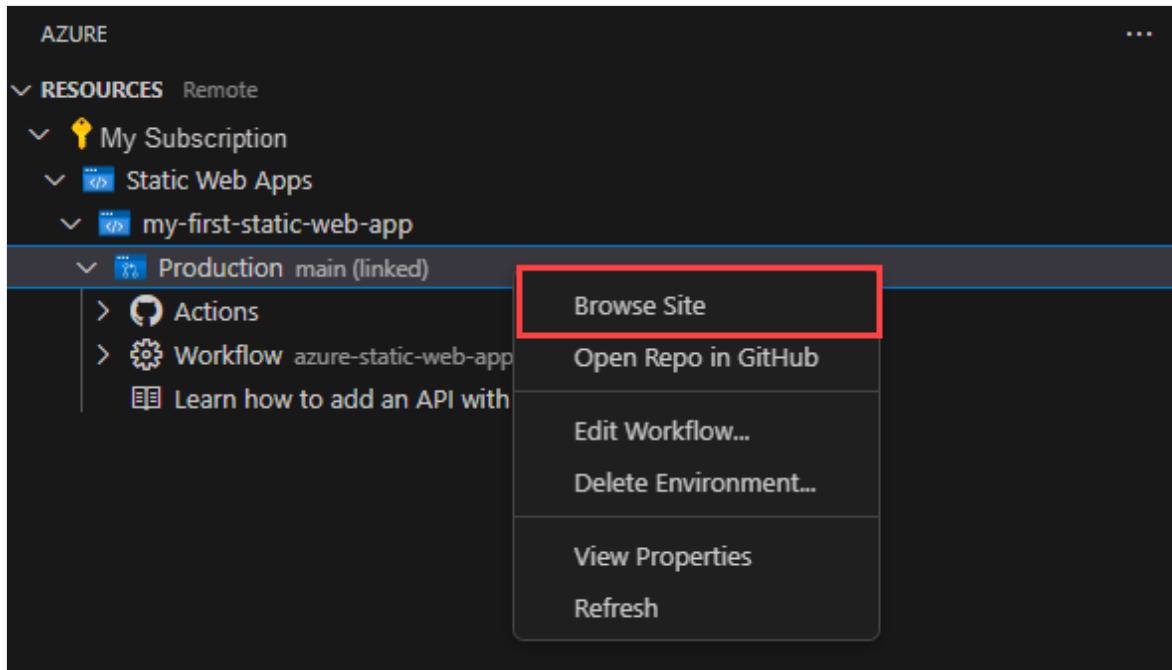
If GitHub presents you with a button labeled **Enable Actions on this repository**, select the button to allow the build action to run on your repository.

As the deployment is in progress, the Visual Studio Code extension reports the build status to you.



Once the deployment is complete, you can navigate directly to your website.

10. To view the website in the browser, right-click the project in the Static Web Apps extension, and select **Browse Site**.



Clean up resources

If you're not going to continue to use this application, you can delete the Azure Static Web Apps instance through the extension.

In the Visual Studio Code Azure window, return to the *Resources* section and under *Static Web Apps*, right-click *my-first-static-web-app* and select **Delete**.

Related content

- Video series: Deploy websites to the cloud with Azure Static Web Apps ↗

Next steps

[Add an API](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create a GitHub Codespaces dev environment with FastAPI and Postgres

06/19/2025

This article shows you how to run FastAPI and Postgres together in a [GitHub Codespaces](#) environment. Codespaces is a cloud-hosted development environment that allows you to create configurable and repeatable development environments.

You can open the sample repo in a [browser](#) or in an integrated development environment (IDE) like [Visual Studio Code](#) with the [GitHub Codespaces extension](#).

Alternatively, you can clone the sample repository locally. When you open the project in Visual Studio Code, you can use Dev Containers to run it using [Dev Containers](#). Dev Containers requires that [Docker Desktop](#) to be installed locally. If Docker isn't installed, you can run the project using GitHub Codespaces as the development environment.

When using GitHub Codespaces, keep in mind that you have a fixed number of core hours free per month. This tutorial requires less than one core hour to complete. For more information, see [About billing for GitHub Codespaces](#).

You can also use this setup as a starting point and modify the sample to run other Python web frameworks such as Django or Flask.

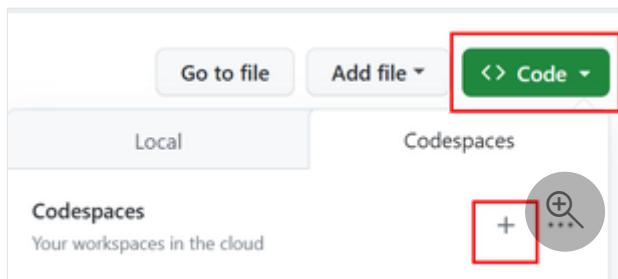
Start the dev environment in Codespaces

This tutorial introduces one of many possible ways to create and work with GitHub Codespaces.

1. Go to sample app repo <https://github.com/Azure-Samples/msdocs-fastapi-postgres-codespace>.

The sample repo has all the configuration needed to create an environment with a FastAPI app using a Postgres database. You can create a similar project following the steps in [Setting up a Python project for GitHub Codespaces](#).

2. Select **Code, Codespaces** tab, and + to create a new codespace.



3. When the container finishes building, confirm that you see **Codespaces** in the lower left corner of the browser, and see the sample repo.

The codespace key configuration files are *devcontainer.json*, *Dockerfile*, and *docker-compose.yml*. For more information, see [GitHub Codespaces overview](#).

💡 Tip

You can also run the codespace in Visual Studio Code. Select **Codespaces** in lower left corner of the browser or (`Ctrl + Shift + P` / `Ctrl + Command + P`) and type "Codespaces". Then select **Open in VS Code**. Also, if you stop the codespace and go back to the repo and open it again in GitHub Codespaces, you have the option to open it in VS Code or a browser.

4. Select the *.env.devcontainer* file and create a copy called *.env* with the same contents.

The *.env* contains environment variables that are used in the code to connect to the database.

5. If a terminal window isn't already open, open one by opening the Command Palette (`Ctrl + Shift + P` / `Ctrl + Command + P`), typing "Terminal: Create New Terminal", and selecting it to create a new terminal.
6. Select the **PORTS** tab in the terminal window to confirm that PostgreSQL is running on port 5432.
7. In the terminal window, run the FastAPI app.

```
Bash
uvicorn main:app --reload
```

8. Select the notification **Open in Browser**.

If you don't see or miss the notification, go to **PORTS** and find the **Local Address** for port 8000. Use the URL listed there.

9. Add `/docs` on the end of the preview URL to see the [Swagger UI](#), which allows you to test the API methods.

The API methods are generated from the OpenAPI interface that FastAPI creates from the code.

The screenshot shows the FastAPI Swagger UI interface. At the top, it displays "FastAPI 0.1.0 OAS3" and a link to "/openapi.json". Below this, the "default" endpoint is listed with the following methods:

- GET / Root**
- GET /restaurant/{id} Get Restaurant**
- POST /restaurant Set Restaurant**
- GET /all Get All Restaurants**

Each method row has a dropdown arrow icon on the right. A magnifying glass icon with a plus sign is located at the bottom right of the list.

10. On the Swagger page, run the POST method to add a restaurant.

a. Expand the **POST** method.

b. Select **Try it out**.

c. Fill in the request body.

The screenshot shows the "Try it out" section for the POST /restaurant method. The "JSON" tab is selected, displaying the following request body:

```
{  
  "name": "Restaurant 1",  
  "address": "Restaurant 1 address"  
}
```

d. Select **Execute** to commit the change

Connect to the database and view the data

1. Go back to the GitHub Codespace for the project, select the SQLTools extension, and then select **Local database** to connect.

The SQLTools extension should be installed when the container is created. If the SQLTools extension doesn't appear in the Activity Bar, close the codespace and reopen.

2. Expand the **Local database** node until you find the *restaurants* table, right select **Show Table Records**.

You should see the restaurant you added.

The screenshot shows the SQLTOOLS interface. On the left, there's a sidebar with icons for file, connections, search, and tools. Under 'CONNECTIONS', it lists 'Azure database' and 'Local database'. The 'Local database' section is expanded, showing 'postres database', 'Schemas' (with 'public' selected), and 'Tables'. The 'restaurants' table is selected, indicated by a red box around its icon. A context menu is open over the table, with the 'Show Table Records' option highlighted by another red box. The menu also includes 'Describe Table', 'Generate Insert Query', 'Add Name(s) To Cursor', and 'Copy Value(s)'. To the right of the menu, a table titled 'Local database: 1 records on 'restaurants' table' is displayed, showing one record: id 1, name Restaurant 1, and address Restaurant 1 address.

id	name	address
1	Restaurant 1	Restaurant 1 address

Clean up

To stop using the codespace, close the browser. (Or, close VS Code if you opened it that way.)

If you plan on using the codespace again, you can keep it. Only running codespaces incur CPU charges. A stopped codespace incurs only storage costs.

If you want to remove the codespace, go to <https://github.com/codespaces> to manage your codespaces.

Next steps

- [Develop a Python web app](#)
- [Develop a container app](#)
- [Learn to use the Azure libraries for Python](#)

Configure a custom startup file for Python apps on Azure App Service

Article • 04/22/2025

In this article, you learn when and how to configure a custom startup file for a Python web app hosted on Azure App Service. While a startup file isn't required for local development, Azure App Service runs your deployed web app within a Docker container that can utilize startup commands if provided.

You need a custom startup file in the following situations:

- **Custom Gunicorn Arguments:** You want to start the [Gunicorn](#) default web server with extra arguments beyond its defaults, which are `--bind=0.0.0.0 --timeout 600`.
- **Alternative Frameworks or Servers:** Your app is built with a framework other than Flask or Django, or you want to use a different web server besides Gunicorn.
- **Non-Standard Flask Application Structure:** You have a Flask app whose main code file is named something **other** than `app.py` or `application.py`*, or the `app` object is named something **other** than `app`.

In other words, a custom startup command is required unless your project has an `app.py` or `application.py` file in the root folder with a Flask app object named `app`.

For more information, see [Configure Python Apps - Container startup process](#).

Create a startup file

When you need a custom startup file, use the following steps:

1. Create a file in your project named `startup.txt`, `startup.sh`, or another name of your choice that contains your startup commands. See the later sections in this article for specifics on Django, Flask, and other frameworks.

A startup file can include multiple commands if needed.

2. Commit the file to your code repository so it can be deployed with the rest of the app.
3. In Visual Studio Code, select the Azure icon in the Activity Bar, expand **RESOURCES**, find and expand your subscription, expand **App Services**, and right-click the App Service, and select **Open in Portal**.

4. In the [Azure portal](#), on the **Configuration** page for the App Service, select **General settings**, enter the name of your startup file (like `startup.txt` or `startup.sh`) under **Stack settings > Startup Command**, then select **Save**.

 **Note**

Instead of using a startup command file, you can put the startup command itself directly in the **Startup Command** field on the Azure portal. Using a startup command file is recommended because it stores your configuration in your repository. This enables version control to track changes and simplifies redeployment to other Azure App Service instances.

5. Select **Continue** when prompted to restart the App Service.

If you access your Azure App Service site before deploying your application code, an "Application Error" appears because no code is available to process the request.

Django startup commands

By default, Azure App Service locates the folder containing your `wsgi.py` file and starts Gunicorn with the following command:

```
sh  
  
# <module> is the folder that contains wsgi.py. If you need to use a subfolder,  
# specify the parent of <module> using --chdir.  
gunicorn --bind=0.0.0.0 --timeout 600 <module>.wsgi
```

If you want to modify any Gunicorn arguments, such as increasing the timeout value to 1,200 seconds(`--timeout 1200`), create a custom startup command file. This allows you to override the default settings with your specific requirements. For more information, see [Container startup process - Django app](#).

Flask startup commands

By default, App Service on Linux assumes that your Flask application meets the following criteria:

- The WSGI callable is named `app`.
- The application code is contained in a file named `application.py` or `app.py`.
- The application file is located in the app's root folder.

If your project differs from this structure, then your custom startup command must identify the app object's location in the format *file:app_object*:

- **Different file name and/or app object name:** If the app's main code file is *hello.py* and the app object is named `myapp`, the startup command is as follows:

text

```
gunicorn --bind=0.0.0.0 --timeout 600 hello:myapp
```

- **Startup file is in a subfolder:** If the startup file is *myapp/website.py* and the app object is `app`, then use Gunicorn's `--chdir` argument to specify the folder and then name the startup file and app object as usual:

text

```
gunicorn --bind=0.0.0.0 --timeout 600 --chdir myapp website:app
```

- **Startup file is within a module:** In the [python-sample-vscode-flask-tutorial](#) code, the *webapp.py* startup file is contained within the folder *hello_app*, which is itself a module with an `_init_.py` file. The app object is named `app` and is defined in `_init_.py` and *webapp.py* uses a relative import.

Because of this arrangement, pointing Gunicorn to `webapp:app` produces the error, "Attempted relative import in non-package," and the app fails to start.

In this situation, create a shim file that imports the app object from the module, and then have Gunicorn launch the app using the shim. The [python-sample-vscode-flask-tutorial](#) code, for example, contains *startup.py* with the following contents:

Python

```
from hello_app.webapp import app
```

The startup command is then:

txt

```
gunicorn --bind=0.0.0.0 --workers=4 startup:app
```

For more information, see [Container startup process - Flask app](#).

Other frameworks and web servers

The App Service container that runs Python apps has Django and Flask installed by default, along with the Gunicorn web server.

To use a framework other than Django or Flask (such as [Falcon](#), [FastAPI](#), etc.), or to use a different web server:

- Include the framework and/or web server in your *requirements.txt* file.
- In your startup command, identify the WSGI callable as described in the [previous section for Flask](#).
- To launch a web server other than Gunicorn, use a `python -m` command instead of invoking the server directly. For example, the following command starts the [uvicorn](#) server, assuming that the WSGI callable is named `app` and is found in *application.py*:

```
sh  
python -m uvicorn application:app --host 0.0.0.0
```

You use `python -m` because web servers installed via *requirements.txt* aren't added to the Python global environment and therefore can't be invoked directly. The `python -m` command invokes the server from within the current virtual environment.

Deploy Python web apps to App Service by using GitHub Actions (Linux)

06/23/2025

This article describes how to use the continuous integration and continuous delivery (CI/CD) platform in GitHub Actions to deploy a Python web app to Azure App Service on Linux. Your GitHub Actions workflow automatically builds the code and deploys it to the App Service instance whenever there's a commit to the repository. You can add other automation in your GitHub Actions workflow, such as test scripts, security checks, and multistages deployment.

Create repository for app code

To complete the procedures in this article, you need a Python web app committed to a GitHub repository.

- **Existing app:** To use an existing Python web app, make sure the app is committed to a GitHub repository.
- **New app:** If you need a new Python web app, you can fork and clone the <https://github.com/Microsoft/python-sample-vscode-flask-tutorial> GitHub repository. The sample code supports the [Flask in Visual Studio Code](#) tutorial, and provides a functioning Python application.

ⓘ Note

If your app uses [Django](#) and a [SQLite](#) database, it won't work for these procedures. SQLite is not supported in most cloud-hosted environments due to its local file-based storage limitations. Consider switching to a cloud-compatible database such as PostgreSQL or Azure Cosmos DB. For more information, see [Review Django considerations](#) later in this article.

Create target App Service instance

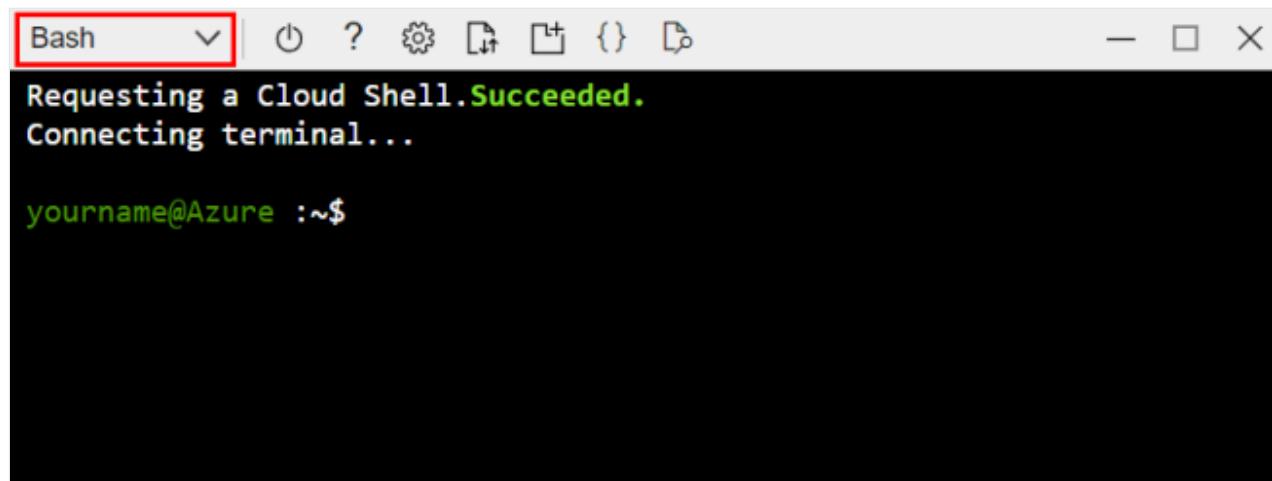
The quickest way to create an App Service instance is to use the [Azure command-line interface](#) (CLI) through the interactive [Azure Cloud Shell](#). The Cloud Shell includes [Git](#) and the Azure CLI. In the following procedure, you use the `az webapp up` command to both create the App Service instance and do the initial deployment of your app.

1. Sign in to the Azure portal at <https://portal.azure.com>.

2. Open the Azure CLI by selecting the Cloud Shell option on the portal toolbar:



3. In Cloud Shell, select the **Bash** option from the dropdown menu:



4. In Cloud Shell, clone your repository by using the [git clone](#) command.

💡 Tip

To paste commands or text into Cloud Shell, use the **Ctrl+Shift+V** keyboard shortcut, or right-click and select **Paste** from the context menu.

- For the Flask sample app, you can use the following command. Replace the `<github-user>` portion with the name of the GitHub account where you forked the repo:

```
Bash
```

```
git clone https://github.com/<github-user>/python-sample-vscode-flask-tutorial.git
```

- If your app is in a different repo, set up GitHub Actions for the particular repo. Replace the `<github-user>` portion with the name of the GitHub account where you forked the repo, and provide the actual repo name in the `<repo-name>` placeholder:

```
Bash
```

```
git clone https://github.com/<github-user>/<repo-name>.git
```

ⓘ Note

Cloud Shell is backed by an Azure Storage account in a resource group named `cloud-shell-storage-<your-region>`. That storage account contains an image of the Cloud Shell file system, which stores the cloned repository. There's a small cost for this storage. You can delete the storage account after you complete this article, along with other resources you create.

5. In Cloud Shell, change directory into the repository folder for your Python app, so the `az webapp up` command recognizes the app as Python. For the Flask sample app, you use the following command:

```
Bash
```

```
cd python-sample-vscode-flask-tutorial
```

6. In Cloud Shell, use the `az webapp up` command to create an App Service instance and do the initial deployment for your app:

```
Bash
```

```
az webapp up --name <app-service-name> --runtime "PYTHON:3.9"
```

- For the `<app-service-name>` placeholder, specify an App Service name that's unique in Azure. The name must be 3-60 characters long and can contain only letters, numbers, and hyphens. The name must start with a letter and end with a letter or number.
- For a list of available runtimes on your system, use the `az webapp list-runtimes` command.
- When you enter the runtime value in the command, use the `PYTHON:x.y` format, where `x.y` is the Python major and minor version.
- You can also specify the region location of the App Service instance by using the `--location` parameter. For a list of available locations, use the `az account list-locations --output table` command.

7. If your app has a custom startup script, use the `az webapp config` command to initiate the script.

- If your app doesn't have a custom startup script, continue to the next step.
- For the Flask sample app, you need to access the startup script in the `startup.txt` file by running the following command:

Bash

```
az webapp config set \
--resource-group <resource-group-name> \
--name <app-service-name> \
--startup-file startup.txt
```

Provide your resource group name and App Service instance name in the `<resource-group-name>` and `<app-service-name>` placeholders. To find the resource group name, check the output from the previous `az webapp up` command. The resource group name includes the Azure account name followed by the `_rg` suffix, as in `<azure-account-name>_rg_`.

8. To view the running app, open a browser and go to the deployment endpoint for your App Service instance. In the following URL, replace the `<app-service-name>` placeholder with your App Service instance name:

URL

```
http://<app-service-name>.azurewebsites.net
```

If you see a generic page, wait a few seconds for the App Service instance to start, and refresh the page.

- If you continue to see a generic page, confirm you deployed from the correct folder.
- For the Flask sample app, confirm you deployed from the `python-sample-vscode-flask-tutorial` folder. Also check that you set the startup command correctly.

Set up continuous deployment in App Service

In the next procedure, you set up continuous delivery (CD), which means a new code deployment occurs whenever a workflow triggers. The trigger in the article example is any change to the `main` branch of your repository, such as with a pull request (PR).

1. In Cloud Shell, confirm you're in the root directory for your system (`~`) and not in an app subfolder, such as `python-sample-vscode-flask-tutorial`.
2. Add GitHub Actions with the `az webapp deployment github-actions add` command. Replace any placeholders with your specific values:

Bash

```
az webapp deployment github-actions add \
--repo "<github-user>/<github-repo>" \
```

```
--resource-group <resource-group-name> \
--branch <branch-name> \
--name <app-service-name> \
--login-with-github
```

- The `--login-with-github` parameter uses an interactive method to retrieve a personal access token. Follow the prompts and complete the authentication.
- If the system encounters an existing workflow file with the same App Service instance name, follow the prompts to choose whether to overwrite the workflow. You can use the `--force` parameter with the command to automatically overwrite any conflicting workflows.

The `add` command completes the following tasks:

- Creates a new workflow file at the `.github/workflows/<workflow-name>.yml` path in your repo. The file name contains the name of your App Service instance.
- Fetches a publish profile with secrets for your App Service instance and adds it as a GitHub action secret. The name of the secret begins with `AZUREAPPSERVICE_PUBLISHPROFILE_`. This secret is referenced in the workflow file.

3. Get the details of a source control deployment configuration with the [az webapp deployment source show](#) command. Replace the placeholder parameters with your specific values:

Bash

```
az webapp deployment source show \
  --name <app-service-name> \
  --resource-group <resource-group-name>
```

4. In the command output, confirm the values for the `repoUrl` and `branch` properties. These values should match the values you specified with the `add` command.

Examine GitHub workflow and actions

A workflow definition is specified in a YAML (`.yml`) file in the `./github/workflows/` path in your repository. This YAML file contains the various steps and parameters that make up the workflow, an automated process associated with a GitHub repository. You can build, test, package, release, and deploy any project on GitHub with a workflow.

Each workflow is made up of one or more jobs, and each job is a set of steps. Each step is a shell script or an action. Each job has an **Action** section in the workflow file.

In terms of the workflow set up with your Python code for deployment to Azure App Service, the workflow has the following actions:

[] [Expand table](#)

Action	Description
checkout	Check out the repository on a <i>runner</i> , a GitHub Actions agent.
setup-python	Install Python on the runner.
appservice-build	Build the web app.
webapps-deploy	Deploy the web app by using a publish profile credential to authenticate in Azure. The credential is stored in a GitHub secret .

The workflow template used to create the workflow is [Azure/actions-workflow-samples](#).

The workflow is triggered on push events to the specified branch. The event and branch are defined at the beginning of the workflow file. For example, the following code snippet shows the workflow is triggered on push events to the *main* branch:

YAML

```
on:  
  push:  
    branches:  
      - main
```

OAuth authorized apps

When you set up continuous deployment, you authorize Azure App Service as an authorized OAuth App for your GitHub account. App Service uses the authorized access to create a GitHub action YAML file at the *.github/workflows/<workflow-name>.yml* path in your repo.

To see your authorized apps and revoke permissions under your GitHub accounts, go to **Settings > Integrations/Applications**:

The screenshot shows the GitHub 'Applications' settings page. On the left, there's a sidebar with links like 'Public profile', 'Account', 'Appearance', 'Accessibility', and 'Notifications'. Below that is a 'Access' section with 'Billing and plans', 'Emails', 'Password and authentication', and 'Sessions'. The main area is titled 'Applications' and has tabs for 'Installed GitHub Apps', 'Authorized GitHub Apps', and 'Authorized OAuth Apps'. The 'Authorized OAuth Apps' tab is selected and highlighted with a red box. It lists two applications: 'Azure CLI' (last used within the last week, owned by AzureAppServiceCLI) and 'Git Credential Manager' (last used within the last week, owned by git-ecosystem). A 'Revoke all' button is at the top right.

Workflow publish profile secret

In the `.github/workflows/<workflow-name>.yml` workflow file added to your repo, there's a placeholder for publish profile credentials required for the deploy job of the workflow. The publish profile information is stored encrypted in the repository.

To view the secret, go to **Settings > Security > Secret and variables > Actions:**

The screenshot shows the GitHub 'Repository secrets' settings page. On the left, there's a sidebar with 'Code security', 'Deploy keys', 'Secrets and variables' (which is expanded), 'Actions' (which is highlighted with a red box), 'Codespaces', and 'Dependabot'. The main area is titled 'Repository secrets' and shows a table with one row. The row contains the name 'AZUREAPPSERVICE_PUBLISHPROFILE_0000AAAA1111BBBB2222CCCC3333DDDD', the last updated time '24 minutes ago', and edit and delete icons. A green 'New repository secret' button is at the top right.

In this article, the GitHub action authenticates with a publish profile credential. There are other ways to authenticate, such as with a service principal or OpenID Connect. For more information, see [Deploy to App Service using GitHub Actions](#).

Run and test workflow

The last step is to test the workflow by making a change to the repo.

1. In a browser, go to your fork of the sample repository (or the repository you used), and select the branch you set as part of the trigger:

This branch is 3 commits ahead of [microsoft/python-sample-vscode-flask-tutorial:main](#).

[Contribute](#) [Sync fork](#)

Author	Commit Message	Date
Username	Create workflow using Azure CLI	386f4ae · 2 hours ago
.devcontainer	Adding dev container and updating co...	8 months ago
.github	Create workflow using Azure CLI	2 hours ago
.vscode	Adding dev container and updating co...	8 months ago
hello_app	Update home.html	5 hours ago
.dockerignore	Adding docker files	6 years ago

2. Make a small change to your Python web app.

For the Flask tutorial, here's a simple change:

- Go to the `/hello-app/templates/home.html` file of the trigger branch.
- Select **Edit** (pencil).
- In the Editor, locate the print `<p>` statement, and add the text "Redeployed!"

3. Commit the change directly to the branch you're working in.

- In the Editor, select **Commit changes** at the top right. The **Commit changes** window opens.
- In the **Commit changes** window, modify the commit message as desired, and select **Commit changes**.

The commit process triggers the GitHub Actions workflow.

You can also trigger the workflow manually:

- Go to the **Actions** tab of the repo set up for continuous deployment.
- Select the workflow in the list of workflows, and then select **Run workflow**.

Troubleshoot failed workflow

You can check the status of a workflow on the **Actions** tab for the app repo. When you examine the workflow file created in this article, you see two jobs: **build** and **deploy**. As a reminder, the workflow is based on the [Azure/actions-workflow-samples](#) template.

For a failed job, look at the output of job tasks for an indication of the failure.

Here are some common issues to investigate:

- If the app fails because of a missing dependency, then your `requirements.txt` file wasn't processed during deployment. This behavior happens if you created the web app directly on the portal rather than by using the `az webapp up` command as shown in this article.
- If you provisioned the app service through the portal, the build action `SCM_DO_BUILD_DURING_DEPLOYMENT` setting might not be set. This setting must be set to `true`. The `az webapp up` command sets the build action automatically.
- If you see an error message regarding "TLS handshake timeout," run the workflow manually by selecting **Trigger auto deployment** under the **Actions** tab of the app repo. You can determine if the timeout is a temporary issue.
- If you set up continuous deployment for the container app as shown in this article, the initial workflow file `.github/workflows/<workflow-name>.yml` is created automatically for you. If you modified the file, remove the modifications to see if they're causing the failure.

Run post-deployment script

A post-deployment script can complete several tasks, such as defining environment variables expected by the app code. You add the script as part of the app code and execute the script by using the startup command.

To avoid hard-coding variable values in your workflow YAML file, consider configuring the variables in GitHub and referring to the variable names in the script. You can create encrypted secrets for a repository or for an environment (account repository). For more information, see [Using secrets in GitHub Actions](#).

Review Django considerations

As noted earlier in this article, you can use GitHub Actions to deploy Django apps to Azure App Service on Linux, if you use a separate database. You can't use a SQLite database because App Service locks the `db.sqlite3` file, which prevents both reads and writes. This behavior doesn't affect an external database.

The [Configure Python app on App Service - Container startup process](#) article describes how App Service automatically looks for a `wsgi.py` file within your app code, which typically contains the app object. When you used the `webapp config set` command to set the startup command, you used the `--startup-file` parameter to specify the file that contains the app object. The `webapp config set` command isn't available in the `webapps-deploy` action. Instead, you can

use the `startup-command` parameter to specify the startup command. For example, the following code shows how to specify the startup command in the workflow file:

YAML

```
startup-command: startup.txt
```

When you use Django, you typically want to migrate the data models by using the `python manage.py migrate` command after you deploy the app code. You can run the `migrate` command in a post-deployment script.

Disconnect GitHub Actions

Disconnecting GitHub Actions from your App Service instance allows you to reconfigure the app deployment. You can choose what happens to your workflow file after you disconnect, and whether to save or delete the file.

Azure CLI

Disconnect GitHub Actions with the following Azure CLI [az webapp deployment github-actions remove](#) command. Replace any placeholders with your specific values:

Bash

```
az webapp deployment github-actions remove \
--repo "<github-username>/<github-repo>" \
--resource-group <resource-group-name> \
--branch <branch-name> \
--name <app-service-name> \
--login-with-github
```

Clean up resources

To avoid incurring charges on the Azure resources created in this article, delete the resource group that contains the App Service instance and the App Service Plan.

Azure CLI

Anywhere the Azure CLI is installed, including the Azure Cloud Shell, you can use the [az group delete](#) command to delete a resource group:

Bash

```
az group delete --name <resource-group-name>
```

Delete storage account

To delete the storage account that maintains the file system for Cloud Shell, which incurs a small monthly charge, delete the resource group that begins with *cloud-shell-storage-*. If you're the only user of the group, it's safe to delete the resource group. If there are other users, you can delete a storage account in the resource group.

Update GitHub account and repo

If you delete the Azure resource group, consider making the following modifications to the GitHub account and repo that was connected for continuous deployment:

- In the app repository, remove the *.github/workflows/<workflow-name>.yml* file.
- In the app repository settings, remove the *AZUREAPPSERVICE_PUBLISHPROFILE_* secret key created for the workflow.
- In the GitHub account settings, remove Azure App Service as an authorized Oauth App for your GitHub account.

Related content

- [Common repositories and workflows for GitHub Actions ↗](#)
- [Command reference - az webapp deployment github-actions](#)

Use Azure Pipelines to build and deploy a Python web app to Azure App Service

04/16/2025

Azure DevOps Services

Use Azure Pipelines for continuous integration and continuous delivery (CI/CD) to build and deploy a Python web app to Azure App Service on Linux. Your pipeline automatically builds and deploys your Python web app to App Service whenever there's a commit to the repository.

In this article, you learn how to:

- ✓ Create a web app in Azure App Service.
- ✓ Create a project in Azure DevOps.
- ✓ Connect your DevOps project to Azure.
- ✓ Create a Python-specific pipeline.
- ✓ Run the pipeline to build and deploy your app to your web app in App Service.

Prerequisites

[] Expand table

Product	Requirements
Azure DevOps	<ul style="list-style-type: none">- An Azure DevOps project.- An ability to run pipelines on Microsoft-hosted agents. You can either purchase a parallel job or you can request a free tier.- Basic knowledge of YAML and Azure Pipelines. For more information, see Create your first pipeline.- Permissions:<ul style="list-style-type: none">- To create a pipeline: you must be in the Contributors group and the group needs to have <i>Create build pipeline</i> permission set to Allow. Members of the Project Administrators group can manage pipelines.- To create service connections: You must have the <i>Administrator</i> or <i>Creator</i> role for service connections.
GitHub	<ul style="list-style-type: none">- A GitHub account.- A GitHub service connection to authorize Azure Pipelines.
Azure	An Azure subscription .

Create a repository for your app code

Fork the sample repository at <https://github.com/Microsoft/python-sample-vscode-flask-tutorial> to your GitHub account.

On your local host, clone your GitHub repository. Use the following command, replacing `<repository-url>` with the URL of your forked repository.

```
git  
git clone <repository-url>
```

Test your app locally

Build and run the app locally to make sure it works.

1. Change to the cloned repository folder.

```
Bash  
cd python-sample-vscode-flask-tutorial
```

2. Build and run the app

```
Linux  
Bash  
python -m venv .env  
source .env/bin/activate  
pip install --upgrade pip  
pip install -r ./requirements.txt  
export FLASK_APP=hello_app.webapp  
python3 -m flask run
```

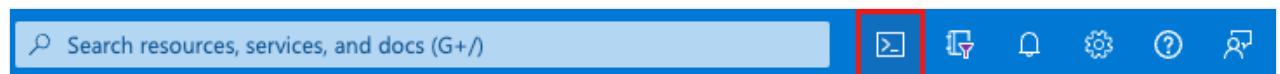
3. To view the app, open a browser window and go to `http://localhost:5000`. Verify that you see the title `Visual Studio Flask Tutorial`.

4. When you're finished, close the browser window and stop the Flask server with `ctrl + c`.

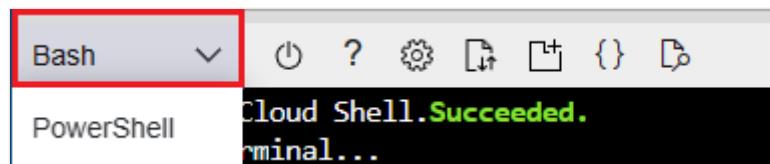
Open a Cloud Shell

1. Sign in to the Azure portal at <https://portal.azure.com>.

2. Open the Azure CLI by selecting the Cloud Shell button on the portal toolbar.



3. The Cloud Shell appears along the bottom of the browser. Select **Bash** from the dropdown menu.



4. To give you more space to work, select the maximize button.

Create an Azure App Service web app

Create your Azure App Service web app from the Cloud Shell in the Azure portal.

💡 Tip

To paste into the Cloud Shell, use `ctrl + Shift + V` or right-click and select **Paste** from the context menu.

1. Clone your repository with the following command, replacing `<repository-url>` with the URL of your forked repository.

```
Bash  
git clone <repository-url>
```

2. Change directory to the cloned repository folder, so the `az webapp up` command recognizes the app as a Python app.

```
Bash  
cd python-sample-vscode-flask-tutorial
```

3. Use the `az webapp up` command to both provision the App Service and do the first deployment of your app. Replace `<your-web-app-name>` with a name that is unique across Azure. Typically, you use a personal or company name along with an app identifier, such as `<your-name>-flaskpipelines`. The app URL becomes `<your-appservice>.azurewebsites.net`.

```
Azure CLI
```

```
az webapp up --name <your-web-app-name>
```

The JSON output of the `az webapp up` command shows:

JSON

```
{  
    "URL": <your-web-app-url>,  
    "appserviceplan": <your-app-service-plan-name>,  
    "location": <your-azure-location>,  
    "name": <your-web-app-name>,  
    "os": "Linux",  
    "resourcegroup": <your-resource-group>,  
    "runtime_version": "python|3.11",  
    "runtime_version_detected": "-",  
    "sku": <sku>,  
    "src_path": <repository-source-path>  
}
```

Note the `URL` and the `runtime_version` values. You use the `runtime_version` in the pipeline YAML file. The `URL` is the URL of your web app. You can use it to verify that the app is running.

ⓘ Note

The `az webapp up` command does the following actions:

- Create a default [resource group](#).
- Create a default [App Service plan](#).
- [Create an app](#) with the specified name.
- [Zip deploy](#) all files from the current working directory, with build automation enabled.
- Cache the parameters locally in the `.azure/config` file so that you don't need to specify them again when deploying later with `az webapp up` or other `az webapp` commands from the project folder. The cached values are used automatically by default.

You can override the default action with your own values using the command parameters. For more information, see [az webapp up](#).

4. The `python-sample-vscode-flask-tutorial` app has a `startup.txt` file that contains the specific startup command for the web app. Set the web app `startup-file` configuration property to `startup.txt`.

- a. From the `az webapp up` command output, copy the `resourcegroup` value.
- b. Enter the following command, using the resource group and your app name.

```
Azure CLI
```

```
az webapp config set --resource-group <your-resource-group> --name <your-web-app-name> --startup-file startup.txt
```

When the command completes, it shows JSON output that contains all of the configuration settings for your web app.

5. To see the running app, open a browser and go to the `URL` shown in the `az webapp up` command output. If you see a generic page, wait a few seconds for the App Service to start, then refresh the page. Verify that you see the title `Visual Studio Flask Tutorial`.

Create an Azure DevOps project

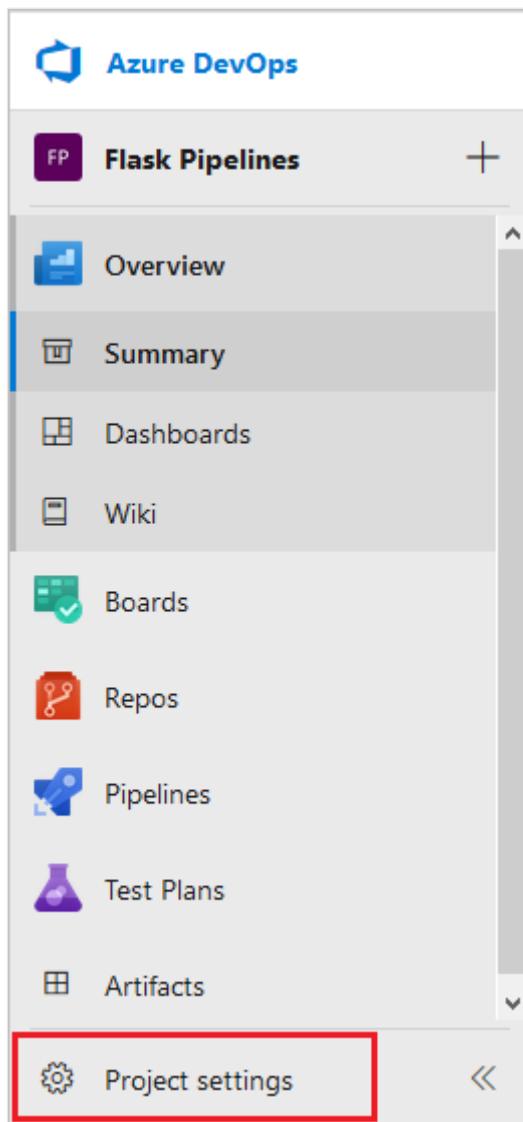
Create a new Azure DevOps project.

1. In a browser, go to dev.azure.com and sign in.
2. Select your organization.
3. Create a new project by selecting **New project** or **Create project** if creating the first project in the organization.
4. Enter a **Project name**.
5. Select the **Visibility** for your project.
6. Select **Create**.

Create a service connection

A service connection allows you to create a connection to provide authenticated access from Azure Pipelines to external and remote services. To deploy to your Azure App Service web app, create a service connection to the resource group containing the web app.

1. On project page, select **Project settings**.



2. Select **Service connections** in the **Pipelines** section of the menu.
3. Select **Create service connection**.
4. Select **Azure Resource Manager** and select **Next**.

New service connection

X

Choose a service or connection type

Search connection types

 Azure Artifacts upstream feed

 Azure Classic

 Azure Repos/Team Foundation Server

 Azure Resource Manager

 Azure Service Bus

 Bitbucket Cloud

 Cargo

[Learn more](#)

5. Select your authentication method and select **Next**.

6. In the **New Azure service connection** dialog, enter the information specific to the selected authentication method. For more information about authentication methods, see [Connect to Azure by using an Azure Resource Manager service connection](#).

For example, if you're using a **Workload Identity federation (automatic)** or **Service principal (automatic)** authentication method, enter the required information.

New Azure service connection

X

Azure Resource Manager using Workload Identity federation
with OpenID Connect (automatic)

Scope level

- Subscription
- Management Group
- Machine Learning Workspace

Subscription

<your Azure subscription>



Resource group

PythonWebApp



Details

Service connection name

Azure resource manager connection

Description (optional)

Security

- Grant access permission to all pipelines

[Learn more](#)

[Troubleshoot](#)

Back

Save

Expand table

Field	Description
Scope level	Select Subscription .
Subscription	Your Azure subscription name.
Resource group	The name of the resource group containing your web app.
Service connection name	A descriptive name for the connection.
Grant access permissions to all pipelines	Select this option to grant access to all pipelines.

7. Select Save.

The new connection appears in the **Service connections** list, and is ready for use in your Azure Pipeline.

Create a pipeline

Create a pipeline to build and deploy your Python web app to Azure App Service. To understand pipeline concepts, watch:

<https://learn-video.azurefd.net/vod/player?id=20e737aa-cadc-4603-9685-3816085087e9&locale=en-us&embedUrl=%2Fazure%2Fdevops%2Fpipelines%2Fecosystems%2Fpython-webapp>

1. On the left navigation menu, select **Pipelines**.

The screenshot shows the Azure DevOps interface for the 'Flask Pipelines' project. The left sidebar has a red box around the 'Pipelines' item. The main area features a cartoon illustration of a person working at a desk with a laptop, accompanied by a dog. Below the illustration, the text 'Welcome to the project!' is displayed, followed by a question 'What service would you like to start with?' and four buttons: Boards, Repos, Pipelines, and Test Plans. The 'Pipelines' button is highlighted.

2. Select **Create Pipeline**.

The screenshot shows the 'Create your first Pipeline' wizard. It features a cartoon illustration of a robot and a person working on a laptop. Below the illustration, the text 'Create your first Pipeline' is prominently displayed in large, bold, black font. A subtext reads: 'Automate your build and release processes using our wizard, and go from code to cloud-hosted within minutes.' At the bottom, there is a blue 'Create Pipeline' button with a red border, and a vertical ellipsis icon to its right.

3. In the **Where is your code** dialog, select **GitHub**. You might be prompted to sign into GitHub.

Connect Select Configure Review

New pipeline

Where is your code?

-  Azure Repos Git YAML
Free private Git repositories, pull requests, and code search
-  Bitbucket Cloud YAML
Hosted by Atlassian
-  GitHub YAML
Home to the world's largest community of developers
-  GitHub Enterprise Server YAML
The self-hosted version of GitHub Enterprise
-  Other Git
Any generic Git repository
-  Subversion
Centralized version control by Apache

4. On the **Select a repository** screen, select the forked sample repository.

✓ Connect **Select** Configure Review

New pipeline

Select a repository

Filter by keywords My repositories ▾ X

-  Microsoft/vscode-docs 2h ago
-  Microsoft/vscode-website private Yesterday
-  Mycode/python-sample-vscode-flask-tutorial fork Yesterday

5. You might be prompted to enter your GitHub password again as a confirmation.

6. If the Azure Pipelines extension isn't installed on GitHub, GitHub prompts you to install the **Azure Pipelines** extension.

The screenshot shows the GitHub marketplace page for the "Azure Pipelines" extension. At the top, there's a search bar and navigation links for "Pull requests", "Issues", "Marketplace", and "Explore". The extension card for "Azure Pipelines" is displayed, featuring its logo (a blue gear with a white play button), the name "Azure Pipelines", a status message "(Installed 7 days ago)", developer information ("Developed by AzurePipelines"), and a link to the Azure Pipelines website (<https://azure.microsoft.com/services/devops/pipelines/>). On the left, a sidebar menu lists "Personal settings", "Profile", "Account", "Emails", and "Notifications". Below the extension card, a section titled "Continuously build, test, and deploy to any platform and cloud" describes the service.

On this page, scroll down to the **Repository access** section, choose whether to install the extension on all repositories or only selected ones, and then select **Approve and install**.

The screenshot shows the "Repository access" configuration dialog. It contains two options: "All repositories" (selected) and "Only select repositories". The "All repositories" option is highlighted with a red border and includes a note: "This applies to all current *and* future repositories." Below these options is a "Select repositories" dropdown menu with a downward arrow. At the bottom of the dialog are two buttons: "Approve and install" (highlighted with a green border) and "Cancel".

7. In the **Configure your pipeline** dialog, select **Python to Linux Web App on Azure**.
8. Select your Azure subscription and select **Continue**.
9. If you're using your username and password to authenticate, a browser opens for you to sign in to your Microsoft account.
10. Select your web app name from the dropdown list and select **Validate and configure**.

Azure Pipelines creates a `azure-pipelines.yml` file and displays it in the YAML pipelines editor. The pipeline file defines your CI/CD pipeline as a series of *stages*, *Jobs*, and *steps*, where each step contains the details for different *tasks* and *scripts*. Take a look at the pipeline to see what it does. Make sure all the default inputs are appropriate for your code.

YAML pipeline file

The following explanation describes the YAML pipeline file. To learn about the pipeline YAML file schema, see [YAML schema reference](#).

The complete example pipeline YAML file is shown below:

```
yml

trigger:
- main

variables:
# Azure Resource Manager connection created during pipeline creation
azureServiceConnectionId: '<GUID>'

# Web app name
webAppName: '<your-webapp-name>'

# Agent VM image name
vmImageName: 'ubuntu-latest'

# Environment name
environmentName: '<your-webapp-name>'

# Project root folder. Point to the folder containing manage.py file.
projectRoot: $(System.DefaultWorkingDirectory)

pythonVersion: '3.11'

stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: BuildJob
    pool:
      vmImage: $(vmImageName)
    steps:
    - task: UsePythonVersion@0
      inputs:
        versionSpec: '$(pythonVersion)'
      displayName: 'Use Python $(pythonVersion)'

    - script: |
        python -m venv antenv
        source antenv/bin/activate
        python -m pip install --upgrade pip
        pip install setuptools
        pip install -r requirements.txt
      workingDirectory: $(projectRoot)
      displayName: "Install requirements"

    - task: ArchiveFiles@2
      displayName: 'Archive files'
      inputs:
        rootFolderOrFile: '$(projectRoot)'
        includeRootFolder: false
        archiveType: zip
        archiveFile: $(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip
```

```

replaceExistingArchive: true

- upload: $(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip
  displayName: 'Upload package'
  artifact: drop

- stage: Deploy
  displayName: 'Deploy Web App'
  dependsOn: Build
  condition: succeeded()
  jobs:
    - deployment: DeploymentJob
      pool:
        vmImage: $(vmImageName)
      environment: $(environmentName)
      strategy:
        runOnce:
          deploy:
            steps:
              - task: UsePythonVersion@0
                inputs:
                  versionSpec: '$(pythonVersion)'
                displayName: 'Use Python version'

              - task: AzureWebApp@1
                displayName: 'Deploy Azure Web App : $(webAppName)'
                inputs:
                  azureSubscription: $(azureServiceConnectionId)
                  appName: $(webAppName)
                  package: $(Pipeline.Workspace)/drop/$(Build.BuildId).zip

```

Variables

The `variables` section contains the following variables:

yml

```

variables:
# Azure Resource Manager connection created during pipeline creation
azureServiceConnectionId: '<GUID>'

# Web app name
webAppName: '<your-webapp-name>'

# Agent VM image name
vmImageName: 'ubuntu-latest'

# Environment name
environmentName: '<your-webapp-name>'

# Project root folder.

```

```

projectRoot: $(System.DefaultWorkingDirectory)

# Python version: 3.11. Change this to match the Python runtime version running on
# your web app.
pythonVersion: '3.11'

```

[\[+\] Expand table](#)

Variable	Description
azureServiceConnectionId	The ID or name of the Azure Resource Manager service connection.
webAppName	The name of the Azure App Service web app.
vmlImageName	The name of the operating system to use for the build agent.
environmentName	The name of the environment used in the deployment stage. The environment is automatically created when the stage job is run.
projectRoot	The root folder containing the app code.
pythonVersion	The version of Python to use on the build and deployment agents.

Build stage

The build stage contains a single job that runs on the operating system defined in the vmlImageName variable.

```

yml

- job: BuildJob
  pool:
    vmlImage: $(vmlImageName)

```

The job contains multiple steps:

1. The [UsePythonVersion](#) task selects the version of Python to use. The version is defined in the `pythonVersion` variable.

```

yml

- task: UsePythonVersion@0
  inputs:
    versionSpec: '$(pythonVersion)'
    displayName: 'Use Python $(pythonVersion)'

```

2. This step uses a script to create a virtual Python environment and install the app's dependencies contained in the `requirements.txt`. The `workingDirectory` parameter specifies the location of the app code.

```
yml  
  
- script: |  
    python -m venv antenv  
    source antenv/bin/activate  
    python -m pip install --upgrade pip  
    pip install setuptools  
    pip install -r ./requirements.txt  
  workingDirectory: $(projectRoot)  
  displayName: "Install requirements"
```

3. The [ArchiveFiles](#) task creates the `.zip` archive containing the web app. The `.zip` file is uploaded to the pipeline as the artifact named `drop`. The `.zip` file is used in the deployment stage to deploy the app to the web app.

```
yml  
  
- task: ArchiveFiles@2  
  displayName: 'Archive files'  
  inputs:  
    rootFolderOrFile: '$(projectRoot)'  
    includeRootFolder: false  
    archiveType: zip  
    archiveFile: $(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip  
    replaceExistingArchive: true  
  
- upload: $(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip  
  displayName: 'Upload package'  
  artifact: drop
```

[] [Expand table](#)

Parameter	Description
<code>rootFolderOrFile</code>	The location of the app code.
<code>includeRootFolder</code>	Indicates whether to include the root folder in the <code>.zip</code> file. Set this parameter to <code>false</code> otherwise, the contents of the <code>.zip</code> file are put in a folder named <code>s</code> and App Service on Linux container can't find the app code.
<code>archiveType</code>	The type of archive to create. Set to <code>zip</code> .
<code>archiveFile</code>	The location of the <code>.zip</code> file to create.

Parameter	Description
<code>replaceExistingArchive</code>	Indicates whether to replace an existing archive if the file already exists. Set to <code>true</code> .
<code>upload</code>	The location of the <code>.zip</code> file to upload.
<code>artifact</code>	The name of the artifact to create.

Deployment stage

The deployment stage is run if the build stage completes successfully. The following keywords define this behavior:

yml

```
dependsOn: Build
condition: succeeded()
```

The deployment stage contains a single deployment job configured with the following keywords:

yml

```
- deployment: DeploymentJob
  pool:
    vmImage: $(vmImageName)
    environment: $(environmentName)
```

 [Expand table](#)

Keyword	Description
<code>deployment</code>	Indicates that the job is a deployment job targeting an environment .
<code>pool</code>	Specifies deployment agent pool. The default agent pool if the name isn't specified. The <code>vmImage</code> keyword identifies the operating system for the agent's virtual machine image
<code>environment</code>	Specifies the environment to deploy to. The environment is automatically created in your project when the job is run.

The `strategy` keyword is used to define the deployment strategy. The `runOnce` keyword specifies that the deployment job runs once. The `deploy` keyword specifies the steps to run in the deployment job.

```
yml
```

```
strategy:  
  runOnce:  
    deploy:  
      steps:
```

The `steps` in the pipeline are:

1. Use the [UsePythonVersion](#) task to specify the version of Python to use on the agent. The version is defined in the `pythonVersion` variable.

```
yml
```

```
- task: UsePythonVersion@0  
  inputs:  
    versionSpec: '$(pythonVersion)'  
    displayName: 'Use Python version'
```

2. Deploy the web app using the [AzureWebApp@1](#). This task deploys the pipeline artifact `drop` to your web app.

```
yml
```

```
- task: AzureWebApp@1  
  displayName: 'Deploy Azure Web App : <your-web-app-name>'  
  inputs:  
    azureSubscription: $(azureServiceConnectionId)  
    appName: $(webAppName)  
    package: $(Pipeline.Workspace)/drop/$(Build.BuildId).zip
```

[+] [Expand table](#)

Parameter	Description
<code>azureSubscription</code>	The Azure Resource Manager service connection ID or name to use.
<code>appName</code>	The name of the web app.
<code>package</code>	The location of the .zip file to deploy.

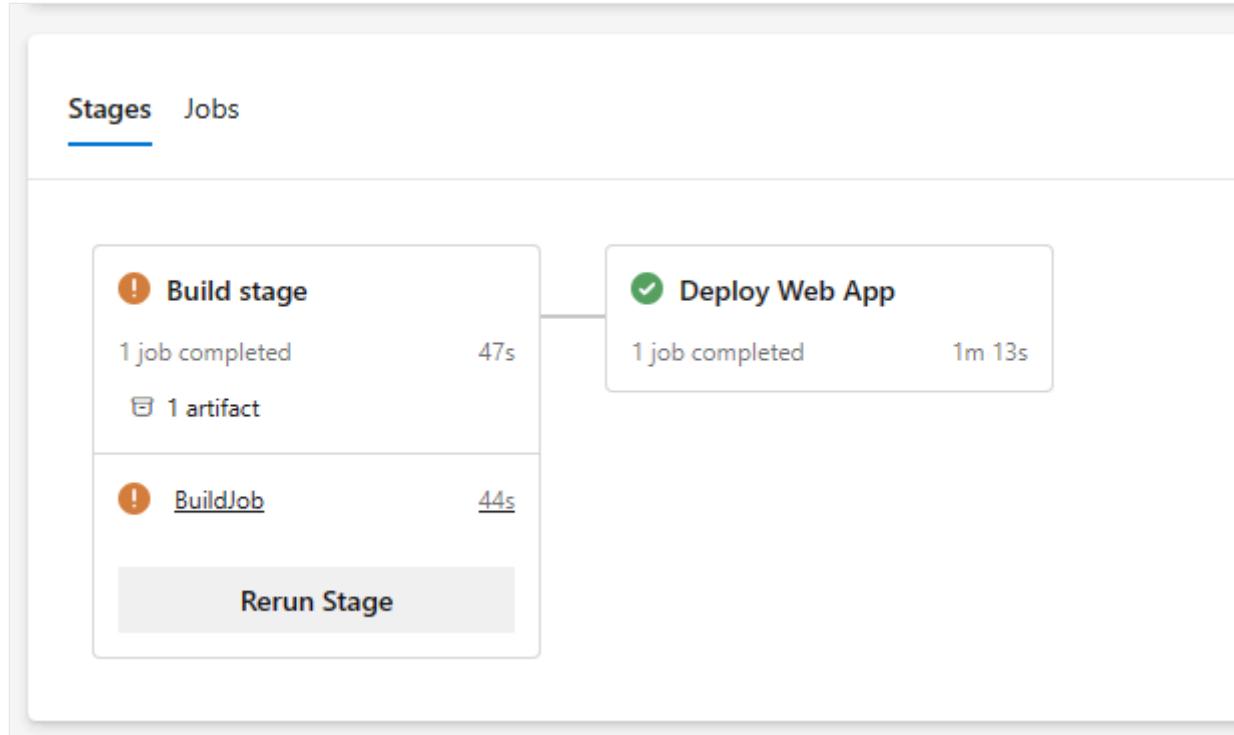
Also, because the *python-vscode-flask-tutorial* repository contains the same startup command in a file named *startup.txt*, you can specify that file by adding the parameter: `startUpCommand: 'startup.txt'`.

Run the pipeline

You're now ready to try it out!

1. In the editor, select **Save and run**.
2. In the **Save and run** dialog, add a commit message then select **Save and run**.

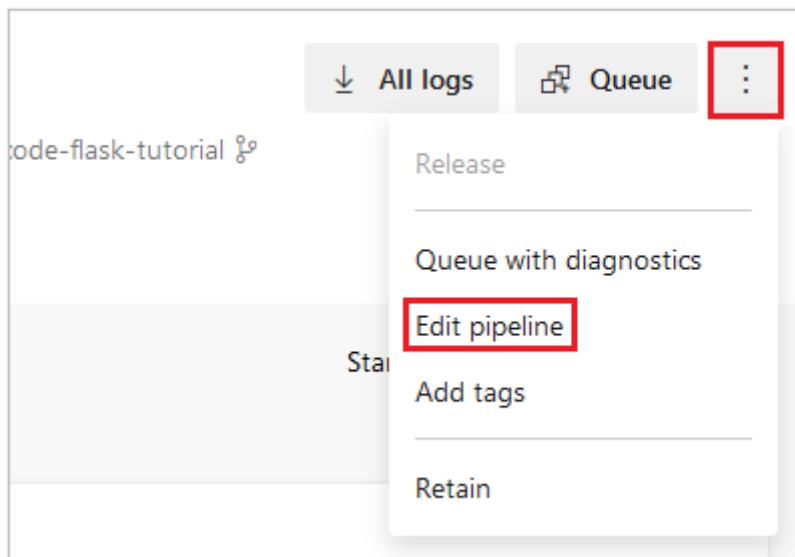
You can watch the pipeline as it runs by selecting the Stages or Jobs in the pipeline run summary.



There are green check marks next to each stage and job as it completes successfully. If errors occur, they're displayed in the summary or in the job steps.

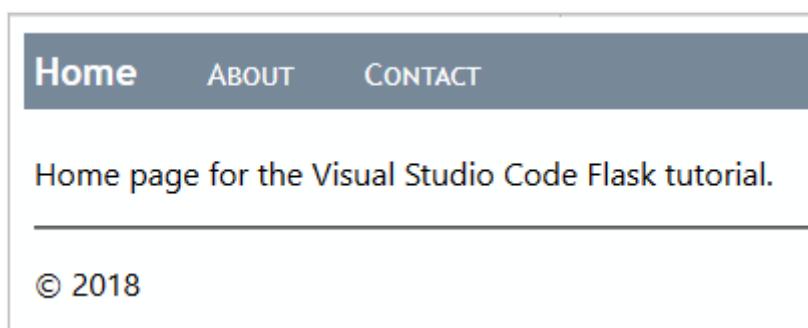
← Jobs in run #20240312.1		
username.python-sample-vscode-flask-tutorial		
Build stage		
✓	BuildJob	54s
✓	Initialize job	5s
✓	Checkout username/py...	2s
✓	Use Python 3.12	<1s
✓	Install requirements	10s
✓	Archive files	2s
✓	Upload package	6s
✓	Post-job: Checkout us...	<1s
✓	Finalize Job	<1s
Deploy Web App		
✓	DeploymentJob	59s
✓	Initialize job	4s
✓	Download Artifact	4s
✓	Use Python version	<1s
✓	Deploy Azure Web Ap...	36s
✓	Finalize Job	<1s

You can quickly return to the YAML editor by selecting the vertical dots at the upper right of the **Summary** page and selecting **Edit pipeline**:



3. From the deployment job, select the **Deploy Azure Web App** task to display its output. To visit the deployed site, hold down `ctrl` and select the URL after `App Service Application URL`.

If you're using the sample app, the app should appear as follows:



ⓘ Important

If your app fails because of a missing dependency, then your `requirements.txt` file was not processed during deployment. This behavior happens if you created the web app directly on the portal rather than using the `az webapp up` command as shown in this article.

The `az webapp up` command specifically sets the build action

`SCM_DO_BUILD_DURING_DEPLOYMENT` to `true`. If you provisioned the app service through the portal, this action is not automatically set.

The following steps set the action:

1. Open the [Azure portal](#), select your App Service, then select Configuration.
2. Under the Application Settings tab, select New Application Setting.
3. In the popup that appears, set Name to `SCM_DO_BUILD_DURING_DEPLOYMENT`, set Value to `true`, and select OK.

4. Select **Save** at the top of the **Configuration** page.
5. Run the pipeline again. Your dependencies should be installed during deployment.

Trigger a pipeline run

To trigger a pipeline run, commit a change to the repository. For example, you can add a new feature to the app, or update the app's dependencies.

1. Go to your GitHub repository.
2. Make a change to the code, such as changing the title of the app.
3. Commit the change to your repository.
4. Go to your pipeline and verify a new run is created.
5. When the run completes, verify the new build is deployed to your web app.
 - a. In the Azure portal, go to your web app.
 - b. Select **Deployment Center** and select the **Logs** tab.
 - c. Verify that the new deployment is listed.

Considerations for Django

You can use Azure Pipelines to deploy Django apps to Azure App Service on Linux if you're using a separate database. You can't use a SQLite database, because App Service locks the `db.sqlite3` file, preventing both reads and writes. This behavior doesn't affect an external database.

As described in [Configure Python app on App Service - Container startup process](#), App Service automatically looks for a `wsgi.py` file within your app code, which typically contains the app object. If you want to customize the startup command in any way, use the `startUpCommand` parameter in the `AzureWebApp@1` step of your YAML pipeline file, as described in the previous section.

When using Django, you typically want to migrate the data models using `manage.py migrate` after deploying the app code. You can add `startUpCommand` with a post-deployment script for this purpose. For example, here's the `startUpCommand` property in the `AzureWebApp@1` task.

yml

```
- task: AzureWebApp@1
  displayName: 'Deploy Azure Web App : $(webAppName)'
  inputs:
    azureSubscription: $(azureServiceConnectionId)
    appName: $(webAppName)
```

```
package: $(Pipeline.Workspace)/drop/$(Build.BuildId).zip
startUpCommand: 'python manage.py migrate'
```

Run tests on the build agent

As part of your build process, you might want to run tests on your app code. Tests run on the build agent, so you need to install your dependencies into a virtual environment on the build agent. After the tests run, delete the virtual environment before you create the `.zip` file for deployment. The following script elements illustrate this process. Place them before the `ArchiveFiles@2` task in the `azure-pipelines.yml` file. For more information, see [Run cross-platform scripts](#).

yml

```
# The | symbol is a continuation character, indicating a multi-line script.
# A single-line script can immediately follow "- script:".
- script: |
    python -m venv .env
    source .env/bin/activate
    pip install setuptools
    pip install -r requirements.txt

# The displayName shows in the pipeline UI when a build runs
displayName: 'Install dependencies on build agent'

- script: |
    # Put commands to run tests here
    displayName: 'Run tests'

- script: |
    echo Deleting .env
    deactivate
    rm -rf .env
displayName: 'Remove .env before zip'
```

You can also use a task like `PublishTestResults@2` to publish the test results to your pipeline. For more information, see [Build Python apps - Run tests](#).

Clean up resources

To avoid incurring charges on the Azure resources created in this tutorial:

- Delete the project that you created. Deleting the project deletes the pipeline and service connection.

- Delete the Azure resource group that contains the App Service and the App Service Plan. In the Azure portal, go to the resource group, select **Delete resource group**, and follow the prompts.
- Delete the storage account that maintains the file system for Cloud Shell. Close the Cloud Shell then go to the resource group that begins with **cloud-shell-storage-**, select **Delete resource group**, and follow the prompts.

Next steps

- [Customize Python apps in Azure Pipelines](#)
- [Configure Python app on App Service](#)

Quickstart: Sign in users in a sample web app

04/08/2025

Applies to:  Workforce tenants  External tenants ([learn more](#))

In this quickstart, you use a sample web app to show you how to sign in users and call Microsoft Graph API in your workforce tenant. The sample app uses the [Microsoft Authentication Library](#) to handle authentication.

Before you begin, use the **Choose a tenant type** selector at the top of this page to select tenant type. Microsoft Entra ID provides two tenant configurations, [workforce](#) and [external](#). A workforce tenant configuration is for your employees, internal apps, and other organizational resources. An external tenant is for your customer-facing apps.

Prerequisites

- An Azure account with an active subscription. If you don't already have one, [Create an account for free](#).
- This Azure account must have permissions to manage applications. Any of the following Microsoft Entra roles include the required permissions:
 - Application Administrator
 - Application Developer
- A workforce tenant. You can use your Default Directory or [set up a new tenant](#).
- [Visual Studio Code](#) or another code editor.

Node

- Register a new app in the [Microsoft Entra admin center](#), configured for *Accounts in this organizational directory only*. Refer to [Register an application](#) for more details. Record the following values from the application **Overview** page for later use:
 - Application (client) ID
 - Directory (tenant) ID
- Add the following redirect URIs using the **Web** platform configuration. Refer to [How to add a redirect URI in your application](#) for more details.
 - **Redirect URI:** `http://localhost:3000/auth/redirect`
 - **Front-channel logout URL:** `https://localhost:5001/signout-callback-oidc`
- Add a client secret to your app registration. **Do not** use client secrets in production apps. Use certificates or federated credentials instead. For more information, see [add credentials to your application](#).

- [Node.js](#)

Clone or download sample web application

To obtain the sample application, you can either clone it from GitHub or download it as a `.zip` file.

Node

- [Download the `.zip` file](#), then extract it to a file path where the length of the name is fewer than 260 characters or clone the repository:
- To clone the sample, open a command prompt and navigate to where you wish to create the project, and enter the following command:

Console

```
git clone https://github.com/Azure-Samples/ms-identity-node.git
```

Configure the sample web app

For you to sign in users with the sample app, you need to update it with your app and tenant details:

Node

In the *ms-identity-node* folder, open the *App/.env* file, then replace the following placeholders:

[Expand table](#)

Variable	Description	Example(s)
<code>Enter_the_Cloud_Instance_Id_Here</code>	The Azure cloud instance in which your application is registered	https://login.microsoftonline.com/ (include the trailing forward-slash)

Variable	Description	Example(s)
Enter_the_Tenant_Info_here	Tenant ID or Primary domain	contoso.microsoft.com or aaaabbbb-0000-cccc-1111-dddd2222eeee
Enter_the_Application_Id_Here	Client ID of the application you registered	00001111-aaaa-2222-bbbb-3333cccc4444
Enter_the_Client_Secret_Here	Client secret of the application you registered	A1b-C2d_E3f.H4i,J5k?L6m!N7o-P8q_R9s.T0u
Enter_the_Graph_Endpoint_Here	The Microsoft Graph API cloud instance that your app calls	https://graph.microsoft.com/ (include the trailing forward-slash)
Enter_the_Express_Session_Secret_Here	A random string of characters used to sign the Express session cookie	A1b-C2d_E3f.H4...

After you make changes, your file should look similar to the following snippet:

```
env

CLOUD_INSTANCE=https://login.microsoftonline.com/
TENANT_ID=aaaabbbb-0000-cccc-1111-dddd2222eeee
CLIENT_ID=00001111-aaaa-2222-bbbb-3333cccc4444
CLIENT_SECRET=A1b-C2d_E3f.H4...

REDIRECT_URI=http://localhost:3000/auth/redirect
POST_LOGOUT_REDIRECT_URI=http://localhost:3000

GRAPH_API_ENDPOINT=https://graph.microsoft.com/

EXPRESS_SESSION_SECRET=6DP6v09eLiW7f1E65B8k
```

Run and test sample web app

You've configured your sample app. You can proceed to run and test it.

Node

1. To start the server, run the following commands from within the project directory:

```
Console
```

```
cd App  
npm install  
npm start
```

2. Go to <http://localhost:3000/>.

3. Select **Sign in** to start the sign-in process.

The first time you sign in, you're prompted to provide your consent to allow the application to sign you in and access your profile. After you're signed in successfully, you'll be redirected back to the application home page.

How the app works

The sample hosts a web server on localhost, port 3000. When a web browser accesses this address, the app renders the home page. Once the user selects **Sign in**, the app redirects the browser to Microsoft Entra sign-in screen, via the URL generated by the MSAL Node library. After user consents, the browser redirects the user back to the application home page, along with an ID and access token.

Related content

Node

- Learn how to build a Node.js web app that signs in users and calls Microsoft Graph API in [Tutorial: Sign in users and acquire a token for Microsoft Graph in a Node.js & Express web app](#).

Quickstart: Azure Key Vault secret client library for Python

Article • 04/14/2025

Get started with the Azure Key Vault secret client library for Python. Follow these steps to install the package and try out example code for basic tasks. By using Key Vault to store secrets, you avoid storing secrets in your code, which increases the security of your app.

[API reference documentation](#) | [Library source code](#) | [Package \(Python Package Index\)](#)

Prerequisites

- An Azure subscription - [create one for free](#).
- [Python 3.7+](#).
- [Azure CLI](#) or [Azure PowerShell](#).

This quickstart assumes you're running [Azure CLI](#) or [Azure PowerShell](#) in a Linux terminal window.

Set up your local environment

This quickstart is using Azure Identity library with Azure CLI or Azure PowerShell to authenticate user to Azure Services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls, for more information, see [Authenticate the client with Azure Identity client library](#).

Sign in to Azure

Azure CLI

1. Run the `az login` command.

Azure CLI

`az login`

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

Install the packages

1. In a terminal or command prompt, create a suitable project folder, and then create and activate a Python virtual environment as described on [Use Python virtual environments](#).
2. Install the Microsoft Entra identity library:

terminal

```
pip install azure-identity
```

3. Install the Key Vault secrets library:

terminal

```
pip install azure-keyvault-secrets
```

Create a resource group and key vault

Azure CLI

1. Use the `az group create` command to create a resource group:

Azure CLI

```
az group create --name myResourceGroup --location eastus
```

You can change "eastus" to a location nearer to you, if you prefer.

2. Use `az keyvault create` to create the key vault:

Azure CLI

```
az keyvault create --name <your-unique-keyvault-name> --resource-group  
myResourceGroup
```

Replace `<your-unique-keyvault-name>` with a name that's unique across all of Azure.

You typically use your personal or company name along with other numbers and

identifiers.

Set the KEY_VAULT_NAME environmental variable

Our script will use the value assigned to the `KEY_VAULT_NAME` environment variable as the name of the key vault. You must therefore set this value using the following command:

Console

```
export KEY_VAULT_NAME=<your-unique-keyvault-name>
```

Grant access to your key vault

To gain permissions to your key vault through [Role-Based Access Control \(RBAC\)](#), assign a role to your "User Principal Name" (UPN) using the Azure CLI command [az role assignment create](#).

Azure CLI

```
az role assignment create --role "Key Vault Secrets Officer" --assignee "<upn>" --scope "/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>/providers/Microsoft.KeyVault/vaults/<your-unique-keyvault-name>"
```

Replace `<upn>`, `<subscription-id>`, `<resource-group-name>` and `<your-unique-keyvault-name>` with your actual values. Your UPN will typically be in the format of an email address (e.g., `username@domain.com`).

Create the sample code

The Azure Key Vault secret client library for Python allows you to manage secrets. The following code sample demonstrates how to create a client, set a secret, retrieve a secret, and delete a secret.

Create a file named `kv_secrets.py` that contains this code.

Python

```
import os
from azure.keyvault.secrets import SecretClient
from azure.identity import DefaultAzureCredential

keyVaultName = os.environ["KEY_VAULT_NAME"]
KVUri = f"https://{{keyVaultName}}.vault.azure.net"
```

```
credential = DefaultAzureCredential()
client = SecretClient(vault_url=KVUri, credential=credential)

secretName = input("Input a name for your secret > ")
secretValue = input("Input a value for your secret > ")

print(f"Creating a secret in {keyVaultName} called '{secretName}' with the value '{secretValue}' ...")

client.set_secret(secretName, secretValue)

print(" done.")

print(f"Retrieving your secret from {keyVaultName}.") 

retrieved_secret = client.get_secret(secretName)

print(f"Your secret is '{retrieved_secret.value}'.")
print(f"Deleting your secret from {keyVaultName} ...")

poller = client.begin_delete_secret(secretName)
deleted_secret = poller.result()

print(" done.")
```

Run the code

Make sure the code in the previous section is in a file named *kv_secrets.py*. Then run the code with the following command:

terminal

```
python kv_secrets.py
```

- If you encounter permissions errors, make sure you ran the [az keyvault set-policy](#) or [Set-AzKeyVaultAccessPolicy command](#).
- Rerunning the code with the same secret name may produce the error, "(Conflict) Secret <name> is currently in a deleted but recoverable state." Use a different secret name.

Code details

Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential class](#) provided by the [Azure Identity client library](#) is the recommended

approach for implementing passwordless connections to Azure services in your code.

`DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In the example code, the name of your key vault is expanded using the value of the `KVUri` variable, in the format: "https://<your-key-vault-name>.vault.azure.net".

Python

```
credential = DefaultAzureCredential()  
client = SecretClient(vault_url=KVUri, credential=credential)
```

Save a secret

Once you've obtained the client object for the key vault, you can store a secret using the [set_secret](#) method:

Python

```
client.set_secret(secretName, secretValue)
```

Calling `set_secret` generates a call to the Azure REST API for the key vault.

When Azure handles the request, it authenticates the caller's identity (the service principal) using the credential object you provided to the client.

Retrieve a secret

To read a secret from Key Vault, use the [get_secret](#) method:

Python

```
retrieved_secret = client.get_secret(secretName)
```

The secret value is contained in `retrieved_secret.value`.

You can also retrieve a secret with the Azure CLI command [az keyvault secret show](#) or the Azure PowerShell cmdlet [Get-AzKeyVaultSecret](#).

Delete a secret

To delete a secret, use the [begin_delete_secret](#) method:

Python

```
poller = client.begin_delete_secret(secretName)
deleted_secret = poller.result()
```

The `begin_delete_secret` method is asynchronous and returns a poller object. Calling the poller's `result` method waits for its completion.

You can verify that the secret had been removed with the Azure CLI command [az keyvault secret show](#) or the Azure PowerShell cmdlet [Get-AzKeyVaultSecret](#).

Once deleted, a secret remains in a deleted but recoverable state for a time. If you run the code again, use a different secret name.

Clean up resources

If you want to also experiment with [certificates](#) and [keys](#), you can reuse the Key Vault created in this article.

Otherwise, when you're finished with the resources created in this article, use the following command to delete the resource group and all its contained resources:

Azure CLI

Azure CLI

```
az group delete --resource-group myResourceGroup
```

Next steps

- [Overview of Azure Key Vault](#)
- [Azure Key Vault developer's guide](#)

- Key Vault security overview
- Authenticate with Key Vault

Quickstart: Create and deploy function code to Azure using Visual Studio Code

08/11/2025

Use Visual Studio Code to create a function that responds to HTTP requests from a template. Use GitHub Copilot to improve the generated function code, verify code updates locally, and then deploy it to the serverless Flex Consumption hosting plan in Azure Functions.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Make sure to select your preferred development language at the top of the article.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Visual Studio Code](#) on one of the [supported platforms](#).
- The [Azure Functions extension](#) for Visual Studio Code.
- Python versions that are [supported by Azure Functions](#). For more information, see [How to install Python](#).
- The [Python extension](#) for Visual Studio Code.

Install or update Core Tools

The Azure Functions extension for Visual Studio Code integrates with Azure Functions Core Tools so that you can run and debug your functions locally in Visual Studio Code using the Azure Functions runtime. Before getting started, it's a good idea to install Core Tools locally or update an existing installation to use the latest version.

In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command tries to either start a package-based installation of the latest version of Core Tools or update an existing package-based installation. If you don't have npm or Homebrew installed on your local computer, you must instead [manually install or update Core Tools](#).

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in your preferred language. Later in the article, you update, run, and then publish your function code to Azure.

1. In Visual Studio Code, press `F1` to open the command palette and search for and run the command `Azure Functions: Create New Project....`.
2. Choose the directory location for your project workspace and choose **Select**. You should either create a new folder or choose an empty folder for the project workspace. Don't choose a project folder that is already part of a workspace.
3. Provide the following information at the prompts:

 Expand table

Prompt	Selection
Select a language	Choose <code>Python</code> .
Select a Python interpreter to create a virtual environment	Choose your preferred Python interpreter. If an option isn't shown, type in the full path to your Python binary.
Select a template for your project's first function	Choose <code>HTTP trigger</code> .
Name of the function you want to create	Enter <code>HttpExample</code> .
Authorization level	Choose <code>ANONYMOUS</code> , which lets anyone call your function endpoint. For more information, see Authorization level .
Select how you would like to open your project	Choose <code>Open in current window</code> .

Using this information, Visual Studio Code generates a code project for Azure Functions with an HTTP trigger function endpoint. You can view the local project files in the Explorer. To learn more about files that are created, see [Generated project files](#).

4. In the `local.settings.json` file, update the `AzureWebJobsStorage` setting as in the following example:

JSON

```
"AzureWebJobsStorage": "UseDevelopmentStorage=true",
```

This tells the local Functions host to use the storage emulator for the storage connection required by the Python v2 model. When you publish your project to Azure, this setting

uses the default storage account instead. If you're using an Azure Storage account during local development, set your storage account connection string here.

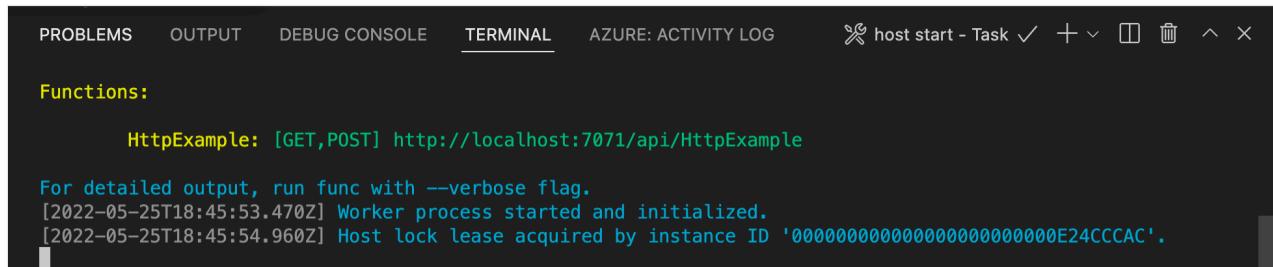
Start the emulator

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azurite: Start`.
2. Check the bottom bar and verify that Azurite emulation services are running. If so, you can now run your function locally.

Run the function locally

Visual Studio Code integrates with [Azure Functions Core tools](#) to let you run this project on your local development computer before you publish to Azure.

1. To start the function locally, press `F5` or the **Run and Debug** icon in the left-hand side Activity bar. The **Terminal** panel displays the Output from Core Tools. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.



The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The output window displays the following text:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    AZURE: ACTIVITY LOG    ⚙ host start - Task ✓ + × ☒ □ ☞ ^ ×

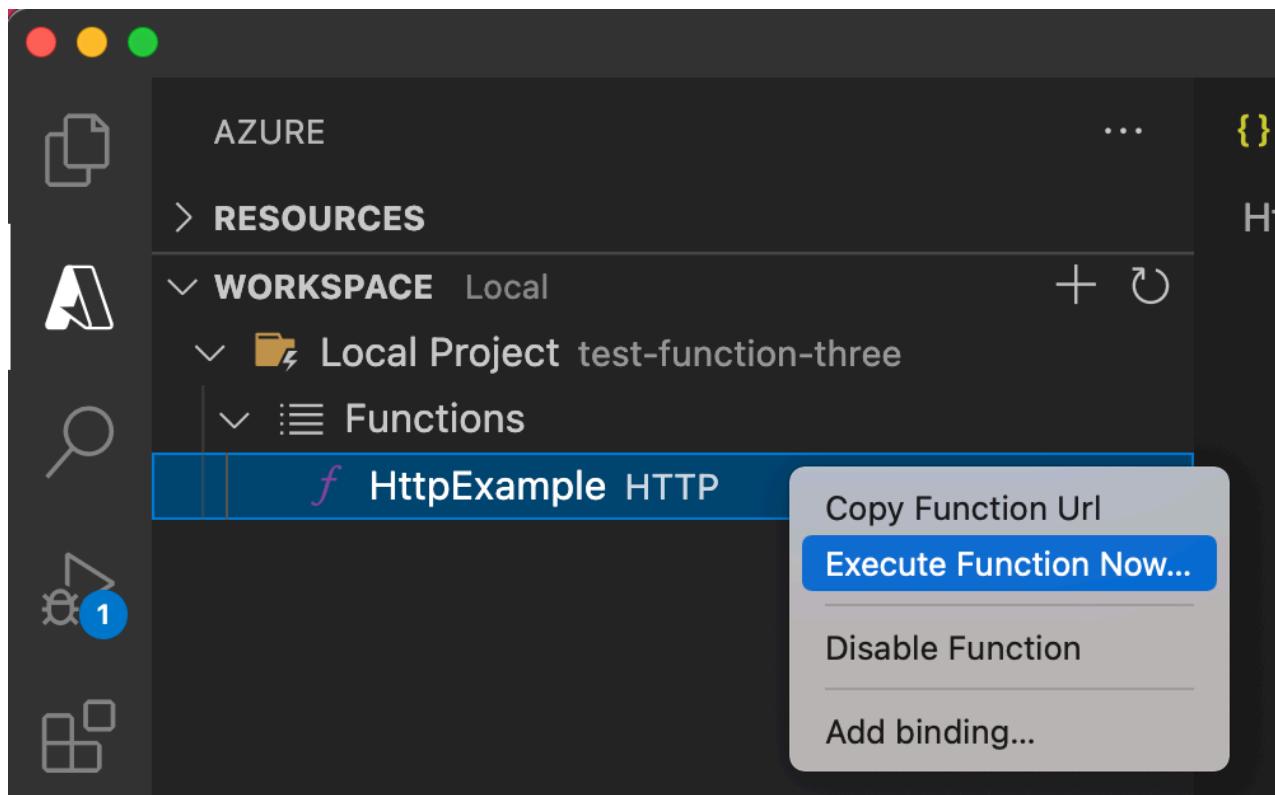
Functions:

HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2022-05-25T18:45:53.470Z] Worker process started and initialized.
[2022-05-25T18:45:54.960Z] Host lock lease acquired by instance ID '000000000000000000000000E24CCCAC'.
```

If you have trouble running on Windows, make sure that the default terminal for Visual Studio Code isn't set to **WSL Bash**.

2. With Core Tools still running in **Terminal**, choose the Azure icon in the activity bar. In the **Workspace** area, expand **Local Project > Functions**. Right-click (Windows) or `Ctrl -` click (macOS) the new function and choose **Execute Function Now....**



3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`.
Press Enter to send this request message to your function.
4. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.
5. With the **Terminal** panel focused, press `ctrl + c` to stop Core Tools and disconnect the debugger.

After you verify that the function runs correctly on your local computer, you can optionally use AI tools, such as GitHub Copilot in Visual Studio Code, to update template-generated function code.

Use AI to normalize and validate input

This is an example prompt for Copilot Chat that updates the existing function code to retrieve parameters from either the query string or JSON body, apply formatting or type conversions, and return them as JSON in the response:

Copilot prompt

Modify the function to accept name, email, and age from the JSON body of the request. If any of these parameters are missing from the query string, read them from the JSON body. Return all three parameters in the JSON response, applying these rules:
Title-case the name
Lowercase the email

```
Convert age to an integer if possible, otherwise return "not provided"
Use sensible defaults if any parameter is missing
```

You can customize your prompt to add specifics as needed, then run the app again locally and verify that it works as expected after the code changes. This time, use a message body like:

JSON

```
{ "name": "devon torres", "email": "torres.devon@contoso.com", "age": "34" }
```

💡 Tip

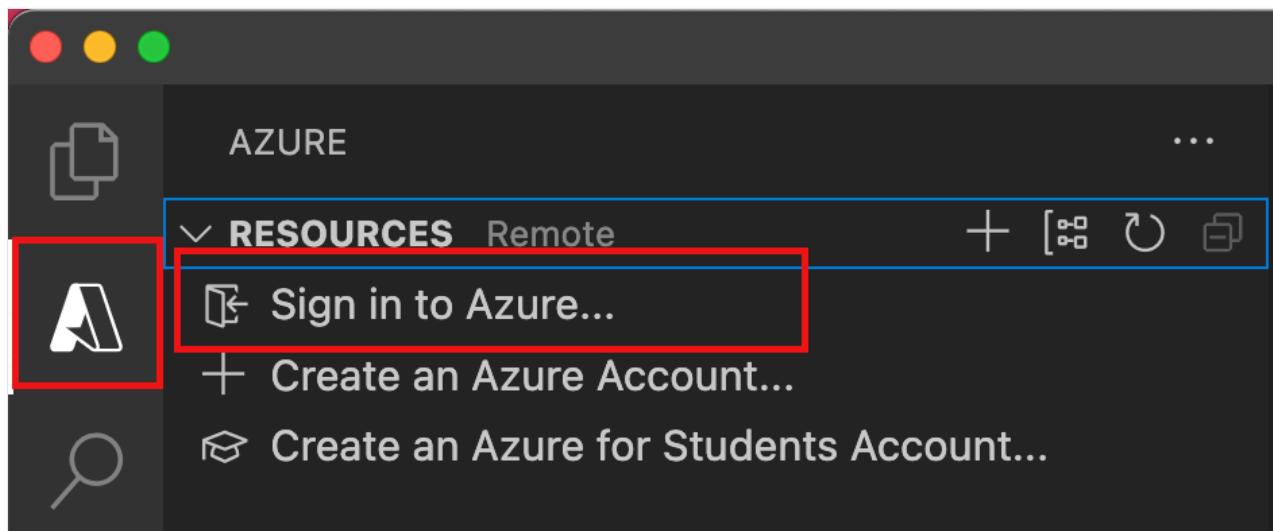
GitHub Copilot is powered by AI, so surprises and mistakes are possible. If you encounter any errors during execution, paste the error message in the chat window, select **Agent** mode, and ask Copilot to help resolve the error. For more information, see [Copilot FAQs](#).

When you are satisfied with your app, you can use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can create Azure resources or publish your app, you must sign in to Azure.

1. If you aren't already signed in, in the **Activity bar**, select the Azure icon. Then under **Resources**, select **Sign in to Azure**.



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, select **Create an Azure Account**. Students can select **Create an Azure for Students Account**.

- When you are prompted in the browser, select your Azure account and sign in by using your Azure account credentials. If you create a new account, you can sign in after your account is created.
- After you successfully sign in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the side bar.

Create the function app in Azure

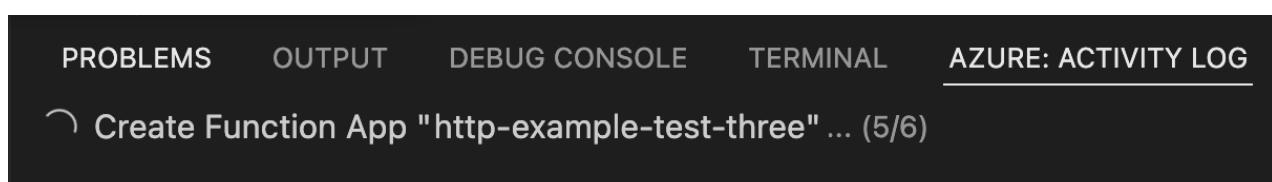
In this section, you create a function app in the Flex Consumption plan along with related resources in your Azure subscription. Many of the resource creation decisions are made for you based on default behaviors. For more control over the created resources, you must instead [create your function app with advanced options](#).

- In Visual Studio Code, select F1 to open the command palette. At the prompt (>), enter and then select **Azure Functions: Create Function App in Azure**.
- At the prompts, provide the following information:

[+] Expand table

Prompt	Action
Select subscription	Select the Azure subscription to use. The prompt doesn't appear when you have only one subscription visible under Resources.
Enter a new function app name	Enter a globally unique name that's valid in a URL path. The name you enter is validated to make sure that it's unique in Azure Functions.
Select a location for new resources	Select an Azure region. For better performance, select a region near you. Only regions supported by Flex Consumption plans are displayed.
Select a runtime stack	Select the language version you currently run locally.
Select resource authentication type	Select Managed identity , which is the most secure option for connecting to the default host storage account .

In the **Azure: Activity Log** panel, the Azure extension shows the status of individual resources as they're created in Azure.



3. When the function app is created, the following related resources are created in your Azure subscription. The resources are named based on the name you entered for your function app.

- A [resource group](#), which is a logical container for related resources.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Azure App Service plan, which defines the underlying host for your function app.
- A standard [Azure Storage account](#), which is used by the Functions host to maintain state and other information about your function app.
- An Application Insights instance that's connected to the function app, and which tracks the use of your functions in the app.
- A user-assigned managed identity that's added to the [Storage Blob Data Contributor](#) role in the new default host storage account.

A notification is displayed after your function app is created and the deployment package is applied.

Tip

By default, the Azure resources required by your function app are created based on the name you enter for your function app. By default, the resources are created with the function app in the same, new resource group. If you want to customize the names of the associated resources or reuse existing resources, [publish the project with advanced create options](#).

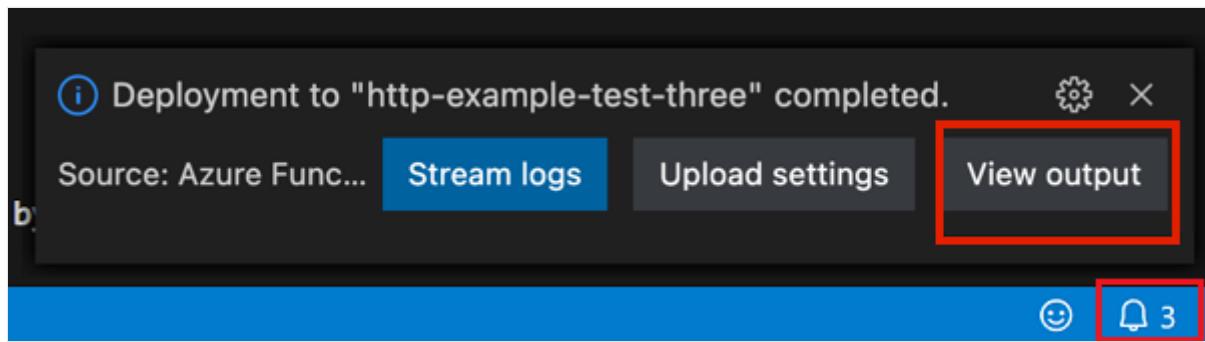
Deploy the project to Azure

Important

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the command palette, enter and then select **Azure Functions: Deploy to Function App**.
2. Select the function app you just created. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. When deployment is completed, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select

the bell icon in the lower-right corner to see it again.



Run the function in Azure

1. Press `F1` to display the command palette, then search for and run the command `Azure Functions:Execute Function Now....`. If prompted, select your subscription.
2. Select your new function app resource and `HttpExample` as your function.
3. In `Enter request body type` `{ "name": "Contoso", "email": "me@contoso.com", "age": "34" }`, then press Enter to send this request message to your function.
4. When the function executes in Azure, the response is displayed in the notification area. Expand the notification to review the full response.

Troubleshooting

Use the following table to resolve the most common issues encountered when using this article.

[] [Expand table](#)

Problem	Solution
Can't create a local function project?	Make sure you have the Azure Functions extension installed.
Can't run the function locally?	Make sure you have the latest version of Azure Functions Core Tools installed. When running on Windows, make sure that the default terminal shell for Visual Studio Code isn't set to WSL Bash.
Can't deploy function to Azure?	Review the Output for error information. The bell icon in the lower right corner is another way to view the output. Did you publish to an existing function app? That action overwrites the content of that app in Azure.

Problem	Solution
Couldn't run the cloud-based Function app?	Remember to use the query string to send in parameters.

Clean up resources

When you continue to the [next step](#) and add an Azure Storage queue binding to your function, you'll need to keep all your resources in place to build on what you've already done.

Otherwise, you can use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, select the Azure icon to open the Azure explorer.
2. In the Resource Groups section, find your resource group.
3. Right-click the resource group and select **Delete**.

To learn more about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You have used [Visual Studio Code](#) to create a function app with a simple HTTP-triggered function. In the next articles, you expand that function by connecting to either Azure Cosmos DB or Azure Storage. To learn more about connecting to other Azure services, see [Add bindings to an existing function in Azure Functions](#). If you want to learn more about security, see [Securing Azure Functions](#).

[Connect to Azure Cosmos DB](#)

[Connect to Azure Queue Storage](#)

Quickstart: Create a function in Azure from the command line

07/22/2025

In this article, you use local command-line tools to create a function that responds to HTTP requests. After verifying your code locally, you deploy it to a serverless Flex Consumption hosting plan in Azure Functions.

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

Make sure to select your preferred development language at the top of the article.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Python 3.11](#).
- [Azure CLI](#)
- The [jq command line JSON processor](#), used to parse JSON output, and is also available in Azure Cloud Shell.

Install the Azure Functions Core Tools

The recommended way to install Core Tools depends on the operating system of your local development computer.

Windows

The following steps use a Windows installer (MSI) to install Core Tools v4.x. For more information about other package-based installers, see the [Core Tools readme](#).

Download and run the Core Tools installer, based on your version of Windows:

- [v4.x - Windows 64-bit](#) (Recommended. [Visual Studio Code debugging](#) requires 64-bit.)
- [v4.x - Windows 32-bit](#)

If you previously used Windows installer (MSI) to install Core Tools on Windows, you should uninstall the old version from Add Remove Programs before installing the latest version.

Create and activate a virtual environment

In a suitable folder, run the following commands to create and activate a virtual environment named `.venv`. Make sure to use one of the [Python versions](#) supported by Azure Functions.

bash

Bash

```
python -m venv .venv
```

Bash

```
source .venv/bin/activate
```

If Python didn't install the `venv` package on your Linux distribution, run the following command:

Bash

```
sudo apt-get install python3-venv
```

You run all subsequent commands in this activated virtual environment.

Create a local code project and function

In Azure Functions, your code project is an app that contains one or more individual functions that each responds to a specific trigger. All functions in a project share the same configurations and are deployed as a unit to Azure. In this section, you create a code project that contains a single function.

1. In a terminal or command prompt, run this [func init](#) command to create a function app project in the current folder:

Console

```
func init --worker-runtime python
```

2. Use this `func new` command to add a function to your project:

Console

```
func new --name HttpExample --template "HTTP trigger" --authlevel "anonymous"
```

A new code file is added to your project. In this case, the `--name` argument is the unique name of your function (`HttpExample`) and the `--template` argument specifies an HTTP trigger.

The project root folder contains various files for the project, including configurations files named `local.settings.json` and `host.json`. Because `local.settings.json` can contain secrets downloaded from Azure, the file is excluded from source control by default in the `.gitignore` file.

Run the function locally

Verify your new function by running the project locally and calling the function endpoint.

1. Use this command to start the local Azure Functions runtime host in the root of the project folder:

Console

```
func start
```

Toward the end of the output, the following lines should appear:

```
...
```

```
Now listening on: http://0.0.0.0:7071  
Application started. Press Ctrl+C to shut down.
```

```
Http Functions:
```

```
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

```
...
```

 Note

If the `HttpExample` endpoint doesn't appear as expected, you likely started the host from outside the root folder of the project. In that case, use **`Ctrl+C`** to stop the host, navigate to the project's root folder, and run the previous command again.

2. Copy the URL of your `HttpExample` function from this output to a browser and browse to the function URL and you should receive success response with a "hello world" message.
3. When you're done, use **`Ctrl+C`** and choose `y` to stop the functions host.

Create supporting Azure resources for your function

Before you can deploy your function code to Azure, you need to create these resources:

- A [resource group](#), which is a logical container for related resources.
- A default [Storage account](#), which is used by the Functions host to maintain state and other information about your functions.
- A [user-assigned managed identity](#), which the Functions host uses to connect to the default storage account.
- A function app, which provides the environment for executing your function code. A function app maps to your local function project and lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

Use the Azure CLI commands in these steps to create the required resources.

1. If you haven't done so already, sign in to Azure:

```
Azure CLI
```

```
az login
```

The [az login](#) command signs you into your Azure account. Skip this step when running in Azure Cloud Shell.

2. If you haven't already done so, use this [az extension add](#) command to install the Application Insights extension:

```
Azure CLI
```

```
az extension add --name application-insights
```

3. Use this [az group create](#) command to create a resource group named

AzureFunctionsQuickstart-rg in your chosen region:

Azure CLI

```
az group create --name "AzureFunctionsQuickstart-rg" --location "<REGION>"
```

In this example, replace `<REGION>` with a region near you that supports the Flex Consumption plan. Use the [az functionapp list-fleconsumption-locations](#) command to view the list of currently supported regions.

4. Use this [az storage account create](#) command to create a general-purpose storage account in your resource group and region:

Azure CLI

```
az storage account create --name <STORAGE_NAME> --location "<REGION>" --resource-group "AzureFunctionsQuickstart-rg" \
--sku "Standard_LRS" --allow-blob-public-access false --allow-shared-key-access false
```

In this example, replace `<STORAGE_NAME>` with a name that is appropriate to you and unique in Azure Storage. Names must contain three to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account, which is [supported by Functions](#). This new account can only be accessed by using Microsoft Entra-authenticated identities that have been granted permissions to specific resources.

5. Use this script to create a user-assigned managed identity, parse the returned JSON properties of the object using `jq`, and grant `Storage Blob Data Owner` permissions in the default storage account:

Azure CLI

```
output=$(az identity create --name "func-host-storage-user" --resource-group "AzureFunctionsQuickstart-rg" --location <REGION> \
--query "{userId:id, principalId: principalId, clientId: clientId}" -o json)

userId=$(echo $output | jq -r '.userId')
principalId=$(echo $output | jq -r '.principalId')
clientId=$(echo $output | jq -r '.clientId')

storageId=$(az storage account show --resource-group "AzureFunctionsQuickstart-rg" --name <STORAGE_NAME> --query 'id' -o tsv)
az role assignment create --assignee-object-id $principalId --assignee-principal-type ServicePrincipal \
--role "Storage Blob Data Owner" --scope $storageId
```

If you don't have the `jq` utility in your local Bash shell, it's available in Azure Cloud Shell. In this example, replace `<STORAGE_NAME>` and `<REGION>` with your default storage account name and region, respectively.

The `az identity create` command creates an identity named `func-host-storage-user`. The returned `principalId` is used to assign permissions to this new identity in the default storage account by using the `az role assignment create` command. The `az storage account show` command is used to obtain the storage account ID.

6. Use this `az functionapp create` command to create the function app in Azure:

```
Azure CLI

az functionapp create --resource-group "AzureFunctionsQuickstart-rg" --name
<APP_NAME> --flexconsumption-location <REGION> \
--runtime python --runtime-version <LANGUAGE_VERSION> --storage-account
<STORAGE_NAME> \
--deployment-storage-auth-type UserAssignedIdentity --deployment-storage-
auth-value "func-host-storage-user"
```

In this example, replace these placeholders with the appropriate values:

- `<APP_NAME>`: a globally unique name appropriate to you. The `<APP_NAME>` is also the default DNS domain for the function app.
- `<STORAGE_NAME>`: the name of the account you used in the previous step.
- `<REGION>`: your current region.
- `<LANGUAGE_VERSION>`: use the same [supported language stack version](#) you verified locally.

This command creates a function app running in your specified language runtime on Linux in the [Flex Consumption Plan](#), which is free for the amount of usage you incur here. The command also creates an associated Azure Application Insights instance in the same resource group, with which you can use to monitor your function app executions and view logs. For more information, see [Monitor Azure Functions](#). The instance incurs no costs until you activate it.

7. Use this script to add your user-assigned managed identity to the [Monitoring Metrics Publisher](#) role in your Application Insights instance:

```
Azure CLI

appInsights=$(az monitor app-insights component show --resource-group
"AzureFunctionsQuickstart-rg" \
--app <APP_NAME> --query "id" --output tsv)
principalId=$(az identity show --name "func-host-storage-user" --resource-
```

```
group "AzureFunctionsQuickstart-rg" \
    --query principalId -o tsv)
az role assignment create --role "Monitoring Metrics Publisher" --assignee
$principalId --scope $appInsights
```

In this example, replace `<APP_NAME>` with the name of your function app. The [az role assignment create](#) command adds your user to the role. The resource ID of your Application Insights instance and the principal ID of your user are obtained by using the [az monitor app-insights component show](#) and [az identity show](#) commands, respectively.

Update application settings

To enable the Functions host to connect to the default storage account using shared secrets, you must replace the `AzureWebJobsStorage` connection string setting with several settings that are prefixed with `AzureWebJobsStorage_`. These settings define a complex setting that your app uses to connect to storage and Application Insights with a user-assigned managed identity.

1. Use this script to get the client ID of the user-assigned managed identity and uses it to define managed identity connections to both storage and Application Insights:

Azure CLI

```
clientId=$(az identity show --name func-host-storage-user \
    --resource-group AzureFunctionsQuickstart-rg --query 'clientId' -o tsv)
az functionapp config appsettings set --name <APP_NAME> --resource-group
"AzureFunctionsQuickstart-rg" \
    --settings AzureWebJobsStorage__accountName=<STORAGE_NAME> \
    AzureWebJobsStorage__credential=managedidentity
AzureWebJobsStorage__clientId=$clientId \
APPLICATIONINSIGHTS_AUTHENTICATION_STRING="ClientId=$clientId;Authorization=A
D"
```

In this script, replace `<APP_NAME>` and `<STORAGE_NAME>` with the names of your function app and storage account, respectively.

2. Run the [az functionapp config appsettings delete](#) command to remove the existing `AzureWebJobsStorage` connection string setting, which contains a shared secret key:

Azure CLI

```
az functionapp config appsettings delete --name <APP_NAME> --resource-group
"AzureFunctionsQuickstart-rg" --setting-names AzureWebJobsStorage
```

In this example, replace `<APP_NAME>` with the names of your function app.

At this point, the Functions host is able to connect to the storage account securely using managed identities instead of shared secrets. You can now deploy your project code to the Azure resources.

Deploy the function project to Azure

After you've successfully created your function app in Azure, you're now ready to deploy your local functions project by using the [func azure functionapp publish](#) command.

In your root project folder, run this [func azure functionapp publish](#) command:

```
Console
```

```
func azure functionapp publish <APP_NAME>
```

In this example, replace `<APP_NAME>` with the name of your app. A successful deployment shows results similar to the following output (truncated for simplicity):

```
...
Getting site publishing info...
Creating archive for current directory...
Performing remote build for functions project.

...
Deployment successful.
Remote build succeeded!
Syncing triggers...
Functions in msdocs-azurefunctions-qs:
  HttpExample - [httpTrigger]
    Invoke url: https://msdocs-azurefunctions-
qs.azurewebsites.net/api/httpexample
```

Invoke the function on Azure

Because your function uses an HTTP trigger and supports GET requests, you invoke it by making an HTTP request to its URL. It's easiest to do execute a GET request in a browser.

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar.

The endpoint URL should look something like this example:

```
https://contoso-app.azurewebsites.net/api/httpexample
```

When you navigate to this URL, the browser should display similar output as when you ran the function locally.

Clean up resources

If you continue to the [next step](#) and add an Azure Storage queue output binding, keep all your resources in place as you'll build on what you've already done.

Otherwise, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

[Connect to Azure Queue Storage](#)

Connect Azure Functions to Azure Storage using Visual Studio Code

Article • 04/26/2024

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding. Although a function has only one trigger, it can have multiple input and output bindings. To learn more, see [Azure Functions triggers and bindings concepts](#).

In this article, you learn how to use Visual Studio Code to connect Azure Storage to the function you created in the previous quickstart article. The output binding that you add to this function writes data from the HTTP request to a message in an Azure Queue storage queue.

Most bindings require a stored connection string that Functions uses to access the bound service. To make it easier, you use the storage account that you created with your function app. The connection to this account is already stored in an app setting named `AzureWebJobsStorage`.

Configure your local environment

Before you begin, you must meet the following requirements:

- Install the [Azure Storage extension for Visual Studio Code](#).
- Install [Azure Storage Explorer](#). Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.
- Complete the steps in [part 1 of the Visual Studio Code quickstart](#).

This article assumes that you're already signed in to your Azure subscription from Visual Studio Code. You can sign in by running `Azure: Sign In` from the command palette.

Download the function app settings

In the [previous quickstart article](#), you created a function app in Azure along with the required storage account. The connection string for this account is stored securely in the

app settings in Azure. In this article, you write messages to a Storage queue in the same account. To connect to your storage account when running the function locally, you must download app settings to the `local.settings.json` file.

1. Press `F1` to open the command palette, then search for and run the command `Azure Functions: Download Remote Settings....`
2. Choose the function app you created in the previous article. Select **Yes to all** to overwrite the existing local settings.

 **Important**

Because the `local.settings.json` file contains secrets, it never gets published, and is excluded from the source control.

3. Copy the value `AzureWebJobsStorage`, which is the key for the storage account connection string value. You use this connection to verify that the output binding works as expected.

Register binding extensions

Because you're using a Queue storage output binding, you must have the Storage bindings extension installed before you run the project.

Your project has been configured to use [extension bundles](#), which automatically installs a predefined set of extension packages.

Extension bundles is already enabled in the `host.json` file at the root of the project, which should look like the following example:

```
JSON

{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[3.*, 4.0.0)"
  }
}
```

Now, you can add the storage output binding to your project.

Add an output binding

Binding attributes are defined by decorating specific function code in the `function_app.py` file. You use the `queue_output` decorator to add an [Azure Queue storage output binding](#).

By using the `queue_output` decorator, the binding direction is implicitly 'out' and type is Azure Storage Queue. Add the following decorator to your function code in `HttpExample\function_app.py`:

Python

```
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
```

In this code, `arg_name` identifies the binding parameter referenced in your code, `queue_name` is name of the queue that the binding writes to, and `connection` is the name of an application setting that contains the connection string for the Storage account. In quickstarts you use the same storage account as the function app, which is in the `AzureWebJobsStorage` setting. When the `queue_name` doesn't exist, the binding creates it on first use.

Add code that uses the output binding

After the binding is defined, you can use the `name` of the binding to access it as an attribute in the function signature. By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data. The Functions runtime and queue output binding do those tasks for you.

Update `HttpExample\function_app.py` to match the following code, add the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="HttpExample")
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
def HttpExample(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) ->
func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')
    name = req.params.get('name')
```

```

if not name:
    try:
        req_body = req.get_json()
    except ValueError:
        pass
else:
    name = req_body.get('name')

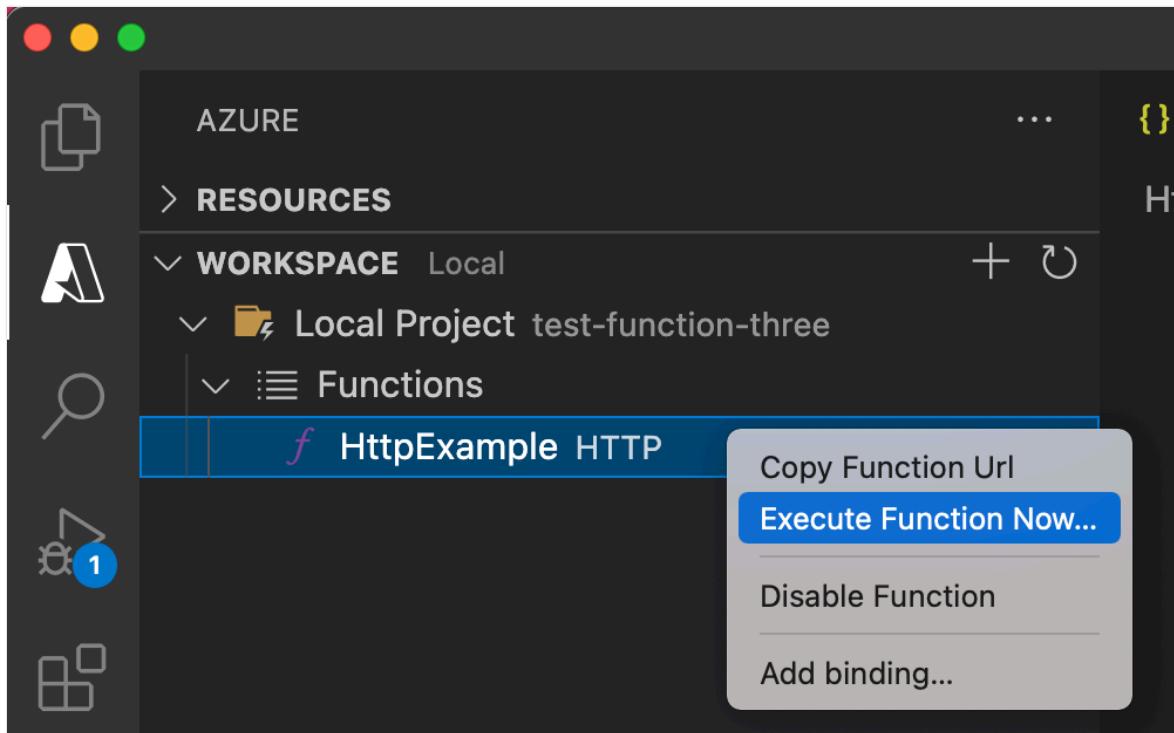
if name:
    msg.set(name)
    return func.HttpResponse(f"Hello, {name}. This HTTP triggered
function executed successfully.")
else:
    return func.HttpResponse(
        "This HTTP triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
        status_code=200
)

```

The `msg` parameter is an instance of the `azure.functions.Out class`. The `set` method writes a string message to the queue. In this case, it's the `name` passed to the function in the URL query string.

Run the function locally

1. As in the previous article, press `F5` to start the function app project and Core Tools.
2. With the Core Tools running, go to the **Azure: Functions** area. Under **Functions**, expand **Local Project > Functions**. Right-click (Ctrl-click on Mac) the `HttpExample` function and select **Execute Function Now....**



3. In the **Enter request body**, you see the request message body value of `{ "name": "Azure" }`. Press **Enter** to send this request message to your function.

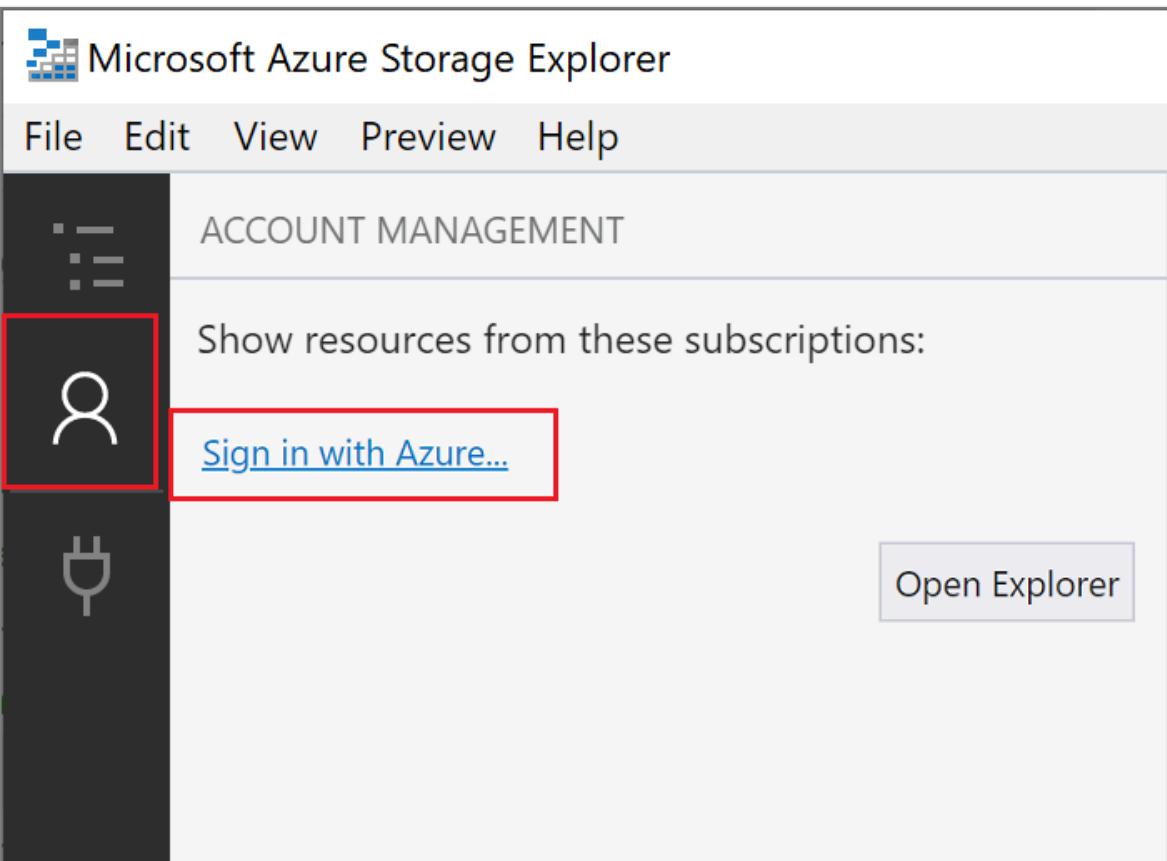
4. After a response is returned, press **Ctrl + C** to stop Core Tools.

Because you're using the storage connection string, your function connects to the Azure storage account when running locally. A new queue named **outqueue** is created in your storage account by the Functions runtime when the output binding is first used. You'll use Storage Explorer to verify that the queue was created along with the new message.

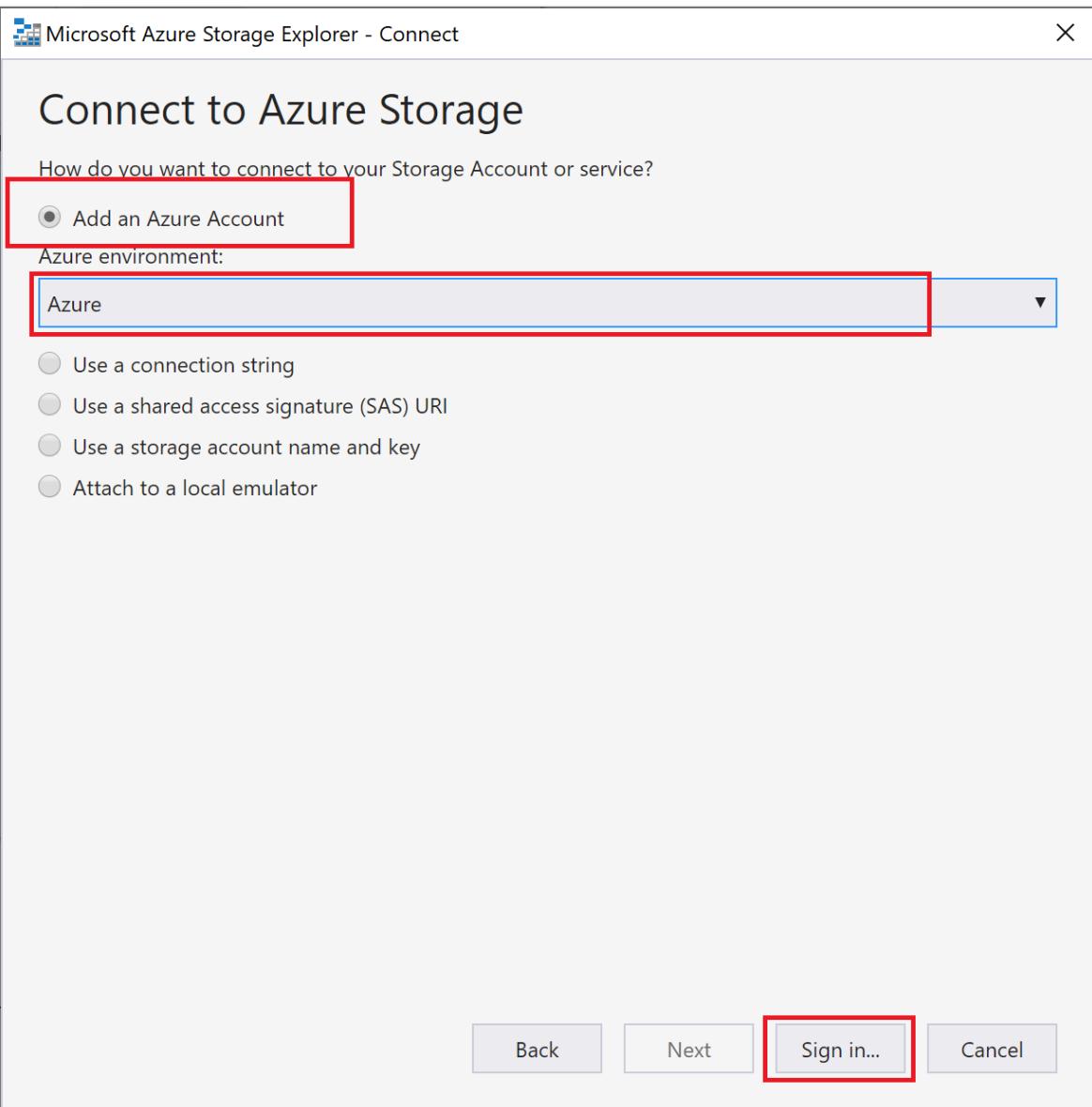
Connect Storage Explorer to your account

Skip this section if you've already installed Azure Storage Explorer and connected it to your Azure account.

1. Run the [Azure Storage Explorer](#) tool, select the connect icon on the left, and select **Add an account**.



2. In the **Connect** dialog, choose **Add an Azure account**, choose your **Azure environment**, and then select **Sign in....**

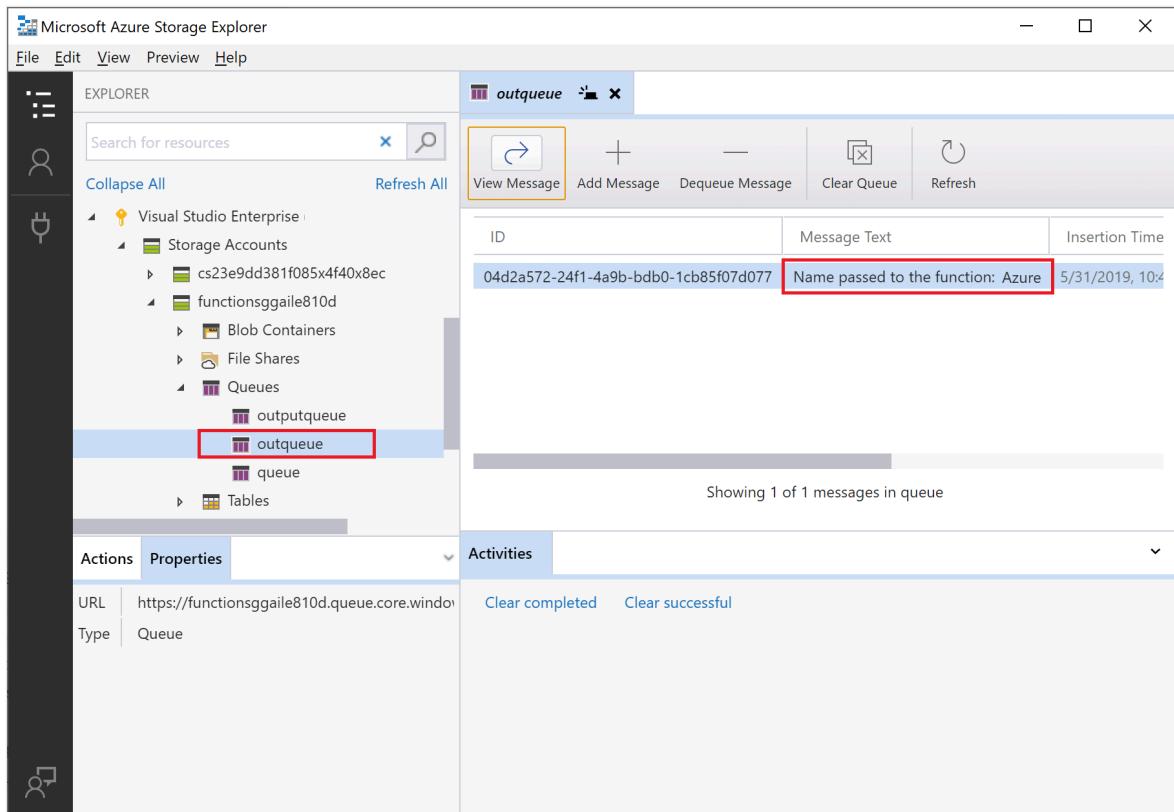


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

Examine the output queue

1. In Visual Studio Code, press **F1** to open the command palette, then search for and run the command **Azure Storage: Open in Storage Explorer** and choose your storage account name. Your storage account opens in the Azure Storage Explorer.
2. Expand the **Queues** node, and then select the queue named **outqueue**.

The queue contains the message that the queue output binding created when you ran the HTTP-triggered function. If you invoked the function with the default **name** value of *Azure*, the queue message is *Name passed to the function: Azure*.



3. Run the function again, send another request, and you see a new message in the queue.

Now, it's time to republish the updated function app to Azure.

Redeploy and verify the updated app

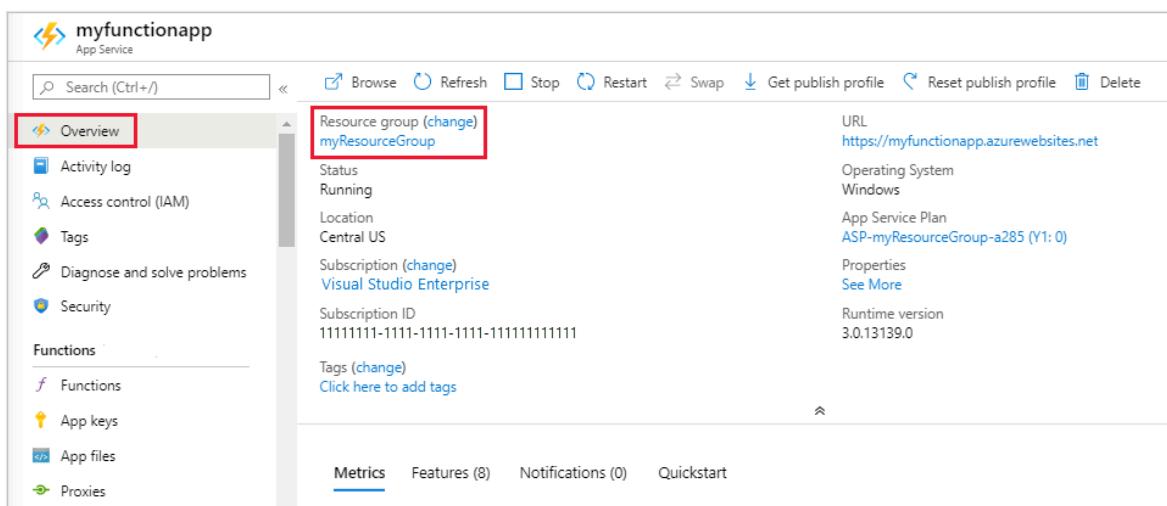
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** to dismiss the warning about overwriting files.
3. After the deployment completes, you can again use the **Execute Function Now...** feature to trigger the function in Azure.
4. Again [view the message in the storage queue](#) to verify that the output binding generates a new message in the queue.

Clean up resources

In Azure, *resources* refer to function apps, functions, storage accounts, and so forth. They're grouped into *resource groups*, and you can delete everything in a group by deleting the group.

You've created resources to complete these quickstarts. You may be billed for these resources, depending on your [account status](#) and [service pricing](#). If you don't need the resources anymore, here's how to delete them:

1. In Visual Studio Code, press `F1` to open the command palette. In the command palette, search for and select `Azure: Open in portal`.
2. Choose your function app and press `Enter`. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.



The screenshot shows the Azure portal interface for a function app named 'myfunctionapp'. The left sidebar has links for Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Functions (with sub-links for Functions, App keys, App files, and Proxies), Metrics, Features (8), Notifications (0), and Quickstart. The main content area is the 'Overview' tab, which displays the following details:

- Resource group (change)**: myResourceGroup
- Status: Running
- Location: Central US
- Subscription (change): Visual Studio Enterprise
- Subscription ID: 11111111-1111-1111-1111-111111111111
- Tags (change): Click here to add tags
- URL: https://myfunctionapp.azurewebsites.net
- Operating System: Windows
- App Service Plan: ASP-myResourceGroup-a285 (Y1: 0)
- Properties: See More
- Runtime version: 3.0.13139.0

4. On the **Resource group** page, review the list of included resources, and verify that they're the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions using Visual Studio Code:

- [Develop Azure Functions using Visual Studio Code](#)
- [Azure Functions triggers and bindings](#).
- [Examples of complete Function projects in Python](#).
- [Azure Functions Python developer guide](#)

Connect Azure Functions to Azure Storage using command line tools

Article • 12/29/2024

In this article, you integrate an Azure Storage queue with the function and storage account you created in the previous quickstart article. You achieve this integration by using an *output binding* that writes data from an HTTP request to a message in the queue. Completing this article incurs no extra costs beyond the few USD cents of the previous quickstart. To learn more about bindings, see [Azure Functions triggers and bindings concepts](#).

Configure your local environment

Before you begin, you must complete the article, [Quickstart: Create an Azure Functions project from the command line](#). If you already cleaned up resources at the end of that article, go through the steps again to recreate the function app and related resources in Azure.

Retrieve the Azure Storage connection string

Important

This article currently shows how to connect to your Azure Storage account by using the connection string, which contains a shared secret key. Using a connection string makes it easier for you to verify data updates in the storage account. For the best security, you should instead use managed identities when connecting to your storage account. For more information, see [Connections](#) in the Developer Guide.

Earlier, you created an Azure Storage account for function app's use. The connection string for this account is stored securely in app settings in Azure. By downloading the setting into the `local.settings.json` file, you can use the connection to write to a Storage queue in the same account when running the function locally.

1. From the root of the project, run the following command, replacing `<APP_NAME>` with the name of your function app from the previous step. This command overwrites any existing values in the file.

```
func azure functionapp fetch-app-settings <APP_NAME>
```

2. Open *local.settings.json* file and locate the value named `AzureWebJobsStorage`, which is the Storage account connection string. You use the name `AzureWebJobsStorage` and the connection string in other sections of this article.

 **Important**

Because the *local.settings.json* file contains secrets downloaded from Azure, always exclude this file from source control. The *.gitignore* file created with a local functions project excludes the file by default.

Add an output binding definition to the function

Although a function can have only one trigger, it can have multiple input and output bindings, which lets you connect to other Azure services and resources without writing custom integration code.

When using the [Python v2 programming model](#), binding attributes are defined directly in the *function_app.py* file as decorators. From the previous quickstart, your *function_app.py* file already contains one decorator-based binding:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp()

@app.function_name(name="HttpTrigger1")
@app.route(route="hello", auth_level=func.AuthLevel.ANONYMOUS)
```

The `route` decorator adds `HttpTrigger` and `HttpOutput` binding to the function, which enables your function be triggered when http requests hit the specified route.

To write to an Azure Storage queue from this function, add the `queue_output` decorator to your function code:

Python

```
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
```

In the decorator, `arg_name` identifies the binding parameter referenced in your code, `queue_name` is name of the queue that the binding writes to, and `connection` is the name of an application setting that contains the connection string for the Storage account. In quickstarts you use the same storage account as the function app, which is in the `AzureWebJobsStorage` setting (from `local.settings.json` file). When the `queue_name` doesn't exist, the binding creates it on first use.

For more information on the details of bindings, see [Azure Functions triggers and bindings concepts](#) and [queue output configuration](#).

Add code to use the output binding

With the queue binding defined, you can now update your function to receive the `msg` output parameter and write messages to the queue.

Update `HttpExample\function_app.py` to match the following code, add the `msg` parameter to the function definition and `msg.set(name)` under the `if name:` statement:

Python

```
import azure.functions as func
import logging

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="HttpExample")
@app.queue_output(arg_name="msg", queue_name="outqueue",
connection="AzureWebJobsStorage")
def HttpExample(req: func.HttpRequest, msg: func.Out[func.QueueMessage]) ->
func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        msg.set(name)
        return func.HttpResponse(f"Hello, {name}. This HTTP triggered")
```

```
        function executed successfully.")
    else:
        return func.HttpResponse(
            "This HTTP triggered function executed successfully. Pass a
            name in the query string or in the request body for a personalized
            response.",
            status_code=200
        )
```

The `msg` parameter is an instance of the [azure.functions.Out class](#). The `set` method writes a string message to the queue. In this case, it's the `name` passed to the function in the URL query string.

Observe that you *don't* need to write any code for authentication, getting a queue reference, or writing data. All these integration tasks are conveniently handled in the Azure Functions runtime and queue output binding.

Run the function locally

1. Run your function by starting the local Azure Functions runtime host from the `LocalFunctionProj` folder.

```
Console
```

```
func start
```

Toward the end of the output, the following lines must appear:

```
Azure Functions Core Tools
Core Tools Version:      4.0.5049 Commit hash: N/A  (64-bit)
Function Runtime Version: 4.15.2.20177

[2023-03-17T03:27:12.372Z] Worker process started and initialized.

Functions:

    HttpExample: [GET,POST] http://localhost:7071/api/HttpExample
```

ⓘ Note

If `HttpExample` doesn't appear as shown above, you likely started the host from outside the root folder of the project. In that case, use **Ctrl+C** to stop the host, go to the project's root folder, and run the previous command again.

2. Copy the URL of your HTTP function from this output to a browser and append the query string `?name=<YOUR_NAME>`, making the full URL like `http://localhost:7071/api/HttpExample?name=Functions`. The browser should display a response message that echoes back your query string value. The terminal in which you started your project also shows log output as you make requests.

3. When you're done, press `ctrl + c` and type `y` to stop the functions host.

Tip

During startup, the host downloads and installs the [Storage binding extension](#) and other Microsoft binding extensions. This installation happens because binding extensions are enabled by default in the `host.json` file with the following properties:

JSON

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[1.*, 2.0.0)"  
    }  
}
```

If you encounter any errors related to binding extensions, check that the above properties are present in `host.json`.

View the message in the Azure Storage queue

You can view the queue in the [Azure portal](#) or in the [Microsoft Azure Storage Explorer](#). You can also view the queue in the Azure CLI, as described in the following steps:

1. Open the function project's `local.setting.json` file and copy the connection string value. In a terminal or command window, run the following command to create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, and paste your specific connection string in place of `<MY_CONNECTION_STRING>`. (This environment variable means you don't need to supply the connection string to each subsequent command using the `--connection-string` argument.)

bash

Bash

```
export AZURE_STORAGE_CONNECTION_STRING=<MY_CONNECTION_STRING>"
```

2. (Optional) Use the `az storage queue list` command to view the Storage queues in your account. The output from this command must include a queue named `outqueue`, which was created when the function wrote its first message to that queue.

Azure CLI

```
az storage queue list --output tsv
```

3. Use the `az storage message get` command to read the message from this queue, which should be the value you supplied when testing the function earlier. The command reads and removes the first message from the queue.

bash

Azure CLI

```
echo `echo $(az storage message get --queue-name outqueue -o tsv --query '[].{Message:content}') | base64 --decode`
```

Because the message body is stored `base64 encoded`, the message must be decoded before it's displayed. After you execute `az storage message get`, the message is removed from the queue. If there was only one message in `outqueue`, you won't retrieve a message when you run this command a second time and instead get an error.

Redeploy the project to Azure

After you verify locally that the function wrote a message to the Azure Storage queue, you can redeploy your project to update the endpoint running on Azure.

In the `LocalFunctionsProj` folder, use the `func azure functionapp publish` command to redeploy the project, replacing `<APP_NAME>` with the name of your app.

```
func azure functionapp publish <APP_NAME>
```

Verify in Azure

1. As in the previous quickstart, use a browser or CURL to test the redeployed function.

Browser

Copy the complete **Invoke URL** shown in the output of the publish command into a browser address bar, appending the query parameter `&name=Functions`. The browser should display the same output as when you ran the function locally.

2. Examine the Storage queue again, as described in the previous section, to verify that it contains the new message written to the queue.

Clean up resources

After you finish, use the following command to delete the resource group and all its contained resources to avoid incurring further costs.

Azure CLI

```
az group delete --name AzureFunctionsQuickstart-rg
```

Next steps

You've updated your HTTP triggered function to write data to a Storage queue. Now you can learn more about developing Functions from the command line using Core Tools and Azure CLI:

- [Work with Azure Functions Core Tools](#)
- [Azure Functions triggers and bindings](#)
- [Examples of complete Function projects in Python.](#)
- [Azure Functions Python developer guide](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Data solutions for Python apps on Azure

06/05/2025

Azure offers a wide range of fully managed database and storage solutions, including relational, NoSQL, and in-memory databases, with support for both proprietary and open-source technologies. You can also choose from object, block, and file storage services. The following articles can help you get started using these options with Python on Azure.

Databases

- **PostgreSQL:** Build scalable, secure, and fully managed enterprise apps using open-source PostgreSQL. You can scale single-node PostgreSQL for high performance or migrate existing PostgreSQL and Oracle workloads to the cloud.
 - [Quickstart: Use Python to connect and query data in Azure Database for PostgreSQL - Flexible Server](#)
 - [Quickstart: Use Python to connect and query data in Azure Database for PostgreSQL - Single Server](#)
 - [Deploy a Python \(Django or Flask\) web app with PostgreSQL in Azure App Service](#)
- **MySQL:** Build scalable applications using a fully managed, intelligent MySQL database in the cloud.
 - [Quickstart: Use Python to connect and query data in Azure Database for MySQL - Flexible Server](#)
 - [Quickstart: Use Python to connect and query data in Azure Database for MySQL](#)
- **Azure SQL:** Build scalable applications with a fully managed and intelligent SQL database platform in the cloud.
 - [Quickstart: Use Python to query a database in Azure SQL Database or Azure SQL Managed Instance](#)

NoSQL, blobs, tables, files, graphs, and caches

- **Cosmos DB:** Build low-latency, high-availability apps at global scale, or migrate Cassandra, MongoDB, and other NoSQL workloads to the cloud.
 - [Quickstart: Azure Cosmos DB for NoSQL client library for Python](#)
 - [Quickstart: Azure Cosmos DB for MongoDB for Python with MongoDB driver](#)
 - [Quickstart: Build a Cassandra app with Python SDK and Azure Cosmos DB](#)
 - [Quickstart: Build an API for Table app with Python SDK and Azure Cosmos DB](#)
 - [Quickstart: Azure Cosmos DB for Apache Gremlin library for Python](#)

- **Blob storage:** Secure, massively scalable object storage for cloud-native apps, data lakes, archives, high-performance computing (HPC), and machine learning.
 - [Quickstart: Azure Blob Storage client library for Python](#)
 - [Azure Storage samples using v12 Python client libraries](#)
- **Azure Data Lake Storage Gen2:** Scalable, secure data lake optimized for high-performance analytics.
 - [Use Python to manage directories and files in Azure Data Lake Storage Gen2](#)
 - [Use Python to manage ACLs in Azure Data Lake Storage Gen2](#)
- **File storage:** Simple, secure, and serverless enterprise-grade cloud file shares.
 - [Develop for Azure Files with Python](#)
- **Redis Cache:** Accelerate application performance with a scalable, in-memory data store compatible with open source.
 - [Quickstart: Use Azure Cache for Redis in Python](#)

Big data and analytics

- **Azure Data Lake analytics:** Fully managed, pay-per-job analytics service that delivers powerful parallel data processing with built-in enterprise-grade security, auditing, and support.
 - [Manage Azure Data Lake Analytics using Python](#)
 - [Develop U-SQL with Python for Azure Data Lake Analytics in Visual Studio Code](#)
- **Azure Data Factory:** A fully managed data integration service that lets you visually build, orchestrate, and automate data movement and transformation across various data sources.
 - [Quickstart: Create a data factory and pipeline using Python](#)
 - [Transform data by running a Python activity in Azure Databricks](#)
- **Azure Event Hubs:** A fully managed, hyper-scale telemetry ingestion service designed to collect, transform, and store millions of events per second from connected devices and applications.
 - [Send events to or receive events from event hubs by using Python](#)
 - [Capture Event Hubs data in Azure Storage and read it by using Python \(azure-eventhub\)](#)
- **HDInsight:** A fully managed cloud service that runs popular open-source frameworks like Hadoop and Spark, backed by a 99.9% SLA for enterprise-grade big data analytics.
 - [Use Spark & Hive Tools for Visual Studio Code](#)

- **Azure Databricks:** A fully managed, fast, easy and collaborative Apache® Spark™ based analytics platform optimized for big data and AI workloads on Azure.
 - [Connect to Azure Databricks from Excel, Python, or R](#)
 - [Get Started with Azure Databricks](#)
 - [Tutorial: Azure Data Lake Storage Gen2, Azure Databricks & Spark](#)
- **Azure Synapse Analytics:** A fully managed analytics service that unifies data integration, enterprise data warehousing, and big data analytics into a single platform.
 - [Quickstart: Use Python to query a database in Azure SQL Database or Azure SQL Managed Instance \(includes Azure Synapse Analytics\)](#)

Machine learning for Python apps on Azure

06/12/2025

The following articles help you get started with Azure Machine Learning. Azure Machine Learning v2 REST APIs, Azure CLI extension, and Python SDK are designed to streamline the entire machine learning lifecycle and accelerate production workflows. The links in this article target v2, which is recommended if you're starting a new machine learning project.

Getting started

In Azure Machine Learning, the workspace is the main resource that organizes and manages everything you create, such as datasets, models, and experiments.

- [Quickstart: Get started with Azure Machine Learning](#)
- [Manage Azure Machine Learning workspaces in the portal or with the Python SDK \(v2\)](#)
- [Run Jupyter notebooks in your workspace](#)
- [Tutorial: Model development on a cloud workstation](#)

Deploy models

Deploy models for low-latency, real-time machine learning predictions.

- [Tutorial: Designer - deploy a machine learning model](#)
- [Deploy and score a machine learning model by using an online endpoint](#)

Automated machine learning

Automated ML (AutoML) refers to the process of streamlining machine learning model development by automating its repetitive and time-consuming tasks.

- [Train a regression model with AutoML and Python \(SDK v1\)](#)
- [Set up AutoML training for tabular data with the Azure Machine Learning CLI and Python SDK \(v2\)](#)

Data access

With Azure Machine Learning, you can import data from your local computer or connect to existing cloud storage services.

- [Create and manage data assets](#)

- [Tutorial: Upload, access and explore your data in Azure Machine Learning](#)
- [Access data in a job](#)

Machine learning pipelines

Use machine learning pipelines to build workflows that connect different stages of the ML process.

- [Use Azure Pipelines with Azure Machine Learning](#)
- [Create and run machine learning pipelines using components with the Azure Machine Learning SDK v2](#)
- [Tutorial: Create production ML pipelines with Python SDK v2 in a Jupyter notebook](#)

Overview of Python Container Apps in Azure

Article • 04/23/2025

This article explains how to take a Python project—such as a web application—and deploy it as a Docker container in Azure. It covers the general containerization workflow, Azure deployment options for containers, and Python-specific container configurations within Azure. Building and deploying Docker containers in Azure follows a standard process across languages, with Python-specific configurations in the Dockerfile, requirements.txt, and settings for web frameworks like [Django](#), [Flask](#), and [FastAPI](#).

Container workflow scenarios

For Python container development, some typical workflows for moving from code to container are discussed in the following table.

[] Expand table

Scenario	Description	Workflow
Dev	Build Python Docker images locally in your development environment.	<p>Code: Clone your app code locally using Git (with Docker installed).</p> <p>Build: Use Docker CLI, VS Code (with extensions), PyCharm (with Docker plugin). Described in section Working with Python Docker images and containers.</p> <p>Test: Run and test the container locally.</p> <p>Push: Push the image to a container registry like Azure Container Registry, Docker Hub, or private registry.</p> <p>Deploy: Deploy the container from the registry to an Azure service.</p>
Hybrid	Build Docker images in Azure, but initiate the process from your local environment.	<p>Code: Clone the code locally (not necessary for Docker to be installed).</p> <p>Build: To trigger builds in Azure, use VS Code (with remote extensions) or the Azure CLI.</p> <p>Push: Push the built image to Azure Container Registry.</p>

Scenario	Description	Workflow
		<p>Deploy: Deploy the container from the registry to an Azure service.</p>
Azure	<p>Use Azure Cloud Shell to build and deploy containers entirely in the cloud.</p>	<p>Code: Clone the GitHub repo in Azure Cloud Shell.</p> <p>Build: Use Azure CLI or Docker CLI in Cloud Shell.</p> <p>Push: Push the image to a registry like Azure Container Registry, Docker Hub, or private registry.</p> <p>Deploy: Deploy the container from the registry to an Azure service.</p>

The end goal of these workflows is to have a container running in one of the Azure resources supporting Docker containers as listed in the next section.

A dev environment can be:

- Your local workstation with Visual Studio Code or PyCharm
- [Codespaces](#) (a development environment hosted in the cloud)
- [Visual Studio Dev Containers](#) (a container as a development environment)

Deployment container options in Azure

Python container apps are supported in the following services.

[] [Expand table](#)

Service	Description
Web App for Containers	<p>Azure App Service is a fully managed hosting platform for containerized web applications, including websites and web APIs. It supports scalable deployments and integrates seamlessly with CI/CD workflows using Docker Hub, Azure Container Registry, and GitHub. This service is ideal for developers who want a simple and efficient path to deploy containerized apps, while also benefiting from the full capabilities of the Azure App Service platform. By packaging your application and all its dependencies into a single deployable container, you gain both portability and ease of management—without needing to manage infrastructure.</p> <p>Example: Deploy a Flask or FastAPI web app on Azure App Service.</p>
Azure Container Apps (ACA)	<p>Azure Container Apps (ACA) is a fully managed serverless container service powered by Kubernetes and open-source technologies like Dapr, KEDA, and envoy. Its design incorporates industry best practices and is optimized for executing general-purpose containers. ACA abstracts away the complexity of managing a Kubernetes infrastructure.</p>

Service	Description
	<p>—direct access to the Kubernetes API isn't required or supported. Instead, it offers higher-level application constructs such as revisions, scaling, certificates, and environments to simplify development and deployment workflows. This service is ideal for development teams looking to build and deploy containerized microservices with minimal operational overhead, allowing them to focus on application logic instead of infrastructure management.</p> <p>Example: Deploy a Flask or FastAPI web app on Azure Container Apps.</p>
Azure Container Instances (ACI) ↗	<p>Azure Container Instances (ACI) is a serverless offering that provides a single pod of Hyper-V isolated containers on demand. Billing is based on actual resource consumption rather than pre-allocated infrastructure, making it well-suited for short-lived or burstable workloads. Unlike other container services, ACI doesn't include built-in support for concepts like scaling, load balancing, or TLS certificates. Instead, it typically functions as a foundational container building block, often integrated with Azure services like Azure Kubernetes Service (AKS) for orchestration. ACI excels as a lightweight choice when the higher-level abstractions and features of Azure Container Apps aren't needed</p> <p>Example: Create a container image for deployment to Azure Container Instances. (The tutorial isn't Python-specific, but the concepts shown apply to all languages.)</p>
Azure Kubernetes Service (AKS) ↗	<p>Azure Kubernetes Service (AKS) is a fully managed Kubernetes option in Azure that gives you complete control over your Kubernetes environment. It supports direct access to the Kubernetes API and can run any standard Kubernetes workload. The full cluster resides in your subscription, with the cluster configurations and operations within your control and responsibility. ACI is ideal for teams seeking a fully managed container solution, while AKS gives you full control over the Kubernetes cluster, requiring you to manage configurations, networking, scaling, and operations. Azure handles the control plane and infrastructure provisioning, but the day-to-day operation and security of the cluster are within your team's control. This service is ideal for teams that want the flexibility and power of Kubernetes with the added benefit of Azure's managed infrastructure, while still maintaining full ownership over the cluster environment.</p> <p>Example: Deploy an Azure Kubernetes Service cluster using the Azure CLI.</p>
Azure Functions ↗	<p>Azure Functions offers an event-driven, serverless Functions-as-a-Service (FaaS) platform that lets you run small pieces of code (functions) in response to events—without managing infrastructure. Azure Functions shares many characteristics with Azure Container Apps around scale and integration with events, but is optimized for short-lived functions deployed as either code or containers. al for teams looking to trigger the execution of functions on events; for example, to bind to other data sources. Like Azure Container Apps, Azure Functions supports automatic scaling and integration with event sources (for example, HTTP requests, message queues, or blob storage updates). This service is ideal for teams building lightweight, event-triggered workflows, such as processing file uploads or responding to database changes, in Python or other languages.</p> <p>Example: Create a function on Linux using a custom container.</p>

For a more detailed comparison of these services, see [Comparing Container Apps with other Azure container options](#).

Virtual environments and containers

Virtual environments in Python isolate project dependencies from system-level Python installations, ensuring consistency across development environments. A virtual environment includes its own isolated Python interpreter, along with the libraries and scripts needed to run the specific project code within that environment. Dependencies for Python projects are managed through the *requirements.txt* file. By specifying dependencies in a *requirements.txt* file, developers can reproduce the exact environment needed for their project. This approach facilitates smoother transitions to containerized deployments like Azure App Service, where environment consistency is essential for reliable application performance.

Tip

In containerized Python projects, virtual environments are typically unnecessary because Docker containers provide isolated environments with their own Python interpreter and dependencies. However, you might use virtual environments for local development or testing. To keep Docker images lean, exclude virtual environments using a *.dockerignore* file, which prevents copying unnecessary files into the image.

You can think of Docker containers as offering capabilities similar to Python virtual environments—but with broader advantages in reproducibility, isolation, and portability. Unlike virtual environments, Docker containers can run consistently across different operating systems and environments, as long as a container runtime is available.

A Docker container includes your Python project code along with everything it needs to run, such as dependencies, environment settings, and system libraries. To create a container, you first build a Docker image from your project code and configuration, and then start a container, which is a runnable instance of that image.

For containerizing Python projects, the key files are described in the following table:

[] Expand table

Project file	Description
<i>requirements.txt</i>	This file contains the definitive list of Python dependencies needed for your application. Docker uses this list during the image build process to install all required packages. This ensures consistency between development and deployment environments.

Project file	Description
<i>Dockerfile</i>	This file contains instructions for building your Python Docker image, including the base image selection, dependency installation, code copying, and container startup commands. It defines the complete execution environment for your application. For more information, see the section Dockerfile instructions for Python .
<i>.dockerignore</i>	This file specifies the files and directories that should be excluded when copying content to the Docker image with the <code>COPY</code> command in the Dockerfile. This file uses patterns similar to <code>.gitignore</code> for defining exclusions. The <code>.dockerignore</code> file supports exclusion patterns similar to <code>.gitignore</code> files. For more information, see .dockerignore file .

Excluding files helps image build performance, but should also be used to avoid adding sensitive information to the image where it can be inspected. For example, the `.dockerignore` should contain lines to ignore `.env` and `.venv` (virtual environments).

Container settings for web frameworks

Web frameworks typically bind to default ports (such as 5000 for Flask, 8000 for FastAPI). When you are deploying containers to Azure services, such as Azure Container Instances, Azure Kubernetes Service (AKS), or App Service for Containers, it's crucial that you explicitly expose and configure the container's listening port to ensure proper routing of inbound traffic. Configuring the correct port ensures that Azure's infrastructure can direct requests to the correct endpoint inside your container.

[Expand table](#)

Web framework	Port
Django	8000
Flask	5000 or 5002
FastAPI (uvicorn)	8000 or 80

The following table shows how to set the port for different Azure container solutions.

[Expand table](#)

Azure container solution	How to set web app port
Web App for Containers	By default, App Service assumes your custom container is listening on either port 80 or port 8080. If your container listens to a different port, set the <code>WEBSITES_PORT</code> app

Azure container solution	How to set web app port
	setting in your App Service app. For more information, see Configure a custom container for Azure App Service .
Azure Containers Apps	Azure Container Apps lets you expose your container app to the public web, to your virtual network, or to other container apps within the same environment by enabling ingress. Set the ingress <code>targetPort</code> to the port your container listens to for incoming requests. Application ingress endpoint is always exposed on port 443. For more information, see Set up HTTPS or TCP ingress in Azure Container Apps .
Azure Container Instances, Azure Kubernetes	You define the port on which your app is listening during container or pod creation. Your container image should include a web framework, an application server (for example, gunicorn, uvicorn), and optionally a web server (for example, nginx). In more complex scenarios, you might split responsibilities across two containers—one for the application server and another for the web server. In that case, the web server container typically exposes ports 80 or 443 for external traffic.

Python Dockerfile

A Dockerfile is a text file that contains instructions for building a Docker image for a Python application. The first instruction typically specifies the base image to start from. Subsequent instructions then detail actions such as installing necessary software, copying application files, and configuring the environment to create a runnable image. The following table provides Python-specific examples for commonly used Dockerfile instructions.

[] [Expand table](#)

Instruction	Purpose	Example
FROM	Sets the base image for subsequent instructions.	<code>FROM python:3.8-slim</code>
EXPOSE	Tells Docker that the container listens to a specified port at runtime.	<code>EXPOSE 5000</code>
COPY	Copies files or directories from the specified source and adds them to the filesystem of the container at the specified destination path.	<code>COPY . /app</code>
RUN	Runs a command inside the Docker image. For example, pull in dependencies. The command runs once at build time.	<code>RUN python -m pip install -r requirements.txt</code>
CMD	The command provides the default for executing a container. There can only be one CMD instruction.	<code>CMD ["gunicorn", "--bind", "0.0.0.0:5000", "wsgi:app"]</code>

The Docker build command builds Docker images from a Dockerfile and a context. A build's context is the set of files located in the specified path or URL. Typically, you build an image from the root of your Python project and the path for the build command is "." as shown in the following example.

Bash

```
docker build --rm --pull --file "Dockerfile" --tag "mywebapp:latest" .
```

The build process can refer to any of the files in the context. For example, your build can use a COPY instruction to reference a file in the context. Here's an example of a Dockerfile for a Python project using the [Flask](#) framework:

Dockerfile

```
FROM python:3.8-slim

EXPOSE 5000

# Keeps Python from generating .pyc files in the container.
ENV PYTHONDONTWRITEBYTECODE=1

# Turns off buffering for easier container logging
ENV PYTHONUNBUFFERED=1

# Install pip requirements.
COPY requirements.txt .
RUN python -m pip install -r requirements.txt

WORKDIR /app
COPY . /app

# Creates a non-root user with an explicit UID and adds permission to access the
# /app folder.
RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser
/app
USER appuser

# Provides defaults for an executing container; can be overridden with Docker CLI.
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "wsgi:app"]
```

You can create a Dockerfile by hand or create it automatically with VS Code and the Docker extension. For more information, see [Generating Docker files](#).

The Docker build command is part of the Docker CLI. When you use IDEs like VS Code or PyCharm, the UI commands for working with Docker images call the build command for you and automate specifying options.

Working with Python Docker images and containers

VS Code and PyCharm

Integrated development environments (IDEs) like Visual Studio Code (VS Code) and PyCharm streamline Python container development by integrating Docker tasks into your workflow. With extensions or plugins, these IDEs simplify building Docker images, running containers, and deploying to Azure services like App Service or Container Instances. Here are some of the things you can do with VS Code and PyCharm.

- Download and build Docker images.
 - Build images in your dev environment.
 - Build Docker images in Azure without Docker installed in dev environment. (For PyCharm, use the Azure CLI to build images in Azure.)
- Create and run Docker containers from an existing image, a pulled image, or directly from a Dockerfile.
- Run multicontainer applications with Docker Compose.
- Connect and work with container registries like Docker Hub, GitLab, JetBrains Space, Docker V2, and other self-hosted Docker registries.
- (VS Code only) Add a Dockerfile and Docker compose files that are tailored for your Python project.

To set up VS Code and PyCharm to run Docker containers in your dev environment, use the following steps.

VS Code

If you haven't already, install [Azure Tools for VS Code](#).

[+] [Expand table](#)

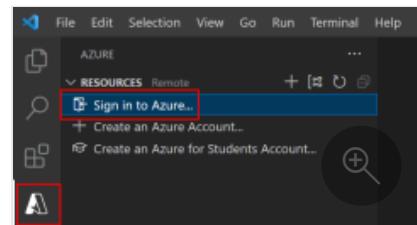
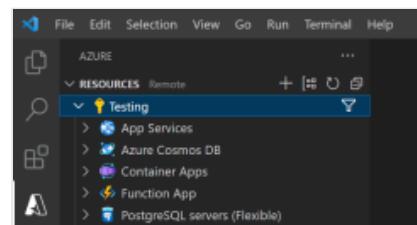
Instructions

Step 1: Use SHIFT + ALT + A to open the Azure extension and confirm you're connected to Azure.

You can also select the Azure icon on the VS Code extensions bar.

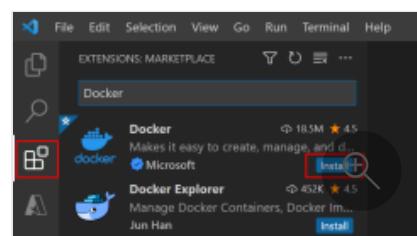
If you aren't signed in, select **Sign in to Azure** and follow the prompts.

If you have trouble accessing your Azure subscription, it may be because you're behind a proxy. To resolve connection issues, see [Network Connections in Visual Studio Code](#).

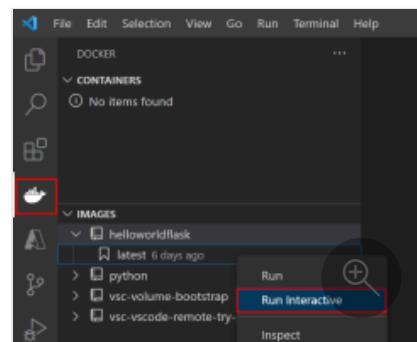


Step 2: Use CTRL + SHIFT + X to open Extensions, search for the [Docker extension](#), and install the extension.

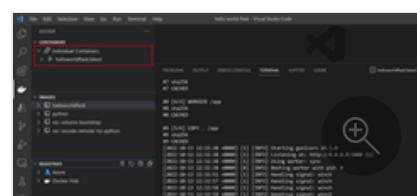
You can also select the **Extensions** icon on the VS Code extensions bar.



Step 3: Select the Docker icon in the extension bar, expand images, and right-click a Docker image run it as a container.



Step 4: Monitor the Docker run output in the **Terminal** window.



Azure CLI and Docker CLI

You can also work with Python Docker images and containers using the [Azure CLI](#) and [Docker CLI](#). Both VS Code and PyCharm have terminals where you can run these CLIs.

Use a CLI when you want finer control over build and run arguments, and for automation. For example, the following command shows how to use the Azure CLI `az acr build` to specify the Docker image name.

```
az acr build --registry <registry-name> \
--resource-group <resource-group> \
--target pythoncontainererwebapp:latest .
```

As another example, consider the following command that shows how to use the Docker CLI [run](#) command. The example shows how to run a Docker container that communicates to a MongoDB instance in your dev environment, outside the container. The different values to complete the command are easier to automate when specified in a command line.

Bash

```
docker run --rm -it \
--publish <port>:<port> --publish 27017:27017 \
--add-host mongoservice:<your-server-IP-address> \
--env CONNECTION_STRING=mongodb://mongoservice:27017 \
--env DB_NAME=<database-name> \
--env COLLECTION_NAME=<collection-name> \
containermongo:latest
```

For more information about this scenario, see [Build and test a containerized Python web app locally](#).

Environment variables in containers

Python projects commonly use environment variables to pass configuration data into the application code. This approach allows for greater flexibility across different environments. For instance, database connection details can be stored in environment variables, making it easy to switch between development, testing, and production databases without modifying the code. This separation of configuration from code promotes cleaner deployments and enhances security and maintainability.

Packages like [python-dotenv](#) are often used to read key-value pairs from an `.env` file and set them as environment variables. An `.env` file is useful when running in a virtual environment but isn't recommended when working with containers. **Don't copy the `.env` file into the Docker image, especially if it contains sensitive information and the container will be made public.** Use the `.dockerignore` file to exclude files from being copied into the Docker image. For more information, see the section [Virtual environments and containers](#) in this article.

You can pass environment variables to containers in a few ways:

1. Defined in the Dockerfile as [ENV](#) instructions.
2. Passed in as `--build-arg` arguments with the Docker [build](#) command.

3. Passed in as `--secret` arguments with the Docker build command and [BuildKit](#) backend.

4. Passed in as `--env` or `--env-file` arguments with the Docker [run](#) command.

The first two options have the same drawback as noted with `.env` files, namely that you're hardcoding potentially sensitive information into a Docker image. You can inspect a Docker image and see the environment variables, for example, with the command [docker image inspect](#).

The third option with BuildKit allows you to pass secret information to be used in the Dockerfile for building docker images in a safe way that won't end up stored in the final image.

The fourth option of passing in environment variables with the Docker run command means the Docker image doesn't contain the variables. However, the variables are still visible inspecting the container instance (for example, with [docker container inspect](#)). This option may be acceptable when access to the container instance is controlled or in testing or dev scenarios.

Here's an example of passing environment variables using the Docker CLI run command and using the `--env` argument.

Bash

```
# PORT=8000 for Django and 5000 for Flask
export PORT=<port-number>

docker run --rm -it \
  --publish $PORT:$PORT \
  --env CONNECTION_STRING=<connection-info> \
  --env DB_NAME=<database-name> \
  <dockerimagename:tag>
```

In VS Code (Docker extension) or PyCharm (Docker plugin), UI tools simplify managing Docker images and containers by executing standard docker CLI commands (such as `docker build`, `docker run`) in the background.

Finally, specifying environment variables when deploying a container in Azure is different than using environment variables in your dev environment. For example:

- For Web App for Containers, you configure application settings during configuration of App Service. These settings are available to your app code as environment variables and accessed using the standard [os.environ](#) pattern. You can change values after initial deployment when needed. For more information, see [Access app settings as environment variables](#).

- For Azure Container Apps, you configure environment variables during initial configuration of the container app. Subsequent modification of environment variables creates a *revision* of the container. In addition, Azure Container Apps allows you to define secrets at the application level and then reference them in environment variables. For more information, see [Manage secrets in Azure Container Apps](#).

As another option, you can use [Service Connector](#) to help you connect Azure compute services to other backing services. This service configures the network settings and connection information (for example, generating environment variables) between compute services and target backing services in management plane.

Viewing container logs

View container instance logs to see diagnostic messages output from code and to troubleshoot issues in your container's code. Here are several ways you can view logs when running a container in your ***dev environment***:

- Running a container with VS Code or PyCharm, as shown in the section [VS Code and PyCharm](#), you can see logs in terminal windows opened when Docker run executes.
- If you're using the Docker CLI [run ↗](#) command with the interactive flag `-it`, you see output following the command.
- In [Docker Desktop ↗](#), you can also view logs for a running container.

When you deploy a container in **Azure**, you also have access to container logs. Here are several Azure services and how to access container logs in Azure portal.

 [Expand table](#)

Azure service	How to access logs in Azure portal
Web App for Containers	Go to the Diagnose and solve problems resource to view logs. Diagnostics is an intelligent and interactive experience to help you troubleshoot your app with no configuration required. For a real-time view of logs, go to the Monitoring - Log stream . For more detailed log queries and configuration, see the other resources under Monitoring .
Azure Container Apps	Go to the environment resource Diagnose and solve problems to troubleshoot environment problems. More often, you want to see container logs. In the container resource, under Application - Revision management , select the revision and from there you can view system and console logs. For more detailed log queries and configuration, see the resources under Monitoring .

Azure service

How to access logs in Azure portal

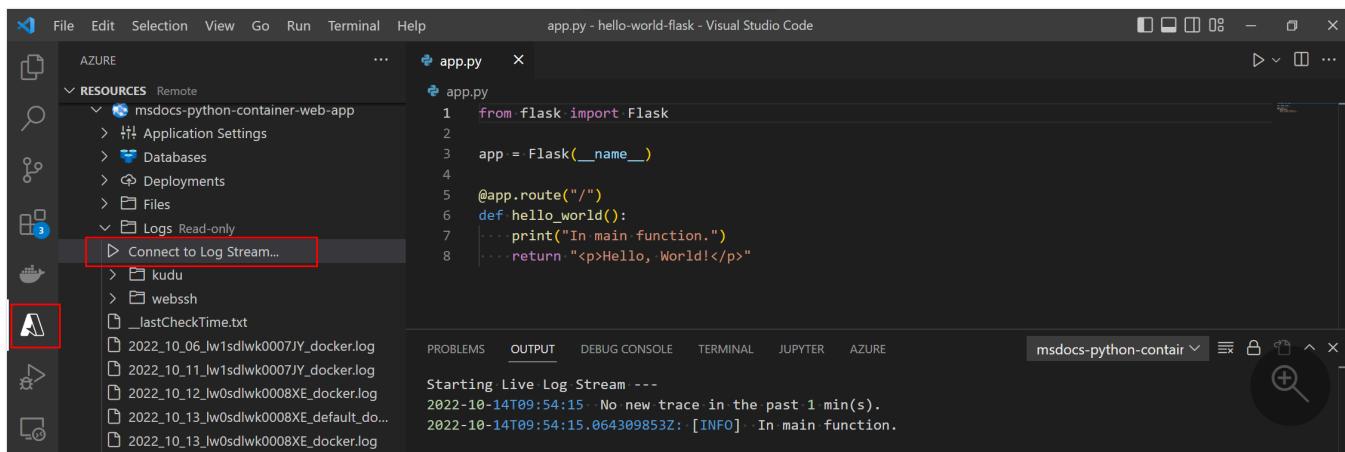
Azure Container Instances	Go to the Containers resource and select Logs .
---------------------------	---

For these services, here are the Azure CLI commands to access logs.

[\[+\] Expand table](#)

Azure service	Azure CLI command to access logs
Web App for Containers	az webapp log
Azure Container Apps	az containerapps logs
Azure Container Instances	az container logs

There's also support for viewing logs in VS Code. You must have [Azure Tools for VS Code](#) installed. Below is an example of viewing Web Apps for Containers (App Service) logs in VS Code.



Next steps

- Containerized Python web app on Azure with MongoDB
- Deploy a Python web app on Azure Container Apps with PostgreSQL

Deploy a containerized Flask or FastAPI web app on Azure App Service

06/10/2025

This tutorial shows you how to deploy a Python [Flask](#) or [FastAPI](#) web app to [Azure App Service](#) using the [Web App for Containers](#) feature. This approach provides a streamlined path for developers who want the benefits of a fully managed platform while deploying their app as a single containerized artifact with all dependencies included. For more information about using containers in Azure, see [Comparing Azure container options](#).

In this tutorial, you use the [Docker CLI](#) and [Docker](#) to optionally build and test a Docker image locally. You then use the [Azure CLI](#) to push the Docker image to [Azure Container Registry](#) (ACR) and deploy it to Azure App Service. The web app is configured with its system-assigned [managed identity](#) for secure, passwordless access to pull the image from ACR using Azure role-based access control (RBAC). You can also deploy with [Visual Studio Code](#) with the [Azure Tools Extension](#) installed.

For an example of building and creating a Docker image to run on Azure Container Apps, see [Deploy a Flask or FastPI web app on Azure Container Apps](#).

! Note

This tutorial demonstrates how to create a Docker image that can be deployed to Azure App Service. However, using a Docker image is not required to deploy to App Service. You can also deploy your application code directly from your local workspace to App Service without creating a Docker image. For an example, see [Quickstart: Deploy a Python \(Django or Flask\) web app to Azure App Service](#).

Prerequisites

To complete this tutorial, you need:

- An Azure account where you can deploy a web app to [Azure App Service](#) and [Azure Container Registry](#). If you don't have an Azure subscription, create a [free account](#) before you begin.
- [Azure CLI](#) to create a Docker image and deploy it to App Service. And optionally, [Docker](#) and the [Docker CLI](#) to create a Docker and test it in your local environment.

Get the sample code

In your local environment, get the code.

Flask

Bash

```
git clone https://github.com/Azure-Samples/msdocs-python-flask-webapp-quickstart.git  
cd msdocs-python-flask-webapp-quickstart
```

Add Dockerfile and .dockerignore files

Add a *Dockerfile* to instruct Docker how to build the image. The *Dockerfile* specifies the use of [Gunicorn](#), a production-level web server that forwards web requests to the Flask and FastAPI frameworks. The ENTRYPOINT and CMD commands instruct Gunicorn to handle requests for the app object.

Flask

Dockerfile

```
# syntax=docker/dockerfile:1  
  
FROM python:3.11  
  
WORKDIR /code  
  
COPY requirements.txt .  
  
RUN pip3 install -r requirements.txt  
  
COPY . .  
  
EXPOSE 50505  
  
ENTRYPOINT ["gunicorn", "app:app"]
```

50505 is used for the container port (internal) in this example, but you can use any free port.

Check the *requirements.txt* file to make sure it contains `gunicorn`.

Python

Flask==3.1.0

gunicorn

Add a `.dockerignore` file to exclude unnecessary files from the image.

`dockerignore`

```
.git*
**/*.pyc
.venv/
```

Configure gunicorn

Gunicorn can be configured with a `gunicorn.conf.py` file. When the `gunicorn.conf.py` file is located in the same directory where gunicorn is run, you don't need to specify its location in the `Dockerfile`. For more information about specifying the configuration file, see [Gunicorn settings](#).

In this tutorial, the suggested configuration file configures gunicorn to increase its number of workers based on the number of CPU cores available. For more information about `gunicorn.conf.py` file settings, see [Gunicorn configuration](#).

Flask

text

```
# Gunicorn configuration file
import multiprocessing

max_requests = 1000
max_requests_jitter = 50

log_file = "-"

bind = "0.0.0.0:50505"

workers = (multiprocessing.cpu_count() * 2) + 1
threads = workers

timeout = 120
```

Build and run the image locally

Build the image locally.

Flask

Bash

```
docker build --tag flask-demo .
```

! Note

If the `docker build` command returns an error, make sure the docker deamon is running.

On Windows, make sure that Docker Desktop is running.

Run the image locally in a Docker container.

Flask

Bash

```
docker run --detach --publish 5000:50505 flask-demo
```

Open the `http://localhost:5000` URL in your browser to see the web app running locally.

The `--detach` option runs the container in the background. The `--publish` option maps the container port to a port on the host. The host port (external) is first in the pair, and the container port (internal) is second. For more information, see [Docker run reference ↗](#).

Create a resource group and Azure Container Registry

1. Run the `az login` command to [sign in to Azure](#).

Azure CLI

```
az login
```

2. Run the `az upgrade` command to make sure your version of the Azure CLI is current.

```
Azure CLI
```

```
az upgrade
```

3. Create a group with the [az group create](#) command.

```
Azure CLI
```

```
RESOURCE_GROUP_NAME=<resource-group-name>
LOCATION=<location>
az group create --name $RESOURCE_GROUP_NAME --location $LOCATION
```

An Azure resource group is a logical container into which Azure resources are deployed and managed. When creating a resource group, you specify a location such as *eastus*. Replace `<location>` with the location you choose. Certain SKUs are unavailable in certain locations, so you might get an error indicating this. Use a different location and try again.

4. Create an Azure Container Registry with the [az acr create](#) command. Replace `<container-registry-name>` with a unique name for your instance.

```
Azure CLI
```

```
COUNTAINER_REGISTRY_NAME=<container-registry-name>
az acr create --resource-group $RESOURCE_GROOP_NAME \
--name $CONTAINER_REGISTRY_NAME --sku Basic
```

ⓘ Note

The registry name must be unique in Azure. If you get an error, try a different name. Registry names can consist of 5-50 alphanumeric characters. Hyphens and underscores are not allowed. To learn more, see [Azure Container Registry name rules](#). If you use a different name, make sure that you use your name rather than `webappacr123` in the commands that reference the registry and registry artifacts in following sections.

An Azure Container Registry is a private Docker registry that stores images for use in Azure Container Instances, Azure App Service, Azure Kubernetes Service, and other services. When creating a registry, you specify a name, SKU, and resource group.

Build the image in Azure Container Registry

Build the Docker image in Azure with the [az acr build](#) command. The command uses the Dockerfile in the current directory, and pushes the image to the registry.

Azure CLI

```
az acr build \
--resource-group $RESOURCE_GROUP_NAME \
--registry $CONTAINER_REGISTRY_NAME \
--image webappssimple:latest .
```

The `--registry` option specifies the registry name, and the `--image` option specifies the image name. The image name is in the format `registry.azurecr.io/repository:tag`.

Deploy web app to Azure

1. Create an App Service plan with the [az appservice plan](#) command.

Azure CLI

```
az appservice plan create \
--name webplan \
--resource-group $RESOURCE_GROUP_NAME \
--sku B1 \
--is-linux
```

2. Set an environment variable to your subscription ID. It's used in the `--scope` parameter in the next command.

Azure CLI

```
SUBSCRIPTION_ID=$(az account show --query id --output tsv)
```

The command for creating the environment variable is shown for the Bash shell. Change the syntax as appropriate for other environments.

3. Create the web app with the [az webapp create](#) command.

Azure CLI

```
export MSYS_NO_PATHCONV=1 # This line is for Windows users to prevent path
conversion issues in Git Bash.
az webapp create \
--resource-group $RESOURCE_GROUP_NAME \
--plan webplan --name <container-registry-name> \
--assign-identity [system] \
--role AcrPull \
```

```
--scope /subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP_NAME \
--acr-use-identity --acr-identity [system] \
--container-image-name
$CONTAINER_REGISTRY_NAME.azurecr.io/webappsimple:latest
```

Notes:

- The web app name must be unique in Azure. If you get an error, try a different name. The name can consist of alphanumeric characters and hyphens, but can't start or end with a hyphen. To learn more, see [Microsoft.Web name rules](#).
- If you're using a name different than `webappacr123` for your Azure Container Registry, make sure you update the `--container-image-name` parameter appropriately.
- The `--assign-identity`, `--role`, and `--scope` parameters enable the system-assigned managed identity on the web app and assign it the [AcrPull role](#) on the resource group. This gives the managed identity permission to pull images from any Azure Container Registry in the resource group.
- The `--acr-use-identity` and `--acr-identity` parameters configure the web app to use its system-assigned managed identity to pull images from the Azure Container Registry.
- It can take a few minutes for the web app to be created. You can check the deployment logs with the `az webapp log tail` command. For example, `az webapp log tail --resource-group web-app-simple-rg --name webappsimple123`. If you see entries with "warmup" in them, the container is being deployed.
- The URL of the web app is `<web-app-name>.azurewebsites.net`, for example, `https://webappsimple123.azurewebsites.net`.

Make updates and redeploy

After you make code changes, you can redeploy to App Service with the `az acr build` and `az webapp update` commands.

Clean up

All the Azure resources created in this tutorial are in the same resource group. Removing the resource group removes all resources in the resource group and is the fastest way to remove all Azure resources used for your app.

To remove resources, use the [az group delete](#) command.

Azure CLI

```
az group delete --name $RESOURCE_GROUP_NAME --yes --no-wait
```

You can also remove the group in the [Azure portal](#) or in [Visual Studio Code](#) and the [Azure Tools Extension](#).

Next steps

For more information, see the following resources:

- [Deploy a Python web app on Azure Container Apps](#)
- [Quickstart: Deploy a Python \(Django or Flask\) web app to Azure App Service](#)

Overview: Containerized Python web app on Azure with MongoDB

Article • 04/02/2025

This tutorial series shows you how to containerize a Python web app and then either run it locally or deploy it to [Azure App Service](#). App Service [Web App for Containers](#) allows you to focus on building your containers without worrying about managing and maintaining an underlying container orchestrator. When you are building web apps, Azure App Service is a good option for taking your first steps with containers. This container web app can use either a local MongoDB instance or [MongoDB for Azure Cosmos DB](#) to store data. For more information about using containers in Azure, see [Comparing Azure container options](#).

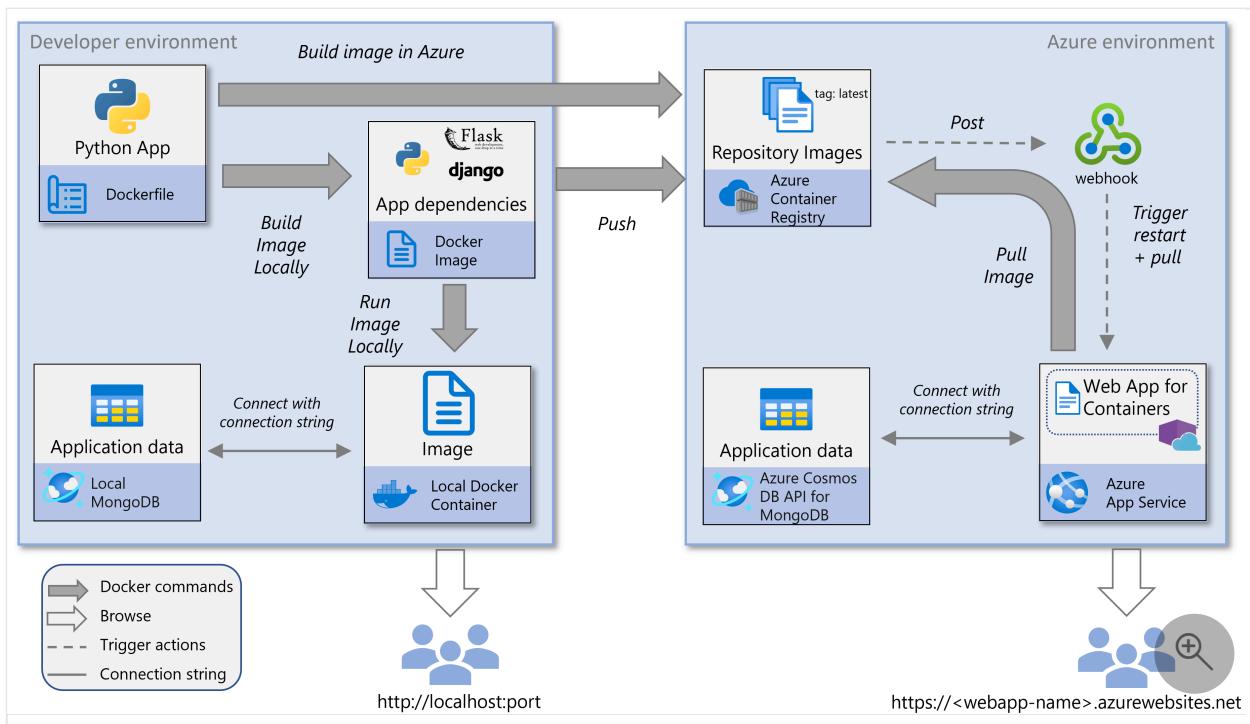
In this tutorial you:

- Build and run a [Docker](#) container locally. See [Build and run a containerized Python web app locally](#).
- Build a [Docker](#) container image directly in Azure. See [Build a containerized Python web app in Azure](#).
- Configure an App Service to create a web app based on the Docker container image. See [Deploy a containerized Python app to App Service](#).

After completing the articles in this tutorial series, you'll have the basis for Continuous Integration (CI) and Continuous Deployment (CD) of a Python web app to Azure.

Service overview

The service diagram supporting this tutorial shows two environments: developer environment and Azure environment. It highlights the key Azure services used in the development process.



Developer environment

The components supporting the developer environment in this tutorial include:

- **Local Development System:** A personal computer used for coding, building, and testing the Docker container.
- **Docker Containerization:** Docker is employed to package the app and its dependencies into a portable container.
- **Development Tools:** Includes a code editor and other necessary tools for software development.
- **Local MongoDB Instance:** A local MongoDB database utilized for data storage during development.
- **MongoDB Connection:** Access to the local MongoDB database provided through a connection string.

Azure environment

The components supporting the Azure environment in this tutorial include:

- [Azure App Service](#)
 - In Azure App Service, Web App for Containers uses the [Docker](#) container technology to provide container hosting of both built-in images and custom images using Docker.

- Web App for Containers uses a webhook in the Azure Container Registry (ACR) to get notified of new images. When a new image is pushed to the registry, the webhook notification triggers App Service to pull the update and restart the app.
- [Azure Container Registry](#)
 - Azure Container Registry allows you to store and manage Docker images and their components in Azure. It provides a registry located near your deployments in Azure that gives you the ability to control access using your Microsoft Entra groups and permissions.
 - In this tutorial, Azure Container Registry is the registry source, but you can also use Docker Hub or a private registry with minor modifications.
- [Azure Cosmos DB for MongoDB](#)
 - The Azure Cosmos DB for MongoDB is a NoSQL database used in this tutorial for data storage.
 - The containerized application connects to and accesses the Azure Cosmos DB resource using a connection string, which is stored as an environment variable and provided to the app.

Authentication

In this tutorial, you build a Docker image, either locally or in Azure, and then deploy it to Azure App Service. The App Service pulls the container image from an Azure Container Registry repository.

To securely pull images from the repository, App Service utilizes a system-assigned managed identity. This managed identity grants the web app permissions to interact with other Azure resources, eliminating the need for explicit credentials. For this tutorial, the managed identity is configured during setup of App Service to use a registry container image.

The tutorial sample web app uses MongoDB to store data. The sample code connects to Azure Cosmos DB via a connection string.

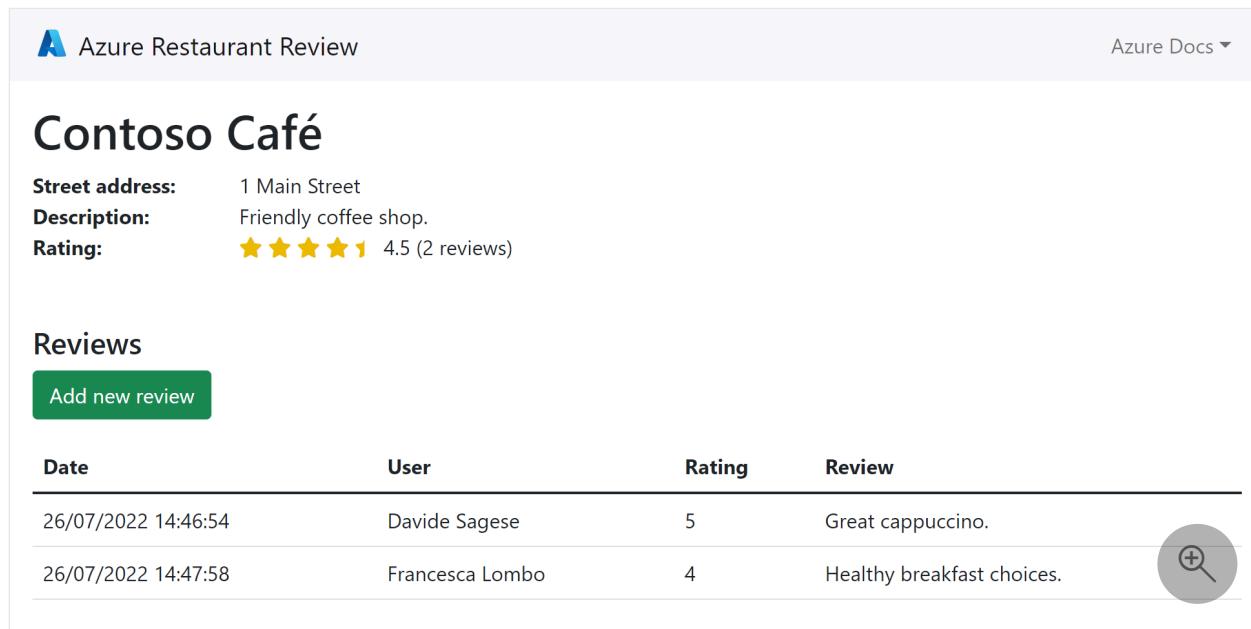
Prerequisites

To complete this tutorial, you need:

- An Azure account where you can create:
 - [Azure Container Registry](#)
 - [Azure App Service](#)
 - [Azure Cosmos DB for MongoDB](#) (or access to an equivalent). To create an Azure Cosmos DB for MongoDB database, follow the steps in [part 2 of this tutorial](#).
- [Visual Studio Code](#) or [Azure CLI](#), depending on your tool of choice. If you use Visual Studio Code, you need the [Docker extension](#) and [Azure App Service extension](#).
- These Python packages:
 - [MongoDB Shell \(mongosh\)](#) for connecting to MongoDB.
 - [Flask](#) or [Django](#) as a web framework.
- [Docker](#) installed locally.

Sample app

The end result of this tutorial is a restaurant review app, deployed and running in Azure, that looks like the following screenshot.



The screenshot shows a web application titled "Azure Restaurant Review". At the top, there's a header with the Azure logo and "Azure Restaurant Review" on the left, and "Azure Docs" with a dropdown arrow on the right. Below the header, the main content area displays a restaurant profile for "Contoso Café". The profile includes the street address "1 Main Street", the description "Friendly coffee shop.", and a rating of "4.5 (2 reviews)" with five yellow stars. Under the heading "Reviews", there's a green button labeled "Add new review". A table lists two reviews: one from "Davide Sagese" on 26/07/2022 at 14:46:54 rating 5 with the review "Great cappuccino.", and another from "Francesca Lombo" on 26/07/2022 at 14:47:58 rating 4 with the review "Healthy breakfast choices.". To the right of the reviews table is a circular icon containing a magnifying glass.

Date	User	Rating	Review
26/07/2022 14:46:54	Davide Sagese	5	Great cappuccino.
26/07/2022 14:47:58	Francesca Lombo	4	Healthy breakfast choices.

In this tutorial, you build a Python restaurant review app that utilizes MongoDB for data storage. For an example app using PostgreSQL, see [Create and deploy a Flask web app to Azure with a managed identity](#).

Next step

[Build and test locally](#)

Feedback

Was this page helpful?

 Yes

 No

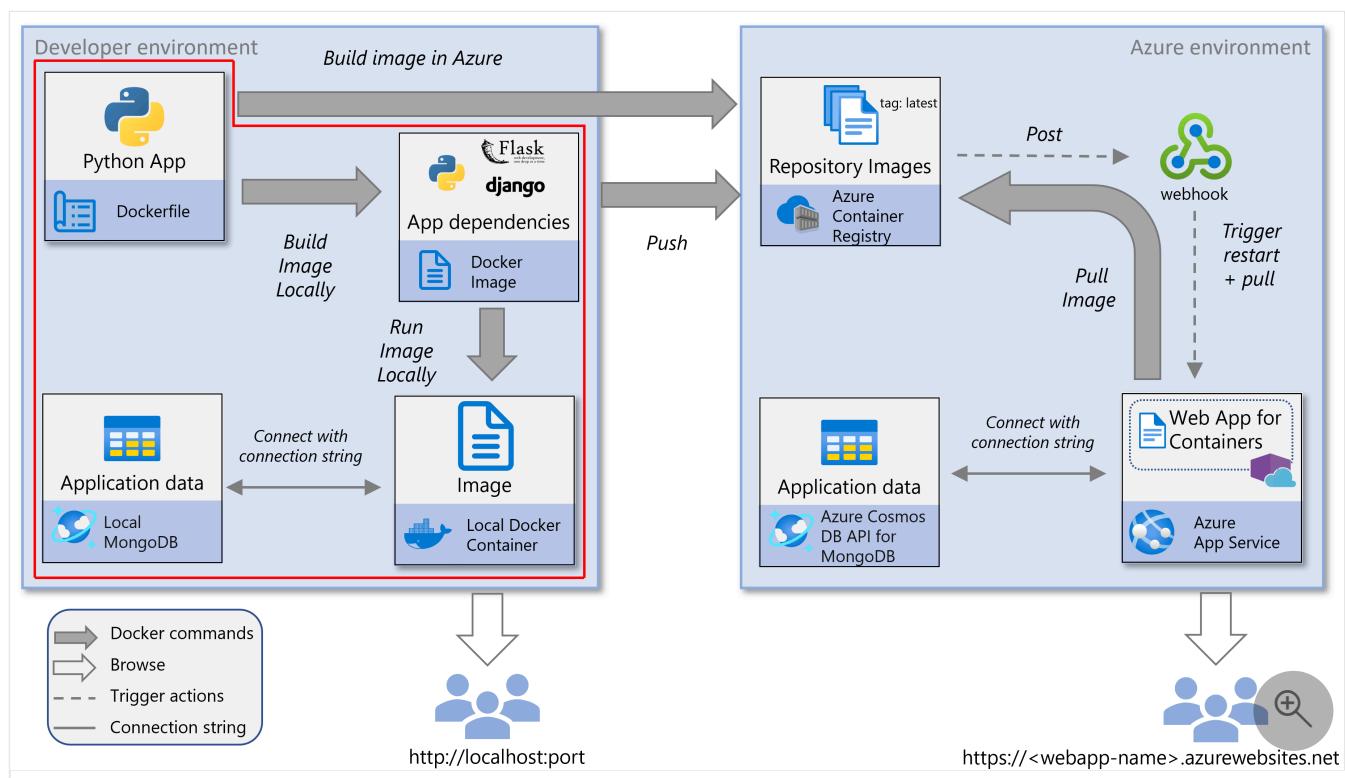
[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Build and run a containerized Python web app locally

Article • 04/17/2025

In this part of the tutorial series, you learn how to build and run a containerized [Django](#) or [Flask](#) Python web app on your local computer. To store data for this app, you can use either a local MongoDB instance or [Azure Cosmos DB for MongoDB](#). This article is part 2 of a 5-part tutorial series. We recommend that you complete [part 1](#) before starting this article.

The following service diagram highlights the local components covered in this article. In this article, you also learn how to use Azure Cosmos DB for MongoDB with a local Docker image, rather than a local instance of MongoDB.



Clone or download the sample Python app

In this section, you clone or download the sample Python app that you use to build a Docker image. You can choose between a Django or Flask Python web app. If you have your own Python web app, you can choose to use that instead. If you use your own Python web app, make sure your app has a *Dockerfile* in the root folder and can connect to a MongoDB database.

Git clone

1. Clone either the Django or Flask repository into a local folder by using one of the following commands:

Django

Console

```
# Django
git clone https://github.com/Azure-Samples/msdocs-python-django-
container-web-app.git
```

2. Navigate to the root folder for your cloned repository.

Django

Console

```
# Django
cd msdocs-python-django-container-web-app
```

Build a Docker image

In this section, you build a Docker image for the Python web app using either Visual Studio Code or the Azure CLI. The Docker image contains the Python web app, its dependencies, and the Python runtime. The Docker image is built from a *Dockerfile* that defines the image's contents and behavior. The *Dockerfile* is in the root folder of the sample app you cloned or downloaded (or provided yourself).

Tip

If you're new to the Azure CLI, see [Get started with Azure CLI](#) to learn how to download and install the Azure CLI locally or how to run Azure CLI commands in Azure Cloud Shell.

Azure CLI

[Docker](#) is required to build the Docker image using the Docker CLI. Once Docker is installed, open a terminal window and navigate to the sample folder.

① Note

The steps in this section require the Docker daemon to be running. In some installations, for example on Windows, you need to open [Docker Desktop](#), which starts the daemon, before proceeding.

1. Confirm that Docker is accessible by running the following command in the root folder of the sample app.

Console

```
docker
```

If, after running this command, you see help for the [Docker CLI](#), Docker is accessible. Otherwise, make sure Docker is installed and that your shell has access to the Docker CLI.

2. Build the Docker image for the Python web app by using the [Docker build](#) command.

The general form of the command is `docker build --rm --pull --file "<path-to-project-root>/Dockerfile" --label "com.microsoft.created-by=docker-cli" --tag "<container-name>:latest" "<path-to-project-root>"`.

If you're at the root folder of the project, use the following command to build the Docker image. The dot (".") at the end of the command refers to the current directory in which the command runs. To force a rebuild, add `--no-cache`.

Bash

Console

```
#!/bin/bash
docker build --rm --pull \
  --file "Dockerfile" \
  --label "com.microsoft.create-by=docker-cli" \
  --tag "msdocspythoncontainerwebapp:latest" \
  .
```

3. Confirm the image was built successfully by using the [Docker images](#) command.

Console

```
docker images
```

The command returns a list of images by REPOSITORY name, TAG, and CREATED date among other image characteristics.

At this point, you have a local Docker image named "msdocspythoncontainerwebapp" with the tag "latest". Tags help define version details, intended use, stability, and other relevant information. For more information, see [Recommendations for tagging and versioning container images](#).

 **Note**

Images that are built from VS Code or by using the Docker CLI directly can also be viewed with the [Docker Desktop](#) application.

Set up MongoDB

Your Python web app requires a MongoDB database named *restaurants_reviews* and a collection named *restaurants_reviews* are required to store data. In this tutorial, you use both a local installation of MongoDB and a [Azure Cosmos DB for MongoDB](#) instance to create and access the database and collection.

 **Important**

Don't use a MongoDB database you use in production. In this tutorial, you store the MongoDB connection string to the one of these MongoDB instances in an environment variable (which is observable by anyone capable of inspecting your container - such as by using `docker inspect`).

Local MongoDB

Let's start by creating a local instance of MongoDB using the Azure CLI.

1. Install [MongoDB](#) (if it isn't already installed).

You can check for the installation of MongoDB by using the [MongoDB Shell \(mongosh\)](#). If the following commands don't work, you may need to explicitly [install mongosh](#) or [connect mongosh to your MongoDB server](#).

- Use the following command to open the MongoDB shell and get the version of both the MongoDB shell and the MongoDB server:

```
Console
```

```
mongosh
```

💡 Tip

To return just the version of MongoDB server installed on your system, close and reopen the MongoDB shell and use the following command: `mongosh --quiet --exec 'db.version()'`

In some setups, you can also directly invoke the Mongo daemon in your bash shell.

```
Console
```

```
mongod --version
```

2. Edit the `mongod.cfg` file in the `\MongoDB\Server\8.0\bin` folder and add your computer's local IP address to the `bindIP` key.

The `bindip` key in the [MongoD configuration file](#) defines the hostnames and IP addresses that MongoDB listens for client connections. Add the current IP of your local development computer. The sample Python web app running locally in a Docker container communicates to the host computer with this address.

For example, part of the configuration file should look like this:

```
yml
```

```
net:  
  port: 27017  
  bindIp: 127.0.0.1,<local-ip-address>
```

3. Save your changes to this configuration file.

ⓘ Important

You need administrative privileges to save the changes you make to this configuration file.

4. Restart MongoDB to pick up changes to the configuration file.
5. Open a MongoDB shell and run the following command to set the database name to "restaurants_reviews" and the collection name to "restaurants_reviews". You can also create a database and collection with the VS Code [MongoDB extension](#) or any other MongoDB-aware tool.

```
mongosh  
> help  
> use restaurants_reviews  
> db.restaurants_reviews.insertOne({})  
> show dbs  
> exit
```

After you complete the previous step, the local MongoDB connection string is "mongodb://127.0.0.1:27017/", the database name is "restaurants_reviews", and the collection name is "restaurants_reviews".

Azure Cosmos DB for MongoDB

Now, let's also create an Azure Cosmos DB for MongoDB instance using the Azure CLI.

! Note

In part 4 of this tutorial series, you use the Azure Cosmos DB for MongoDB instance to run the web app in Azure App Service.

Before running the following script, replace the location, the resource group, and Azure Cosmos DB for MongoDB account name with appropriate values (optional). We recommend using the same resource group for all the Azure resources created in this tutorial to make them easier to delete when you're finished.

The script takes a few minutes to run.

Bash

Azure CLI

```
#!/bin/bash  
# LOCATION: The Azure region. Use the "az account list-locations -o table"  
# command to find a region near you.  
LOCATION='westus'  
# RESOURCE_GROUP_NAME: The resource group name, which can contain underscores,
```

```

hyphens, periods, parenthesis, letters, and numbers.
RESOURCE_GROUP_NAME='msdocs-web-app-rg'
# ACCOUNT_NAME: The Azure Cosmos DB for MongoDB account name, which can contain
lowercase letters, hyphens, and numbers.
ACCOUNT_NAME='msdocs-cosmos-db-account-name'

# Create a resource group
echo "Creating resource group $RESOURCE_GROUP_NAME in $LOCATION..."
az group create --name $RESOURCE_GROUP_NAME --location $LOCATION

# Create a Cosmos account for MongoDB API
echo "Creating $ACCOUNT_NAME. This command may take a while to complete."
az cosmosdb create --name $ACCOUNT_NAME --resource-group $RESOURCE_GROUP_NAME
--kind MongoDB

# Create a MongoDB API database
echo "Creating database restaurants_reviews"
az cosmosdb mongodb database create --account-name $ACCOUNT_NAME --resource-
group $RESOURCE_GROUP_NAME --name restaurants_reviews

# Create a MongoDB API collection
echo "Creating collection restaurants_reviews"
az cosmosdb mongodb collection create --account-name $ACCOUNT_NAME --resource-
group $RESOURCE_GROUP_NAME --database-name restaurants_reviews --name
restaurants_reviews

# Get the connection string for the MongoDB database
echo "Get the connection string for the MongoDB account"
az cosmosdb keys list --name $ACCOUNT_NAME --resource-group
$RESOURCE_GROUP_NAME --type connection-strings

echo "Copy the Primary MongoDB Connection String from the list above"

```

When the script completes, copy the *Primary MongoDB Connection String* from the output of the last command to your clipboard or other location.

Output

```
{
  "connectionStrings": [
    {
      "connectionString": ""mongodb://msdocs-cosmos-
db:pnaMGVtGIRAZHUjsg4GJBCZMBJ0trV4eg2IcZf1TqV...5oONz0WX14Ph0Ha5IeYACDbuVrBPA==@ms
docs-cosmos-db.mongo.cosmos.azure.com:10255/?ssl=true&replicaSet=globaldb&retrywrites=false&maxIdleTimeMS=120000&appName=@msdoc
s-cosmos-db@"",
      "description": "Primary MongoDB Connection String",
      "keyKind": "Primary",
      "type": "MongoDB"
    },
    ...
  ]
}
```

```
]  
}
```

After you complete the previous step, you have an Azure Cosmos DB for MongoDB connection string of the form `mongodb://<server-name>:<password>@<server-name>.mongo.cosmos.azure.com:10255/?ssl=true&<other-parameters>`, a database named `restaurants_reviews`, and a collection named `restaurants_reviews`.

For more information about how to use the Azure CLI to create a Cosmos DB for MongoDB account and to create databases and collections, see [Create a database and collection for MongoDB for Azure Cosmos DB using Azure CLI](#). You can also use [PowerShell](#), the VS Code [Azure Databases extension](#), and [Azure portal](#).

💡 Tip

In the VS Code Azure Databases extension, you can right-click on the MongoDB server and get the connection string.

Run the image locally in a container

You're now ready to run the Docker container locally using either your local MongoDB instance or your Cosmos DB for MongoDB instance. In this section of the tutorial, you learn to use either VS Code or the Azure CLI to run the image locally. The sample app expects the MongoDB connection information to be passed in to it with environment variables. There are several ways to get environment variables passed to container locally. Each has advantages and disadvantages in terms of security. You should avoid checking in any sensitive information or leaving sensitive information in code in the container.

ⓘ Note

When the web app is deployed to Azure, the web app gets connection information from environment values set as App Service configuration settings and none of the modifications for the local development environment scenario apply.

Azure CLI

MongoDB local

Use the following commands with your local instance of MongoDB to run the Docker image locally.

1. Run the latest version of the image.

```
Bash
```

```
Bash
```

```
#!/bin/bash

# Define variables
# Set the port number based on the framework being used:
# 8000 for Django, 5000 for Flask
export PORT=<port-number> # Replace with actual port (e.g., 8000 or 5000)

# Set your computer's IP address (replace with actual IP)
export YOUR_IP_ADDRESS=<your-computer-ip-address> # Replace with actual IP address

# Run the Docker container with the required environment variables
docker run --rm -it \
    --publish "$PORT:$PORT" \
    --publish 27017:27017 \
    --add-host "mongoservice:$YOUR_IP_ADDRESS" \
    --env CONNECTION_STRING=mongodb://mongoservice:27017 \
    --env DB_NAME=restaurants_reviews \
    --env COLLECTION_NAME=restaurants_reviews \
    --env SECRET_KEY="supersecretkeythatispassedtopythonapp" \
    msdocspythoncontainerwebapp:latest
```

2. Confirm that the container is running. In another console window, run the [docker container ls](#) command.

```
Console
```

```
docker container ls
```

See your container "msdocspythoncontainerwebapp:latest:latest" in the list. Notice the `NAMES` column of the output and the `PORTS` column. Use the container name to stop the container.

3. Test the web app.

Go to "http://127.0.0.1:8000" for Django and "http://127.0.0.1:5000/" for Flask.

4. Shut down the container.

```
Console
```

```
docker container stop <container-name>
```

Azure Cosmos DB for MongoDB

Use the following commands with your Azure Cosmos DB for MongoDB instance to run the Docker image in Azure.

1. Run the latest version of the image.

```
Bash
```

```
Bash
```

```
#!/bin/bash
# PORT=8000 for Django and 5000 for Flask
export PORT=<port-number>
export CONNECTION_STRING="<connection-string>

docker run --rm -it \
    --publish $PORT:$PORT/tcp \
    --env CONNECTION_STRING=$CONNECTION_STRING \
    --env DB_NAME=restaurants_reviews \
    --env COLLECTION_NAME=restaurants_reviews \
    --env SECRET_KEY=supersecretkeythatyougenerate \
    msdocspythoncontainerwebapp:latest
```

Passing in sensitive information is only shown for demonstration purposes. The connection string information can be viewed by inspecting the container with the command [docker container inspect](#). Another way to handle secrets is to use the [BuildKit](#) functionality of Docker.

2. Open a new console window, run the following [docker container ls](#) command to confirm that the container is running.

```
Console
```

```
docker container ls
```

See your container "msdocspythoncontainerwebapp:latest:latest" in the list. Notice the `NAMES` column of the output and the `PORTS` column. Use the container name to

stop the container.

3. Test the web app.

Go to "http://127.0.0.1:8000" for Django and "http://127.0.0.1:5000/" for Flask.

4. Shut down the container.

Console

```
docker container stop <container-name>
```

You can also start a container from an image and stop it with the [Docker Desktop ↗](#) application.

Next step

[Build a container image in Azure](#)

Build a containerized Python web app in Azure

Article • 04/17/2025

In this part of the tutorial series, you learn how to build a containerized Python web app directly in [Azure Container Registry](#) without installing Docker locally. Building the Docker image in Azure is often faster and easier than creating the image locally and then pushing it to the Azure Container Registry. Additionally, cloud-based image building eliminates the need for Docker to run in your development environment.

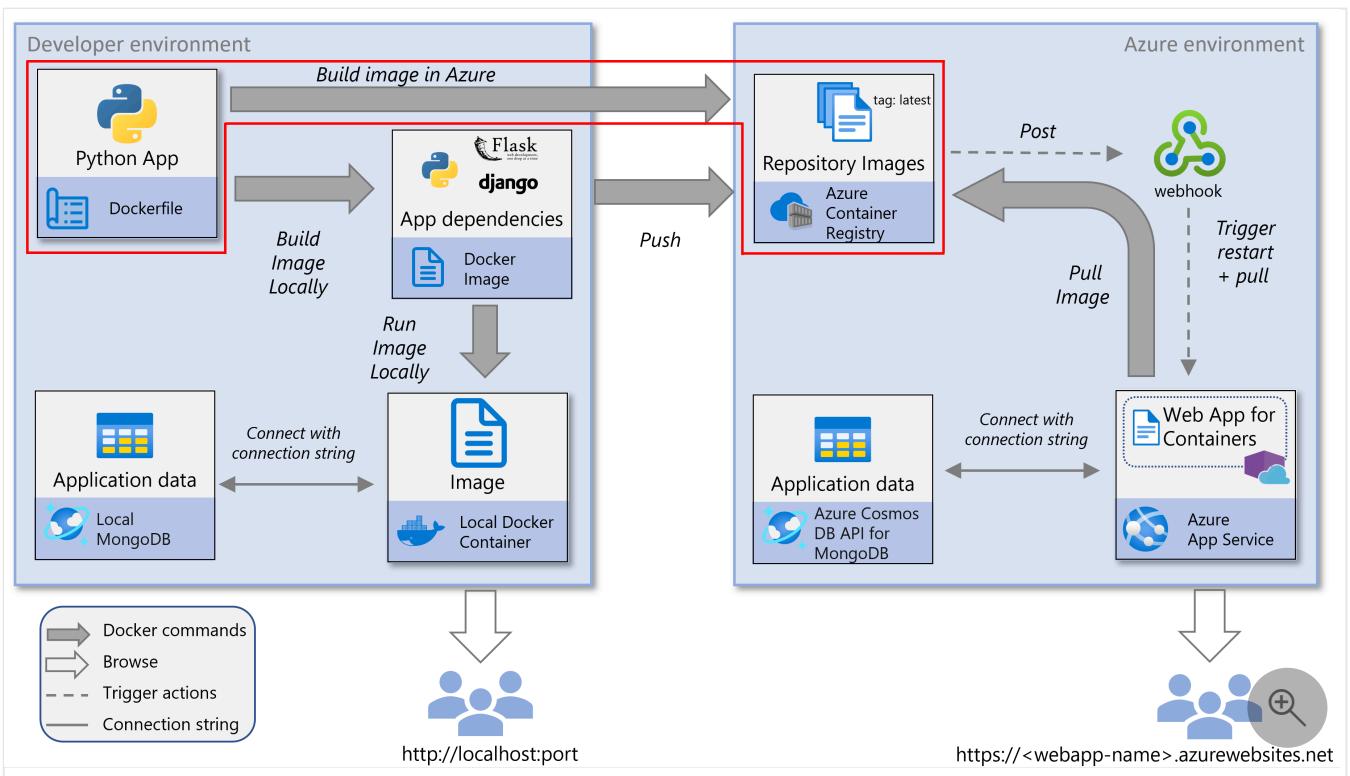
App Service enables you to run containerized web apps and deploy them through the continuous integration/continuous deployment (CI/CD) capabilities of Docker Hub, Azure Container Registry, and Visual Studio Team Services. This article is part 3 of a 5-part tutorial series about how to containerize and deploy a Python web app to Azure App Service. In this part of the tutorial, you learn how to build the containerized Python web app in Azure.

Azure App Service lets you deploy and run containerized web apps using CI/CD pipelines from platforms like Docker Hub, Azure Container Registry, and Azure DevOps. This article is part 3 of a 5-part tutorial series.

In [part 2 of this tutorial](#) series, you built and ran the container image locally. In contrast, in this part of the tutorial, you build (containerize) the same Python web app directly into a Docker image in the [Azure Container Registry](#). Building the image in Azure is typically faster and easier than building locally and then pushing the image to a registry. Also, building in the cloud doesn't require Docker to be running in your dev environment.

Once the Docker image is in Azure Container Registry, it can be deployed to Azure App service.

This service diagram highlights the components covered in this article.



Azure CLI

Create an Azure Container Registry

If you have an existing Azure Container Registry you wish to use, skip this next step and proceed to the next step. Otherwise, create a new Azure Container Registry using the Azure CLI.

Azure CLI commands can be run in the [Azure Cloud Shell](#) or in your local development environment with the [Azure CLI installed](#).

! Note

Use the same names as in part 2 of this tutorial series.

1. Create an Azure container registry with the `az acr create` command.

Bash

Azure CLI

```
#!/bin/bash
# Use the resource group that you created in part 2 of this tutorial
series.
```

```
RESOURCE_GROUP_NAME='msdocs-web-app-rg'
# REGISTRY_NAME must be unique within Azure and contain 5-50
# alphanumeric characters.
REGISTRY_NAME='msdocscontainerregistryname'

echo "Creating Azure Container Registry $REGISTRY_NAME..."
az acr create -g $RESOURCE_GROUP_NAME -n $REGISTRY_NAME --sku
Standard
```

In the JSON output of the command, locate the `loginServer` value. This value represents the fully qualified registry name (all lowercase) and contains the registry name.

2. If you're using the Azure CLI on your local machine, execute the `az acr login` command to log in to the container registry.

Azure CLI

```
az acr login -n $REGISTRY_NAME
```

The command adds "azurecr.io" to the name to create the fully qualified registry name. If successful, you see the message "Login Succeeded".

 **Note**

In the Azure Cloud Shell, the `az acr login` command isn't necessary, as authentication is handled automatically through your Cloud Shell session. However, if you encounter authentication issues, you can still use it.

Build an image in Azure Container Registry

You can generate the container image directly in Azure through various approaches:

- The Azure Cloud Shell allows you to construct the image entirely in the cloud, independent of your local environment.
- Alternatively, you can use VS Code or the Azure CLI to create it in Azure from your local setup, without needing Docker to be running locally.

Azure CLI commands can be run in your local development environment with the [Azure CLI installed](#) or in [Azure Cloud Shell](#).

1. In the console, navigate to the root folder for your cloned repository from part 2 of this tutorial series.
2. Build the container image using the `az acr build` command.

Azure CLI

```
az acr build -r $REGISTRY_NAME -g $RESOURCE_GROUP_NAME -t  
msdocspythoncontainerwebapp:latest .  
# When using Azure Cloud Shell, run one of the following commands  
instead:  
# az acr build -r $REGISTRY_NAME -g $RESOURCE_GROUP_NAME -t  
msdocspythoncontainerwebapp:latest https://github.com/Azure-  
Samples/msdocs-python-django-container-web-app.git  
# az acr build -r $REGISTRY_NAME -g $RESOURCE_GROUP_NAME -t  
msdocspythoncontainerwebapp:latest https://github.com/Azure-  
Samples/msdocs-python-flask-container-web-app.git
```

The last argument in the command is the fully qualified path to the repo. When running in Azure Cloud Shell, use <https://github.com/Azure-Samples/msdocs-python-django-container-web-app.git> ↗ for the Django sample app and <https://github.com/Azure-Samples/msdocs-python-flask-container-web-app.git> ↗ for the Flask sample app.

3. Confirm the container image was created with the `az acr repository list` command.

Azure CLI

```
az acr repository list -n $REGISTRY_NAME
```

Next step

[Deploy web app](#)

Deploy a containerized Python app to App Service

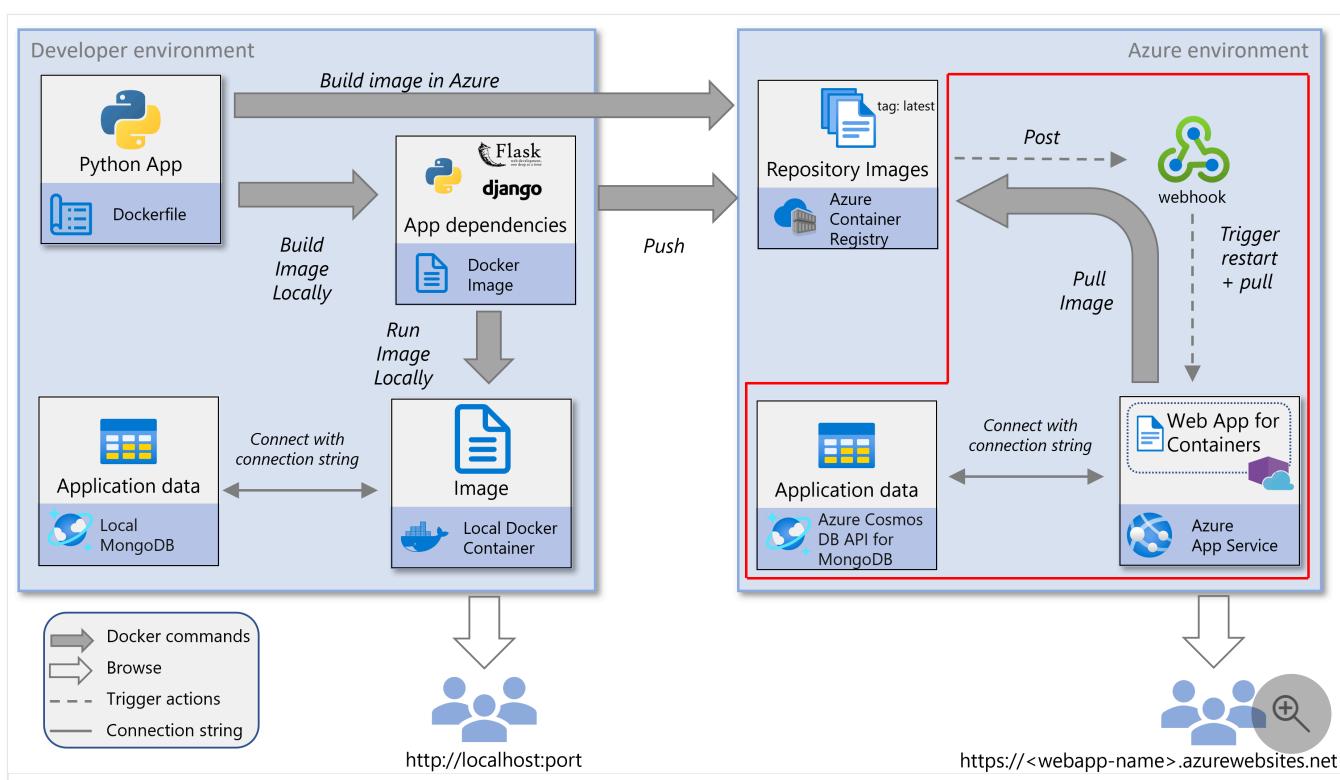
05/21/2025

In this part of the tutorial series, you learn how to deploy a containerized Python web application to [Azure App Service Web App for Containers](#). This fully managed service lets you run containerized apps without having to maintain your own container orchestrator.

App Service simplifies deployment through continuous integration/continuous deployment (CI/CD) pipelines that work with Docker Hub, Azure Container Registry, Azure Key Vault, and other DevOps tools. This tutorial is part 4 of a 5-part tutorial series.

At the end of this article, you have a secure, production-ready App Service web app running from a Docker container image. The app uses a **system-assigned managed identity** to pull the image from Azure Container Registry and retrieve secrets from Azure Key Vault.

This service diagram highlights the components covered in this article.



Azure CLI

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a local machine with the [Azure CLI installed](#).

Important

We recommend using **Azure Cloud Shell** for all CLI-based steps in this tutorial because it:

- Comes pre-authenticated with your Azure account, avoiding login issues
- Includes all required Azure CLI extensions out of the box
- Ensures consistent behavior regardless of your local OS or environment
- Requires no local installation, ideal for users without admin rights
- Provides direct access to Azure services from the portal—no local Docker or network setup required
- Avoids local firewall or network configuration issues

Create Key Vault with RBAC Authorization

Azure Key Vault is a secure service for storing secrets, API keys, connection strings, and certificates. In this script, it stores the **MongoDB connection string** and the web app's **SECRET_KEY**.

The Key Vault is configured to use **role-based access control (RBAC)** to manage access through Azure roles instead of traditional access policies. The web app uses its **system-assigned managed identity** to retrieve secrets securely at runtime.

ⓘ Note

Creating the Key Vault early ensures that roles can be assigned before any attempt to access secrets. It also helps avoid propagation delays in role assignments. Since Key Vault doesn't depend on the App Service, provisioning it early improves reliability and sequencing.

1. In this step, you use the `az keyvault create` command to create an Azure Key Vault with RBAC enabled.

Bash

Azure CLI

```
#!/bin/bash
RESOURCE_GROUP_NAME="msdocs-web-app-rg"
LOCATION="westus"
KEYVAULT_NAME="${RESOURCE_GROUP_NAME}-kv"

az keyvault create \
```

```
--name "$KEYVAULT_NAME" \
--resource-group "$RESOURCE_GROUP_NAME" \
--location "$LOCATION" \
--enable-rbac-authorization true
```

Create the app service plan and web app

The App Service Plan defines the compute resources, pricing tier, and region for your web app. The web app runs your containerized application and is provisioned with a system-assigned managed identity that is used to securely authenticate to Azure Container Registry (ACR) and Azure Key Vault.

In this step, you perform the following tasks:

- Create an App Service Plan
- Create the web app with its managed identity
- Configure the web app to deploy using a specific container image
- Prepare for continuous deployment via ACR

! Note

The web app must be created before assigning access to ACR or Key Vault because the **managed identity is only created at deployment time**. Also, assigning the container image during creation ensures the app starts up correctly with the intended configuration.

1. In this step, you use the [az appservice plan create](#) command to provision the compute environment for your app.

Bash

Azure CLI

```
#!/bin/bash
APP_SERVICE_PLAN_NAME="msdocs-web-app-plan"

az appservice plan create \
--name "$APP_SERVICE_PLAN_NAME" \
--resource-group "$RESOURCE_GROUP_NAME" \
--sku B1 \
--is-linux
```

2. In this step, you use the `az webapp create` command to create the web app. This command also enables a [system-assigned managed identity](#) and sets the container image that the app runs.

Bash

Azure CLI

```
#!/bin/bash
APP_SERVICE_NAME="msdocs-website-name" #APP_SERVICE_NAME must be
globally unique as it becomes the website name in the URL
`https://<website-name>.azurewebsites.net`.
# Use the same registry name as in part 2 of this tutorial series.
REGISTRY_NAME="msdocscontainerregistryname" #REGISTRY_NAME is the
registry name you used in part 2 of this tutorial.
CONTAINER_NAME="$REGISTRY_NAME.azurecr.io/msdocspythoncontainerwebapp
:latest" #CONTAINER_NAME is of the form
"yourregistryname.azurecr.io/repo_name:tag".

az webapp create \
--resource-group "$RESOURCE_GROUP_NAME" \
--plan "$APP_SERVICE_PLAN_NAME" \
--name "$APP_SERVICE_NAME" \
--assign-identity '[system]' \
--deployment-container-image-name "$CONTAINER_NAME"
```

 **Note**

When running this command, you may see the following error:

Output

```
No credential was provided to access Azure Container Registry. Trying
to look up...
Retrieving credentials failed with an exception:'Failed to retrieve
container registry credentials. Please either provide the credentials
or run 'az acr update -n msdocscontainerregistryname --admin-enabled
true' to enable admin first.'
```

This error occurs because the web app tries to use admin credentials to access ACR, which credentials are disabled by default. It's safe to ignore this message: the next step configures the web app to use its managed identity to authenticate with ACR.

Grant Secrets Officer Role to logged-in user

To store secrets in Azure Key Vault, the user running the script must have the **Key Vault Secrets Officer** role. This role allows creating and managing secrets within the vault.

In this step, the script assigns that role to the currently logged-in user. This user can then securely store application secrets, such as the MongoDB connection string and the app's `SECRET_KEY`.

This role assignment is the first of two Key Vault–related role assignments. Later, the web app's system-assigned managed identity is granted access to retrieve secrets from the vault.

Using **Azure RBAC** ensures secure, auditable access based on identity, eliminating the need for hard-coded credentials.

! Note

The user must be assigned the **Key Vault Secrets Officer** role **before** attempting to store any secrets in the key vault. This assignment is done using the [az role assignment create](#) command scoped to the Key Vault.

1. In this step, you use the [az role assignment create](#) command to assign the role at the Key Vault scope.

Bash

Azure CLI

```
#!/bin/bash
CALLER_ID=$(az ad signed-in-user show --query id -o tsv)
echo $CALLER_ID # Verify this value retrieved successfully. In
production, poll to verify this value is retrieved successfully.

az role assignment create \
    --role "Key Vault Secrets Officer" \
    --assignee "$CALLER_ID" \
    --scope "/subscriptions/$(az account show --query id -o
tsv)/resourceGroups/$RESOURCE_GROUP_NAME/providers/Microsoft.KeyVault
/vaults/$KEYVAULT_NAME"
```

Grant web access to ACR using managed identity

To pull images from Azure Container Registry (ACR) securely, the web app must be configured to use its **system-assigned managed identity**. Using managed identity avoids the need for admin credentials and supports secure, credential-free deployment.

This process involves two key actions:

- Enabling the web app to use its managed identity when accessing ACR
 - Assigning the **AcrPull** role to that identity on the target ACR
1. In this step, you retrieve the **principal ID** (unique object ID) of the web app's managed identity using the [az webapp identity show](#) command. Next, you enable the use of the managed identity for ACR authentication by setting the `acrUseManagedIdentityCreds` property to `true` using [az webapp config set](#). You then assign the **AcrPull** role to the web app's managed identity using the [az role assignment create](#) command. This role grants the web app permission to pull images from the registry.

Bash

Azure CLI

```
#!/bin/bash
PRINCIPAL_ID=$(az webapp identity show \
    --name "$APP_SERVICE_NAME" \
    --resource-group "$RESOURCE_GROUP_NAME" \
    --query principalId \
    -o tsv)
echo $PRINCIPAL_ID # Verify this value retrieved successfully. In
# production, poll for successful 'AcrPull' role assignment using `az
role assignment list`.

az webapp config set \
    --resource-group "$RESOURCE_GROUP_NAME" \
    --name "$APP_SERVICE_NAME" \
    --generic-configurations '{"acrUseManagedIdentityCreds": true}'

az role assignment create \
    --role "AcrPull" \
    --assignee "$PRINCIPAL_ID" \
    --scope "/subscriptions/$(az account show --query id -o
tsv)/resourceGroups/$RESOURCE_GROUP_NAME/providers/Microsoft.ContainerRegistry/registries/$REGISTRY_NAME"
```

Grant key vault access to the web app's managed identity

The web app needs permission to access secrets like the MongoDB connection string and the `SECRET_KEY`. To grant these permissions, you must assign the **Key Vault Secrets User** role to the web app's **system-assigned managed identity**.

1. In this step, you use the unique identifier (principal ID) of the web app's system-assigned managed identity to grant the web app access to the Key Vault with the **Key Vault Secrets User** role using the [az role assignment create](#) command.

Bash

Azure CLI

```
#!/bin/bash

az role assignment create \
--role "Key Vault Secrets User" \
--assignee "$PRINCIPAL_ID" \
--scope "/subscriptions/$(az account show --query id -o
tsv)/resourceGroups/$RESOURCE_GROUP_NAME/providers/Microsoft.KeyVault
/vaults/$KEYVAULT_NAME"
```

Store Secrets in Key Vault

To avoid hardcoding secrets in your application, this step stores the **MongoDB connection string** and the web app's **secret key** in Azure Key Vault. These secrets can then be securely accessed by the web app at runtime through its managed identity, without the need to store credentials in code or configuration.

Note

While this tutorial stores only the connection string and secret key in the key vault, you can optionally store other application settings such as the MongoDB database name or collection name in Key Vault as well.

1. In this step, you use the [az cosmosdb keys list](#) command to retrieve the MongoDB connection string. You then use the [az keyvault secret set](#) command to store both the connection string and a randomly generated secret key in Key Vault.

Bash

Azure CLI

```
#!/bin/bash
ACCOUNT_NAME="msdocs-cosmos-db-account-name"

MONGO_CONNECTION_STRING=$(az cosmosdb keys list \
    --name "$ACCOUNT_NAME" \
    --resource-group "$RESOURCE_GROUP_NAME" \
    --type connection-strings \
    --query "connectionStrings[?description=='Primary MongoDB
Connection String'].connectionString" -o tsv)

SECRET_KEY=$(openssl rand -base64 32 | tr -dc 'a-zA-Z0-9')
# This key is cryptographically secure, using OpenSSL's strong random
number generator.

az keyvault secret set \
    --vault-name "$KEYVAULT_NAME" \
    --name "MongoConnectionString" \
    --value "$MONGO_CONNECTION_STRING"

az keyvault secret set \
    --vault-name "$KEYVAULT_NAME" \
    --name "MongoSecretKey" \
    --value "$SECRET_KEY"
```

Configure web app to use Key Vault secrets

To access secrets securely at runtime, the web app must be configured to reference the secrets stored in Azure Key Vault. This step is done using **Key Vault references**, which inject the secret values into the app's environment through its **system-assigned managed identity**.

This approach avoids hardcoding secrets and allows the app to securely retrieve sensitive values like the MongoDB connection string and secret key during execution.

1. In this step, you use the `az webapp config appsettings set` command to add application settings that reference the Key Vault secrets. Specifically, this sets the `MongoConnectionString` and `MongoSecretKey` app settings to reference the corresponding secrets stored in Key Vault.

Bash

Azure CLI

```
#!/bin/bash
MONGODB_NAME="restaurants_reviews"
MONGODB_COLLECTION_NAME="restaurants_reviews"

az webapp config appsettings set \
    --resource-group "$RESOURCE_GROUP_NAME" \
    --name "$APP_SERVICE_NAME" \
    --settings \
        CONNECTION_STRING="@Microsoft.KeyVault(SecretUri=https://$KEYVAULT_NAME.vault.azure.net/secrets/MongoConnectionString)" \
        SECRET_KEY="@Microsoft.KeyVault(SecretUri=https://$KEYVAULT_NAME.vault.azure.net/secrets/MongoSecretKey)" \
        DB_NAME="$MONGODB_NAME" \
        COLLECTION_NAME="$MONGODB_COLLECTION_NAME"
```

Enable continuous deployment from ACR

Enabling continuous deployment allows the web app to automatically pull and run the latest container image whenever one is pushed to Azure Container Registry (ACR). This reduces manual deployment steps and helps ensure your app stays up to date.

ⓘ Note

In the next step, you'll register a webhook in ACR to notify the web app when a new image is pushed.

1. In this step, you use the `az webapp deployment container config` command to enable continuous deployment from ACR to the web app.

Bash

Azure CLI

```
#!/bin/bash
az webapp deployment container config \
    --name "$APP_SERVICE_NAME" \
    --resource-group "$RESOURCE_GROUP_NAME" \
    --enable-cd true
```

Register an ACR Webhook for continuous deployment

To automate deployments, register a webhook in Azure Container Registry (ACR) that notifies the web app whenever a new container image is pushed. The webhook allows the app to automatically pull and run the latest version.

The webhook configured in Azure Container Registry (ACR) sends a POST request to the web app's SCM endpoint (SERVICE_URI) whenever a new image is pushed to the msdocpythoncontainerwebapp repository. This action triggers the web app to pull and deploy the updated image, completing the continuous deployment pipeline between ACR and Azure App Service.

ⓘ Note

The webhook URI must follow this format:

```
https://<app-name>.scm.azurewebsites.net/api/registry/webhook
```

It **must end** with `/api/registry/webhook`. If you receive a URI error, confirm that the path is correct.

1. In this step, use the `az acr webhook create` command to register the webhook and configure it to trigger on `push` events.

Bash

Azure CLI

```
#!/bin/bash
CREDENTIAL=$(az webapp deployment list-publishing-credentials \
    --resource-group "$RESOURCE_GROUP_NAME" \
    --name "$APP_SERVICE_NAME" \
    --query publishingPassword --output tsv)
# Web app publishing credentials may not be available immediately. In
# production, poll until non-empty.

SERVICE_URI="https://$APP_SERVICE_NAME:$CREDENTIAL@$APP_SERVICE_NAME.
scm.azurewebsites.net/api/registry/webhook"

az acr webhook create \
    --name webhookforwebapp \
    --registry "$REGISTRY_NAME" \
    --scope msdocpythoncontainerwebapp:* \
    --uri "$SERVICE_URI" \
```

```
--actions push
```

Browse the site

To verify that the web app is running, open `https://<website-name>.azurewebsites.net`, replacing `<website-name>` with the name of your App Service. You should see the restaurant review sample app. It may take a few moments to load the first time.

Once the site appears, try adding a restaurant and submitting a review to confirm that the app is functioning correctly.

ⓘ Note

The `az webapp browse` command isn't supported in Cloud Shell. If you're using Cloud Shell, manually open a browser and navigate to the site URL.

If you're using the Azure CLI locally, you can use the `az webapp browse` command to open the site in your default browser:

Azure CLI

```
az webapp browse --name $APP_SERVICE_NAME --resource-group  
$RESOURCE_GROUP_NAME
```

ⓘ Note

The `az webapp browse` command isn't supported in Cloud Shell. Open a browser window and navigate to the website URL instead.

Troubleshoot deployment

If you don't see the sample app, try the following steps.

- With container deployment and App Service, always check the **Deployment Center / Logs** page in the Azure portal. Confirm that the container was pulled and is running. The initial pull and running of the container can take a few moments.
- Try to restart the App Service and see if that resolves your issue.

- If there are programming errors, those errors show up in the application logs. On the Azure portal page for the App Service, select **Diagnose and solve problems/Application logs**.
- The sample app relies on a connection to Azure Cosmos DB for MongoDB. Confirm that the App Service has application settings with the correct connection info.
- Confirm that managed identity is enabled for the App Service and is used in the Deployment Center. On the Azure portal page for the App Service, go to the App Service **Deployment Center** resource and confirm that **Authentication** is set to **Managed Identity**.
- Check that the webhook is defined in the Azure Container Registry. The webhook enables the App Service to pull the container image. In particular, check that Service URI ends with "/api/registry/webhook". If not, add it.
- [Different Azure Container Registry skus](#) have different features, including number of webhooks. If you're reusing an existing registry, you could see the message: "Quota exceeded for resource type webhooks for the registry SKU Basic. Learn more about different SKU quotas and upgrade process: <https://aka.ms/acr/tiers>". If you see this message, use a new registry, or reduce the number of [registry webhooks](#) in use.

Next step

[Clean up resources](#)

Containerize tutorial cleanup and next steps

Article • 04/17/2025

In this part of the tutorial series, you learn how to clean up resources used in Azure so you don't incur other charges and help keep your Azure subscription uncluttered.

Clean up resources

At the end of a tutorial or project, it's important to clean up any Azure resources you no longer need. This helps you:

- Avoid unnecessary charges – Resources left running can continue to accrue costs.
- Keep your Azure subscription organized – Removing unused resources makes it easier to manage and navigate your subscription.

In this tutorial, all the Azure resources were created in the same resource group. Removing the resource group removes all resources in the resource group and is the fastest way to remove all Azure resources used for your app.

💡 Tip

If you plan to continue development or testing, you can leave the resources running. Just be aware of potential costs.

Azure CLI

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI installed](#).

Delete the resource group by using the [az group delete](#) command.

You can optionally add the `--no-wait` argument to allow the command to return before the operation is complete.

VS Code

To work with Azure resources in VS Code, you must have the [Azure Tools extension pack](#) installed and be signed into Azure from VS Code.

1. In the Azure view in VS Code (from the Azure Tools extension), expand **RESOURCES** and find your subscription.
2. Make sure the view is set to **Group by Resource Group**.
3. Locate the resource group you want to delete.
4. Right-click the resource group and select **Delete Resource Group**.
5. In the confirmation dialog, enter the exact name of the resource group.
6. Press **ENTER** to confirm and delete the resource group.

Next steps

After completing this tutorial, here are some next steps you can take to build upon what you learned and move the tutorial code and deployment closer to production ready:

- [Deploy a web app from a geo-replicated Azure container registry](#)
- [Review Security in Azure Cosmos DB](#)
- Map a custom DNS name to your app, see [Tutorial: Map custom DNS name to your app](#).
- Monitor App Service for availability, performance, and operation, see [Monitoring App Service](#) and [Set up Azure Monitor for your Python application](#).
- Enable continuous deployment to Azure App Service, see [Continuous deployment to Azure App Service](#), [Use CI/CD to deploy a Python web app to Azure App Service on Linux](#), and [Design a CI/CD pipeline using Azure DevOps](#).
- Create reusable infrastructure as code with [Azure Developer CLI \(azd\)](#).

Related Learn modules

The following are some Learn modules that explore the technologies and themes covered in this tutorial:

- [Introduction to Python](#)
- [Get started with Django](#)
- [Create views and templates in Django](#)

- Create data-driven websites by using the Python framework Django
- Deploy a Django application to Azure by using PostgreSQL
- Get Started with the MongoDB API in Azure Cosmos DB
- Migrate on-premises MongoDB databases to Azure Cosmos DB
- Build a containerized web application with Docker

Deploy a Flask or FastAPI web app on Azure Container Apps

Article • 12/16/2024

This tutorial shows you how to containerize a Python [Flask](#) or [FastAPI](#) web app and deploy it to [Azure Container Apps](#). Azure Container Apps uses [Docker](#) container technology to host both built-in images and custom images. For more information about using containers in Azure, see [Comparing Azure container options](#).

In this tutorial, you use the [Docker CLI](#) and the [Azure CLI](#) to create a Docker image and deploy it to Azure Container Apps. You can also deploy with [Visual Studio Code](#) and the [Azure Tools Extension](#).

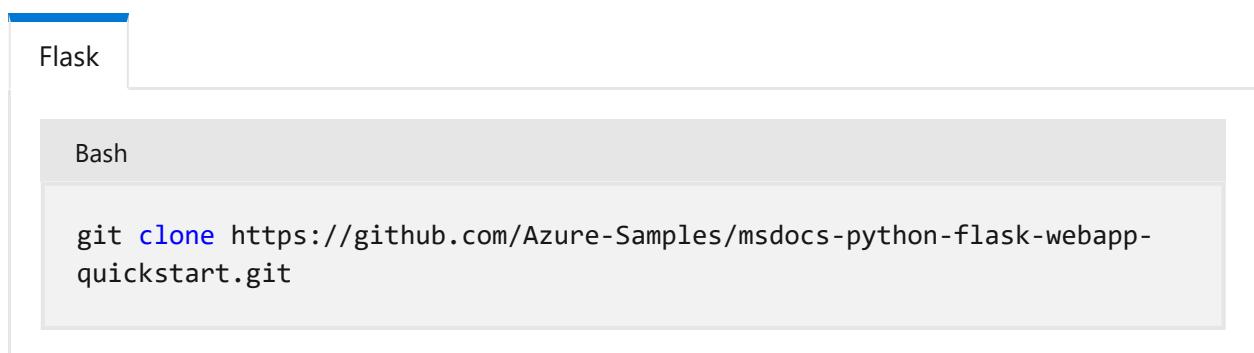
Prerequisites

To complete this tutorial, you need:

- An Azure account where you can deploy a web app to [Azure Container Apps](#). (An [Azure Container Registry](#) and [Log Analytics workspace](#) are created for you in the process.)
- [Azure CLI](#), [Docker](#), and the [Docker CLI](#) installed in your local environment.

Get the sample code

In your local environment, get the code.



```
Flask
Bash
git clone https://github.com/Azure-Samples/msdocs-python-flask-webapp-quickstart.git
```

Add Dockerfile and .dockerignore files

Add a *Dockerfile* to instruct Docker how to build the image. The *Dockerfile* specifies the use of [Gunicorn](#), a production-level web server that forwards web requests to the

Flask and FastAPI frameworks. The `ENTRYPOINT` and `CMD` commands instruct Gunicorn to handle requests for the `app` object.

Flask

Dockerfile

```
# syntax=docker/dockerfile:1

FROM python:3.11

WORKDIR /code

COPY requirements.txt .

RUN pip3 install -r requirements.txt

COPY . .

EXPOSE 50505

ENTRYPOINT ["gunicorn", "app:app"]
```

`50505` is used for the container port (internal) in this example, but you can use any free port.

Check the `requirements.txt` file to make sure it contains `gunicorn`.

Python

```
Flask==2.2.2
gunicorn
Werkzeug==2.2.2
```

Configure gunicorn

Gunicorn can be configured with a `gunicorn.conf.py` file. When the `gunicorn.conf.py` file is located in the same directory where `gunicorn` is run, you don't need to specify its location in the `ENTRYPOINT` or `CMD` instruction of the `Dockerfile`. For more information about specifying the configuration file, see [Gunicorn settings](#).

In this tutorial, the suggested configuration file configures Gunicorn to increase its number of workers based on the number of CPU cores available. For more information about `gunicorn.conf.py` file settings, see [Gunicorn configuration](#).

```
text

# Gunicorn configuration file
import multiprocessing

max_requests = 1000
max_requests_jitter = 50

log_file = "-"

bind = "0.0.0.0:50505"

workers = (multiprocessing.cpu_count() * 2) + 1
threads = workers

timeout = 120
```

Add a `.dockerignore` file to exclude unnecessary files from the image.

```
dockerignore

.git*
**/*.pyc
.venv/
```

Build and run the image locally

Build the image locally.

Flask

Bash

```
docker build --tag flask-demo .
```

Run the image locally in a Docker container.

Flask

Bash

```
docker run --detach --publish 5000:50505 flask-demo
```

Open the `http://localhost:5000` URL in your browser to see the web app running locally.

The `--detach` option runs the container in the background. The `--publish` option maps the container port to a port on the host. The host port (external) is first in the pair, and the container port (internal) is second. For more information, see [Docker run reference ↗](#).

Deploy web app to Azure

To deploy the Docker image to Azure Container Apps, use the `az containerapp up` command. (The following commands are shown for the Bash shell. Change the continuation character (`\`) as appropriate for other shells.)



A screenshot of a terminal window titled "Flask". The title bar is blue. The main area shows the command "az containerapp up" followed by several options: "--resource-group web-flask-aca-rg --name web-aca-app \ --ingress external --target-port 50505 --source .". The "az containerapp up" part is highlighted in blue, indicating it's a command.

When deployment completes, you have a resource group with the following resources inside of it:

- An Azure Container Registry
- A Container Apps Environment
- A Container App running the web app image
- A Log Analytics workspace

The URL for the deployed app is in the output of the `az containerapp up` command. Open the URL in your browser to see the web app running in Azure. The form of the URL will look like the following `https://web-aca-app.<generated-text>.<location-info>.azurecontainerapps.io`, where the `<generated-text>` and `<location-info>` are unique to your deployment.

Make updates and redeploy

After you make code updates, you can run the previous `az containerapp up` command again, which rebuilds the image and redeploys it to Azure Container Apps. Running the

command again takes in account that the resource group and app already exist, and updates just the container app.

In more complex update scenarios, you can redeploy with the [az acr build](#) and [az containerapp update](#) commands together to update the container app.

Clean up

All the Azure resources created in this tutorial are in the same resource group. Removing the resource group removes all resources in the resource group and is the fastest way to remove all Azure resources used for your app.

To remove resources, use the [az group delete](#) command.



```
az group delete --name web-flask-aca-rg
```

You can also remove the group in the [Azure portal](#) or in [Visual Studio Code](#) and the [Azure Tools Extension](#).

Next steps

For more information, see the following resources:

- [Deploy Azure Container Apps with the az containerapp up command](#)
- [Quickstart: Deploy to Azure Container Apps using Visual Studio Code](#)
- [Azure Container Apps image pull with managed identity](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Tutorial: Learn overview concepts for deploying a Python web app on Azure Container Apps

Article • 01/13/2025

This tutorial series shows you how to containerize a Python web app and deploy it to [Azure Container Apps](#). A sample web app is containerized, and the Docker image is stored in [Azure Container Registry](#). Azure Container Apps is configured to pull the Docker image from Container Registry and create a container. The sample app connects to [Azure Database for PostgreSQL](#) to demonstrate communication between Container Apps and other Azure resources.

There are several options to build and deploy cloud-native and containerized Python web apps on Azure. This tutorial series covers Azure Container Apps. Container Apps is good for running general-purpose containers, especially for applications that span many microservices deployed in containers.

In this tutorial series, you create one container. To deploy a Python web app as a container to Azure App Service, see [Containerized Python web app on Azure with MongoDB](#).

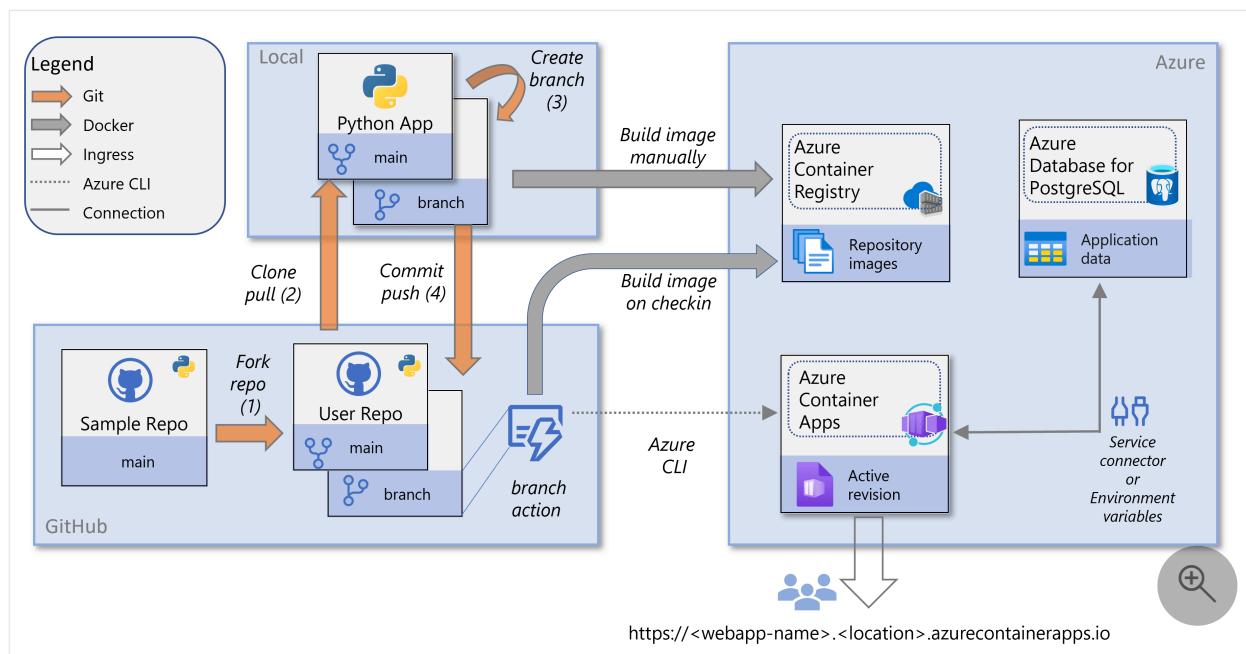
The procedures in this tutorial series guide you to complete these tasks:

- ✓ Build a [Docker](#) image from a Python web app and store the image in [Azure Container Registry](#).
- ✓ Configure [Azure Container Apps](#) to host the Docker image.
- ✓ Set up [GitHub Actions](#) to update the container with a new Docker image triggered by changes to your GitHub repository. *This step is optional.*
- ✓ Set up continuous integration and continuous delivery (CI/CD) of a Python web app to Azure.

In this first part of the series, you learn foundational concepts for deploying a Python web app on Azure Container Apps.

Service overview

The following diagram shows how you'll use your local environment, GitHub repositories, and Azure services in this tutorial series.



The diagram includes these components:

- **Azure Container Apps:**

Azure Container Apps enables you to run microservices and containerized applications on a serverless platform. A serverless platform means that you enjoy the benefits of running containers with minimal configuration. With Azure Container Apps, your applications can dynamically scale based on characteristics such as HTTP traffic, event-driven processing, or CPU or memory load.

Container Apps pulls Docker images from Azure Container Registry. Changes to container images trigger an update to the deployed container. You can also configure GitHub Actions to trigger updates.

- **Azure Container Registry:**

Azure Container Registry enables you to work with Docker images in Azure. Because Container Registry is close to your deployments in Azure, you have control over access. You can use your Microsoft Entra groups and permissions to control access to Docker images.

In this tutorial series, the registry source is Azure Container Registry. But you can also use Docker Hub or a private registry with minor modifications.

- **Azure Database for PostgreSQL:**

The sample code stores application data in a PostgreSQL database. The container app connects to PostgreSQL by using a [user-assigned managed identity](#). Connection information is stored in environment variables configured explicitly or through an [Azure service connector](#).

- [GitHub](#) :

The sample code for this tutorial series is in a GitHub repo that you fork and clone locally. To set up a CI/CD workflow with [GitHub Actions](#), you need a GitHub account.

You can still follow along with this tutorial series without a GitHub account, if you work locally or in [Azure Cloud Shell](#) to build the container image from the sample code repo.

Revisions and CI/CD

To make code changes and push them to a container, you create a new Docker image with your changes. Then, you push the image to Container Registry and create a new [revision](#) of the container app.

To automate this process, an optional step in the tutorial series shows you how to build a CI/CD pipeline by using GitHub Actions. The pipeline automatically builds and deploys your code to Container Apps whenever a new commit is pushed to your GitHub repository.

Authentication and security

In this tutorial series, you build a Docker container image directly in Azure and deploy it to Azure Container Apps. Container Apps runs in the context of an [environment](#), which is supported by an [Azure virtual network](#). Virtual networks are a fundamental building block for your private network in Azure. Container Apps allows you to expose your container app to the public web by enabling ingress.

To set up CI/CD, you authorize Azure Container Apps as an [OAuth app](#) for your GitHub account. As an OAuth app, Container Apps writes a GitHub Actions workflow file to your repo with information about Azure resources and jobs to update them. The workflow updates Azure resources by using the credentials of a Microsoft Entra service principal (or an existing one) with role-based access for Container Apps and a username and password for Azure Container Registry. Credentials are stored securely in your GitHub repo.

Finally, the sample web app in this tutorial series stores data in a PostgreSQL database. The sample code connects to PostgreSQL via a connection string. When the app is running in Azure, it connects to the PostgreSQL database by using a user-assigned managed identity. The code uses [DefaultAzureCredential](#) to dynamically update the password in the connection string with a Microsoft Entra access token during runtime.

This mechanism prevents the need to hardcode the password in the connection string or an environment variable, and it provides an extra layer of security.

The tutorial series walks you through creating the managed identity and granting it an appropriate PostgreSQL role and permissions so that it can access and update the database. During the configuration of Container Apps, the tutorial series walks you through configuring the managed identity on the app and setting up environment variables that contain connection information for the database. You can also use an Azure service connector to accomplish the same thing.

Prerequisites

To complete this tutorial series, you need:

- An Azure account where you can create:
 - An Azure Container Registry instance.
 - An Azure Container Apps environment.
 - An Azure Database for PostgreSQL instance.
- [Visual Studio Code](#) or the [Azure CLI](#), depending on what tool you use:
 - For Visual Studio Code, you need the [Container Apps extension](#).
 - You can use the Azure CLI through [Azure Cloud Shell](#).
- Python packages:
 - [psycopg2-binary](#) for connecting to PostgreSQL.
 - [Flask](#) or [Django](#) as a web framework.

Sample app

The Python sample app is a restaurant review app that saves restaurant and review data in PostgreSQL. At the end of the tutorial series, you'll have a restaurant review app deployed and running in Azure Container Apps that looks like the following screenshot.

Contoso Café

Street address: 1 Main Street

Description: Friendly coffee shop.

Rating:  4.5 (2 reviews)

Reviews

[Add new review](#)

Date	User	Rating	Review	
26/07/2022 14:46:54	Davide Sagese	5	Great cappuccino.	
26/07/2022 14:47:58	Francesca Lombo	4	Healthy breakfast choices.	

Next step

[Build and deploy a Python web app with Azure Container Apps and PostgreSQL](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Tutorial: Build and deploy a Python web app with Azure Container Apps and PostgreSQL

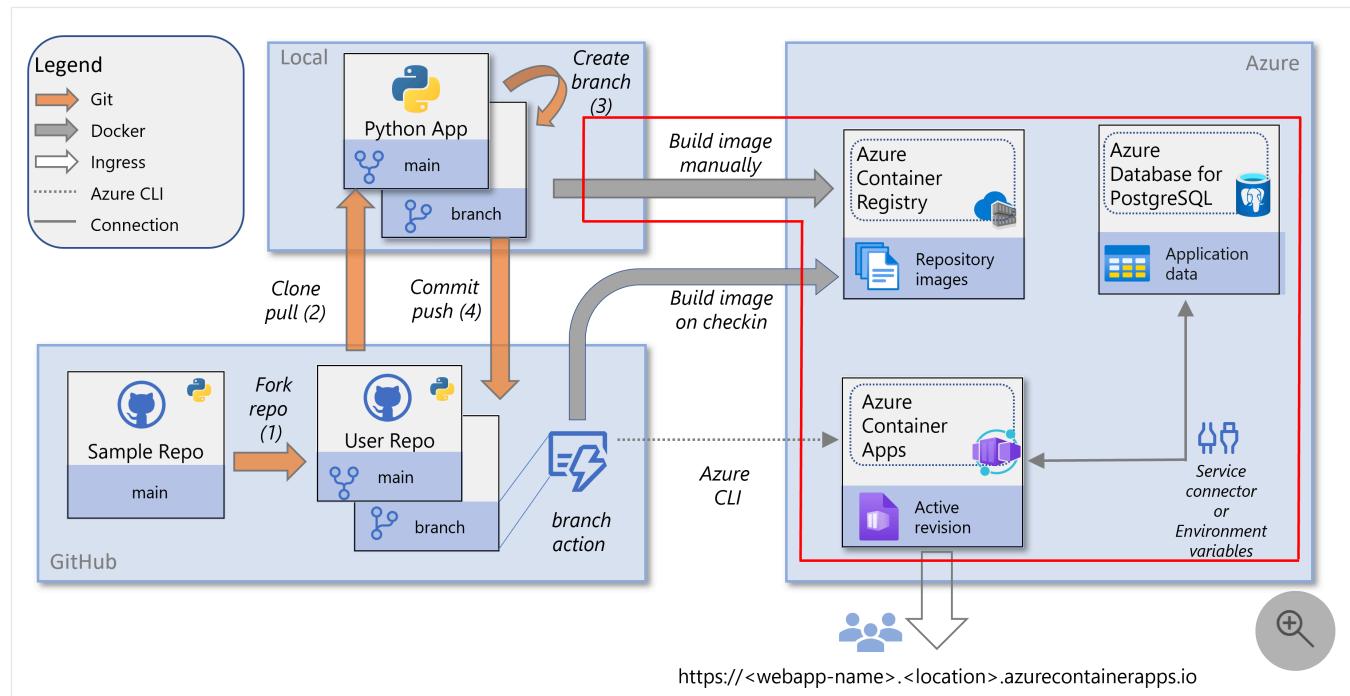
06/18/2025

This article is part of a tutorial series about how to containerize and deploy a Python web app to [Azure Container Apps](#). Container Apps enables you to deploy containerized apps without managing complex infrastructure.

In this tutorial, you:

- ✓ Containerize a Python sample web app (Django or Flask) by building a container image in the cloud.
- ✓ Deploy the container image to Azure Container Apps.
- ✓ Define environment variables that enable the container app to connect to an [Azure Database for PostgreSQL - Flexible Server](#) instance, where the sample app stores data.

The following diagram highlights the tasks in this tutorial: building and deploying a container image.



Prerequisites

If you don't have an Azure subscription, create a [free account](#) before you begin.

Azure CLI

You can run Azure CLI commands in [Azure Cloud Shell](#) or on a workstation with the [Azure CLI](#) installed.

If you're running locally, follow these steps to sign in and install the necessary modules for this tutorial:

1. Sign in to Azure and authenticate, if necessary:

```
Azure CLI
```

```
az login
```

2. Make sure you're running the latest version of the Azure CLI:

```
Azure CLI
```

```
az upgrade
```

3. Install or upgrade the *containerapp* and *rdbms-connect* Azure CLI extensions by using the [az extension add](#) command:

```
Azure CLI
```

```
az extension add --name containerapp --upgrade  
az extension add --name rdbms-connect --upgrade
```

① Note

To list the extensions installed on your system, you can use the [az extension list](#) command. For example:

```
Azure CLI
```

```
az extension list --query [].name --output tsv
```

Get the sample app

Fork and clone the sample code to your developer environment:

1. Go to the GitHub repository of the sample app ([Django](#) or [Flask](#)) and select **Fork**.

Follow the steps to fork the repo to your GitHub account. You can also download the code repo directly to your local machine without forking or a GitHub account. But if you use the download method, you won't be able to set up continuous integration and continuous delivery (CI/CD) in the next tutorial in this series.

2. At the command prompt in your console, use the [git clone](#) command to clone the forked repo into the *python-container* folder:

Console

```
# Django
git clone https://github.com/<github-username>/msdocs-python-django-azure-
container-apps.git python-container

# Flask
# git clone https://github.com/<github-username>/msdocs-python-flask-azure-
container-apps.git python-container
```

3. Change the directory:

Console

```
cd python-container
```

Build a container image from web app code

After you follow these steps, you'll have an Azure Container Registry instance that contains a Docker container image built from the sample code.

Azure CLI

1. If you're running commands in a Git Bash shell on a Windows computer, enter the following command before proceeding:

Azure CLI

```
#!/bin/bash
export MSYS_NO_PATHCONV=1
```

2. Create a resource group by using the [az group create](#) command:

Azure CLI

```
#!/bin/bash
RESOURCE_GROUP_NAME=<resource-group-name>
LOCATION=<location>
az group create \
    --name $RESOURCE_GROUP_NAME \
    --location $LOCATION
```

3. Create a container registry by using the [az acr create](#) command:

Azure CLI

```
#!/bin/bash
REGISTRY_NAME=<registry-name> #The name that you use for *\<registry-name>* must be unique within Azure, and it must contain 5 to 50 alphanumeric characters.
az acr create \
    --resource-group $RESOURCE_GROUP_NAME \
    --name $REGISTRY_NAME \
    --sku Basic \
    --admin-enabled true
```

4. Sign in to the registry by using the [az acr login](#) command:

Azure CLI

```
az acr login --name $REGISTRY_NAME
```

The command adds "azurecr.io" to the name to create the fully qualified registry name. If the sign-in is successful, the message "Login Succeeded" appears. If you're accessing the registry from a subscription that's different from the one in which you created the registry, use the `--suffix` switch.

If sign-in fails, make sure the Docker daemon is running on your system.

5. Build the image by using the [az acr build](#) command:

Azure CLI

```
#!/bin/bash
az acr build \
    --registry $REGISTRY_NAME \
    --resource-group $RESOURCE_GROUP_NAME \
    --image pythoncontainer:latest .
```

These considerations apply:

- The dot (.) at the end of the command indicates the location of the source code to build. If you aren't running this command in the sample app's root directory, specify the path to the code.
- If you're running the command in Azure Cloud Shell, use `git clone` to first pull the repo into the Cloud Shell environment. Then change directory into the root of the project so that the dot (.) is interpreted correctly.
- If you leave out the `-t` (same as `--image`) option, the command queues a local context build without pushing it to the registry. Building without pushing can be useful to check that the image builds.

6. Confirm that the container image was created by using the [az acr repository list](#) command:

Azure CLI

```
az acr repository list --name $REGISTRY_NAME
```

! Note

The steps in this section create a container registry in the Basic service tier. This tier is cost-optimized, with a feature set and throughput targeted for developer scenarios, and is suitable for the requirements of this tutorial. In production scenarios, you would most likely use either the Standard or Premium service tier. These tiers provide enhanced levels of storage and throughput.

To learn more, see [Azure Container Registry service tiers](#). For information about pricing, see [Azure Container Registry pricing](#).

Create a PostgreSQL Flexible Server instance

The sample app ([Django](#) or [Flask](#)) stores restaurant review data in a PostgreSQL database. In these steps, you create the server that will contain the database.

Azure CLI

1. Use the [az postgres flexible-server create](#) command to create the PostgreSQL server in Azure. It isn't uncommon for this command to run for a few minutes before it

finishes.

Azure CLI

```
#!/bin/bash
ADMIN_USERNAME=demoadmin
ADMIN_PASSWORD=<admin-password> # Use a strong password that meets the
                                # requirements for PostgreSQL.
POSTGRES_SERVER_NAME=<postgres-server-name>
az postgres flexible-server create \
    --resource-group $RESOURCE_GROUP_NAME \
    --name $POSTGRES_SERVER_NAME \
    --location $LOCATION \
    --admin-user $ADMIN_USERNAME \
    --admin-password $ADMIN_PASSWORD \
    --version 16 \
    --tier Burstable \
    --sku-name Standard_B1ms \
    --public-access 0.0.0.0 \
    --microsoft-entra-auth Enabled \
    --storage-size 32 \
    --backup-retention 7 \
    --high-availability Disabled \
    --yes
```

Use these values:

- <*postgres-server-name*>: The PostgreSQL database server name. This name must be unique across all of Azure. The server endpoint is `https://<postgres-server-name>.postgres.database.azure.com`. Allowed characters are A to z, 0 to 9, and hyphen (-).
- <*location*>: Use the same location that you used for the web app. <*location*> is one of the Azure location `Name` values from the output of the command `az account list-locations -o table`.
- <*admin-username*>: The username for the administrator account. It can't be `azure_superuser`, `admin`, `administrator`, `root`, `guest`, or `public`. Use `demoadmin` for this tutorial.
- <*admin-password*>: The password of the administrator user. It must contain 8 to 128 characters from three of the following categories: English uppercase letters, English lowercase letters, numbers, and non-alphanumeric characters.

 **Important**

When you're creating usernames or passwords, *do not* use the dollar sign (\$) character. Later, when you create environment variables with these values, that character has a special meaning within the Linux container that you use to run Python apps.

- `--version`: Use `16`. It specifies the PostgreSQL version to use for the server.
- `--tier`: Use `Burstable`. It specifies the pricing tier for the server. The Burstable tier is a lower-cost option for workloads that don't need the full CPU continuously, and is suitable for the requirements of this tutorial.
- `--sku-name`: The name of the pricing tier and compute configuration; for example, `Standard_B1ms`. For more information, see [Azure Database for PostgreSQL pricing](#). To list available tiers, use `az postgres flexible-server list-skus --location <location>`.
- `--public-access`: Use `0.0.0.0`. It allows public access to the server from any Azure service, such as Container Apps.
- `--microsoft-entra-auth`: Use `Enabled`. It enables Microsoft Entra authentication on the server.
- `--storage-size`: Use `32`. It specifies the storage size in gigabytes (GB) for the server. The minimum is 32 GB.
- `--backup-retention`: Use `7`. It specifies the number of days to retain backups for the server. The minimum is 7 days.
- `--high-availability`: Use `Disabled`. It disables high availability for the server. High availability is not required for this tutorial.
- `--yes`: It accepts the terms of use for the PostgreSQL server.

Note

If you plan to work with the PostgreSQL server from your local workstation by using tools, you need to add a firewall rule for your workstation's IP address by using the [`az postgres flexible-server firewall-rule create`](#) command.

2. Use the [`az ad signed-in-user show`](#) command to get the object ID of your user account. You use this ID in the next command.

Azure CLI

```
#!/bin/bash
CALLER_OBJECT_ID=$(az ad signed-in-user show --query id -o tsv)
CALLER_DISPLAY_NAME=$(az ad signed-in-user show --query userPrincipalName
-o tsv)
```

3. Use the [az postgres flexible-server ad-admin create](#) command to add your user account as a Microsoft Entra administrator on the PostgreSQL server:

Azure CLI

```
#!/bin/bash
az postgres flexible-server microsoft-entra-admin create \
--server-name "$POSTGRES_SERVER_NAME" \
--resource-group "$RESOURCE_GROUP_NAME" \
--display-name "$CALLER_DISPLAY_NAME" \
--object-id "$CALLER_OBJECT_ID" \
--type User
```

4. Use the [az postgres flexible-server firewall-rule create](#) command to add a rule that allows your web app to access the PostgreSQL flexible server. In the following command, you configure the server's firewall to accept connections from your development workstation by using your public IP address:

Azure CLI

```
MY_IP=$(curl -s ifconfig.me)
az postgres flexible-server firewall-rule create \
--name "$POSTGRES_SERVER_NAME" \
--resource-group "$RESOURCE_GROUP_NAME" \
--rule-name allow-my-ip \
--start-ip-address "$MY_IP" \
--end-ip-address "$MY_IP"
```
```

#### ⓘ Note

The steps in this section create a PostgreSQL server with a single vCore and limited memory in the **Burstable** pricing tier. The **Burstable** tier is a lower-cost option for workloads that don't need the full CPU continuously, and is suitable for the requirements of this tutorial. For production workloads, you might upgrade to either the **General**

Purpose or Memory Optimized pricing tier. These tiers provide higher performance but increase costs.

To learn more, see [Compute options in Azure Database for PostgreSQL - Flexible Server](#).

For information about pricing, see [Azure Database for PostgreSQL pricing](#).

## Create a database on the server

At this point, you have a PostgreSQL server. In this section, you create a database on the server.

Azure CLI

Use the [az postgres flexible-server db create](#) command to create a database named *restaurants\_reviews*:

```
#!/bin/bash
DATABASE_NAME=restaurants_reviews
az postgres flexible-server db create \
 --resource-group $RESOURCE_GROUP_NAME \
 --server-name $POSTGRES_SERVER_NAME \
 --database-name $DATABASE_NAME
```

You could also use the [az postgres flexible-server connect](#) command to connect to the database and then work with [psql](#) commands. When you're working with psql, it's often easier to use [Azure Cloud Shell](#) because the shell includes all the dependencies for you.

You can also connect to the Azure Database for PostgreSQL flexible server and create a database by using [psql](#) or an IDE that supports PostgreSQL, like [Azure Data Studio](#). For steps using psql, see [Configure the managed identity on the PostgreSQL database](#) later in this article.

## Create a user-assigned managed identity

Create a user-assigned managed identity to use as the identity for the container app when it's running in Azure.

 Note

To create a user-assigned managed identity, your account needs the [Managed Identity Contributor](#) role assignment.

## Azure CLI

Use the [az identity create](#) command to create a user-assigned managed identity:

### Azure CLI

```
UA_MANAGED_IDENTITY_NAME=<managed-identity-name> # Use a unique name for the
managed identity, such as -"my-ua-managed-id".
az identity create \
 --name $UA_MANAGED_IDENTITY_NAME
 --resource-group $RESOURCE_GROUP_NAME
```

## Configure the managed identity on the PostgreSQL database

Configure the managed identity as a role on the PostgreSQL server and then grant it necessary permissions for the *restaurants\_reviews* database. Whether you're using the Azure CLI or psql, you must connect to the Azure PostgreSQL server with a user who's configured as a Microsoft Entra admin on your server instance. Only Microsoft Entra accounts configured as a PostgreSQL admin can configure managed identities and other Microsoft admin roles on your server.

## Azure CLI

1. Get an access token for your Azure account by using the [az account get-access-token](#) command. You use the access token in the next steps.

### Azure CLI

```
#!/bin/bash
MY_ACCESS_TOKEN=$(az account get-access-token --resource-type oss-rdbms -
 -output tsv --query accessToken)
echo $MY_ACCESS_TOKEN
```

2. Add the user-assigned managed identity as database role on your PostgreSQL server by using the [az postgres flexible-server execute](#) command:

### Azure CLI

```
#!/bin/bash
az postgres flexible-server execute \
 --name "$POSTGRES_SERVER_NAME" \
 --admin-user "$CALLER_DISPLAY_NAME" \
 --query "CREATE ROLE $UA_MANAGED_IDENTITY_NAME WITH LOGIN;"
```

```
--admin-password "$ACCESS_TOKEN" \
--database-name postgres \
--querytext "SELECT * FROM
pgadaauth_create_principal('$UA_MANAGED_IDENTITY_NAME', false, false);"
```

### ⓘ Note

If you're running the `az postgres flexible-server execute` command on your local workstation, make sure that you added a firewall rule for your workstation's IP address. You can add a rule by using the [az postgres flexible-server firewall-rule create](#) command. The same requirement also exists for the command in the next step.

3. Grant the user-assigned managed identity the necessary permissions on the *restaurants\_reviews* database by using the following [az postgres flexible-server execute](#) command:

#### Azure CLI

```
#!/bin/bash
SQL_GRANTS=$(cat <<EOF
GRANT CONNECT ON DATABASE $DATABASE_NAME TO "$UA_MANAGED_IDENTITY_NAME";
GRANT USAGE, CREATE ON SCHEMA public TO "$UA_MANAGED_IDENTITY_NAME";
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO
"$UA_MANAGED_IDENTITY_NAME";
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT, INSERT, UPDATE,
DELETE ON TABLES TO "$UA_MANAGED_IDENTITY_NAME";
EOF
)

az postgres flexible-server execute \
--name "$POSTGRES_SERVER_NAME" \
--admin-user "$CALLER_DISPLAY_NAME" \
--admin-password "$MY_ACCESS_TOKEN" \
--database-name "$DATABASE_NAME" \
--querytext "$SQL_GRANTS"
```

This Azure CLI command connects to the *restaurants\_reviews* database on the server and issues the following SQL commands:

#### SQL

```
GRANT CONNECT ON DATABASE restaurants_reviews TO "my-ua-managed-id";
GRANT USAGE ON SCHEMA public TO "my-ua-managed-id";
GRANT CREATE ON SCHEMA public TO "my-ua-managed-id";
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO "my-ua-managed-
```

```
id";
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO "my-ua-managed-id";
```

# Deploy the web app to Container Apps

Container apps are deployed to Azure Container Apps *environments*, which act as a secure boundary. In the following steps, you create the environment and a container inside the environment. You then configure the container so that the website is visible externally.

Azure CLI

These steps require the Azure Container Apps extension, *containerapp*.

1. Create a Container Apps environment by using the [az containerapp env create](#) command:

Azure CLI

```
#!/bin/bash
APP_ENV_NAME=<app-env-name> # Use a unique name for the environment, such
as "python-container-env".
az containerapp env create \
--name python-container-env \
--resource-group $RESOURCE_GROUP_NAME \
--location $LOCATION
```

2. Get the sign-in credentials for the Azure Container Registry instance by using the [az acr credential show](#) command:

Azure CLI

```
#!/bin/bash
REGISTRY_CREDS=$(az acr credential show -n "$REGISTRY_NAME" --query "
[username,passwords[0].value]" -o tsv)
REGISTRY_USERNAME=$(echo "$REGISTRY_CREDS" | head -n1)
REGISTRY_PASSWORD=$(echo "$REGISTRY_CREDS" | tail -n1)
```

You use the username and one of the passwords returned from the command's output when you create the container app in step 5.

3. Use the [az identity show](#) command to get the client ID and resource ID of the user-assigned managed identity:

#### Azure CLI

```
UA_CLIENT_ID=$(az identity show \
 --name "$UA_MANAGED_IDENTITY_NAME" \
 --resource-group "$RESOURCE_GROUP" \
 --query clientId -o tsv)
UA_RESOURCE_ID=$(az identity show \
 --name "$UA_MANAGED_IDENTITY_NAME" \
 --resource-group "$RESOURCE_GROUP" \
 --query id -o tsv)
```

You use the value of the client ID (GUID) and the resource ID from the command's output when you create the container app in step 5. The resource ID has the following form: /subscriptions/<subscription-id>/resourcegroups/pythoncontainer-rg/providers/Microsoft.ManagedIdentity/userAssignedIdentities/my-ua-managed-id.

4. Run the following command to generate a secret key value:

#### Azure CLI

```
AZURE_SECRET_KEY=$(python -c 'import secrets;
print(secrets.token_hex())')
```

You use the secret key value to set an environment variable when you create the container app in step 5.

#### ⚠ Note

The command that this step shows is for a Bash shell. Depending on your environment, you might need to invoke Python by using `python3`. On Windows, you need to enclose the command in the `-c` parameter in double quotation marks rather than single quotation marks. You also might need to invoke Python by using `py` or `py -3`, depending on your environment.

5. Create a container app in the environment by using the `az containerapp create` command:

#### Azure CLI

```
az containerapp create \
 --name "$CONTAINER_APP_NAME" \
 --resource-group "$RESOURCE_GROUP" \
 --environment "$APP_ENV" \
 --image "$REGISTRY_NAME.azurecr.io/$IMAGE_NAME" \
 --target-port "$TARGET_PORT" \
```

```
--ingress external \
--registry-server "$REGISTRY_NAME.azurecr.io" \
--registry-username "$REGISTRY_USERNAME" \
--registry-password "$REGISTRY_PASSWORD" \
--user-assigned "$UA_RESOURCE_ID" \
--env-vars \
 DBHOST="$POSTGRES_SERVER_NAME" \
 DBNAME="$DATABASE_NAME" \
 DBUSER="$UA_MANAGED_IDENTITY_NAME" \
 RUNNING_IN_PRODUCTION=1 \
 AZURE_CLIENT_ID="$UA_CLIENT_ID" \
 AZURE_SECRET_KEY="$AZURE_SECRET_KEY"
```
```

6. For Django only, migrate and create a database schema. (In the Flask sample app, it's done automatically, and you can skip this step.)

Connect by using the `az containerapp exec` command:

Azure CLI

```
az containerapp exec \
  --name $CONTAINER_APP_NAME \
  --resource-group $RESOURCE_GROUP_NAME
```

Then, at the shell command prompt, enter `python manage.py migrate`.

You don't need to migrate for revisions of the container.

7. Test the website.

The `az containerapp create` command that you entered previously outputs an application URL that you can use to browse to the app. The URL ends in `azurecontainerapps.io`. Go to the URL in a browser. Alternatively, you can use the `az containerapp browse` command.

Here's an example of the sample website after the addition of a restaurant and two reviews.

A screenshot of the Azure Restaurant Review application. At the top left is the logo and name "Azure Restaurant Review". At the top right is a three-line menu icon. Below the header is a section titled "Restaurants". A table lists one restaurant: "Contoso Café" with a 4.0 rating from 2 reviews. To the right of the table is a blue "Details" button. Below the table is a green button labeled "Add new restaurant" with a magnifying glass icon. A circular callout highlights the magnifying glass icon.

Troubleshoot deployment

You forgot the application URL to access the website

In the Azure portal:

- Go to the **Overview** page of the container app and look for **Application Url**.

In VS Code:

1. Go to the **Azure view** (Ctrl+Shift+A) and expand the subscription that you're working in.
2. Expand the **Container Apps** node, expand the managed environment, right-click **python-container-app**, and then select **Browse**. VS Code opens the browser with the application URL.

In the Azure CLI:

- Use the command `az containerapp show -g pythoncontainer-rg -n python-container-app --query properties.configuration.ingress.fqdn`.

In VS Code, the Build Image in Azure task returns an error

If you see the message "Error: failed to download context. Please check if the URL is incorrect" in the VS Code **Output** window, refresh the registry in the Docker extension. To refresh, select the Docker extension, go to the **Registries** section, find the registry, and select it.

If you run the **Build Image in Azure** task again, check whether your registry from a previous run exists. If so, use it.

In the Azure portal, an access error appears during the creation of a container app

An access error that contains "Cannot access ACR '<name>.azurecr.io'" occurs when admin credentials on an Azure Container Registry instance are disabled.

To check admin status in the portal, go to your Azure Container Registry instance, select the **Access keys** resource, and ensure that **Admin user** is enabled.

Your container image doesn't appear in the Azure Container Registry instance

- Check the output of the Azure CLI command or VS Code output and look for messages to confirm success.
- Check that the name of the registry was specified correctly in your build command with the Azure CLI or in the VS Code task prompts.
- Make sure your credentials aren't expired. For example, in VS Code, find the target registry in the Docker extension and refresh. In the Azure CLI, run `az login`.

Website returns "Bad Request (400)"

If you get a "Bad Request (400)" error, check the PostgreSQL environment variables passed in to the container. The 400 error often indicates that the Python code can't connect to the PostgreSQL instance.

The sample code used in this tutorial checks for the existence of the container environment variable `RUNNING_IN_PRODUCTION`, which can be set to any value (like `1`).

Website returns "Not Found (404)"

- Check the **Application Url** value on the **Overview** page for the container. If the application URL contains the word "internal," ingress isn't set correctly.
- Check the ingress of the container. For example, in the Azure portal, go to the **Ingress** resource of the container. Make sure that **HTTP Ingress** is enabled and **Accepting traffic from anywhere** is selected.

Website doesn't start, you get "stream timeout," or nothing is returned

- Check the logs:
 - In the Azure portal, go to the container app's revision management resource and check **Provision Status** for the container:
 - If the status is **Provisioning**, wait until provisioning finishes.

- If the status is **Failed**, select the revision and view the console logs. Choose the order of the columns to show **Time Generated**, **Stream_s**, and **Log_s**. Sort the logs by most recent and look for Python `stderr` and `stdout` messages in the **Stream_s** column. Python `print` output is `stdout` messages.
- In the Azure CLI, use the [az containerapp logs show](#) command.
- If you're using the Django framework, check to see if the *restaurants_reviews* tables exist in the database. If not, use a console to access the container and run `python manage.py migrate`.

Next step

[Configure continuous deployment for a Python web app in Azure Container Apps](#)

Tutorial: Configure continuous deployment for a Python web app in Azure Container Apps

06/18/2025

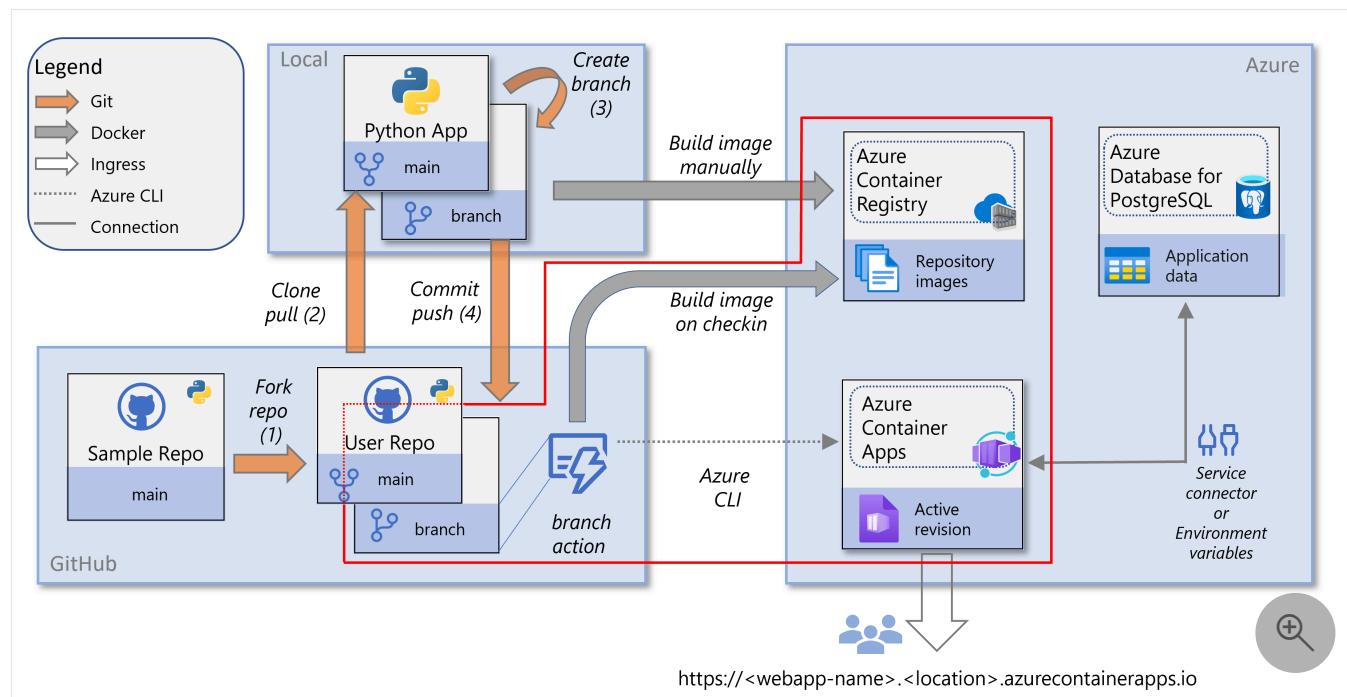
This article is part of a tutorial series about how to containerize and deploy a Python web app to [Azure Container Apps](#). Container Apps enables you to deploy containerized apps without managing complex infrastructure.

In this tutorial, you:

- ✓ Configure continuous deployment for a container app by using a [GitHub Actions](#) workflow.
- ✓ Make a change to a copy of the sample repository to trigger the GitHub Actions workflow.
- ✓ Troubleshoot problems that you might encounter with configuring continuous deployment.
- ✓ Remove resources that you don't need when you finish the tutorial series.

Continuous deployment is related the DevOps practice of continuous integration and continuous delivery (CI/CD), which is automation of your app development workflow.

The following diagram highlights the tasks in this tutorial.



Prerequisites

To set up continuous deployment, you need:

- The resources (and their configuration) that you created in the [previous tutorial](#), which includes an instance of [Azure Container Registry](#) and a container app in [Azure Container Apps](#).
- A GitHub account where you forked the sample code ([Django](#) or [Flask](#)) and that you can connect to from Azure Container Apps. (If you downloaded the sample code instead of forking, be sure to push your local repo to your GitHub account.)
- Optionally, [Git](#) installed in your development environment to make code changes and push to your repo in GitHub. Alternatively, you can make the changes directly in GitHub.

Configure continuous deployment for a container

In the previous tutorial, you created and configured a container app in Azure Container Apps. Part of the configuration was pulling a Docker image from an Azure Container Registry instance. The container image is pulled from the registry when you create a container [revision](#), such as when you first set up the container app.

In this section, you set up continuous deployment by using a GitHub Actions workflow. With continuous deployment, a new Docker image and container revision are created based on a trigger. The trigger in this tutorial is any change to the *main* branch of your repository, such as with a pull request. When the workflow is triggered, it creates a new Docker image, pushes it to the Azure Container Registry instance, and updates the container app to a new revision by using the new image.

Azure CLI

You can run Azure CLI commands in [Azure Cloud Shell](#) or on a workstation where [Azure CLI](#) is installed.

If you're running commands in a Git Bash shell on a Windows computer, enter the following command before proceeding:

Bash

```
export MSYS_NO_PATHCONV=1
```

1. Create a [service principal](#) by using the `az ad sp create-for-rbac` command:

Azure CLI

```
az ad sp create-for-rbac \
--name <app-name> \
--role Contributor \
--scopes "/subscriptions/<subscription-ID>/resourceGroups/<resource-group-name>"
```

In the command:

- <app-name> is an optional display name for the service principal. If you leave off the `--name` option, a GUID is generated as the display name.
- <subscription-ID> is the GUID that uniquely identifies your subscription in Azure. If you don't know your subscription ID, you can run the [az account show](#) command and copy it from the `id` property in the output.
- <resource-group-name> is the name of a resource group that contains the Azure Container Apps container. Role-based access control (RBAC) is on the resource group level. If you followed the steps in the previous tutorial, the name of the resource group is `pythoncontainer-rg`.

Save the output of this command for the next step. In particular, save the client ID (`appId` property), client secret (`password` property), and tenant ID (`tenant` property).

2. Configure GitHub Actions by using the [az containerapp github-action add](#) command:

Azure CLI

```
az containerapp github-action add \
--resource-group <resource-group-name> \
--name python-container-app \
--repo-url <https://github.com/userid/repo> \
--branch main \
--registry-url <registry-name>.azurecr.io \
--service-principal-client-id <client-id> \
--service-principal-tenant-id <tenant-id> \
--service-principal-client-secret <client-secret> \
--login-with-github
```

In the command:

- <resource-group-name> is the name of the resource group. In this tutorial, it's `pythoncontainer-rg`.
- <<https://github.com/userid/repo>> is the URL of your GitHub repository. In this tutorial, it's either `https://github.com/userid/msdocs-python-django-azure-container-apps` or `https://github.com/userid/msdocs-python-flask-azure-container-apps`. In those URLs, `userid` is your GitHub user ID.

- <*registry-name*> is the existing Azure Container Registry instance that you created in the previous tutorial, or one that you can use.
- <*client-id*> is the value of the `appId` property from the previous `az ad sp create-for-rbac` command. The ID is a GUID of the form `00000000-0000-0000-0000-000000000000`.
- <*tenant-id*> is the value of the `tenant` property from the previous `az ad sp create-for-rbac` command. The ID is also a GUID that's similar to the client ID.
- <*client-secret*> is the value of the `password` property from the previous `az ad sp create-for-rbac` command.

In the configuration of continuous deployment, a [service principal](#) enables GitHub Actions to access and modify Azure resources. The roles assigned to the service principal restrict access to resources. The service principal was assigned the built-in [Contributor](#) role on the resource group that contains the container app.

If you followed the steps for the portal, the service principal was automatically created for you. If you followed the steps for the Azure CLI, you explicitly created the service principal before you configured continuous deployment.

Redeploy the web app with GitHub Actions

In this section, you make a change to your forked copy of the sample repository. After that, you can confirm that the change is automatically deployed to the website.

If you haven't already, make a [fork](#) of the sample repository ([Django](#) ↗ or [Flask](#) ↗). You can make your code change directly in [GitHub](#) ↗ or in your development environment from a command line with [Git](#) ↗.

GitHub

1. Go to your fork of the sample repository and start in the *main* branch.

username / msdocs-python-flask-azure-container-app

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main

Go to file Add file Code

A Python/Flask sample web app targeting deployment in Azure Container Apps.

Readme MIT license

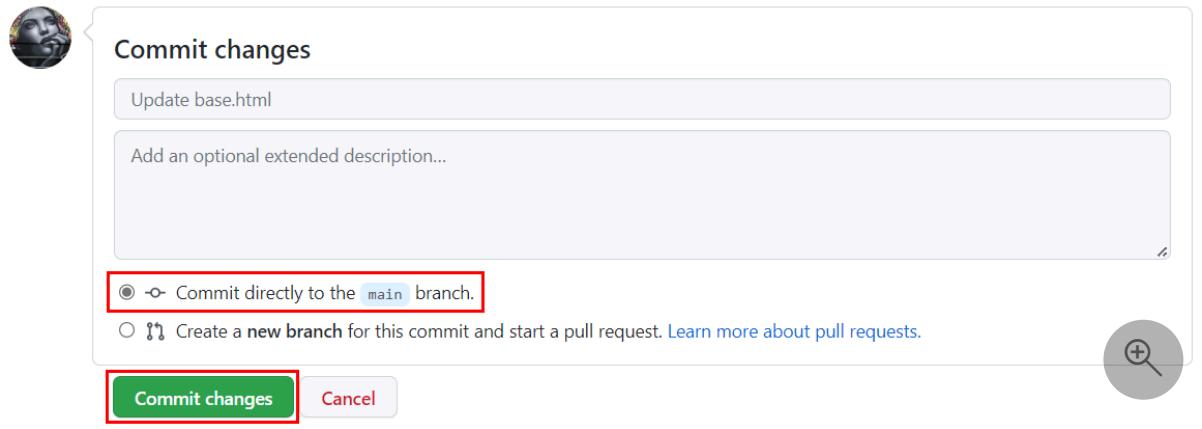
2. Make a change:

- a. Go to the `/templates/base.html` file. (For Django, the path is `restaurant_review/templates/restaurant_review/base.html`.)
 - b. Select **Edit** and change the phrase `Azure Restaurant Review` to `Azure Restaurant Review - Redeployed`.

```
43 lines (42 sloc) | 2.96 KB Raw Blame ⚙️ ⏮ ⌂ ⚡
```

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      {% block head %}
5          <title>Flask web app with PostgreSQL in Azure - {{ block.title }}{% endblock %}</title>
6          <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" integrity="sha384-1TtWZ5P4WLcYqFqZoJzWjE5GqLd0QHvZwWZUxQWZLWZlNQD0N6Q==" crossorigin="anonymous" referrerpolicy="no-referrer"/>
7          <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css" integrity="sha512-9usyDm6QrT1ZfSsO+XgXnqGZD8ew7EeZq5K+tQV9PZj41ZfQwBZJLWZLWZlNQD0N6Q==" crossorigin="anonymous" referrerpolicy="no-referrer"/>
8          <link rel="stylesheet" href="{{ url_for('static', filename='restaurant.css') }}"/>
9          <link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}"/>
10         {% endblock %}
11     </head>
12     <body>
13         <nav class="navbar navbar-expand-md navbar-light fixed-top bg-light">
14             <div class="container">
15                 <a class="navbar-brand" href="/">
16                     
17                     Azure Restaurant Review - Redeployed
18                 </a>
19                 <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse" aria-controls="navba
20                     <span class="navbar-toggler-icon"></span>
21                 </button>
```

3. On the bottom of the page you're editing, make sure that **Commit directly to the main branch** is selected. Then select the **Commit changes** button.



The commit kicks off the GitHub Actions workflow.

ⓘ Note

This tutorial shows making a change directly in the *main* branch. In typical software workflows, you make a change in a branch other than *main* and then create a pull request to merge the change into *main*. Pull requests also kick off the workflow.

Understand GitHub Actions

Viewing workflow history

If you need to view the workflow history, use one of the following procedures.

GitHub

On [GitHub](#), go to your fork of the sample repository and open the **Actions** tab.

The screenshot shows the GitHub Actions interface for a repository named 'msdocs-python-flask-azure-container-app'. The 'Actions' tab is active. A specific workflow run, 'Update base.html', is highlighted with a red box. This run was triggered by a push to the 'main' branch 2 minutes ago and is currently in progress.

Workflow secrets

The `.github/workflows/<workflow-name>.yml` workflow file that was added to the repo includes placeholders for credentials that are needed for the build and container app update jobs of the workflow. The credential information is stored encrypted in the repository's **Settings** area, under **Security > Secrets and variables > Actions**.

The screenshot shows the GitHub Settings page for a repository. The 'Actions' tab is selected in the sidebar. In the main area, the 'Repository secrets' section is displayed, listing three secrets:

Name	Last updated	Actions
PYTHONCONTAINERAPP_AZURE_CREDENTIALS	2 hours ago	
PYTHONCONTAINERAPP_REGISTRY_PASSWORD	2 hours ago	
PYTHONCONTAINERAPP_REGISTRY_USERNAME	2 hours ago	

If credential information changes, you can update it here. For example, if the Azure Container Registry passwords are regenerated, you need to update the `REGISTRY_PASSWORD` value. For more information, see [Encrypted secrets](#) in the GitHub documentation.

OAuth authorized apps

When you set up continuous deployment, you designate Azure Container Apps as an authorized OAuth app for your GitHub account. Container Apps uses the authorized access to create a GitHub Actions YAML file in `.github/workflows/<workflow-name>.yml`. You can view your authorized apps, and revoke permissions in your account, under **Integrations > Applications**.

The screenshot shows the GitHub 'Applications' page under the 'Authorized OAuth Apps' tab. It lists 9 applications with their last used date and owner. Two specific entries are highlighted with red boxes: 'Azure App Service' (last used 2 months ago, owned by AzureAppService) and 'Azure App Service Container Apps' (last used 1 week ago, owned by AzureAppService). A search icon is visible on the right.

Application	Last used	Owner
Azure App Service	Last used within the last 2 months	Owned by AzureAppService
Azure App Service Container Apps	Last used within the last week	Owned by AzureAppService

Troubleshoot

You get errors while setting up a service principal via the Azure CLI

This section can help you troubleshoot errors that you get while setting up a service principal by using the Azure CLI `az ad sp create-for-rba` command.

If you get an error that contains "InvalidSchema: No connection adapters were found":

- Check the shell that you're running in. If you're using a Bash shell, set the `MSYS_NO_PATHCONV` variables as `export MSYS_NO_PATHCONV=1`.

For more information, see the GitHub issue [Unable to create service principal with Azure CLI from Git Bash shell ↗](#).

If you get an error that contains "More than one application have the same display name":

- The name is already taken for the service principal. Choose another name, or leave off the `--name` argument. A GUID will be automatically generated as a display name.

GitHub Actions workflow failed

To check a workflow's status, go to the **Actions** tab of the repo:

- If there's a failed workflow, drill into the workflow file. There should be two jobs: build and deploy. For a failed job, check the output of the job's tasks to look for problems.
- If there's an error message that contains "TLS handshake timeout," run the workflow manually. In the repo, on the **Actions** tab, select **Trigger auto deployment** to see if the timeout is a temporary issue.
- If you set up continuous deployment for the container app as shown in this tutorial, the workflow file (`.github/workflows/<workflow-name>.yml`) is created automatically for you.

You shouldn't need to modify this file for this tutorial. If you did, revert your changes and try the workflow.

Website doesn't show changes that you merged in the main branch

In GitHub:

- Check that the GitHub Actions workflow ran and that you checked the change into the branch that triggers the workflow.

In the Azure portal:

- Check the Azure Container Registry instance to see if a new Docker image was created with a time stamp after your change to the branch.
- Check the logs of the container app to see if there's a programming error:
 1. Go to the container app, and then go to **Revision Management** > *<active container>* > **Revision details** > **Console logs**.
 2. Choose the order of the columns to show **Time Generated**, **Stream_s**, and **Log_s**.
 3. Sort the logs by most recent and look for Python `stderr` and `stdout` messages in the **Stream_s** column. Python `print` output is `stdout` messages.

In the Azure CLI:

- Use the [az containerapp logs show](#) command.

You want to stop continuous deployment

Stopping continuous deployment means disconnecting your container app from your repo.

In the Azure portal:

- Go to the container app. On the service menu, select **Continuous deployment**, and then select **Disconnect**.

In the Azure CLI:

- Use the [az container app github-action remove](#) command.

After you disconnect:

- The `.github/workflows/<workflow-name>.yml` file is removed from your repo.
- Secret keys aren't removed from the repo.

- Azure Container Apps remains as an authorized OAuth app for your GitHub account.
- In Azure, the container is left with the last deployed container. You can reconnect the container app with the Azure Container Registry instance, so that new container revisions pick up the latest image.
- In Azure, service principals that you created and used for continuous deployment aren't deleted.

Remove resources

If you're done with the tutorial series and you don't want to incur extra costs, remove the resources that you used.

Removing a resource group removes all resources in the group and is the fastest way to remove resources. For an example of how to remove resource groups, see [Containerize tutorial cleanup](#).

Related content

If you plan to build on this tutorial, here are some next steps that you can take:

- [Set scaling rules in Azure Container Apps](#)
- [Bind custom domain names and certificates in Azure Container Apps](#)
- [Monitor an app in Azure Container Apps](#)

Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using Azure CLI

08/19/2025



Deploy to Azure



Azure Kubernetes Service (AKS) is a managed Kubernetes service that lets you quickly deploy and manage clusters. In this quickstart, you learn how to:

- Deploy an AKS cluster using the Azure CLI.
- Run a sample multi-container application with a group of microservices and web front ends that simulate a retail scenario.

ⓘ Note

This article includes steps to deploy a cluster with default settings for evaluation purposes only. Before you deploy a production-ready cluster, we recommend that you familiarize yourself with our [baseline reference architecture](#) to consider how it aligns with your business requirements.

Before you begin

This quickstart assumes a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#).

- If you don't have an Azure account, create a [free account](#) before you begin.
- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Get started with Azure Cloud Shell](#).



Launch Cloud Shell



- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Authenticate to Azure using Azure CLI](#).

- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use and manage extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.
- Make sure that the identity you're using to create your cluster has the appropriate minimum permissions. For more information on access and identity for AKS, see [Access and identity options for Azure Kubernetes Service \(AKS\)](#).
 - If you have multiple Azure subscriptions, select the appropriate subscription ID in which the resources should be billed using the `az account set` command. For more information, see [How to manage Azure subscriptions – Azure CLI](#).
 - Dependent upon your Azure subscription, you might need to request a vCPU quota increase. For more information, see [Increase VM-family vCPU quotas](#).

Register resource providers

You might need to register resource providers in your Azure subscription. For example, `Microsoft.ContainerService` is required.

Run the following command to check the registration status.

Azure CLI

```
az provider show --namespace Microsoft.ContainerService --query registrationState
```

If necessary, register the resource provider.

Azure CLI

```
az provider register --namespace Microsoft.ContainerService
```

Define environment variables

Define the following environment variables for use throughout this quickstart.

Azure CLI

```
export RANDOM_ID=$(openssl rand -hex 3)
export MY_RESOURCE_GROUP_NAME="myAKSResourceGroup$RANDOM_ID"
export REGION="westus"
```

```
export MY_AKS_CLUSTER_NAME="myAKSCluster$RANDOM_ID"
export MY_DNS_LABEL="mydnslabel$RANDOM_ID"
```

The `RANDOM_ID` variable's value is a six character alphanumeric value appended to the resource group and cluster name so that the names are unique. Use the `echo` command to view variable values like `echo $RANDOM_ID`.

Create a resource group

An [Azure resource group](#) is a logical group in which Azure resources are deployed and managed. When you create a resource group, you're prompted to specify a location. This location is the storage location of your resource group metadata and where your resources run in Azure if you don't specify another region during resource creation.

Create a resource group using the [az group create](#) command.

Azure CLI

```
az group create --name $MY_RESOURCE_GROUP_NAME --location $REGION
```

The result looks like the following example.

Output

```
{
  "id": "/subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-
eeeeee4e4e/resourceGroups/myAKSResourceGroup<randomIDValue>",
  "location": "westus",
  "managedBy": null,
  "name": "myAKSResourceGroup<randomIDValue>",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": "Microsoft.Resources/resourceGroups"
}
```

Create an AKS cluster

Create an AKS cluster using the [az aks create](#) command. The following example creates a cluster with one node and enables a system-assigned managed identity.

Azure CLI

```
az aks create \
--resource-group $MY_RESOURCE_GROUP_NAME \
--name $MY_AKS_CLUSTER_NAME \
--node-count 1 \
--generate-ssh-keys
```

ⓘ Note

When you create a new cluster, AKS automatically creates a second resource group to store the AKS resources. For more information, see [Why are two resource groups created with AKS?](#)

Connect to the cluster

To manage a Kubernetes cluster, use the Kubernetes command-line client, [kubectl](#). `kubectl` is already installed if you use Azure Cloud Shell. To install `kubectl` locally, use the [az aks install-cli](#) command.

1. Configure `kubectl` to connect to your Kubernetes cluster using the [az aks get-credentials](#) command. This command downloads credentials and configures the Kubernetes CLI to use them.

Azure CLI

```
az aks get-credentials --resource-group $MY_RESOURCE_GROUP_NAME --name $MY_AKS_CLUSTER_NAME
```

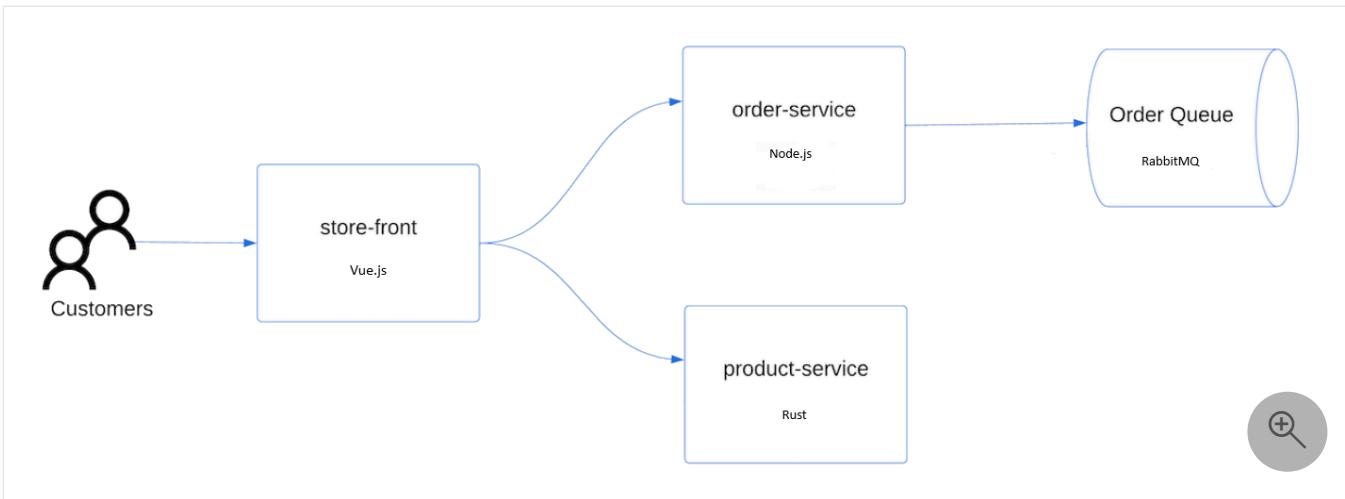
2. Verify the connection to your cluster using the [kubectl get](#) command. This command returns a list of the cluster nodes.

Azure CLI

```
kubectl get nodes
```

Deploy the application

To deploy the application, you use a manifest file to create all the objects required to run the [AKS Store application](#). A Kubernetes manifest file defines a cluster's desired state, such as which container images to run. The manifest includes the following Kubernetes deployments and services:



- Store front: Web application for customers to view products and place orders.
- Product service: Shows product information.
- Order service: Places orders.
- RabbitMQ: Message queue for an order queue.

! Note

We don't recommend running stateful containers, such as RabbitMQ, without persistent storage for production. We use it here for simplicity, but we recommend using managed services, such as Azure CosmosDB or Azure Service Bus.

1. Create a file named `aks-store-quickstart.yaml` and copy in the following manifest.

YAML

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rabbitmq
spec:
  serviceName: rabbitmq
  replicas: 1
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      nodeSelector:
        "kubernetes.io/os": linux
      containers:
        - name: rabbitmq
          image: mcr.microsoft.com/mirror/docker/library/rabbitmq:3.10-
  
```

```

management-alpine
  ports:
    - containerPort: 5672
      name: rabbitmq-amqp
    - containerPort: 15672
      name: rabbitmq-http
  env:
    - name: RABBITMQ_DEFAULT_USER
      value: "username"
    - name: RABBITMQ_DEFAULT_PASS
      value: "password"
  resources:
    requests:
      cpu: 10m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  volumeMounts:
    - name: rabbitmq-enabled-plugins
      mountPath: /etc/rabbitmq/enabled_plugins
      subPath: enabled_plugins
  volumes:
    - name: rabbitmq-enabled-plugins
      configMap:
        name: rabbitmq-enabled-plugins
        items:
          - key: rabbitmq_enabled_plugins
            path: enabled_plugins
  ---
apiVersion: v1
data:
  rabbitmq_enabled_plugins: |
    [rabbitmq_management,rabbitmq_prometheus,rabbitmq_amqp1_0].
kind: ConfigMap
metadata:
  name: rabbitmq-enabled-plugins
  ---
apiVersion: v1
kind: Service
metadata:
  name: rabbitmq
spec:
  selector:
    app: rabbitmq
  ports:
    - name: rabbitmq-amqp
      port: 5672
      targetPort: 5672
    - name: rabbitmq-http
      port: 15672
      targetPort: 15672
  type: ClusterIP
  ---
apiVersion: apps/v1

```

```
kind: Deployment
metadata:
  name: order-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
  spec:
    nodeSelector:
      "kubernetes.io/os": linux
    containers:
      - name: order-service
        image: ghcr.io/azure-samples/aks-store-demo/order-service:latest
        ports:
          - containerPort: 3000
        env:
          - name: ORDER_QUEUE_HOSTNAME
            value: "rabbitmq"
          - name: ORDER_QUEUE_PORT
            value: "5672"
          - name: ORDER_QUEUE_USERNAME
            value: "username"
          - name: ORDER_QUEUE_PASSWORD
            value: "password"
          - name: ORDER_QUEUE_NAME
            value: "orders"
          - name: FASTIFY_ADDRESS
            value: "0.0.0.0"
        resources:
          requests:
            cpu: 1m
            memory: 50Mi
          limits:
            cpu: 75m
            memory: 128Mi
    startupProbe:
      httpGet:
        path: /health
        port: 3000
        failureThreshold: 5
        initialDelaySeconds: 20
        periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /health
        port: 3000
        failureThreshold: 3
        initialDelaySeconds: 3
        periodSeconds: 5
    livenessProbe:
```

```
    httpGet:
      path: /health
      port: 3000
      failureThreshold: 5
      initialDelaySeconds: 3
      periodSeconds: 3
    initContainers:
    - name: wait-for-rabbitmq
      image: busybox
      command: ['sh', '-c', 'until nc -zv rabbitmq 5672; do echo waiting for rabbitmq; sleep 2; done;']
    resources:
      requests:
        cpu: 1m
        memory: 50Mi
      limits:
        cpu: 75m
        memory: 128Mi
    ---
apiVersion: v1
kind: Service
metadata:
  name: order-service
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 3000
    targetPort: 3000
  selector:
    app: order-service
  ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: product-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: product-service
  template:
    metadata:
      labels:
        app: product-service
  spec:
    nodeSelector:
      "kubernetes.io/os": linux
    containers:
    - name: product-service
      image: ghcr.io/azure-samples/aks-store-demo/product-service:latest
      ports:
      - containerPort: 3002
      env:
      - name: AI_SERVICE_URL
```

```
      value: "http://ai-service:5001/"
  resources:
    requests:
      cpu: 1m
      memory: 1Mi
    limits:
      cpu: 2m
      memory: 20Mi
  readinessProbe:
    httpGet:
      path: /health
      port: 3002
    failureThreshold: 3
    initialDelaySeconds: 3
    periodSeconds: 5
  livenessProbe:
    httpGet:
      path: /health
      port: 3002
    failureThreshold: 5
    initialDelaySeconds: 3
    periodSeconds: 3
  ---
apiVersion: v1
kind: Service
metadata:
  name: product-service
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 3002
    targetPort: 3002
  selector:
    app: product-service
  ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: store-front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: store-front
  template:
    metadata:
      labels:
        app: store-front
  spec:
    nodeSelector:
      "kubernetes.io/os": linux
    containers:
    - name: store-front
      image: ghcr.io/azure-samples/aks-store-demo/store-front:latest
```

```

ports:
  - containerPort: 8080
    name: store-front
  env:
    - name: VUE_APP_ORDER_SERVICE_URL
      value: "http://order-service:3000/"
    - name: VUE_APP_PRODUCT_SERVICE_URL
      value: "http://product-service:3002/"
  resources:
    requests:
      cpu: 1m
      memory: 200Mi
    limits:
      cpu: 1000m
      memory: 512Mi
  startupProbe:
    httpGet:
      path: /health
      port: 8080
      failureThreshold: 3
      initialDelaySeconds: 5
      periodSeconds: 5
  readinessProbe:
    httpGet:
      path: /health
      port: 8080
      failureThreshold: 3
      initialDelaySeconds: 3
      periodSeconds: 3
  livenessProbe:
    httpGet:
      path: /health
      port: 8080
      failureThreshold: 5
      initialDelaySeconds: 3
      periodSeconds: 3
  ---
apiVersion: v1
kind: Service
metadata:
  name: store-front
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: store-front
  type: LoadBalancer

```

For a breakdown of YAML manifest files, see [Deployments and YAML manifests](#).

If you create and save the YAML file locally, then you can upload the manifest file to your default directory in Cloud Shell by selecting the **Upload/Download files** button and

selecting the file from your local file system.

2. Deploy the application using the [kubectl apply](#) command and specify the name of your YAML manifest.

```
Azure CLI
```

```
kubectl apply -f aks-store-quickstart.yaml
```

Test the application

You can validate that the application is running by visiting the public IP address or the application URL.

Get the application URL using the following commands:

```
Azure CLI
```

```
runtime="5 minutes"
endtime=$(date -ud "$runtime" +%s)
while [[ $(date -u +%s) -le $endtime ]]
do
    STATUS=$(kubectl get pods -l app=store-front -o 'jsonpath=
{..status.conditions[?(@.type=="Ready")].status}')
    echo $STATUS
    if [ "$STATUS" == 'True' ]
    then
        export IP_ADDRESS=$(kubectl get service store-front --output 'jsonpath=
{..status.loadBalancer.ingress[0].ip}')
        echo "Service IP Address: $IP_ADDRESS"
        break
    else
        sleep 10
    fi
done
```

```
Azure CLI
```

```
curl $IP_ADDRESS
```

Results:

```
HTML
```

```
<!doctype html>
<html lang="">
  <head>
```

```

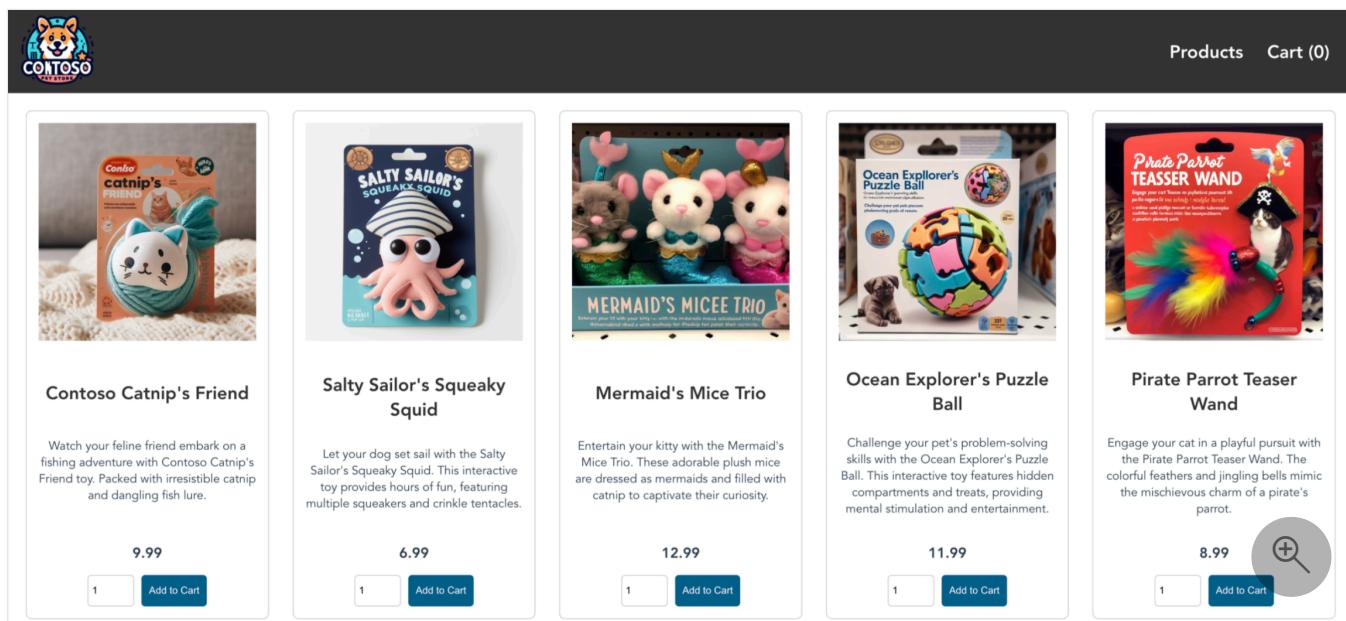
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="icon" href="/favicon.ico">
<title>store-front</title>
<script defer="defer" src="/js/chunk-vendors.df69ae47.js"></script>
<script defer="defer" src="/js/app.7e8cfbb2.js"></script>
<link href="/css/app.a5dc49f6.css" rel="stylesheet">
</head>
<body>
  <div id="app"></div>
</body>
</html>

```

Output

```
echo "You can now visit your web server at $IP_ADDRESS"
```

To view the application website, open a browser and enter the IP address. The page looks like the following example.



Delete the cluster

If you don't plan on going through the [AKS tutorial](#), clean up unnecessary resources to avoid Azure billing charges. You can remove the resource group, container service, and all related resources using the [az group delete](#) command.

Azure CLI

```
az group delete --name $MY_RESOURCE_GROUP_NAME
```

The AKS cluster was created with a system-assigned managed identity, which is the default identity option used in this quickstart. The platform manages this identity so you don't need to manually remove it.

Next steps

In this quickstart, you deployed a Kubernetes cluster and then deployed a simple multi-container application to it. This sample application is for demo purposes only and doesn't represent all the best practices for Kubernetes applications. For guidance about how to create full solutions with AKS for production, see [AKS solution guidance](#).

To learn more about AKS and do a complete code-to-deployment example, continue to the Kubernetes cluster tutorial.

[AKS tutorial](#)

Use the Azure libraries (SDK) for Python

Article • 02/06/2025

The open-source Azure libraries for Python simplify provisioning, managing, and using Azure resources from Python application code.

The details you really want to know

- The Azure libraries are how you communicate with Azure services *from* Python code that you run either locally or in the cloud. (Whether you can run Python code within the scope of a particular service depends on whether that service itself currently supports Python.)
- The libraries support [Python 3.8](#) or later. For more information about supported versions of Python, see [Azure SDKs Python version support policy](#). If you're using [PyPy](#), make sure the version you use at least supports the Python version mentioned previously.
- The Azure SDK for Python is composed solely of over 180 individual Python libraries that relate to specific Azure services. There are no other tools in the SDK.
- When you run code locally, authenticating with Azure relies on environment variables as described in [How to authenticate Python apps to Azure services using the Azure SDK for Python](#).
- To install library packages with pip, use `pip install <library_name>` using library names from the [package index](#). To install library packages in conda environments, use `conda install <package_name>` using names from the [Microsoft channel on anaconda.org](#). For more information, see [Install Azure library packages](#).
- There are distinct **management** and **client** libraries (sometimes referred to as "management plane" and "data plane" libraries). Each set serves different purposes and is used by different kinds of code. For more information, see the following sections later in this article:
 - [Create and manage Azure resources with management libraries](#)
 - [Connect to and use Azure resources with client libraries](#)
- Documentation for the libraries is found on the [Azure for Python Reference](#), which is organized by Azure Service, or the [Python API browser](#), which is organized by package name.

- To try the libraries for yourself, we first recommend [setting up your local dev environment](#). Then you can try any of the following standalone examples (in any order): [Example: Create a resource group](#), [Example: Create and use Azure Storage](#), [Example: Create and deploy a web app](#), [Example: Create and query a MySQL database](#), and [Example: Create a virtual machine](#).
- For demonstration videos, see [Introducing the Azure SDK for Python](#) (PyCon 2021) and [Using Azure SDKs to interact with Azure resources](#) (PyCon 2020).

Non-essential but still interesting details

- Because the [Azure CLI](#) is written in Python using the management libraries, anything you can do with Azure CLI commands you can also do from a Python script. That said, the CLI commands provide many helpful features such as performing multiple tasks together, automatically handling asynchronous operations, formatting output like connection strings, and so on. So, using the CLI (or its equivalent, [Azure PowerShell](#)) for automated creation and management scripts can be more convenient than writing the equivalent Python code, unless you want to have a much more exacting degree of control over the process.
- The Azure libraries for Python build on top of the underlying [Azure REST API](#), allowing you to use those APIs through familiar Python paradigms. However, you can always use the REST API directly from Python code, if desired.
- You can find the source code for the Azure libraries on <https://github.com/Azure/azure-sdk-for-python>. As an open-source project, contributions are welcome!
- Although you can use the libraries with interpreters such as IronPython and Jython that we don't test against, you may encounter isolated issues and incompatibilities.
- The source repo for the library API reference documentation resides on <https://github.com/MicrosoftDocs/azure-docs-sdk-python/>.
- Starting in 2019, we updated Azure Python libraries to share common cloud patterns such as authentication protocols, logging, tracing, transport protocols, buffered responses, and retries. The updated libraries adhere to [current Azure SDK guidelines](#).
 - On 31 March 2023, we retired support for Azure SDK libraries that don't conform to current Azure SDK guidelines. While older libraries can still be used beyond 31 March 2023, they'll no longer receive official support and updates

from Microsoft. For more information, see the notice [Update your Azure SDK libraries](#).

- To avoid missing security and performance updates to the Azure SDKs, upgrade to the [latest Azure SDK libraries](#) by 31 March 2023.
- To check which Python libraries are impacted, see [Azure SDK Deprecated Releases for Python](#).
- For details on the guidelines we apply to the libraries, see the [Python Guidelines: Introduction](#).

Create and manage Azure resources with management libraries

The SDK's *management* (or "management plane") libraries, the names of which all begin with `azure-mgmt-`, help you create, configure, and otherwise manage Azure resources from Python scripts. All Azure services have corresponding management libraries. For more information, see [Azure control plane and data plane](#).

With the management libraries, you can write configuration and deployment scripts to perform the same tasks that you can through the [Azure portal](#) or the [Azure CLI](#). (As noted earlier, the Azure CLI is written in Python and uses the management libraries to implement its various commands.)

The following examples illustrate how to use some of the primary management libraries:

- [Create a resource group](#)
- [List resource groups in a subscription](#)
- [Create an Azure Storage account and a Blob storage container](#)
- [Create and deploy a web app to App Service](#)
- [Create and query an Azure MySQL database](#)
- [Create a virtual machine](#)

For details on working with each management library, see the *README.md* or *README.rst* file located in the library's project folder in the [SDK GitHub repository](#). You can also find more code snippets in the [reference documentation](#) and the [Azure Samples](#).

Migrating from older management libraries

If you're migrating code from older versions of the management libraries, see the following details:

- If you use the `ServicePrincipalCredentials` class, see [Authenticate with token credentials](#).
- The names of async APIs have changed as described on [Library usage patterns - asynchronous operations](#). The names of async APIs in newer libraries start with `begin_`. In most cases, the API signature remains the same.

Connect to and use Azure resources with client libraries

The SDK's *client* (or "data plane") libraries help you write Python application code to interact with already-provisioned services. Client libraries exist only for those services that support a client API.

The article, [Example: Use Azure Storage](#), provides a basic illustration of using client library.

Different Azure services also provide examples using these libraries. See the following index pages for other links:

- [App hosting](#)
- [Cognitive Services](#)
- [Data solutions](#)
- [Identity and security](#)
- [Machine learning](#)
- [Messaging and IoT](#)
- [Other services](#)

For details on working with each client library, see the *README.md* or *README.rst* file located in the library's project folder in the [SDK's GitHub repository](#). You can also find more code snippets in the [reference documentation](#) and the [Azure Samples](#).

Get help and connect with the SDK team

- Visit the [Azure libraries for Python documentation](#)
- Post questions to the community on [Stack Overflow](#)
- Open issues against the SDK on [GitHub](#)
- Mention [@AzureSDK](#) on Twitter
- Complete a short survey about the Azure SDK for Python

Next step

We strongly recommend doing a one-time setup of your local development environment so that you can easily use any of the Azure libraries for Python.

[Set up your local dev environment >>>](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Azure libraries for Python usage patterns

Article • 04/22/2025

The Azure SDK for Python is composed of many independent libraries, which are listed on the [Python SDK package index](#).

All the libraries share certain common characteristics and usage patterns, such as installation and the use of inline JSON for object arguments.

Set up your local development environment

If you haven't already, you can set up an environment where you can run this code. Here are some options:

- Configure a Python virtual environment using `venv` or your tool of choice. You can create the virtual environment locally or in [Azure Cloud Shell](#) and run the code there. Be sure to activate the virtual environment to start using it. To install python, see [Install Python](#).

Bash

```
python -m venv .venv
source .venv/bin/activate # Linux or macOS
.venv\Scripts\activate # Windows
```

- Use a [conda environment](#). To install Conda, see [Install Miniconda](#).
- Use a [Dev Container](#) in [Visual Studio Code](#) or [GitHub Codespaces](#).

Library installation

Choose the installation method that corresponds to your Python environment management tool, either pip or conda.

pip

To install a specific library package, use `pip install`:

Windows Command Prompt

```
REM Install the management library for Azure Storage
pip install azure-mgmt-storage
```

```
Windows Command Prompt
```

```
REM Install the client library for Azure Blob Storage  
pip install azure-storage-blob
```

```
Windows Command Prompt
```

```
REM Install the azure identity library for Azure authentication  
pip install azure-identity
```

`pip install` retrieves the latest version of a library in your current Python environment.

You can also use `pip` to uninstall libraries and install specific versions, including preview versions. For more information, see [How to install Azure library packages for Python](#).

Asynchronous operations

Asynchronous libraries

Many client and management libraries provide async versions (`.aio`). The `asyncio` library has been available since Python 3.4, and the `async/await` keywords were introduced in Python 3.5. The `async` versions of the libraries are intended to be used with Python 3.5 and later.

Examples of Azure Python SDK libraries with `async` versions include: [azure.storage.blob.aio](#), [azure.servicebus.aio](#), [azure.mgmt.keyvault.aio](#), and [azure.mgmt.compute.aio](#).

These libraries need an `async` transport such as `aiohttp` to work. The `azure-core` library provides an `async` transport, `AioHttpTransport`, which is used by the `async` libraries, so you may not need to install `aiohttp` separately.

The following code shows how create a python file that demonstrates how to create a client for the `async` version of the Azure Blob Storage library:

```
Python
```

```
credential = DefaultAzureCredential()  
  
async def run():  
  
    async with BlobClient(  
        storage_url,  
        container_name="blob-container-01",  
        blob_name=f"sample-blob-{str(uuid.uuid4())[0:5]}.txt",  
        credential=credential,
```

```
) as blob_client:

    # Open a local file and upload its contents to Blob Storage
    with open("./sample-source.txt", "rb") as data:
        await blob_client.upload_blob(data)
        print(f"Uploaded sample-source.txt to {blob_client.url}")

    # Close credential
    await credential.close()

asyncio.run(run())
```

The full example is on GitHub at [use_blob_auth_async.py](#). For the synchronous version of this code, see [Example: Upload a blob](#).

Long running operations

Some management operations that you invoke (such as

`ComputeManagementClient.virtual_machines.begin_create_or_update` and

`WebAppsClient.web_apps.begin_create_or_update`) return a poller for long running operations,

`LROPoller[<type>]`, where `<type>` is specific to the operation in question.

! Note

You may notice differences in method names in a library depending on its version and whether it's based on `azure.core`. Older libraries that aren't based on `azure.core` typically use names like `create_or_update`. Libraries based on `azure.core` add the `begin_` prefix to method names to better indicate that they are long polling operations. Migrating old code to a newer `azure.core`-based library typically means adding the `begin_` prefix to method names, as most method signatures remain the same.

The `LROPoller` return type means that the operation is asynchronous. Accordingly, you must call that poller's `result` method to wait for the operation to finish and obtain its result.

The following code, taken from [Example: Create and deploy a web app](#), shows an example of using the poller to wait for a result:

Python

```
# Step 3: With the plan in place, provision the web app itself, which is the
# process that can host
# whatever code we want to deploy to it.
```

```
poller = app_service_client.web_apps.begin_create_or_update(RESOURCE_GROUP_NAME,
    WEB_APP_NAME,
    {
        "location": LOCATION,
        "server_farm_id": plan_result.id,
        "site_config": {
            "linux_fx_version": "python|3.8"
        }
    }
)

web_app_result = poller.result()
```

In this case, the return value of `begin_create_or_update` is of type `AzureOperationPoller[Site]`, which means that the return value of `poller.result()` is a `Site` object.

Exceptions

In general, the Azure libraries raise exceptions when operations fail to perform as intended, including failed HTTP requests to the Azure REST API. For app code, you can use `try...except` blocks around library operations.

For more information on the type of exceptions that may be raised, see the documentation for the operation in question.

Logging

The most recent Azure libraries use the Python standard `logging` library to generate log output. You can set the logging level for individual libraries, groups of libraries, or all libraries. Once you register a logging stream handler, you can then enable logging for a specific client object or a specific operation. For more information, see [Logging in the Azure libraries](#).

Proxy configuration

To specify a proxy, you can use environment variables or optional arguments. For more information, see [How to configure proxies](#).

Optional arguments for client objects and methods

In the library reference documentation, you often see a `**kwargs` or `**operation_config` argument in the signature of a client object constructor or a specific operation method. These placeholders indicate that the object or method in question may support other named

arguments. Typically, the reference documentation indicates the specific arguments you can use. There are also some general arguments that are often supported as described in the following sections.

Arguments for libraries based on `azure.core`

These arguments apply to those libraries listed on [Python - New Libraries ↗](#). For example, here are a subset of the keyword arguments for `azure-core`. For a complete list, see the GitHub README for [azure-core ↗](#).

[+] Expand table

Name	Type	Default	Description
<code>logging_enable</code>	<code>bool</code>	<code>False</code>	Enables logging. For more information, see Logging in the Azure libraries .
<code>proxies</code>	<code>dict</code>	<code>{}</code>	Proxy server URLs. For more information, see How to configure proxies .
<code>use_env_settings</code>	<code>bool</code>	<code>True</code>	If <code>True</code> , allows use of <code>HTTP_PROXY</code> and <code>HTTPS_PROXY</code> environment variables for proxies. If <code>False</code> , the environment variables are ignored. For more information, see How to configure proxies .
<code>connection_timeout</code>	<code>int</code>	<code>300</code>	The timeout in seconds for making a connection to Azure REST API endpoints.
<code>read_timeout</code>	<code>int</code>	<code>300</code>	The timeout in seconds for completing an Azure REST API operation (that is, waiting for a response).
<code>retry_total</code>	<code>int</code>	<code>10</code>	The number of allowable retry attempts for REST API calls. Use <code>retry_total=0</code> to disable retries.
<code>retry_mode</code>	<code>enum</code>	<code>exponential</code>	Applies retry timing in a linear or exponential manner. If 'single', retries are made at regular intervals. If 'exponential', each retry waits twice as long as the previous retry.

Individual libraries aren't obligated to support any of these arguments, so always consult the reference documentation for each library for exact details. Also, each library may support other arguments. For example, for blob storage specific keyword arguments, see the GitHub README for [azure-storage-blob ↗](#).

Inline JSON pattern for object arguments

Many operations within the Azure libraries allow you to express object arguments either as discrete objects or as inline JSON.

For example, suppose you have a [ResourceManagementClient](#) object through which you create a resource group with its [create_or_update](#) method. The second argument to this method is of type [ResourceGroup](#).

To call the `create_or_update` method, you can create a discrete instance of [ResourceGroup](#) directly with its required arguments (`location` in this case):

Python

```
# Provision the resource group.
rg_result = resource_client.resource_groups.create_or_update(
    "PythonSDKExample-rg",
    ResourceGroup(location="centralus")
)
```

Alternately, you can pass the same parameters as inline JSON:

Python

```
# Provision the resource group.
rg_result = resource_client.resource_groups.create_or_update(
    "PythonAzureExample-rg", {"location": "centralus"}
)
```

When you use inline JSON, the Azure libraries automatically convert the inline JSON to the appropriate object type for the argument in question.

Objects can also have nested object arguments, in which case you can also use nested JSON.

For example, suppose you have an instance of the [KeyVaultManagementClient](#) object, and are calling its [create_or_update](#). In this case, the third argument is of type [VaultCreateOrUpdateParameters](#), which itself contains an argument of type [VaultProperties](#). [VaultProperties](#), in turn, contains object arguments of type [Sku](#) and [list\[AccessPolicyEntry\]](#). A [Sku](#) contains a [SkuName](#) object, and each [AccessPolicyEntry](#) contains a [Permissions](#) object.

To call `begin_create_or_update` with embedded objects, you use code like the following (assuming `tenant_id`, `object_id`, and `LOCATION` are already defined). You can also create the necessary objects before the function call.

Python

```

# Provision a Key Vault using inline parameters
poller = keyvault_client.vaults.begin_create_or_update(
    RESOURCE_GROUP_NAME,
    KEY_VAULT_NAME_A,
    VaultCreateOrUpdateParameters(
        location = LOCATION,
        properties = VaultProperties(
            tenant_id = tenant_id,
            sku = Sku(
                name="standard",
                family="A"
            ),
            access_policies = [
                AccessPolicyEntry(
                    tenant_id = tenant_id,
                    object_id = object_id,
                    permissions = Permissions(
                        keys = ['all'],
                        secrets = ['all']
                    )
                )
            ]
        )
    )
)

key_vault1 = poller.result()

```

The same call using inline JSON appears as follows:

Python

```

# Provision a Key Vault using inline JSON
poller = keyvault_client.vaults.begin_create_or_update(
    RESOURCE_GROUP_NAME,
    KEY_VAULT_NAME_B,
    {
        'location': LOCATION,
        'properties': {
            'sku': {
                'name': 'standard',
                'family': 'A'
            },
            'tenant_id': tenant_id,
            'access_policies': [
                {
                    'tenant_id': tenant_id,
                    'object_id': object_id,
                    'permissions': {
                        'keys': ['all'],
                        'secrets': ['all']
                    }
                }
            ]
        }
    }
)

```

```
        }
    }

key_vault2 = poller.result()
```

Because both forms are equivalent, you can choose whichever you prefer and even intermix them. (The full code for these examples can be found on [GitHub](#).)

If your JSON isn't formed properly, you typically get the error, "DeserializationError: Unable to deserialize to object: type, AttributeError: 'str' object has no attribute 'get'". A common cause of this error is that you're providing a single string for a property when the library expects a nested JSON object. For example, using `'sku': 'standard'` in the previous example generates this error because the `sku` parameter is a `Sku` object that expects inline object JSON, in this case `{'name': 'standard'}`, which maps to the expected `SkuName` type.

Next steps

Now that you understand the common patterns for using the Azure libraries for Python, see the following standalone examples to explore specific management and client library scenarios. You can try these examples in any order as they're not sequential or interdependent.

- [Example: Create a resource group](#)
- [Example: Use Azure Storage](#)
- [Example: Create a web app and deploy code](#)
- [Example: Create and query a database](#)
- [Example: Create a virtual machine](#)
- [Use Azure Managed Disks with virtual machines](#)
- [Complete a short survey about the Azure SDK for Python](#)

Authenticate Python apps to Azure services by using the Azure SDK for Python

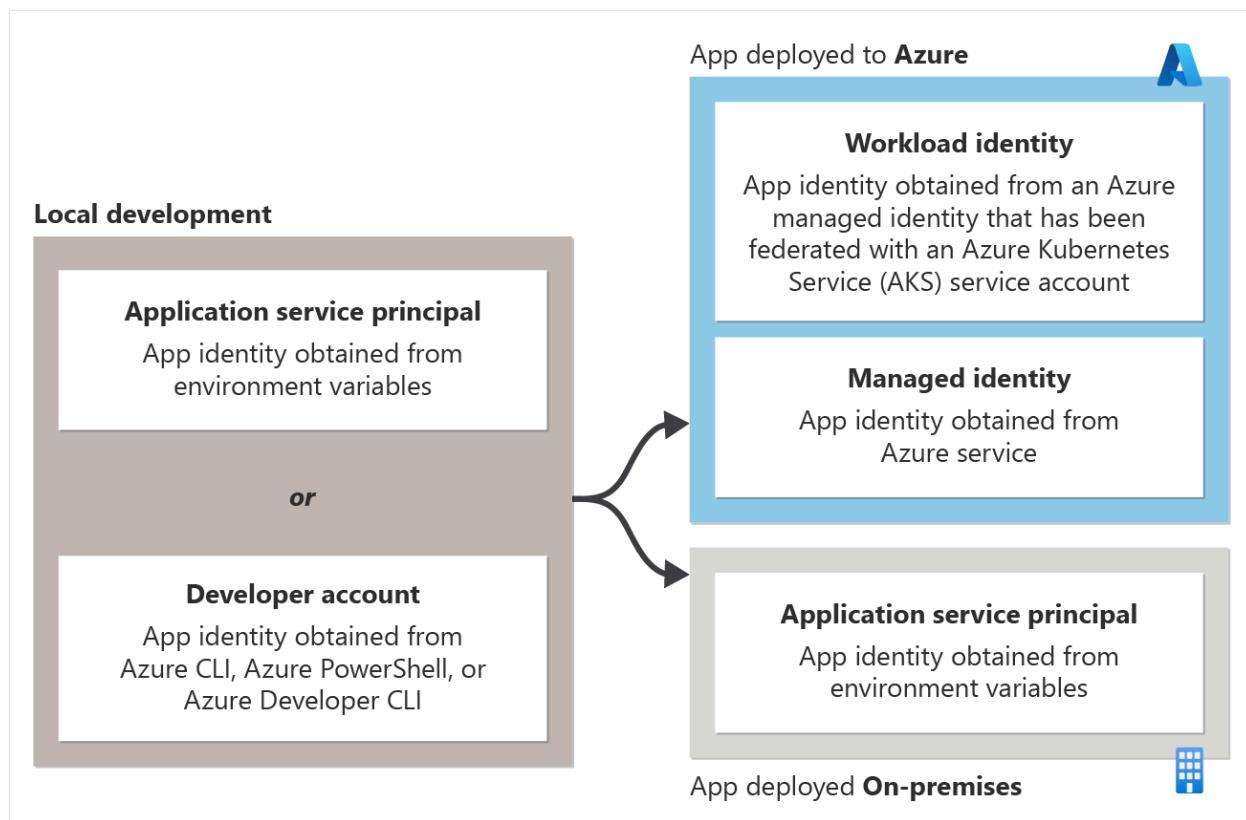
Article • 09/20/2024

When an app needs to access an Azure resource like Azure Storage, Azure Key Vault, or Azure AI services, the app must be authenticated to Azure. This requirement is true for all apps, whether they're deployed to Azure, deployed on-premises, or under development on a local developer workstation. This article describes the recommended approaches to authenticate an app to Azure when you use the Azure SDK for Python.

Recommended app authentication approach

Use token-based authentication rather than connection strings for your apps when they authenticate to Azure resources. The [Azure Identity client library for Python](#) provides classes that support token-based authentication and allow apps to seamlessly authenticate to Azure resources whether the app is in local development, deployed to Azure, or deployed to an on-premises server.

The specific type of token-based authentication an app uses to authenticate to Azure resources depends on where the app is being run. The types of token-based authentication are shown in the following diagram.



- **When a developer is running an app during local development:** The app authenticates to Azure by using either an application service principal for local development or the developer's Azure credentials. These options are discussed in the section [Authentication during local development](#).
- **When an app is hosted on Azure:** The app authenticates to Azure resources by using a managed identity. This option is discussed in the section [Authentication in server environments](#).
- **When an app is hosted and deployed on-premises:** The app authenticates to Azure resources by using an application service principal. This option is discussed in the section [Authentication in server environments](#).

DefaultAzureCredential

The `DefaultAzureCredential` class provided by the Azure Identity client library allows apps to use different authentication methods depending on the environment in which they're run. In this way, apps can be promoted from local development to test environments to production without code changes.

You configure the appropriate authentication method for each environment, and `DefaultAzureCredential` automatically detects and uses that authentication method. The use of `DefaultAzureCredential` is preferred over manually coding conditional logic or feature flags to use different authentication methods in different environments.

Details about using the `DefaultAzureCredential` class are discussed in the section [Use DefaultAzureCredential in an application](#).

Advantages of token-based authentication

Use token-based authentication instead of using connection strings when you build apps for Azure. Token-based authentication offers the following advantages over authenticating with connection strings:

- The token-based authentication methods described in this article allow you to establish the specific permissions needed by the app on the Azure resource. This practice follows the [principle of least privilege](#). In contrast, a connection string grants full rights to the Azure resource.
- Anyone or any app with a connection string can connect to an Azure resource, but token-based authentication methods scope access to the resource to only the apps intended to access the resource.
- With a managed identity, there's no application secret to store. The app is more secure because there's no connection string or application secret that can be compromised.
- The [azure-identity](#) package acquires and manages Microsoft Entra tokens for you. This makes using token-based authentication as easy to use as a connection string.

Limit the use of connection strings to initial proof-of-concept apps or development prototypes that don't access production or sensitive data. Otherwise, the token-based authentication classes available in the Azure Identity client library are always preferred when they're authenticating to Azure resources.

Authentication in server environments

When you're hosting in a server environment, each app is assigned a unique *application identity* per environment where the app runs. In Azure, an application identity is represented by a *service principal*. This special type of security principal identifies and authenticates apps to Azure. The type of service principal to use for your app depends on where your app is running:

[] [Expand table](#)

Authentication method	Description
Apps hosted in Azure	<p>Apps hosted in Azure should use a <i>managed identity service principal</i>. Managed identities are designed to represent the identity of an app hosted in Azure and can only be used with Azure-hosted apps.</p> <p>For example, a Django web app hosted in Azure App Service would be assigned a managed identity. The managed identity assigned to the app would then be used to authenticate the app to other Azure services.</p> <p>Apps running in Azure Kubernetes Service (AKS) can use a Workload identity credential. This credential is based on a managed identity that has a trust relationship with an AKS service account.</p> <p><a data-bbox="467 728 1082 766" href="#">Learn about auth from Azure-hosted apps</p>
Apps hosted outside of Azure (for example, on-premises apps)	<p>Apps hosted outside of Azure (for example, on-premises apps) that need to connect to Azure services should use an <i>application service principal</i>. An application service principal represents the identity of the app in Azure and is created through the application registration process.</p> <p>For example, consider a Django web app hosted on-premises that makes use of Azure Blob Storage. You would create an application service principal for the app by using the app registration process. The <code>AZURE_CLIENT_ID</code>, <code>AZURE_TENANT_ID</code>, and <code>AZURE_CLIENT_SECRET</code> would all be stored as environment variables to be read by the application at runtime and allow the app to authenticate to Azure by using the application service principal.</p> <p><a data-bbox="467 1344 1237 1382" href="#">Learn about auth from apps hosted outside of Azure</p>

Authentication during local development

When an app runs on a developer's workstation during local development, it still must authenticate to any Azure services used by the app. There are two main strategies for authenticating apps to Azure during local development:

[\[+\] Expand table](#)

Authentication method	Description
Create dedicated application service principal objects to be used during local development.	<p>In this method, dedicated <i>application service principal</i> objects are set up by using the app registration process for use during local development. The identity of the service principal is then stored as environment variables to be accessed by the app when it's run in local development.</p>
	<p>This method allows you to assign the specific resource permissions needed by the app to the service principal objects used by developers during local development. This practice makes sure the application only has access to the specific resources it needs and replicates the permissions the app will have in production.</p> <p>The downside of this approach is the need to create separate service principal objects for each developer who works on an application.</p>
	<p><a data-bbox="573 811 1332 856" href="#">Learn about auth using developer service principals</p>
Authenticate the app to Azure by using the developer's credentials during local development.	<p>In this method, a developer must be signed in to Azure from the Azure CLI, Azure PowerShell, or Azure Developer CLI on their local workstation. The application then can access the developer's credentials from the credential store and use those credentials to access Azure resources from the app.</p>
	<p>This method has the advantage of easier setup because a developer only needs to sign in to their Azure account through one of the aforementioned developer tools. The disadvantage of this approach is that the developer's account likely has more permissions than required by the application. As a result, the application doesn't accurately replicate the permissions it will run with in production.</p>
	<p><a data-bbox="573 1463 1209 1508" href="#">Learn about auth using developer accounts</p>

Use `DefaultAzureCredential` in an application

[DefaultAzureCredential](#) is an opinionated, ordered sequence of mechanisms for authenticating to Microsoft Entra ID. Each authentication mechanism is a class that implements the [TokenCredential](#) protocol and is known as a *credential*. At runtime, `DefaultAzureCredential` attempts to authenticate using the first credential. If that credential fails to acquire an access token, the next credential in the sequence is attempted, and so on, until an access token is successfully obtained. In this way, your app can use different credentials in different environments without writing environment-specific code.

To use `DefaultAzureCredential` in a Python app, add the [azure-identity](#) package to your application.

terminal

```
pip install azure-identity
```

Azure services are accessed using specialized client classes from the various Azure SDK client libraries. The following code example shows how to instantiate a `DefaultAzureCredential` object and use it with an Azure SDK client class. In this case, it's a `BlobServiceClient` object used to access Azure Blob Storage.

Python

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

# Acquire a credential object
credential = DefaultAzureCredential()

blob_service_client = BlobServiceClient(
    account_url="https://<my_account_name>.blob.core.windows.net",
    credential=credential)
```

When the preceding code runs on your local development workstation, it looks in the environment variables for an application service principal or at locally installed developer tools, such as the Azure CLI, for a set of developer credentials. Either approach can be used to authenticate the app to Azure resources during local development.

When deployed to Azure, this same code can also authenticate your app to Azure resources. `DefaultAzureCredential` can retrieve environment settings and managed identity configurations to authenticate to Azure services automatically.

Related content

- [Azure Identity client library for Python README on GitHub](#)

Feedback

Was this page helpful?

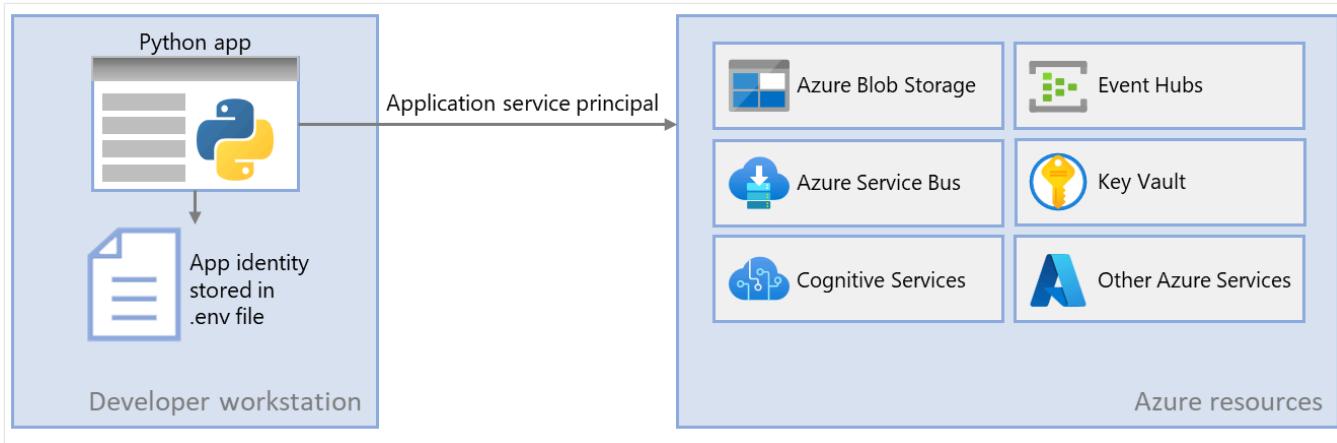
 Yes

 No

Authenticate Python apps to Azure services during local development using service principals

06/02/2025

When building cloud applications, developers often need to run and test their apps locally. Even during local development, the application must authenticate to any Azure services it interacts with. This article explains how to configure dedicated service principal identities specifically for use during local development.



Dedicated application service principals for local development support the principle of least privilege by limiting access to only the Azure resources required by the app during development. Using a dedicated application service principal reduces the risk of unintended access to other resources and helps prevent permission-related bugs when transitioning to production, where broader permissions could lead to issues.

When registering applications for local development in Azure, it's recommended to:

- Create separate app registrations for each developer: This provides each developer with their own service principal, avoiding the need to share credentials and enabling more granular access control.
- Create separate app registrations for each application: This ensures each app only has the permissions it needs, reducing the potential attack surface.

To enable authentication during local development, set environment variables with the application service principal's credentials. The Azure SDK for Python detects these variables and uses them to authenticate requests to Azure services.

1 - Register the application in Azure

Application service principal objects are created when you register an app in Azure. This registration can be performed using either the Azure portal or the Azure CLI. The registration process creates an app registration in Microsoft Entra ID (formerly Azure Active Directory) and generates a service principal object for the app. The service principal object is used to authenticate the app to Azure services. The app registration process also generates a client secret (password) for the app. This secret is used to authenticate the app to Azure services. The client secret is never stored in source control, but rather in a `.env` file in the application directory. The `.env` file is read by the application at runtime to set environment variables that the Azure SDK for Python uses to authenticate the app. The following steps show how to register an app in Azure and create a service principal for the app. The steps are shown for both the Azure CLI and the Azure portal.

Azure CLI

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI installed](#).

First, use the `az ad sp create-for-rbac` command to create a new service principal for the app. The command also creates the app registration for the app at the same time.

Azure CLI

```
SERVICE_PRINCIPAL_NAME=<service-principal-name>
az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NAME
```

The output of this command is similar to the following. Make note of these values or keep this window open as you'll need these values in the next steps and won't be able to view the password (client secret) value again. You can, however, add a new password later without invalidating the service principal or existing passwords if needed.

JSON

```
{
  "appId": "00001111-aaaa-2222-bbbb-3333cccc4444",
  "displayName": "<service-principal-name>",
  "password": "Ee5Ff~6Gg7.-Hh8Ii9Jj0Kk1Ll2Mm3_Nn40o5Pp6",
  "tenant": "aaaabbbb-0000-cccc-1111-dddd2222eeee"
}
```

Next, you need to get the `appId` value and store it into a variable. This value is used to set environment variables in your local development environment so that the Azure SDK for Python can authenticate to Azure using the service principal.

Azure CLI

```
APP_ID=$(az ad sp list \
--all \
--query "[?displayName=='$SERVICE_PRINCIPAL_NAME'].appId | [0]" \
--output tsv)
```

2 - Create a Microsoft Entra security group for local development

Since there are typically multiple developers who work on an application, it's recommended to create a Microsoft Entra security group to encapsulate the roles (permissions) the app needs in local development, rather than assigning the roles to individual service principal objects. This offers the following advantages:

- Every developer is assured to have the same roles assigned since roles are assigned at the group level.
- If a new role is needed for the app, it only needs to be added to the Microsoft Entra group for the app.
- If a new developer joins the team, a new application service principal is created for the developer and added to the group, assuring the developer has the right permissions to work on the app.

Azure CLI

The [az ad group create](#) command is used to create security groups in Microsoft Entra ID. The `--display-name` and `--mainNickname` parameters are required. The name given to the group should be based on the name of the application. It's also useful to include a phrase like 'local-dev' in the name of the group to indicate the purpose of the group.

Azure CLI

```
GROUP_DISPLAY_NAME=""
GROUP_MAIL_NICKNAME=""
GROUP_DESCRIPTION=""
az ad group create \
--display-name $GROUP_DISPLAY_NAME \
--mail-nickname $GROUP_MAIL_NICKNAME \
--description $GROUP_DESCRIPTION
```

To add members to the group, you need the object ID of the application service principal, which is different than the application ID. Use the [az ad sp list](#) to list the available service

principals. The `--filter` parameter command accepts OData style filters and can be used to filter the list as shown. The `--query` parameter limits to columns to only those of interest.

Azure CLI

```
SP_OBJECT_ID=$(az ad sp list \
    --filter "startswith(displayName, '$GROUP_DISPLAY_NAME')" \
    --query "[0].id" \
    --output tsv)
```

The [az ad group member add](#) command can then be used to add members to groups.

Azure CLI

```
az ad group member add \
    --group $GROUP_DISPLAY_NAME \
    --member-id $SP_OBJECT_ID
```

! Note

By default, the creation of Microsoft Entra security groups is limited to certain privileged roles in a directory. If you're unable to create a group, contact an administrator for your directory. If you're unable to add members to an existing group, contact the group owner or a directory administrator. To learn more, see [Manage Microsoft Entra groups and group membership](#).

3 - Assign roles to the application

Next, you need to determine what roles (permissions) your app needs on what resources and assign those roles to your app. In this example, the roles are assigned to the Microsoft Entra group created in step 2. Roles can be assigned at a resource, resource group, or subscription scope. This example shows how to assign roles at the resource group scope since most applications group all their Azure resources into a single resource group.

Azure CLI

A user, group, or application service principal is assigned a role in Azure using the [az role assignment create](#) command. You can specify a group with its object ID. You can specify an application service principal with its appId.

Azure CLI

```
RESOURCE_GROUP_NAME=<resource-group-name>
SUBSCRIPTION_ID=$(az account show --query id --output tsv)
ROLE_NAME=<role-name>
az role assignment create \
--assignee "$APP_ID" \
--scope
./subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP_NAME" \
--role "$ROLE_NAME"
```

![NOTE] To prevent Git Bash from treating /subscriptions/... as a file path, prepend ./ to the string for the `scope` parameter and use double quotes around the entire string.

To get the role names that can be assigned, use the [az role definition list](#) command.

Azure CLI

```
az role definition list \
--query "sort_by([].{roleName:roleName, description:description}, &roleName)" \
--output table
```

For example, to allow the application service principal with the appId of `00001111-aaaa-2222-bbbb-3333cccc4444` read, write, and delete access to Azure Storage blob containers and data in all storage accounts in the `msdocs-python-sdk-auth-example` resource group in the subscription with ID `aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e`, you would assign the application service principal to the *Storage Blob Data Contributor* role using the following command.

Azure CLI

```
az role assignment create --assignee 00001111-aaaa-2222-bbbb-3333cccc4444 \
--scope "./subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e/resourceGroups/msdocs-python-sdk-auth-example" \
--role "Storage Blob Data Contributor"
```

For information on assigning permissions at the resource or subscription level using the Azure CLI, see the article [Assign Azure roles using the Azure CLI](#).

4 - Set local development environment variables

The `DefaultAzureCredential` object will look for the service principal information in a set of environment variables at runtime. Since most developers work on multiple applications, it's recommended to use a package like [python-dotenv](#) to access environment from a `.env` file stored in the application's directory during development. This scopes the environment variables used to authenticate the application to Azure such that they can only be used by this application.

The `.env` file is never checked into source control since it contains the application secret key for Azure. The standard [.gitignore](#) file for Python automatically excludes the `.env` file from check-in.

To use the `python-dotenv` package, first install the package in your application.

terminal

```
pip install python-dotenv
```

Then, create a `.env` file in your application root directory. Set the environment variable values with values obtained from the app registration process as follows:

- `AZURE_CLIENT_ID` → The app ID value.
- `AZURE_TENANT_ID` → The tenant ID value.
- `AZURE_CLIENT_SECRET` → The password/credential generated for the app.

Bash

```
AZURE_CLIENT_ID=00001111-aaaa-2222-bbbb-3333cccc4444
AZURE_TENANT_ID=aaaabbbb-0000-cccc-1111-dddd2222eeee
AZURE_CLIENT_SECRET=Ee5Ff~6Gg7.-Hh8Ii9Jj0Kk1Ll2Mm3_Nn4o5Pp6
```

Finally, in the startup code for your application, use the `python-dotenv` library to read the environment variables from the `.env` file on startup.

Python

```
from dotenv import load_dotenv

if ( os.environ['ENVIRONMENT'] == 'development'):
    print("Loading environment variables from .env file")
    load_dotenv(".env")
```

5 - Implement DefaultAzureCredential in your application

To authenticate Azure SDK client objects to Azure, your application should use the `DefaultAzureCredential` class from the `azure.identity` package. In this scenario, `DefaultAzureCredential` will detect the environment variables `AZURE_CLIENT_ID`, `AZURE_TENANT_ID`, and `AZURE_CLIENT_SECRET` are set and read those variables to get the application service principal information to connect to Azure with.

Start by adding the [azure.identity](#) package to your application.

terminal

```
pip install azure-identity
```

Next, for any Python code that creates an Azure SDK client object in your app, you'll want to:

1. Import the `DefaultAzureCredential` class from the `azure.identity` module.
2. Create a `DefaultAzureCredential` object.
3. Pass the `DefaultAzureCredential` object to the Azure SDK client object constructor.

An example of this is shown in the following code segment.

Python

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

# Acquire a credential object
token_credential = DefaultAzureCredential()

blob_service_client = BlobServiceClient(
    account_url="https://<my_account_name>.blob.core.windows.net",
    credential=token_credential)
```

Authenticate Python apps to Azure services during local development using developer accounts

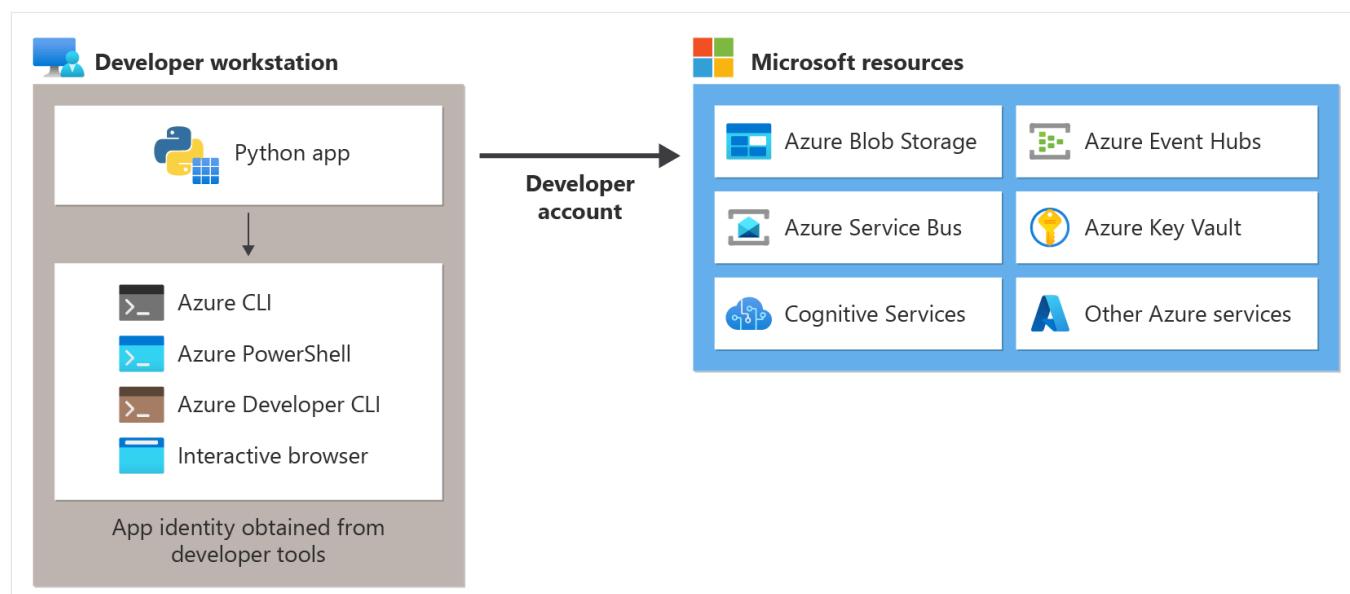
05/30/2025

When developing cloud applications, developers typically build, test, and debug their code locally before deploying it to Azure. However, even during local development, the application needs to authenticate with any Azure services it interacts with, such as Key Vault, Storage, or databases.

This article shows how to configure your application to use the developer's Azure credentials for authentication during local development. This approach enables a seamless and secure development experience without embedding secrets or writing environment-specific logic.

Overview of local development authentication using developer accounts

When developing an application that uses the Azure Identity library for Python, you can authenticate to Azure services during local development using the developer's Azure account. This approach is often the simplest way to authenticate to Azure services during local development since it doesn't require creating and managing service principals or secrets.



To enable an application to authenticate to Azure during local development using the developer's own Azure credentials, the developer must first sign in using one of the supported command-line tools:

- Azure CLI (`az login`)
- Azure Developer CLI (`azd login`)
- Azure PowerShell (`Connect-AzAccount`)

Once signed in, the Azure Identity library for Python can automatically detect the active session and retrieve the necessary tokens from the credentials cache. This capability allows the app to authenticate to Azure services as the signed-in user, without requiring any additional configuration or hardcoded secrets.

This behavior is enabled when using `DefaultAzureCredential`, which transparently falls back to CLI-based credentials in local environments.

Using a developer's signed-in Azure credentials is the easiest setup for local development. It leverages each team member's existing Azure account, enabling seamless access to Azure services without requiring additional configuration.

However, developer accounts typically have broader permissions than the application should have in production. These broader permissions can lead to inconsistencies in testing or inadvertently allow operations that the app wouldn't be authorized to perform in a production environment. To closely mirror production permissions and improve security posture, you can instead create application-specific service principals for local development. These identities:

- Can be assigned only the roles and permissions the application needs
- Support principle of least privilege
- Offer consistent testing of access-related behavior across environments

Developers can configure the local environment to use the service principal via environment variables, and `DefaultAzureCredential` picks it up automatically. For more information, see the article [Authenticate Python apps to Azure services during local development using service principals](#).

1 - Create Microsoft Entra security group for local development

In most development scenarios, multiple developers contribute to the same application. To streamline access control and ensure consistent permissions across the team, we recommend that you first create a Microsoft Entra security group specifically for the application's local development needs.

Assigning Azure roles at the group level—rather than to individual users—offers several key benefits:

- Consistent Role Assignments

All developers in the group automatically inherit the same roles and permissions, ensuring a uniform development environment.

- Simplified Role Management

When the application requires a new role, you only need to add it once to the group. You don't need to update individual user permissions.

- Easy Onboarding

New developers can be granted the necessary permissions simply by adding them to the group. No manual role assignments are required.

If your organization already has a suitable Microsoft Entra security group for the development team, you can reuse it. Otherwise, you can create a new group specifically for the app.

Azure CLI

To create a security group in Microsoft Entra ID, use the [az ad group create](#) Azure CLI command.

This command requires the following parameters:

`--display-name`: A user-friendly name for the group

`--mail-nickname`: A unique identifier used for email and internal reference

We recommend that you base the group name on the application name and include a suffix like `-local-dev` to clearly indicate its purpose.

Bash

```
#!/bin/bash
az ad group create \
    --display-name MyDisplay \
    --mail-nickname MyDisplay \
    --description "<group-description>"
```

PowerShell

```
# PowerShell syntax
az ad group create ` 
    --display-name MyDisplay `
```

```
--mail-nickname MyDisplay `  
--description "<group-description>"
```

After running the `az ad group create` command, copy the value of the `id` property from the command output. You need the `Object ID` of the Microsoft Entra security group for assigning roles in later steps in this article. To retrieve the `Object ID` again later, use the following `az ad group show` command: `az ad group show --group "my-app-local-dev" --query id --output tsv`.

To add a user to the group, you first need to obtain the `Object ID` of the Azure user account you want to add. Use the `az ad user list` command with the `--filter` parameter to search for a specific user by display name. The `--query` parameter helps limit the output to relevant fields:

Bash

```
#!/bin/bash  
az ad user list \  
--filter "startswith(displayName, 'Bob')" \  
--query "[].{objectId:id, displayName:displayName}" \  
--output table
```

PowerShell

```
# PowerShell syntax  
az ad user list `  
    --filter "startswith(displayName, 'Bob')" `  
    --query "[].{objectId:id, displayName:displayName}" `  
    --output table
```

Once you have the `Object ID` of the user, you can add them to the group using the `az ad group member add` command.

Bash

```
#!/bin/bash  
az ad group member add \  
    --group <group-name> \  
    --member-id <object-id>
```

PowerShell

```
# PowerShell syntax  
az ad group member add `
```

```
--group <group-name>
--member-id <object-id>
```

ⓘ Note

By default, the creation of Microsoft Entra security groups is limited to certain privileged roles in a directory. If you're unable to create a group, contact an administrator for your directory. If you're unable to add members to an existing group, contact the group owner or a directory administrator. To learn more, see [Manage Microsoft Entra groups and group membership](#).

2 - Assign roles to the Microsoft Entra group

After creating your Microsoft Entra security group and adding members, the next step is to determine what roles (permissions) your application requires, and assign those roles to the group at the appropriate scope.

- Determine Required Roles

Identify the roles your app needs to function. Common examples include:

- Key Vault Secrets User – to read secrets from Azure Key Vault
- Storage Queue Data Contributor – to send messages to Azure Queue Storage

Refer to the built-in role definitions for more options.

- Choose a Scope for the Role Assignment

Roles can be assigned at different scopes:

- Resource-level (e.g., a single Key Vault or Storage account)
- Resource group-level (recommended for most apps)
- Subscription-level (use with caution—broadest access)

In this example, we assign roles at the resource group scope, which is typical when all application resources are grouped under one resource group.

Azure CLI

A user, group, or application service principal is assigned a role in Azure using the [az role assignment create](#) command. You can specify a group with its Object ID.

Bash

```
#!/bin/bash
az role assignment create --assignee <objectId> \
    --scope /subscriptions/<subscriptionId>/resourceGroups/<resourceGroupName>
\ \
    --role "< roleName >"
```

PowerShell

```
# PowerShell syntax
az role assignment create ` \
    --assignee <objectId> ` \
    --scope /subscriptions/<subscriptionId>/resourceGroups/<resourceGroupName>
` \
    --role "< roleName >"
```

To get the role names that can be assigned, use the [az role definition list](#) command.

Bash

```
#!/bin/bash
az role definition list --query "sort_by([].{roleName:roleName,
description:description}, &roleName)" --output table
```

Powershell

```
# PowerShell syntax
az role definition list --query "sort_by([].{roleName:roleName,
description:description}, &roleName)" --output table
```

To grant read, write, and delete access to Azure Storage blob containers and data for all storage accounts in a specific resource group, assign the Storage Blob Data Contributor role to your Microsoft Entra security group.

Bash

```
#!/bin/bash
az role assignment create --assignee bbbbbbbb-1111-2222-3333-cccccccccc \
    --scope /subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-
eeeeee4e4e/resourceGroups/msdocs-python-sdk-auth-example \
    --role "Storage Blob Data Contributor"
```

Powershell

```
# PowerShell syntax
az role assignment create --assignee bbbbbbbb-1111-2222-3333-cccccccccc \
    --scope /subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-
```

```
eeeeee4e4e/resourceGroups/msdocs-python-sdk-auth-example  
    --role "Storage Blob Data Contributor"
```

For information on assigning permissions at the resource or subscription level using the Azure CLI, see the article [Assign Azure roles using the Azure CLI](#).

3 - Sign-in to Azure using the Azure CLI, Azure PowerShell, Azure Developer CLI, or in a browser

To authenticate with your Azure account, choose one of the following methods:

Azure CLI

Open a terminal on your developer workstation and sign-in to Azure from the [Azure CLI](#).

Azure CLI

```
az login
```

4 - Implement DefaultAzureCredential in your application

To authenticate Azure SDK client objects with Azure, your application should use the [DefaultAzureCredential](#) class from the `azure-identity` package. This is the recommended authentication method for both local development and production deployments.

In a local development scenario, `DefaultAzureCredential` works by sequentially checking for available authentication sources. Specifically, it looks for active sessions in the following tools:

- Azure CLI (`az login`)
- Azure PowerShell (`Connect-AzAccount`)
- Azure Developer CLI (`azd auth login`)

If the developer is signed in to Azure using any of these tools, `DefaultAzureCredential` automatically detects the session and uses those credentials to authenticate the application with Azure services. This allows developers to securely authenticate without storing secrets or modifying code for different environments.

Start by adding the [azure.identity](#) package to your application.

Console

```
pip install azure-identity
```

Next, for any Python code that creates an Azure SDK client object in your app, you want to:

1. Import the `DefaultAzureCredential` class from the `azure.identity` module.
2. Create a `DefaultAzureCredential` object.
3. Pass the `DefaultAzureCredential` object to the Azure SDK client object constructor.

An example of these steps is shown in the following code segment.

Python

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

# Acquire a credential object
token_credential = DefaultAzureCredential()

blob_service_client = BlobServiceClient(
    account_url="https://<my_account_name>.blob.core.windows.net",
    credential=token_credential)
```

Authenticating Azure-hosted apps to Azure resources with the Azure SDK for Python

Article • 08/22/2024

When you host an app in Azure using services like Azure App Service, Azure Virtual Machines, or Azure Container Instances, the recommended approach to authenticate an app to Azure resources is with [managed identity](#).

A managed identity provides an identity for your app such that it can connect to other Azure resources without the need to use a secret key or other application secret.

Internally, Azure knows the identity of your app and what resources it's allowed to connect to. Azure uses this information to automatically obtain Microsoft Entra tokens for the app to allow it to connect to other Azure resources, all without you having to manage any application secrets.

ⓘ Note

Apps running on Azure Kubernetes Service (AKS) can use a workload identity to authenticate with Azure resources. In AKS, a workload identity represents a trust relationship between a managed identity and a Kubernetes service account. If an application deployed to AKS is configured with a Kubernetes service account in such a relationship, `DefaultAzureCredential` authenticates the app to Azure by using the managed identity. Authentication by using a workload identity is discussed in [Use Microsoft Entra Workload ID with Azure Kubernetes Service](#). For steps on how to configure workload identity, see [Deploy and configure workload identity on an Azure Kubernetes Service \(AKS\) cluster](#).

Managed identity types

There are two types of managed identities:

- **System-assigned managed identities** - This type of managed identity is provided by and tied directly to an Azure resource. When you enable managed identity on an Azure resource, you get a system-assigned managed identity for that resource. A system-assigned managed identity is tied to the lifecycle of the Azure resource it's associated with. When the resource is deleted, Azure automatically deletes the identity for you. Since all you have to do is enable managed identity for the Azure

resource hosting your code, this approach is the easiest type of managed identity to use.

- **User-assigned managed identities** - You can also create a managed identity as a standalone Azure resource. This approach is most frequently used when your solution has multiple workloads that run on multiple Azure resources that all need to share the same identity and same permissions. For example, if your solution had components that run on multiple App Service and virtual machine instances that all need access to the same set of Azure resources, then a user-assigned managed identity used across those resources makes sense.

This article covers the steps to enable and use a system-assigned managed identity for an app. If you need to use a user-assigned managed identity, see the article [Manage user-assigned managed identities](#) to see how to create a user-assigned managed identity.

1 - Enable managed identity in the Azure resource hosting the app

The first step is to enable managed identity on Azure resource hosting your app. For example, if you're hosting a Django application using Azure App Service, you need to enable managed identity for the App Service web app that is hosting your app. If you're using a virtual machine to host your app, you would enable your VM to use managed identity.

You can enable managed identity to be used for an Azure resource using either the Azure portal or the Azure CLI.

Azure CLI

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI installed](#).

The Azure CLI commands used to enable managed identity for an Azure resource are of the form `az <command-group> identity --resource-group <resource-group-name> --name <resource-name>`. Specific commands for popular Azure services are shown below.

Azure App Service

Azure CLI

```
az webapp identity assign --resource-group <resource-group-name> -  
name <web-app-name>
```

The output will look like the following.

JSON

```
{  
    "principalId": "aaaaaaaa-bbbb-cccc-1111-222222222222",  
    "tenantId": "aaaabbbb-0000-cccc-1111-ddd2222eeee",  
    "type": "SystemAssigned",  
    "userAssignedIdentities": null  
}
```

The `principalId` value is the unique ID of the managed identity. Keep a copy of this output as you'll need these values in the next step.

2 - Assign roles to the managed identity

Next, you need to determine what roles (permissions) your app needs and assign the managed identity to those roles in Azure. A managed identity can be assigned roles at a resource, resource group, or subscription scope. This example shows how to assign roles at the resource group scope since most applications group all their Azure resources into a single resource group.

Azure CLI

A managed identity is assigned a role in Azure using the `az role assignment create` command. For the assignee, use the `principalId` you copied in step 1.

Azure CLI

```
az role assignment create --assignee <managedIdentityprincipalId> \  
--scope  
/subscriptions/<subscriptionId>/resourceGroups/<resourceGroupName> \  
--role "<roleName>"
```

To get the role names that a service principal can be assigned to, use the `az role definition list` command.

Azure CLI

```
az role definition list \
    --query "sort_by([].{roleName:roleName, description:description}, &roleName)" \
    --output table
```

For example, to allow the managed identity with the ID of `aaaaaaaa-bbbb-cccc-1111-222222222222` read, write, and delete access to Azure Storage blob containers and data in all storage accounts in the *msdocs-python-sdk-auth-example* resource group in the subscription with ID `aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e`, you would assign the application service principal to the *Storage Blob Data Contributor* role using the following command.

Azure CLI

```
az role assignment create --assignee aaaaaaaaa-bbbb-cccc-1111-222222222222 \
    --scope /subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e/resourceGroups/msdocs-python-sdk-auth-example \
    --role "Storage Blob Data Contributor"
```

For information on assigning permissions at the resource or subscription level using the Azure CLI, see the article [Assign Azure roles using the Azure CLI](#).

3 - Implement DefaultAzureCredential in your application

When your code is running in Azure and managed identity has been enabled on the Azure resource hosting your app, the `DefaultAzureCredential` determines the credentials to use in the following order:

1. Check the environment for a service principal as defined by the environment variables `AZURE_CLIENT_ID`, `AZURE_TENANT_ID`, and either `AZURE_CLIENT_SECRET` or `AZURE_CLIENT_CERTIFICATE_PATH` and (optionally) `AZURE_CLIENT_CERTIFICATE_PASSWORD`.
2. Check keyword parameters for a user-assigned managed identity. You can pass in a user-assigned managed identity by specifying its client ID in the `managed_identity_client_id` parameter.
3. Check the `AZURE_CLIENT_ID` environment variable for the client ID of a user-assigned managed identity.
4. Use the system-assigned managed identity for the Azure resource if it's enabled.

You can exclude managed identities from the credential by setting the `exclude_managed_identity_credential` keyword parameter `True`.

In this article, we're using the system-assigned managed identity for an Azure App Service web app, so we don't need to configure a managed identity in the environment or pass it in as a parameter. The following steps show you how to use `DefaultAzureCredential`.

First, add the `azure.identity` package to your application.

terminal

```
pip install azure-identity
```

Next, for any Python code that creates an Azure SDK client object in your app, you'll want to:

1. Import the `DefaultAzureCredential` class from the `azure.identity` module.
2. Create a `DefaultAzureCredential` object.
3. Pass the `DefaultAzureCredential` object to the Azure SDK client object constructor.

An example of these steps is shown in the following code segment.

Python

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

# Acquire a credential object
token_credential = DefaultAzureCredential()

blob_service_client = BlobServiceClient(
    account_url="https://<my_account_name>.blob.core.windows.net",
    credential=token_credential)
```

As discussed in the [Azure SDK for Python authentication overview](#) article, `DefaultAzureCredential` supports multiple authentication methods and determines the authentication method being used at runtime. The benefit of this approach is that your app can use different authentication methods in different environments without implementing environment-specific code. When the preceding code is run on your workstation during local development, `DefaultAzureCredential` will use either an application service principal, as determined by environment settings, or developer tool credentials to authenticate with other Azure resources. Thus, the same code can be used

to authenticate your app to Azure resources during both local development and when deployed to Azure.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Authenticate to Azure resources from Python apps hosted on-premises

06/02/2025

Apps hosted outside of Azure (for example on-premises or at a third-party data center) should use an application service principal to authenticate to Azure when accessing Azure resources. Application service principal objects are created using the app registration process in Azure. When an application service principal is created, a client ID and client secret will be generated for your app. The client ID, client secret, and your tenant ID are then stored in environment variables so they can be used by the Azure SDK for Python to authenticate your app to Azure at runtime.

A different app registration should be created for each environment the app is hosted in. This allows environment specific resource permissions to be configured for each service principal and ensures that an app deployed to one environment doesn't talk to Azure resources that are part of another environment.

1 - Register the application in Azure

An app can be registered with Azure using either the Azure portal or the Azure CLI.

```
Azure CLI
Azure CLI
APP_NAME=<app-name>
az ad sp create-for-rbac --name $APP_NAME
```

The output of the command is similar to the following. Make note of these values or keep this window open as you'll need these values in the next steps and won't be able to view the password (client secret) value again.

```
JSON
{
  "appId": "00001111-aaaa-2222-bbbb-3333cccc4444",
  "displayName": "msdocs-python-sdk-auth-prod",
  "password": "Ee5Ff~6Gg7.-Hh8Ii9Jj0Kk1Ll2Mm3_Nn4Oo5Pp6",
  "tenant": "aaaabbbb-0000-cccc-1111-dddd2222eeee"
}
```

Next, you need to get the `appId` value and store it into a variable. This value is used to set environment variables in your local development environment so that the Azure SDK for Python can authenticate to Azure using the service principal.

Azure CLI

```
APP_ID=$(az ad sp create-for-rbac \
--name $APP_NAME --query appId --output tsv)
```

2 - Assign roles to the application service principal

Next, you need to determine what roles (permissions) your app needs on what resources and assign those roles to your app. Roles can be assigned a role at a resource, resource group, or subscription scope. This example shows how to assign roles for the service principal at the resource group scope since most applications group all their Azure resources into a single resource group.

Azure CLI

A service principal is assigned a role in Azure using the [az role assignment create](#) command.

Azure CLI

```
RESOURCE_GROUP_NAME=<resource-group-name>
SUBSCRIPTION_ID=$(az account show --query id --output tsv)
ROLE_NAME=<role-name>

az role assignment create \
--assignee "$APP_ID" \
--scope
"./subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP_NAME" \
--role "$ROLE_NAME"
```

![NOTE] To prevent Git Bash from treating /subscriptions/... as a file path, prepend ./ to the string for the `scope` parameter and use double quotes around the entire string.

To get the role names that a service principal can be assigned to, use the [az role definition list](#) command.

Azure CLI

```
az role definition list \  
    --query "sort_by([].{roleName:roleName, description:description},  
&roleName)" \  
    --output table
```

For example, to allow the service principal with the appId of `00001111-aaaa-2222-bbbb-3333cccc4444` read, write, and delete access to Azure Storage blob containers and data in all storage accounts in the *msdocs-python-sdk-auth-example* resource group in the subscription with ID `aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e`, you would assign the application service principal to the *Storage Blob Data Contributor* role using the following command.

Azure CLI

```
az role assignment create --assignee 00001111-aaaa-2222-bbbb-3333cccc4444 \  
    --scope "./subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-  
eeeeee4e4e4e/resourceGroups/msdocs-python-sdk-auth-example" \  
    --role "Storage Blob Data Contributor"
```

For information on assigning permissions at the resource or subscription level using the Azure CLI, see the article [Assign Azure roles using the Azure CLI](#).

3 - Configure environment variables for application

You must set the `AZURE_CLIENT_ID`, `AZURE_TENANT_ID`, and `AZURE_CLIENT_SECRET` environment variables for the process that runs your Python app to make the application service principal credentials available to your app at runtime. The `DefaultAzureCredential` object looks for the service principal information in these environment variables.

When using [Gunicorn](#) to run Python web apps in a UNIX server environment, environment variables for an app can be specified by using the `EnvironmentFile` directive in the `gunicorn.server` file as shown below.

```
gunicorn.server  
  
[Unit]  
Description=gunicorn daemon  
After=network.target  
  
[Service]  
User=www-user  
Group=www-data  
WorkingDirectory=/path/to/python-app
```

```
EnvironmentFile=/path/to/python-app/py-env/app-environment-variables
ExecStart=/path/to/python-app/py-env/gunicorn --config config.py wsgi:app

[Install]
WantedBy=multi-user.target
```

The file specified in the `EnvironmentFile` directive should contain a list of environment variables with their values as shown below.

Bash

```
AZURE_CLIENT_ID=<value>
AZURE_TENANT_ID=<value>
AZURE_CLIENT_SECRET=<value>
```

4 - Implement DefaultAzureCredential in application

To authenticate Azure SDK client objects to Azure, your application should use the `DefaultAzureCredential` class from the `azure.identity` package.

Start by adding the [azure.identity](#) package to your application.

terminal

```
pip install azure-identity
```

Next, for any Python code that creates an Azure SDK client object in your app, you'll want to:

1. Import the `DefaultAzureCredential` class from the `azure.identity` module.
2. Create a `DefaultAzureCredential` object.
3. Pass the `DefaultAzureCredential` object to the Azure SDK client object constructor.

An example of this is shown in the following code segment.

Python

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

# Acquire a credential object
token_credential = DefaultAzureCredential()

blob_service_client = BlobServiceClient(
```

```
account_url="https://<my_account_name>.blob.core.windows.net",
credential=token_credential)
```

When the above code instantiates the `DefaultAzureCredential` object, `DefaultAzureCredential` reads the environment variables `AZURE_TENANT_ID`, `AZURE_CLIENT_ID`, and `AZURE_CLIENT_SECRET` for the application service principal information to connect to Azure with.

Additional methods to authenticate to Azure resources from Python apps

Article • 03/26/2025

This article lists additional methods that apps can use to authenticate to Azure resources. The methods in this article are less commonly used; when possible, we encourage you to use one of the methods outlined in [authenticating Python apps to Azure using the Azure SDK overview](#).

Interactive browser authentication

This method interactively authenticates an application through [InteractiveBrowserCredential](#) by collecting user credentials in the default system.

Interactive browser authentication enables the application for all operations allowed by the interactive login credentials. As a result, if you're the owner or administrator of your subscription, your code has inherent access to most resources in that subscription without having to assign any specific permissions. For this reason, the use of interactive browser authentication is discouraged for anything but experimentation.

Enable applications for interactive browser authentication

Perform the following steps to enable the application to authenticate through the interactive browser flow. These steps also work for [device code authentication](#) described later. Following this process is necessary only if using `InteractiveBrowserCredential` in your code.

1. On the [Azure portal](#), navigate to Microsoft Entra ID and select **App registrations** on the left-hand menu.
2. Select the registration for your app, then select **Authentication**.
3. Under **Advanced settings**, select **Yes** for **Allow public client flows**.
4. Select **Save** to apply the changes.
5. To authorize the application for specific resources, navigate to the resource in question, select **API Permissions**, and enable **Microsoft Graph** and other resources you want to access. Microsoft Graph is usually enabled by default.

Important

You must also be the admin of your tenant to grant consent to your application when you sign in for the first time.

If you can't configure the device code flow option on your Active Directory, your application might need to be multitenant. To make this change, navigate to the **Authentication** panel, select **Accounts in any organizational directory** (under **Supported account types**), and then select **Yes** for **Allow public client flows**.

Example using InteractiveBrowserCredential

The following example demonstrates using an [InteractiveBrowserCredential](#) to authenticate with the [SubscriptionClient](#):

Python

```
# Show Azure subscription information

import os
from azure.identity import InteractiveBrowserCredential
from azure.mgmt.resource import SubscriptionClient

credential = InteractiveBrowserCredential()
subscription_client = SubscriptionClient(credential)

subscription = next(subscription_client.subscriptions.list())
print(subscription.subscription_id)
```

For more exact control, such as setting redirect URIs, you can supply specific arguments to `InteractiveBrowserCredential` such as `redirect_uri`.

Interactive brokered authentication

This method interactively authenticates an application through [InteractiveBrowserBrokerCredential](#) by collecting user credentials using the system authentication broker. This credential type is provided in the Azure Identity Broker plugin, [azure-identity-broker](#).

A system authentication broker is an app running on a user's machine that manages the authentication handshakes and token maintenance for all connected accounts. Currently, only the Windows authentication broker, Web Account Manager (WAM), is supported. Users on macOS and Linux will be authenticated through a browser.

Personal Microsoft accounts and work or school accounts are supported. If a supported version of Windows is used, the default browser-based UI is replaced with a smoother authentication experience, similar to Windows built-in apps.

Interactive brokered authentication enables the application for all operations allowed by the interactive login credentials. As a result, if you're the owner or administrator of your subscription, your code has inherent access to most resources in that subscription without having to assign any specific permissions.

Enable applications for interactive brokered authentication

Perform the following steps to enable the application to authenticate through the interactive broker flow.

1. On the [Azure portal](#), navigate to Microsoft Entra ID and select **App registrations** on the left-hand menu.
2. Select the registration for your app, then select **Authentication**.
3. Add the WAM redirect URI to your app registration via a platform configuration:
 - a. Under **Platform configurations**, select **+ Add a platform**.
 - b. Under **Configure platforms**, select the tile for your application type (platform) to configure its settings; For example, **mobile and desktop applications**.
 - c. In **Custom redirect URIs**, enter the WAM redirect URI:

text

```
ms-appx-web://microsoft.aad.brokerplugin/{client_id}
```

The `{client_id}` placeholder must be replaced with the Application (client) ID listed on the Overview pane of the app registration.

- d. Select **Configure**.

To learn more, see [Add a redirect URI to an app registration](#).

4. Back on the **Authentication** pane, under **Advanced settings**, select **Yes** for **Allow public client flows**.
5. Select **Save** to apply the changes.

6. To authorize the application for specific resources, navigate to the resource in question, select **API Permissions**, and enable **Microsoft Graph** and other resources you want to access. Microsoft Graph is usually enabled by default.

ⓘ Important

You must also be the admin of your tenant to grant consent to your application when you sign in for the first time.

Example using InteractiveBrowserBrokerCredential

The following example demonstrates using an [InteractiveBrowserBrokerCredential](#) to authenticate with the [BlobServiceClient](#):

Python

```
import win32gui
from azure.identity.broker import InteractiveBrowserBrokerCredential
from azure.storage.blob import BlobServiceClient

# Get the handle of the current window
current_window_handle = win32gui.GetForegroundWindow()

# To authenticate and authorize with an app, use the following line to get a
# credential and
# substitute the <app_id> and <tenant_id> placeholders with the values for
# your app and tenant.
# credential =
# InteractiveBrowserBrokerCredential(parent_window_handle=current_window_handl
e, client_id=<app_id>, tenant_id=<tenant_id>)
credential =
InteractiveBrowserBrokerCredential(parent_window_handle=current_window_handl
e)
client = BlobServiceClient("https://<storage-account-
name>.blob.core.windows.net/", credential=credential)

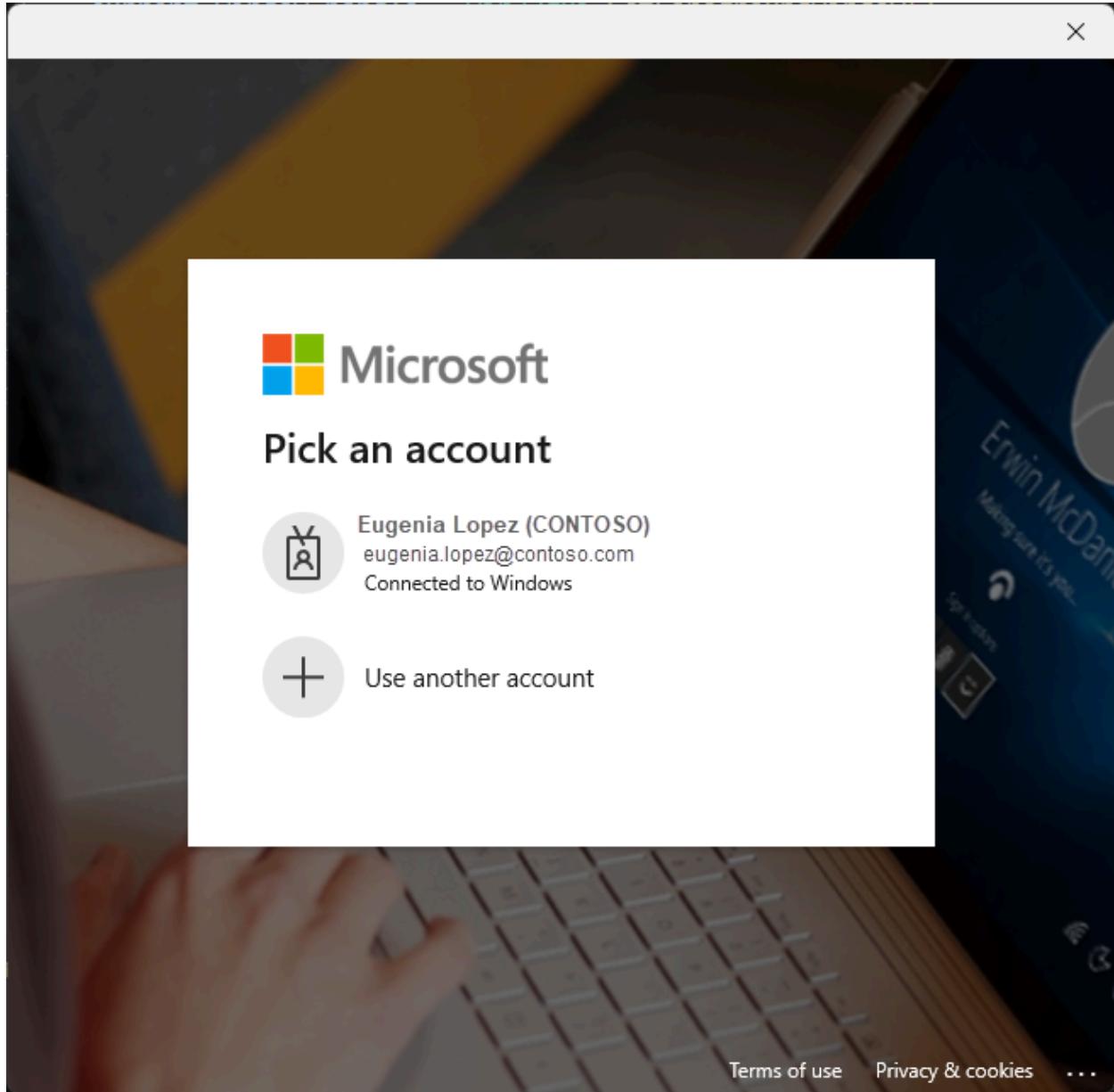
# Prompt for credentials appears on first use of the client
for container in client.list_containers():
    print(container.name)
```

For more exact control, such as setting a timeout, you can supply specific arguments to `InteractiveBrowserBrokerCredential` such as `timeout`.

For the code to run successfully, your user account must be assigned an Azure role on the storage account that allows access to blob containers like "Storage Account Data Contributor". If an app is specified, it must have API permissions set for

`user_impersonation` Access Azure Storage (step 6 in the previous section). This API permission allows the app to access Azure storage on behalf of the signed-in user after consent is granted during sign-in.

The following screenshot shows the user sign-in experience:



Authenticate the default system account via WAM

Many people always sign in to Windows with the same user account and, therefore, only ever want to authenticate using that account. Forcing such individuals to repeatedly select their sole account from an account picker can be aggravating. Fortunately, `InteractiveBrowserBrokerCredential` offers a way for you to enable such individuals to sign in silently with the default system account, which, on Windows, is the signed-in user.

To enable sign-in with the default system account:

1. Make sure you use `azure-identity-broker` version 1.1.0 or greater.
2. Set the `use_default_broker_account` argument to `True` when you create an instance of `InteractiveBrowserBrokerCredential`.

The following example shows how to enable sign-in with the default system account:

Python

```
import win32gui
from azure.identity.broker import InteractiveBrowserBrokerCredential

# code omitted for brevity

window_handle = win32gui.GetForegroundWindow()

credential = InteractiveBrowserBrokerCredential(
    parent_window_handle=window_handle,
    use_default_broker_account=True
)
```

Once you opt into this behavior, the credential type attempts to sign in by asking the underlying Microsoft Authentication Library (MSAL) to perform the sign-in for the default system account. If the sign-in fails, the credential type falls back to displaying the account picker dialog, from which the user can select the appropriate account.

Device code authentication

This method interactively authenticates a user on devices with limited UI (typically devices without a keyboard):

1. When the application attempts to authenticate, the credential prompts the user with a URL and an authentication code.
2. The user visits the URL on a separate browser-enabled device (a computer, smartphone, etc.) and enters the code.
3. The user follows a normal authentication process in the browser.
4. Upon successful authentication, the application is authenticated on the device.

For more information, see [Microsoft identity platform and the OAuth 2.0 device authorization grant flow](#).

Device code authentication in a development environment enables the application for all operations allowed by the interactive login credentials. As a result, if you're the owner or administrator of your subscription, your code has inherent access to most resources in that subscription without having to assign any specific permissions. However, you can

use this method with a specific client ID, rather than the default, for which you can assign specific permissions.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

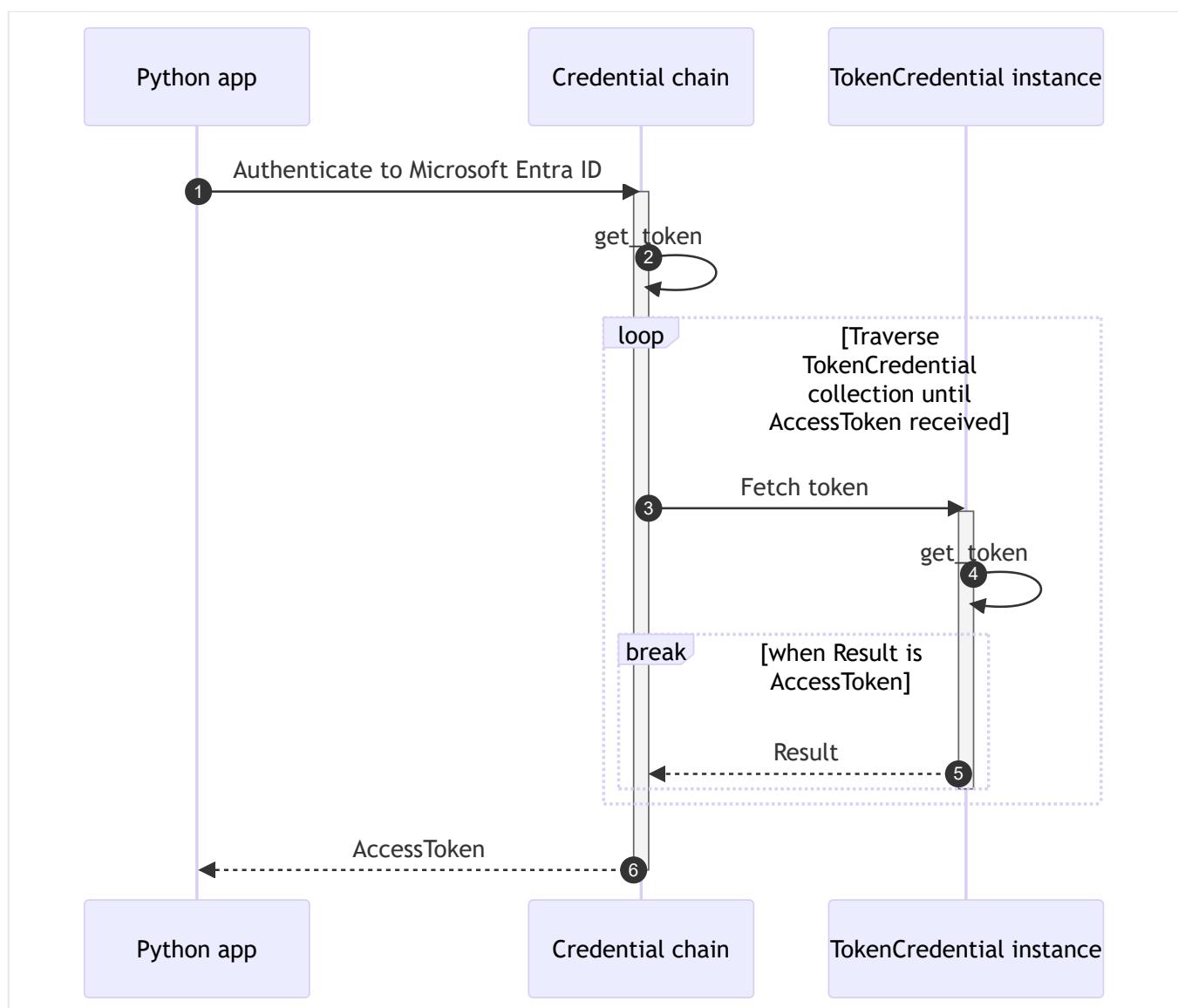
Credential chains in the Azure Identity library for Python

08/07/2025

The Azure Identity library provides *credentials*—public classes that implement the Azure Core library's [TokenCredential](#) protocol. A credential represents a distinct authentication flow for acquiring an access token from Microsoft Entra ID. These credentials can be chained together to form an ordered sequence of authentication mechanisms to be attempted.

How a chained credential works

At runtime, a credential chain attempts to authenticate using the sequence's first credential. If that credential fails to acquire an access token, the next credential in the sequence is attempted, and so on, until an access token is successfully obtained. The following sequence diagram illustrates this behavior:



Why use credential chains

A chained credential can offer the following benefits:

- **Environment awareness:** Automatically selects the most appropriate credential based on the environment in which the app is running. Without it, you'd have to write code like this:

Python

```
# Set up credential based on environment (Azure or local development)
if os.getenv("WEBSITE_HOSTNAME"):
    credential = ManagedIdentityCredential(client_id=user_assigned_client_id)
else:
    credential = AzureCliCredential()
```

- **Seamless transitions:** Your app can move from local development to your staging or production environment without changing authentication code.
- **Improved resiliency:** Includes a fallback mechanism that moves to the next credential when the prior fails to acquire an access token.

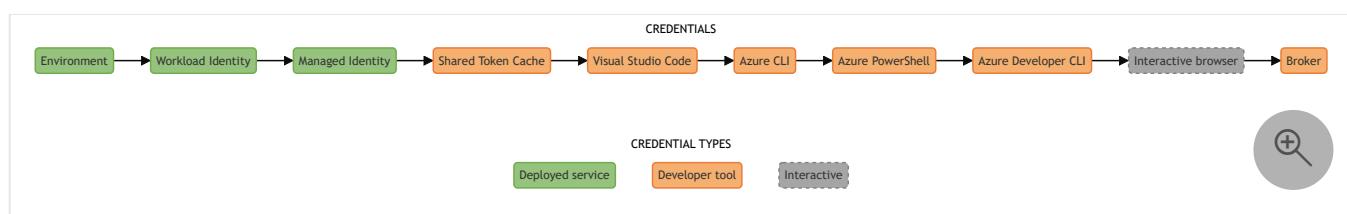
How to choose a chained credential

There are two disparate philosophies to credential chaining:

- **"Tear down" a chain:** Start with a preconfigured chain and exclude what you don't need. For this approach, see the [DefaultAzureCredential overview](#) section.
- **"Build up" a chain:** Start with an empty chain and include only what you need. For this approach, see the [ChainedTokenCredential overview](#) section.

DefaultAzureCredential overview

[DefaultAzureCredential](#) is an opinionated, preconfigured chain of credentials. It's designed to support many environments, along with the most common authentication flows and developer tools. In graphical form, the underlying chain looks like this:



The order in which `DefaultAzureCredential` attempts credentials follows.

Order	Credential	Description	Enabled by default?
1	Environment	Reads a collection of environment variables to determine if an application service principal (application user) is configured for the app. If so, <code>DefaultAzureCredential</code> uses these values to authenticate the app to Azure. This method is most often used in server environments but can also be used when developing locally.	Yes
2	Workload Identity	If the app is deployed to an Azure host with Workload Identity enabled, authenticate that account.	Yes
3	Managed Identity	If the app is deployed to an Azure host with Managed Identity enabled, authenticate the app to Azure using that Managed Identity.	Yes
4	Shared Token Cache	On Windows only, if the developer authenticated to Azure by logging into Visual Studio, authenticate the app to Azure using that same account.	Yes
5	Visual Studio Code	If the developer authenticated via Visual Studio Code's Azure Resources extension and the azure-identity-broker package is installed, authenticate that account.	Yes
6	Azure CLI	If the developer authenticated to Azure using Azure CLI's <code>az login</code> command, authenticate the app to Azure using that same account.	Yes
7	Azure PowerShell	If the developer authenticated to Azure using Azure PowerShell's <code>Connect-AzAccount</code> cmdlet, authenticate the app to Azure using that same account.	Yes
8	Azure Developer CLI	If the developer authenticated to Azure using Azure Developer CLI's <code>azd auth login</code> command, authenticate with that account.	Yes
9	Interactive browser	If enabled, interactively authenticate the developer via the current system's default browser.	No
10	Broker	Authenticates using the default account logged into the OS via a broker. Requires that the azure-identity-broker package is installed, since an instance of <code>InteractiveBrowserBrokerCredential</code> is used.	Yes

In its simplest form, you can use the parameterless version of `DefaultAzureCredential` as follows:

Python

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

# Acquire a credential object
credential = DefaultAzureCredential()

blob_service_client = BlobServiceClient(
    account_url="https://<my_account_name>.blob.core.windows.net",
    credential=credential
)
```

How to customize DefaultAzureCredential

The following sections describe strategies for controlling which credentials are included in the chain.

Exclude an individual credential

To exclude an individual credential from `DefaultAzureCredential`, use the corresponding `exclude`-prefixed [keyword parameter](#). For example:

Python

```
credential = DefaultAzureCredential(
    exclude_environment_credential=True,
    exclude_workload_identity_credential=True,
    managed_identity_client_id=user_assigned_client_id
)
```

In the preceding code sample, `EnvironmentCredential` and `WorkloadIdentityCredential` are removed from the credential chain. As a result, the first credential to be attempted is `ManagedIdentityCredential`. The modified chain looks like this:



! Note

`InteractiveBrowserCredential` is excluded by default and therefore isn't shown in the preceding diagram. To include `InteractiveBrowserCredential`, set the `exclude_interactive_browser_credential` keyword parameter to `False` when you call the `DefaultAzureCredential` constructor.

As more `exclude`-prefixed keyword parameters are set to `True` (credential exclusions are configured), the advantages of using `DefaultAzureCredential` diminish. In such cases, `ChainedTokenCredential` is a better choice and requires less code. To illustrate, these two code samples behave the same way:

DefaultAzureCredential

Python

```
credential = DefaultAzureCredential(  
    exclude_environment_credential=True,  
    exclude_workload_identity_credential=True,  
    exclude_shared_token_cache_credential=True,  
    exclude_visual_studio_code_credential=True,  
    exclude_azure_powershell_credential=True,  
    exclude_azure_developer_cli_credential=True,  
    exclude_broker_credential=True,  
    managed_identity_client_id=user_assigned_client_id  
)
```

Exclude a credential type category

To exclude all `Developer tool` or `Deployed service` credentials, set environment variable `AZURE_TOKEN_CREDENTIALS` to `prod` or `dev`, respectively. When a value of `prod` is used, the underlying credential chain looks as follows:



When a value of `dev` is used, the chain looks as follows:



Important

The `AZURE_TOKEN_CREDENTIALS` environment variable is supported in `azure-identity` package versions 1.23.0 and later.

Use a specific credential

To exclude all credentials except for one, set environment variable `AZURE_TOKEN_CREDENTIALS` to the credential name. For example, you can reduce the `DefaultAzureCredential` chain to `AzureCliCredential` by setting `AZURE_TOKEN_CREDENTIALS` to `AzureCliCredential`. The string comparison is performed in a case-insensitive manner. Valid string values for the environment variable include:

- `AzureCliCredential`
- `AzureDeveloperCliCredential`
- `AzurePowerShellCredential`
- `EnvironmentCredential`
- `InteractiveBrowserCredential`
- `ManagedIdentityCredential`
- `VisualStudioCodeCredential`
- `WorkloadIdentityCredential`

 **Important**

The `AZURE_TOKEN_CREDENTIALS` environment variable supports individual credential names in `azure-identity` package versions 1.24.0 and later.

ChainedTokenCredential overview

[ChainedTokenCredential](#) is an empty chain to which you add credentials to suit your app's needs. For example:

Python

```
credential = ChainedTokenCredential(  
    AzureCliCredential(),  
    AzureDeveloperCliCredential()  
)
```

The preceding code sample creates a tailored credential chain comprised of two development-time credentials. `AzureCliCredential` is attempted first, followed by `AzureDeveloperCliCredential`, if necessary. In graphical form, the chain looks like this:



💡 Tip

For improved performance, optimize credential ordering in `ChainedTokenCredential` from most to least used credential.

Usage guidance for DefaultAzureCredential

`DefaultAzureCredential` is undoubtedly the easiest way to get started with the Azure Identity library, but with that convenience comes tradeoffs. Once you deploy your app to Azure, you should understand the app's authentication requirements. For that reason, replace `DefaultAzureCredential` with a specific `TokenCredential` implementation, such as `ManagedIdentityCredential`.

Here's why:

- **Debugging challenges:** When authentication fails, it can be challenging to debug and identify the offending credential. You must enable logging to see the progression from one credential to the next and the success/failure status of each. For more information, see [Debug a chained credential](#).
- **Performance overhead:** The process of sequentially trying multiple credentials can introduce performance overhead. For example, when running on a local development machine, managed identity is unavailable. Consequently, `ManagedIdentityCredential` always fails in the local development environment, unless explicitly disabled via its corresponding `exclude`-prefixed property.
- **Unpredictable behavior:** `DefaultAzureCredential` checks for the presence of certain [environment variables](#). It's possible that someone could add or modify these environment variables at the system level on the host machine. Those changes apply globally and therefore alter the behavior of `DefaultAzureCredential` at runtime in any app running on that machine.

Debug a chained credential

To diagnose an unexpected issue or to understand what a chained credential is doing, [enable logging](#) in your app. Optionally, filter the logs to only those events emitted from the Azure Identity client library. For example:

Python

```
import logging
from azure.identity import DefaultAzureCredential
```

```

# Set the logging level for the Azure Identity library
logger = logging.getLogger("azure.identity")
logger.setLevel(logging.DEBUG)

# Direct logging output to stdout. Without adding a handler,
# no logging output is visible.
handler = logging.StreamHandler(stream=sys.stdout)
logger.addHandler(handler)

# Optional: Output logging levels to the console.
print(
    f"Logger enabled for ERROR={logger.isEnabledFor(logging.ERROR)}, "
    f"WARNING={logger.isEnabledFor(logging.WARNING)}, "
    f"INFO={logger.isEnabledFor(logging.INFO)}, "
    f"DEBUG={logger.isEnabledFor(logging.DEBUG)}"
)

```

For illustration purposes, assume the parameterless form of `DefaultAzureCredential` is used to authenticate a request to a blob storage account. The app runs in the local development environment, and the developer authenticated to Azure using the Azure CLI. Assume also that the logging level is set to `logging.DEBUG`. When the app is run, the following pertinent entries appear in the output:

Output

```

Logger enabled for ERROR=True, WARNING=True, INFO=True, DEBUG=True
No environment configuration found.
ManagedIdentityCredential will use IMDS
EnvironmentCredential.get_token failed: EnvironmentCredential authentication
unavailable. Environment variables are not fully configured.
Visit https://aka.ms/azsdk/python/identity/environmentcredential/troubleshoot to
troubleshoot this issue.
ManagedIdentityCredential.get_token failed: ManagedIdentityCredential
authentication unavailable, no response from the IMDS endpoint.
SharedTokenCacheCredential.get_token failed: SharedTokenCacheCredential
authentication unavailable. No accounts were found in the cache.
VisualStudioCodeCredential.get_token failed: VisualStudioCodeCredential
authentication unavailable. No Azure account information found in Visual Studio
Code.
AzureCliCredential.get_token succeeded
[Authenticated account] Client ID: 00001111-aaaa-2222-bbbb-3333cccc4444. Tenant
ID: aaaabbbb-0000-cccc-1111-dddd2222eeee. User Principal Name: unavailableUpn.
Object ID (user): aaaaaaaaa-0000-1111-2222-bbbbbbbbbbb
DefaultAzureCredential acquired a token from AzureCliCredential

```

In the preceding output, notice that:

- `EnvironmentCredential`, `ManagedIdentityCredential`, `SharedTokenCacheCredential`, and `VisualStudioCodeCredential` each failed to acquire a Microsoft Entra access token, in that

order.

- The `AzureCliCredential.get_token` call succeeds and the output also indicates that `DefaultAzureCredential` acquired a token from `AzureCliCredential`. Since `AzureCliCredential` succeeded, no credentials beyond it were tried.

ⓘ Note

In the preceding example, the logging level is set to `logging.DEBUG`. Be careful when using this logging level, as it can output sensitive information. For example, in this case, the client ID, tenant ID, and the object ID of the developer's user principal in Azure. All traceback information has been removed from the output for clarity.

Walkthrough: Integrated authentication for Python apps with Azure services

05/28/2025

Microsoft Entra ID, when used with Azure Key Vault, provides a robust and secure approach for authenticating applications to both Azure services and third-party platforms that require access keys or credentials. This combination eliminates the need to hardcode secrets in application code, instead relying on managed identities, role-based access control (RBAC), and centralized secret management via Key Vault. This approach streamlines identity management and enhances security posture in cloud environments.

This walkthrough explores these authentication mechanisms through a practical example provided in the GitHub repository: github.com/Azure-Samples/python-integrated-authentication.

The sample demonstrates how a Python application can:

- Authenticate with Azure using DefaultAzureCredential
- Access Azure Key Vault secrets without storing credentials
- Communicate securely with other Azure services like Azure Storage, Cosmos DB, and more

This article is part of a series that provides a detailed walkthrough of how to authenticate a Python app with Microsoft Entra ID, Azure Key Vault, and Azure Queue Storage by using the Azure Python SDK `azure-identity` library.

Part 1: Background

While many Azure services rely exclusively on role-based access control (RBAC), while others require access via secrets or keys. Such services include Azure Storage, databases, Azure AI services, Key Vault, and Event Hubs

When building cloud applications that interact with these services, developers can use the Azure portal, CLI, or PowerShell to generate and configure service-specific access keys. These keys are tied to particular access policies to prevent unauthorized access. However, this model requires your application to manage keys explicitly and authenticate separately with each service, a process that's both tedious and error-prone.

Embedding secrets directly in code or storing them on developer machines risks exposing them in:

- Source control
- Insecure local environments
- Accidental logs or configuration exports

Azure Offers Two Key Services to Improve Security and Simplify Authentication:

- **Azure Key Vault** Azure Key Vault provides a secure, cloud-based store for secrets, including access keys, connection strings, and certificates. By retrieving secrets from Key Vault only at runtime, applications avoid exposing sensitive data in source code or configuration files.
- With [Microsoft Entra managed identities](#), your application can authenticate once with Microsoft Entra ID. From there, it can access other Azure services—including Key Vault—without managing credentials directly.

This approach provides:

- Credential-free code (no secrets in source control)
- Seamless integration with Azure services
- Environment consistency: the same code runs locally and in the cloud with minimal configuration

This walkthrough shows how to use Microsoft Entra managed identity and Key Vault together in the same app. By using Microsoft Entra ID and Key Vault together, your app never needs to authenticate itself with individual Azure services, and can easily and securely access any keys necessary for third-party services.

 **Important**

This article uses the common, generic term "key" to refer to what are stored as "secrets" in Azure Key Vault, such as an access key for a REST API. This usage shouldn't be confused with Key Vault's management of *cryptographic* keys, which is a separate feature from Key Vault's *secrets*.

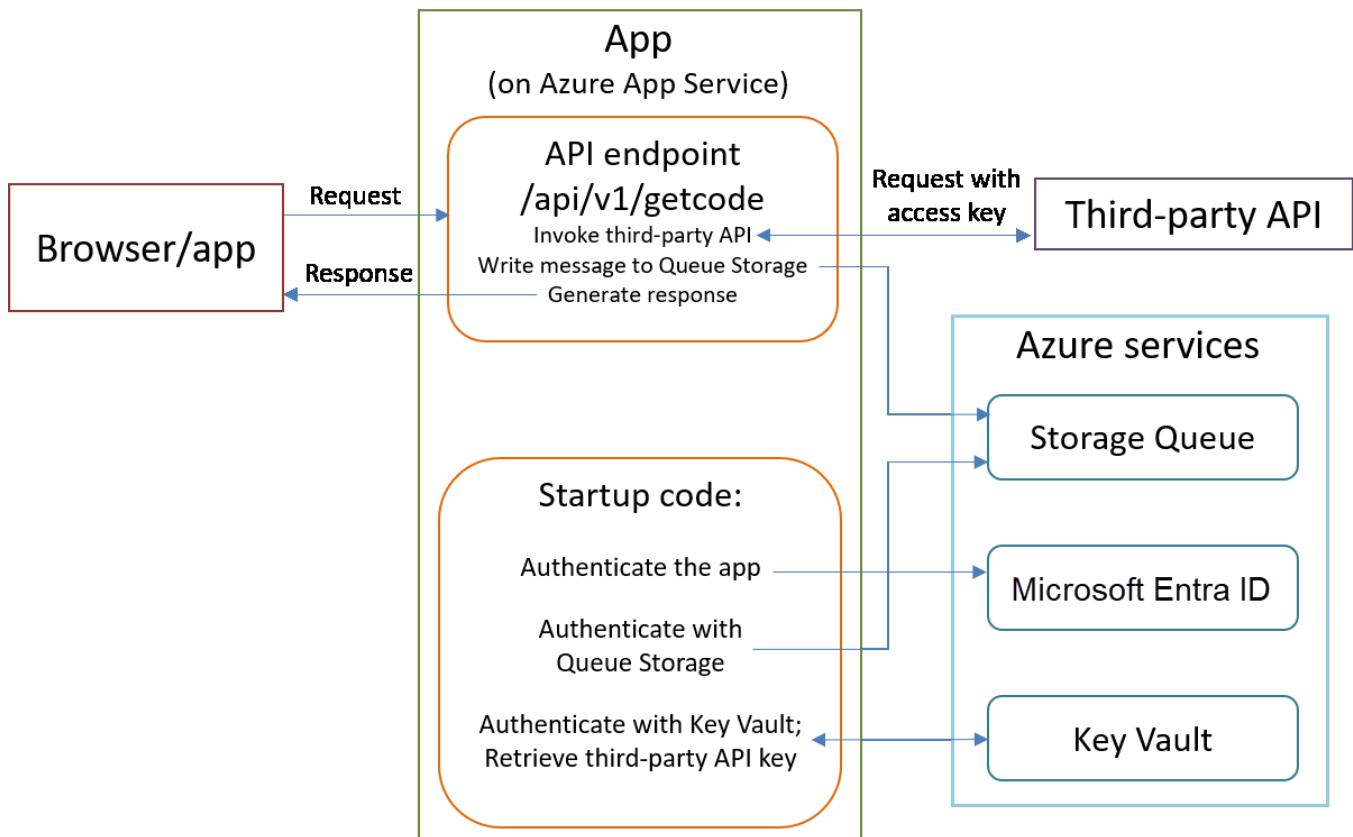
Example cloud app scenario

To understand Azure's authentication process more deeply, consider the following scenario: A Flask web application is deployed to Azure App Service. It exposes a public, unauthenticated API endpoint that returns JSON data in response to HTTP requests.

- To generate its response, the API invokes a third-party API that requires an access key. Instead of storing this key in code or configuration files, the app retrieves it securely at

runtime from Azure Key Vault using Microsoft Entra managed identity.

- Before returning its response to the client, the app writes a message to an Azure Storage Queue for asynchronous processing. The message could represent a task, log, or signal, though the downstream processing isn't the focus of this scenario.



(!) Note

Although public API endpoints are typically protected by their own access keys or authentication mechanisms, this walkthrough assumes the endpoint is open and unauthenticated.

This simplification helps isolate the app's internal authentication requirements—such as accessing Azure Key Vault and Azure Storage—from any authentication concerns related to external clients.

The scenario focuses solely on the app's behavior and doesn't demonstrate or involve an external caller authenticating with the endpoint.

[Part 2 - Authentication requirements >>>](#)

Part 2: Authentication needs in this example scenario

05/29/2025

[Previous part: Introduction and background](#)

In this example scenario, the main application has three distinct authentication requirements:

- Azure Key Vault

The application must authenticate with Azure Key Vault in order to retrieve a securely stored API key needed to call a third-party service.

- Third-party API

Once the API key is retrieved, the application uses it to authenticate with the external third-party API.

- Azure Queue Storage

After processing the request, the application must authenticate with Azure Queue Storage to enqueue a message for asynchronous or deferred processing.

These tasks require the app to manage three sets of credentials:

- Two for Azure resources (Key Vault and Storage)
- One for an external service (third-party API)

Key authentication challenges

Building secure cloud applications requires careful handling of credentials—especially when multiple services are involved. This example scenario presents several critical challenges:

- Circular dependency on Key Vault

The application uses Azure Key Vault to securely store secrets, such as third-party API keys or Azure Storage credentials. However, to retrieve those secrets, the app must first authenticate with Key Vault. This creates a circular problem: The app needs credentials to access Key Vault, but those credentials must themselves be stored securely. Without a secure solution, this could lead to hardcoded credentials or insecure configurations in development environments.

- Secure handling of third-party API keys

After retrieving the API key from Key Vault, the application must use it to call an external third-party service. This key must be handled with extreme care:

- Never hardcoded in source code or configuration files
- Never logged to stdout, stderr, or application logs
- Held only in memory and accessed at runtime, just before use
- Disposed promptly after the request is complete

Failure to follow these practices increases the risk of credential leakage or unauthorized use.

- Securing Azure Queue Storage credentials

To write messages to Azure Queue Storage, the app typically needs a connection string or shared access token. These credentials:

- Must be stored in a secure location, such as Key Vault
- Must not appear in logs, stack traces, or developer tools
- Should be accessed only through secure runtime mechanisms
- Require proper RBAC configuration if using managed identity

- Environment Flexibility

The app must run reliably in both local development and cloud production environments, using the same codebase and minimal conditional logic.

This means:

- No environment-specific secrets embedded in the code
- No need to manually toggle credentials or logic paths
- Consistent use of identity-based authentication across environments

Azure-First authentication with Microsoft Entra ID

As cloud applications scale in complexity and integrate with more services, secure and streamlined authentication becomes essential. Azure offers an “Azure-first” identity model through Microsoft Entra ID, enabling unified identity management and seamless integration with Azure services for secure, credential-free authentication.

Rather than manually managing secrets or embedding credentials in application code—a practice prone to security risks—Microsoft Entra ID enables apps to authenticate securely using managed identities.

The key benefits of Microsoft Entra managed identities are:

- No secrets in code

Applications no longer require hardcoded connection strings, client secrets, or keys.

- Built-in identity for apps

Azure can automatically assign a managed identity to your app, allowing secure access to services, such as Key Vault, Storage, and SQL without additional credentials.

- Environment consistency

The same code and identity model work both in local development and Azure-hosted environments using the Azure SDK's `DefaultAzureCredential`.

Environment-specific identity flow

Applications that use Microsoft Entra ID for authentication benefit from a flexible identity model that works seamlessly in both Azure-hosted and local development environments. This consistency is achieved using the Azure SDK's `DefaultAzureCredential`, which automatically selects the appropriate identity method based on the environment.

Azure environment

When the application is deployed to Azure:

- A managed identity is automatically assigned to the application.
- Azure handles token issuance and credential lifecycle internally—no manual secrets required.
- The application uses Role-Based Access Control (RBAC) or Key Vault access policies to access services

Local development environment

During local development:

- A service principal acts as the app's identity.
- Developers authenticate using the Azure CLI (`az login`), environment variables, or Visual Studio/VS Code integrations.
- The same application code runs without modification—only the identity source changes.

In both environments, Azure SDKs use the `DefaultAzureCredential`, which abstracts away the identity source and selects the right method automatically.

Best practices for secure development

While it's possible to set secrets as environment variables (for example, via Azure App Settings), this approach has downsides:

- You must manually replicate secrets in local environments.
- There's a risk of secrets leaking into source control.
- Additional logic may be required to differentiate between environments.

Instead, the recommended approach is:

- Use Key Vault to store third-party API keys and other secrets.
- Assign managed identity to your deployed app.
- Use a service principal for local development and assign it the same access rights.
- Use `DefaultAzureCredential` in your code to abstract authentication logic.
- Avoid storing or logging any credentials.

Authentication flow in practice

Here's how authentication works at runtime:

- Your code creates a `DefaultAzureCredential` instance.
- You use this credential to instantiate a client (for example, `SecretClient`, `QueueServiceClient`).
- When the app invokes a method (for example, `get_secret()`), the client uses the credential to authenticate the request.
- Azure verifies the identity and checks whether it has the correct role or policy to perform the operation.

This flow ensures that your app can securely access Azure services without embedding secrets in code or configuration files. It also allows you to seamlessly switch between local development and cloud deployment without changing your authentication logic.

[Part 3 - Third-party API implementation >>>](#)

Part 3: Example third-party API implementation

05/29/2025

[Previous part: Authentication requirements](#)

In our example scenario, the main application consumes a third-party API that is secured with an access key. This section demonstrates the API using Azure Functions, but the same principles apply regardless of how or where the API is implemented—whether you host the application on another cloud provider or a traditional web server.

The key aspect is that any client requests to the protected endpoint must include the access key, which the app must manage securely. This section provides an overview of how to implement such an API using Azure Functions, but you can adapt the principles to your specific needs.

Example third-party API implementation

The example third-party API is a simple endpoint that returns a random number between 1 and 999. The API is secured with an access key, which must be provided in the request to access the endpoint. For demonstration purposes, this API is deployed to the endpoint, `https://msdocs-example-api.azurewebsites.net/api/RandomNumber`. To call the API, however, you must provide the access key `d0c5atM1cr0s0ft` either in a `?code=` URL parameter or in an `'x-functions-key'` property of the HTTP header. For example, after you deploy the app and API, try this URL in a browser or curl: `https://msdocs-example-api.azurewebsites.net/api/RandomNumber?code=d0c5atM1cr0s0ft`.

If the access key is valid, the endpoint returns a JSON response that contains a single property, "value", the value of which is a number between 1 and 999, such as `{"value": 959}`.

The endpoint is implemented in Python and deployed to Azure Functions. The code is as follows:

Python

```
import logging
import random
import json

import azure.functions as func
```

```
def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('RandomNumber invoked via HTTP trigger.')

    random_value = random.randint(1, 1000)
    dict = { "value" : random_value }
    return func.HttpResponse(json.dumps(dict))
```

In the sample repository, this code is found under *third_party_api/RandomNumber/_init_.py*. The folder, *RandomNumber*, provides the name of the function and *_init_.py* contains the code. Another file in the folder, *function.json*, describes when the function is triggered. Other files in the *third_party_api* parent folder provide details for the Azure Function app that hosts the function itself.

To deploy the code, the sample's provisioning script performs the following steps:

1. Create a backing storage account for Azure Functions with the Azure CLI command, [az storage account create](#) for managing state and internal operations.
2. Create an Azure Functions app with the Azure CLI command, [az function app create](#).
3. After waiting 60 seconds for the host to be fully provisioned, deploy the code using the [Azure Functions Core Tools](#) command, [func azure functionapp publish](#).
4. Assign the access key, `d0c5atM1cr0s0ft`, to the function. (See [Securing Azure Functions](#) for a background on function keys.)

In the provisioning script, this step is accomplished using the [az functionapp function keys set](#) Azure CLI command.

Comments are included to show how to do this step through a REST API call to the [Functions Key Management API](#) if desired. To call that REST API, another REST API call must be done first to retrieve the Function app's master key.

You can also assign access keys through the [Azure portal](#). On the page for the Functions app, select **Functions**, then select the specific function to secure (which is named `RandomNumber` in this example). On the function's page, select **Function Keys** to open the page where you can create and manage these keys.

[Part 4 - Main app implementation >>>](#)

Part 4: Example main application implementation

05/28/2025

[Previous part: Third-party API implementation](#)

The main app in our scenario is a simple Flask app that's deployed to Azure App Service. The app provides a public API endpoint named `/api/v1/getcode`, which generates a code for some other purpose in the app (for example, with two-factor authentication for human users). The main app also provides a simple home page that displays a link to the API endpoint.

The sample's provisioning script performs the following steps:

1. Create the App Service host and deploy the code with the Azure CLI command, [az webapp up](#).
2. Create an Azure Storage account for the main app (using [az storage account create](#)).
3. Create a Queue in the storage account named "code-requests" (using [az storage queue create](#)).
4. To ensure that the app is allowed to write to the queue, use [az role assignment create](#) to assign the "Storage Queue Data Contributor" role to the app. For more information about roles, see [How to assign role permissions using the Azure CLI](#).

The main app code is as follows; explanations of important details are given in the next parts of this series.

Python

```
from flask import Flask, request, jsonify
import requests, random, string, os
from datetime import datetime
from azure.keyvault.secrets import SecretClient
from azure.identity import DefaultAzureCredential
from azure.storage.queue import QueueClient

app = Flask(__name__)
app.config["DEBUG"] = True

number_url = os.environ["THIRD_PARTY_API_ENDPOINT"]

# Authenticate with Azure. First, obtain the DefaultAzureCredential
credential = DefaultAzureCredential()

# Next, get the client for the Key Vault. You must have first enabled managed
identity
```

```

# on the App Service for the credential to authenticate with Key Vault.
key_vault_url = os.environ["KEY_VAULT_URL"]
keyvault_client = SecretClient(vault_url=key_vault_url, credential=credential)

# Obtain the secret: for this step to work you must add the app's service
principal to
# the key vault's access policies for secret management.
api_secret_name = os.environ["THIRD_PARTY_API_SECRET_NAME"]
vault_secret = keyvault_client.get_secret(api_secret_name)

# The "secret" from Key Vault is an object with multiple properties. The key we
# want for the third-party API is in the value property.
access_key = vault_secret.value

# Set up the Storage queue client to which we write messages
queue_url = os.environ["STORAGE_QUEUE_URL"]
queue_client = QueueClient.from_queue_url(queue_url=queue_url,
credential=credential)

@app.route('/', methods=['GET'])
def home():
    return f'Home page of the main app. Make a request to <a href=".//api/v1/getcode">/api/v1/getcode</a>.'

def random_char(num):
    return ''.join(random.choice(string.ascii_letters) for x in range(num))

@app.route('/api/v1/getcode', methods=['GET'])
def get_code():
    headers = {
        'Content-Type': 'application/json',
        'x-functions-key': access_key
    }

    r = requests.get(url = number_url, headers = headers)

    if (r.status_code != 200):
        return "Could not get you a code.", r.status_code

    data = r.json()
    chars1 = random_char(3)
    chars2 = random_char(3)
    code_value = f'{chars1}-{data["value"]}-{chars2}'
    code = { "code": code_value, "timestamp" : str(datetime.utcnow()) }

    # Log a queue message with the code for, say, a process that invalidates
    # the code after a certain period of time.
    queue_client.send_message(code)

    return jsonify(code)

```

```
if __name__ == '__main__':
    app.run()
```

Part 5 - Dependencies and environment variables >>>

Part 5: Main app dependencies, import statements, and environment variables

05/29/2025

[Previous part: Main app implementation](#)

This section reviews the Python libraries imported by the main application and the environment variables it depends on. When the app is deployed to Azure, these environment variables are supplied through Application Settings in Azure App Service.

Dependencies and import statements

The application relies on the following libraries:

- Flask – to define the web API
- requests – the standard Python HTTP client for making outbound API calls
- azure.identity – for handling Microsoft Entra ID token-based authentication
- azure.keyvault.secrets – to securely retrieve secrets from Azure Key Vault
- azure.storage.queue – to interact with Azure Queue Storage

These dependencies are included in the app's *requirements.txt* file and are installed during deployment or local setup.

txt

```
flask
requests
azure.identity
azure.keyvault.secrets
azure.storage.queue
```

When you deploy the app to Azure App Service, Azure automatically installs these requirements on the host server. When running locally, you install them in your environment with `pip install -r requirements.txt`.

The code file starts with the required import statements for the parts of the libraries used in the code:

Python

```
from flask import Flask, request, jsonify
import requests, random, string, os
from datetime import datetime
```

```
from azure.keyvault.secrets import SecretClient
from azure.identity import DefaultAzureCredential
from azure.storage.queue import QueueClient
```

Environment variables

The app code depends on these four environment variables:

[Expand table]

Variable	Value
THIRD_PARTY_API_ENDPOINT	The URL of the third-party API, such as <code>https://msdocs-example-api.azurewebsites.net/api/RandomNumber</code> described in Part 3 .
KEY_VAULT_URL	The URL of the Azure Key Vault in which you stored the access key for the third-party API.
THIRD_PARTY_API_SECRET_NAME	The name of the secret in Key Vault that contains the access key for the third-party API.
STORAGE_QUEUE_URL	The URL of an Azure Storage Queue that you configure in Azure, such as <code>https://msdocsexamplemainapp.queue.core.windows.net/code-requests</code> (see Part 4). Because the queue name is included at the end of the URL, you don't see the name anywhere in the code.

How you set these variables depends on where the code is running:

- When running the code locally, you create these variables within whatever command shell you're using (such as PowerShell, Bash, or CMD). (If you deploy the app to a virtual machine, you would create similar server-side variables.) You can also use a library like [python-dotenv](#), which reads key-value pairs from an `.env` file and sets them as environment variables
- When the code is deployed to Azure App Service, as is shown in this walkthrough, you don't have access to the server itself. Instead, you define *application settings* with the same names in the App Service configuration. These settings are automatically exposed to the application as environment variables.

The provisioning scripts create these settings using the Azure CLI command, [az webapp config appsettings set](#). All four variables are set with a single command.

To create settings through the Azure portal, see [Configure an App Service app in the Azure portal](#).

When running the code locally, you also need to specify environment variables that contain information about your local service principal. `DefaultAzureCredential` looks for these values. When deployed to App Service, you don't need to set these values as the app's system-assigned managed identity is used instead to authenticate.

 [Expand table](#)

Variable	Value
AZURE_TENANT_ID	The Microsoft Entra tenant (directory) ID.
AZURE_CLIENT_ID	The client (application) ID of an App Registration in the tenant.
AZURE_CLIENT_SECRET	A client secret that was generated for the App Registration.

For more information, see [Authenticate Python apps to Azure services during local development using service principals](#).

[Part 6 - Main app startup code >>>](#)

Part 6: Main app startup code

05/29/2025

[Previous part: Dependencies and environment variables](#)

Immediately following the `import` statements, the app's startup code initializes key variables used throughout the request-handling functions.

First, the application creates the Flask app object, which serves as the foundation for defining routes and handling incoming HTTP requests. Next, it retrieves the third-party API endpoint URL from an environment variable. This allows the endpoint to be easily configured without modifying the codebase:

Python

```
app = Flask(__name__)
app.config["DEBUG"] = True

number_url = os.environ["THIRD_PARTY_API_ENDPOINT"]
```

Next, it obtains the `DefaultAzureCredential` object, which is the recommended credential to use when authenticating with Azure services. See [Authenticate Azure hosted applications with DefaultAzureCredential](#).

Python

```
credential = DefaultAzureCredential()
```

When run locally, `DefaultAzureCredential` looks for the `AZURE_TENANT_ID`, `AZURE_CLIENT_ID`, and `AZURE_CLIENT_SECRET` environment variables that contain information for the service principal that you're using for local development. When run in Azure, `DefaultAzureCredential` defaults to using the system-assigned managed identity enabled on the app. It's possible to override the default behavior with application settings, but in this example scenario, we use the default behavior.

The code next retrieves the third-party API's access key from Azure Key Vault. In the provisioning script, the Key Vault is created using `az keyvault create`, and the secret is stored with `az keyvault secret set`.

The Key Vault resource itself is accessed through a URL, which is loaded from the `KEY_VAULT_URL` environment variable.

Python

```
key_vault_url = os.environ["KEY_VAULT_URL"]
```

To retrieve a secret from Azure Key Vault, the application must create a client object that communicates with the Key Vault service. Since the goal is to read a secret, the app uses the [SecretClient](#) class from the `azure.keyvault.secrets` library. This client requires two inputs:

- The Key Vault URL – typically retrieved from an environment variable
- A credential object – such as the `DefaultAzureCredential` instance created earlier, which represents the identity under which the app is running.

Python

```
keyvault_client = SecretClient(vault_url=key_vault_url, credential=credential)
```

Creating a `SecretClient` object does not immediately authenticate the application. The client is simply a local construct that stores the Key Vault URL and the credential object. Authentication and authorization happen only when you invoke an operation through the client, such as [get_secret](#), which generates a REST API call to the Azure resource.

Python

```
api_secret_name = os.environ["THIRD_PARTY_API_SECRET_NAME"]
vault_secret = keyvault_client.get_secret(api_secret_name)

# The "secret" from Key Vault is an object with multiple properties. The key we
# want for the third-party API is in the value property.
access_key = vault_secret.value
```

Even if an application's identity is authorized to access Azure Key Vault, it must also be explicitly authorized to perform specific operations—such as reading secrets. Without this permission, a call to `get_secret()` fails, even if the identity is otherwise valid. To address this, the provisioning script sets a "get secrets" access policy for the app using the Azure CLI command, [az keyvault set-policy](#). For more information, see [Key Vault Authentication](#) and [Grant your app access to Key Vault](#). The latter article shows how to set an access policy using the Azure portal. (The article is also written for managed identity, but applies equally to a service principle used in local development.)

Finally, the app code sets up the client object through which it can write messages to an Azure Storage Queue. The Queue's URL is in the environment variable `STORAGE_QUEUE_URL`.

Python

```
queue_url = os.environ["STORAGE_QUEUE_URL"]
queue_client = QueueClient.from_queue_url(queue_url=queue_url,
credential=credential)
```

As with Azure Key Vault, the application uses a specific client object from the Azure SDK to interact with Azure Queue Storage. In this case, it uses the [QueueClient](#) class from the `azure-storage-queue` library.

To initialize the client, the app uses the [from_queue_url](#) method, providing the queue's fully qualified URL and a credential object. This credential object is again the `DefaultAzureCredential` instance created earlier, which represents the identity under which the app is running.

As noted earlier in this guide, that authorization is granted by assigning the "Storage Queue Data Contributor" role to the application's identity - either a managed identity in Azure or a service principal during local development. This role assignment is done in the provisioning script using the Azure CLI command [az role assignment create](#).

Assuming all this startup code succeeds, the app has all its internal variables in place to support its `/api/v1/getcode` API endpoint.

[Part 7 - Main app endpoint >>>](#)

Part 7: Main application API endpoint

05/28/2025

[Previous part: Main app startup code](#)

The app URL path `/api/v1/getcode` for the API generates a JSON response that contains an alphanumerical code and a timestamp.

First, the `@app.route` decorator tells Flask that the `get_code` function handles requests to the `/api/v1/getcode` URL.

Python

```
@app.route('/api/v1/getcode', methods=['GET'])
def get_code():
```

Next, the app calls the third-party API, the URL of which is in `number_url`, providing the access key that it retrieves from the key vault in the header.

Python

```
headers = {
    'Content-Type': 'application/json',
    'x-functions-key': access_key
}

r = requests.get(url = number_url, headers = headers)

if (r.status_code != 200):
    return "Could not get you a code.", r.status_code
```

The example third-party API is deployed to the serverless environment of Azure Functions. The `x-functions-key` property in the header is how Azure Functions expects an access key to appear in a header. For more information, see [Azure Functions HTTP trigger - Authorization keys](#). If calling the API fails for any reason, the code returns an error message and the status code.

Assuming that the API call succeeds and returns a numerical value, the app then constructs a more complex code using that number plus some random characters (using its own `random_char` function).

Python

```
data = r.json()
chars1 = random_char(3)
```

```
chars2 = random_char(3)
code_value = f"{chars1}-{data['value']}-{chars2}"
code = { "code": code_value, "timestamp" : str(datetime.utcnow()) }
```

The `code` variable here contains the full JSON response for the app's API, which includes the code value and a timestamp. An example response would be `{"code": "ojE-161-pTv", "timestamp": "2020-04-15 16:54:48.816549"}`.

Before it returns that response, however, it writes a message in the storage queue using the Queue client's `send_message` method:

Python

```
queue_client.send_message(code)

return jsonify(code)
```

Processing queue messages

Messages stored in the queue can be viewed and managed through the [Azure portal](#), with the Azure CLI command `az storage message get` or with [Azure Storage Explorer](#). The sample repository includes a script (`test.cmd` and `test.sh`) to request a code from the app endpoint and then check the message queue. There's also a script to clear the queue using the [az storage message clear](#) command.

Typically, an app like the one in this example would have another process that asynchronously pulls messages from the queue for further processing. As mentioned previously, the response generated by this API endpoint might be used elsewhere in the app with two-factor user authentication. In that case, the app should invalidate the code after a certain period of time, for example 10 minutes. A simple way to do this task would be to maintain a table of valid two-factor authentication codes, which are used by its user sign-in procedure. The app would then have a simple queue-watching process with the following logic (in pseudo-code):

pseudo

```
pull a message from the queue and retrieve the code.

if (code is already in the table):
    remove the code from the table, thereby invalidating it
else:
    add the code to the table, making it valid
    call queue_client.send_message(code, visibility_timeout=600)
```

This pseudo-code employs the `send_message` method's optional `visibility_timeout` parameter, which specifies the number of seconds before the message becomes visible in the queue. Because the default timeout is zero, messages initially written by the API endpoint become immediately visible to the queue-watching process. As a result, that process stores them in the valid code table right away. The process queues the same message again with the timeout, so that it receives the code again 10 minutes later, at which point it removes it from the table.

Implementing the main app API endpoint in Azure Functions

The code shown previously in this article uses the Flask web framework to create its API endpoint. Because Flask needs to run with a web server, such code must be deployed to Azure App Service or to a virtual machine.

An alternate deployment option is the serverless environment of Azure Functions. In this case, all the startup code and the API endpoint code would be contained within the same function that's bound to an HTTP trigger. As with App Service, you use [function application settings](#) to create environment variables for your code.

One piece of the implementation that becomes easier is authenticating with Queue Storage. Instead of obtaining a `QueueClient` object using the queue's URL and a credential object, you create a *queue storage binding* for the function. The binding handles all the authentication behind the scenes. With such a binding, your function is given a ready-to-use client object as a parameter. For more information and example code, see [Connect Azure Functions to Azure Queue Storage](#).

Next steps

Through this tutorial, you learned how apps authenticate with other Azure services using managed identity, and how apps can use Azure Key Vault to store any other necessary secrets for third-party APIs.

The same pattern demonstrated here with Azure Key Vault and Azure Storage applies with all other Azure services. The crucial step is that you assign the correct role for the app within that service's page on the Azure portal, or through the Azure CLI. (See [How to assign Azure roles](#)). Be sure to check the service documentation to see whether you need to configure any other access policies.

Always remember that you need to assign the same roles and access policies to any service principal you're using for local development.

In short, having completed this walkthrough, you can apply your knowledge to any number of other Azure services and any number of other external services.

One subject that we haven't touched upon in this tutorial is authentication of *users*. To explore this area for web apps, begin with [Authenticate and authorize users end-to-end in Azure App Service](#).

See also

- [How to authenticate and authorize Python apps on Azure](#)
- Walkthrough sample: [github.com/Azure-Samples/python-integrated-authentication ↗](https://github.com/Azure-Samples/python-integrated-authentication)
- [Microsoft Entra documentation](#)
- [Azure Key Vault documentation](#)

Authorization in the Azure SDK libraries for Python

07/11/2025

Authorization in Azure determines what actions authenticated users or services can perform on resources. This article explores how to implement authorization using the [Azure SDK for Python](#), covering models, implementation, troubleshooting, and best practices. For detailed authentication setup, refer to [Authenticate Python apps to Azure](#).

Introduction

Authentication (AuthN) verifies the identity of a user or service, while **authorization (AuthZ)** defines what they can do. In Azure, authorization ensures secure access to resources, critical for protecting applications and data. Developers can implement robust authorization with the Azure SDK for Python to control access in various workflows, from managing resources to accessing service-specific data.

Azure authorization models

Azure provides multiple authorization mechanisms to manage access. Understanding these models is essential for effective access control.

Azure Role-Based Access Control

[Azure Role-Based Access Control \(RBAC\)](#) assigns roles to identities at scopes like subscriptions or resource groups. Built-in roles include Owner, Contributor, and Reader, while custom roles allow tailored permissions. When you assign a role at a specific scope, the identity (like a user or service) gets permissions for all resources within that scope and its child scopes. For example, assigning the Contributor role at the subscription level allows management of all resources in that subscription. See [Understand scope for Azure RBAC](#).

Service-specific mechanisms

Some Azure services offer unique authorization methods:

- **Azure Storage:** Uses Shared Access Signatures (SAS) and Access Control Lists (ACLs) for data access. See [Service-Specific Authorization Notes for Azure Storage](#).
- **Azure Key Vault:** Recommends RBAC over legacy access policies. See [Service-Specific Authorization Notes for Azure Key Vault](#).

- **Microsoft Graph:** Employs OAuth 2.0 scopes and application permissions. See [Service-Specific Authorization Notes for Microsoft Graph](#).

Use authorization in Azure SDK for Python

The Azure SDK for Python provides built-in support for handling authentication and authorization through the `azure-identity` package. The `DefaultAzureCredential` class supports various authentication mechanisms, such as managed identities and service principals, adapting to different environments.

Example: List resource groups

This example shows how the Azure SDK for Python uses a credential (via `DefaultAzureCredential`) to authenticate, and how authorization determines whether the identity can successfully list resource groups. If the identity lacks the Reader or higher role on the subscription or resource group scope, this call returns a 403 Forbidden error.

Python

```
from azure.identity import DefaultAzureCredential
from azure.mgmt.resource import ResourceManagementClient

credential = DefaultAzureCredential()
client = ResourceManagementClient(credential, "<subscription-id>")
resource_groups = client.resource_groups.list()
for rg in resource_groups:
    print(rg.name)
```

Replace `<subscription-id>` with your Azure subscription ID, which is usually in the form of `00000000-0000-0000-0000-000000000000`.

Microsoft Graph with scopes

To access Microsoft Graph, use the official [Microsoft Graph SDK for Python](#), which supports both delegated and application permissions.

This example demonstrates how the SDK uses a credential to request an access token with the required authorization scope `https://graph.microsoft.com/.default` and access Microsoft Graph resources. The identity must be authorized in Microsoft Entra ID with appropriate application permissions (such as `User.Read.All`) to retrieve user data; otherwise, the request fails with a 403 Forbidden.

Important

Ensure your app or identity has the `User.Read.All` or other required permissions granted in Microsoft Entra ID.

Python

```
from azure.identity import DefaultAzureCredential
from msgraph.core import GraphClient

credential = DefaultAzureCredential()
client = GraphClient(credential=credential, scopes=[
    "https://graph.microsoft.com/.default"])

response = client.get("/users")
users = response.json().get("value", [])
for user in users:
    print(user["displayName"])
```

Learn more about this SDK in the official [Build Python apps with Microsoft Graph](#) tutorial.

Diagnose authorization errors

Authorization issues often result in HTTP 403 Forbidden errors, indicating insufficient permissions. To diagnose:

- **Check Error Messages:** Review the response for details on missing permissions.
- **Enable Logging:** Use Python logging to inspect requests and responses:

Python

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

- **Verify Access:** Use [Azure CLI](#) or the Azure portal to check role assignments:

Azure CLI

```
az role assignment list --assignee <principal-id> --scope <scope>
```

Replace `<principal-id>` with the object ID of your user, service principal, or managed identity. Replace `<scope>` with an Azure resource scope, such as a subscription ID, a resource group name, or a resource. See [Work with scopes](#).

Work with scopes

Often you'll need to provide a scope, like in the example:

Azure CLI

```
az role assignment list \
--assignee <principal-id> \
--scope <scope>
```

Here are some common scope formats:

 Expand table

Scope Level	Example Value
Subscription	/subscriptions/<subscription-id>
Resource Group	/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>
Resource Name	/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>/providers/<provider-namespace>/<resource-type>/<resource-name>

For example, to list all role assignments for a managed identity at the resource group level:

Azure CLI

```
az role assignment list \
--assignee 12345678-90ab-cdef-1234-567890abcdef \
--scope /subscriptions/<subscription-id>/resourceGroups/my-resource-group
```

Or at the subscription level:

Azure CLI

```
az role assignment list \
--assignee 12345678-90ab-cdef-1234-567890abcdef \
--scope /subscriptions/<subscription-id>
```

You can retrieve the object ID (<principal-id>) of a user or managed identity using:

Azure CLI

```
az ad user show --id <user-email> --query objectId
az identity show --name <identity-name> --resource-group <rg-name> --query
```

principalId

Manage access

Manage access through role assignments using:

- **Azure portal:** Add roles via the "Access control (IAM)" service menu
- **Azure CLI:**

Azure CLI

```
az role assignment create --assignee <principal-id> --role <role-name> --  
scope <scope>
```

Replace `<principal-id>` with the object ID of your user, service principal, or managed identity. Replace `<scope>` with an Azure resource scope, such as a subscription ID, a resource group name, or a resource name. See [Work with scopes](#).

- **ARM Templates:** For declarative management.

For managed identities, assign roles to the identity associated with resources like virtual machines. Use the Azure portal's "Check access" feature or the Azure CLI to verify effective permissions.

Service-specific authorization notes

In the section [Service-specific mechanisms](#), it was noted that some Azure services offer unique authorization methods. This section provides more detail for each of the three Azure services mentioned.

Azure Storage

- **RBAC:** Manages control plane operations.
- **SAS and ACLs:** Control data plane access, with Microsoft Entra authentication also supported.

Example: Access blobs with Microsoft Entra ID

Python

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

credential = DefaultAzureCredential()
client = BlobServiceClient(account_url="https://<account-name>.blob.core.windows.net", credential=credential)
containers = client.list_containers()
for container in containers:
    print(container.name)
```

Replace `<account-name>` with your Azure Storage account name.

Azure Key Vault

RBAC is recommended over legacy access policies for consistency. Access policies are still supported but not preferred.

Example: Retrieve a secret

Python

```
from azure.identity import DefaultAzureCredential
from azure.keyvault.secrets import SecretClient

credential = DefaultAzureCredential()
client = SecretClient(vault_url="https://<vault-name>.vault.azure.net",
                      credential=credential)
secret = client.get_secret("my-secret")
print(secret.value)
```

Replace `<vault-name>` with your Key Vault resource name.

Microsoft Graph

Uses OAuth 2.0 scopes for delegated permissions and application permissions for daemon apps. Specify scopes as shown in the earlier example.

Best practices

- **Least privilege:** Assign only necessary permissions (for example, Reader instead of Contributor).
- **Prefer RBAC:** Especially for Key Vault, for unified access control.
- **Use Managed Identities:** Avoid managing credentials in code.

- **Limit Graph permissions:** Request specific scopes to minimize risks.

Next steps

- [Azure Role-Based Access Control \(RBAC\)](#)
- [Azure CLI Reference](#)
- [Azure SDK for Python Overview](#)
- [Azure Key Vault RBAC Guide](#)
- [Microsoft Graph Permissions Reference](#)

How to install Azure library packages for Python

Article • 02/05/2025

The Azure SDK for Python is composed of many individual libraries that can be installed in standard [Python](#) or [conda](#) environments.

Libraries for standard Python environments are listed in the [package index](#).

Packages for conda environments are listed in the [Microsoft channel on anaconda.org](#). Azure packages have names that begin with `azure-`.

With these Azure libraries, you can create and manage resources on Azure services (using the management libraries, whose package names begin with `azure-mgmt`) and connect with those resources from app code (using the client libraries, whose package names begin with just `azure-`).

Install the latest version of a package

pip

Windows Command Prompt

```
pip install <package>
```

`pip install` retrieves the latest version of a package in your current Python environment.

On Linux systems, you must install a package for each user separately. Installing packages for all users with `sudo pip install` isn't supported.

You can use any package name listed in the [package index](#). On the index page, look in the **Name** column for the functionality you need, and then find and select the PyPI link in the **Package** column.

Install specific package versions

pip

Specify the desired version on the command line with `pip install`.

Windows Command Prompt

```
pip install <package>==<version>
```

You can find version numbers in the [package index](#). On the index page, look in the **Name** column for the functionality you need, and then find and select the PyPI link in the **Package** column. For example, to install a version of the `azure-storage-blob` package you can use: `pip install azure-storage-blob==12.19.0`.

Install preview packages

pip

To install the latest preview of a package, include the `--pre` flag on the command line.

Windows Command Prompt

```
pip install --pre <package>
```

Microsoft periodically releases preview packages that support upcoming features. Preview packages come with the caveat that the package is subject to change and must not be used in production projects.

You can use any package name listed in the [package index](#).

Verify a package installation

pip

To verify a package installation:

Windows Command Prompt

```
pip show <package>
```

If the package is installed, `pip show` displays version and other summary information, otherwise the command displays nothing.

You can also use `pip freeze` or `pip list` to see all the packages that are installed in your current Python environment.

You can use any package name listed in the [package index](#).

Uninstall a package

pip

To uninstall a package:

Windows Command Prompt

```
pip uninstall <package>
```

You can use any package name listed in the [package index](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Example: Use the Azure libraries to create a resource group

05/30/2025

This example demonstrates how to use the Azure SDK management libraries in a Python script to create a resource group. (The [Equivalent Azure CLI command](#) is given later in this article. If you prefer to use the Azure portal, see [Create resource groups](#).)

All the commands in this article work the same in Linux/macOS bash and Windows command shells unless noted.

1: Set up your local development environment

If you haven't already, set up an environment where you can run this code. Here are some options:

- Configure a Python virtual environment using `venv` or your tool of choice. To start using the virtual environment, be sure to activate it. To install python, see [Install Python](#).



A screenshot of a terminal window. The title bar says "Bash". The window contains the following text:

```
#!/bin/bash
# Create a virtual environment
python -m venv .venv
# Activate the virtual environment
source .venv/Scripts/activate # only required for Windows (Git Bash)
```

- Use a [conda environment](#). To install Conda, see [Install Miniconda](#).
- Use a [Dev Container](#) in [Visual Studio Code](#) or [GitHub Codespaces](#).

2: Install the Azure library packages

- In your console, create a `requirements.txt` file that lists the management libraries used in this example:

```
Azure CLI
```

```
azure-mgmt-resource  
azure-identity
```

2. In your console with the virtual environment activated, install the requirements:

Console

```
pip install -r requirements.txt
```

3. Set environment variables

In this step, you set environment variables for use in the code in this article. The code uses the `os.environ` method to retrieve the values.

Bash

Azure CLI

```
#!/bin/bash
export AZURE_RESOURCE_GROUP_NAME=<ResourceGroupName> # Change to your
preferred resource group name
export LOCATION=<Location> # Change to your preferred region
export AZURE_SUBSCRIPTION_ID=$(az account show --query id --output tsv)
```

4: Write code to create a resource group

In this step, you create a Python file named `provision_blob.py` with the following code. This Python script uses the Azure SDK for Python management libraries to create a resource group in your Azure subscription.

Create a Python file named `provision_rg.py` with the following code. The comments explain the details:

Python

```
# Import the needed credential and management objects from the libraries.
import os

from azure.identity import DefaultAzureCredential
from azure.mgmt.resource import ResourceManagementClient

# Acquire a credential object using DefaultAzureCredential.
```

```
credential = DefaultAzureCredential()

# Retrieve subscription ID from environment variable.
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]

# Retrieve resource group name and location from environment variables
RESOURCE_GROUP_NAME = os.environ["AZURE_RESOURCE_GROUP_NAME"]
LOCATION = os.environ["LOCATION"]

# Obtain the management object for resources.
resource_client = ResourceManagementClient(credential, subscription_id)

# Provision the resource group.
rg_result = resource_client.resource_groups.create_or_update(RESOURCE_GROUP_NAME,
    { "location": LOCATION })

print(f"Provisioned resource group {rg_result.name}")

# Within the ResourceManagementClient is an object named resource_groups,
# which is of class ResourceGroupsOperations, which contains methods like
# create_or_update.
#
# The second parameter to create_or_update here is technically a ResourceGroup
# object. You can create the object directly using ResourceGroup(location=
# LOCATION) or you can express the object as inline JSON as shown here. For
# details, see Inline JSON pattern for object arguments at
# https://learn.microsoft.com/azure/developer/python/sdk
# /azure-sdk-library-usage-patterns#inline-json-pattern-for-object-arguments

print(
    f"Provisioned resource group {rg_result.name} in the {rg_result.location}
region"
)

# The return value is another ResourceGroup object with all the details of the
# new group. In this case the call is synchronous: the resource group has been
# provisioned by the time the call returns.

# To update the resource group, repeat the call with different properties, such
# as tags:
rg_result = resource_client.resource_groups.create_or_update(
    RESOURCE_GROUP_NAME,
    {
        "location": LOCATION,
        "tags": {"environment": "test", "department": "tech"},
    },
)

print(f"Updated resource group {rg_result.name} with tags")

# Optional lines to delete the resource group. begin_delete is asynchronous.
# poller = resource_client.resource_groups.begin_delete(rg_result.name)
# result = poller.result()
```

Authentication in the code

Later in this article, you sign in to Azure using the Azure CLI to execute the sample code. If your account has sufficient permissions to create resource groups and storage resources in your Azure subscription, the script should run successfully without additional configuration.

To use this code in a production environment, authenticate using a service principal by setting environment variables. This approach enables secure, automated access without relying on interactive login. For detailed guidance, see [How to authenticate Python apps with Azure services](#).

Ensure that the service principal is assigned a role with sufficient permissions to create resource groups and storage accounts. For example, assigning the Contributor role at the subscription level provides the necessary access. To learn more about role assignments, see [Role-based access control \(RBAC\) in Azure](#).

Reference links for classes used in the code

- [DefaultAzureCredential \(azure.identity\)](#)
- [ResourceManagementClient \(azure.mgmt.resource\)](#)

5: Run the script

1. If you haven't already, sign in to Azure using the Azure CLI:

```
Azure CLI
```

```
az login
```

2. Run the script:

```
Windows Command Prompt
```

```
python provision_rg.py
```

6: Verify the resource group

You can verify that the resource group exists through the Azure portal or the Azure CLI.

- Azure portal: open the [Azure portal](#), select **Resource groups**, and check that the group is listed. If necessary, use the **Refresh** command to update the list.

- Azure CLI: use the [az group show](#) command:

```
Bash
```

```
Azure CLI
```

```
#!/bin/bash
az group show -n $AZURE_RESOURCE_GROUP_NAME
```

7: Clean up resources

Run the [az group delete](#) command if you don't need to keep the resource group created in this example. Resource groups don't incur any ongoing charges in your subscription, but resources in the resource group might continue to incur charges. It's a good practice to clean up any group that you aren't actively using. The `--no-wait` argument allows the command to return immediately instead of waiting for the operation to finish.

```
Bash
```

```
Azure CLI
```

```
#!/bin/bash
az group delete -n $AZURE_RESOURCE_GROUP_NAME --no-wait
```

You can also use the [ResourceManagementClient.resource_groups.begin_delete](#) method to delete a resource group from code. The commented code at the bottom of the script in this article demonstrates the usage.

For reference: equivalent Azure CLI command

The following Azure CLI [az group create](#) command creates a resource group with tags just like the Python script:

```
Azure CLI
```

```
az group create -n PythonAzureExample-rg -l centralus --tags "department=tech"
"environment=test"
```

See also

- Example: List resource groups in a subscription
- Example: Create Azure Storage
- Example: Use Azure Storage
- Example: Create a web app and deploy code
- Example: Create and query a database
- Example: Create a virtual machine
- Use Azure Managed Disks with virtual machines
- Complete a short survey about the Azure SDK for Python ↗

Example: Use the Azure libraries to list resource groups and resources

Article • 04/23/2025

This example demonstrates how to use the Azure SDK management libraries in a Python script to perform two tasks:

- List all the resource groups in an Azure subscription.
- List resources within a specific resource group.

All the commands in this article work the same in Linux/macOS bash and Windows command shells unless noted.

The [Equivalent Azure CLI commands](#) are listed later in this article.

1: Set up your local development environment

If you haven't already, set up an environment where you can run this code. Here are some options:

- Configure a Python virtual environment using `venv` or your tool of choice. To start using the virtual environment, be sure to activate it. To install python, see [Install Python](#).

Bash

```
Azure CLI
#!/bin/bash
# Create a virtual environment
python -m venv .venv
# Activate the virtual environment
source .venv/Scripts/activate # only required for Windows (Git Bash)
```

- Use a [conda environment](#). To install Conda, see [Install Miniconda](#).
- Use a [Dev Container](#) in [Visual Studio Code](#) or [GitHub Codespaces](#).

2: Install the Azure library packages

Create a file named `requirements.txt` with the following contents:

```
txt
```

```
azure-mgmt-resource  
azure-identity
```

In a terminal or command prompt with the virtual environment activated, install the requirements:

Console

```
pip install -r requirements.txt
```

3: Write code to work with resource groups

3a. List resource groups in a subscription

Create a Python file named *list_groups.py* with the following code. The comments explain the details:

Python

```
# Import the needed credential and management objects from the libraries.  
from azure.identity import DefaultAzureCredential  
from azure.mgmt.resource import ResourceManagementClient  
import os  
  
# Acquire a credential object.  
credential = DefaultAzureCredential()  
  
# Retrieve subscription ID from environment variable.  
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]  
  
# Obtain the management object for resources.  
resource_client = ResourceManagementClient(credential, subscription_id)  
  
# Retrieve the list of resource groups  
group_list = resource_client.resource_groups.list()  
  
# Show the groups in formatted output  
column_width = 40  
  
print("Resource Group".ljust(column_width) + "Location")  
print("-" * (column_width * 2))  
  
for group in list(group_list):  
    print(f"{group.name:<{column_width}}{group.location}")
```

3b. List resources within a specific resource group

Create a Python file named *list_resources.py* with the following code. The comments explain the details.

By default, the code lists resources in "myResourceGroup". To use a different resource group, set the `RESOURCE_GROUP_NAME` environment variable to the desired group name.

Python

```
# Import the needed credential and management objects from the libraries.
from azure.identity import DefaultAzureCredential
from azure.mgmt.resource import ResourceManagementClient
import os

# Acquire a credential object.
credential = DefaultAzureCredential()

# Retrieve subscription ID from environment variable.
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]

# Retrieve the resource group to use, defaulting to "myResourceGroup".
resource_group = os.getenv("RESOURCE_GROUP_NAME", "myResourceGroup")

# Obtain the management object for resources.
resource_client = ResourceManagementClient(credential, subscription_id)

# Retrieve the list of resources in "myResourceGroup" (change to any name
# desired).
# The expand argument includes additional properties in the output.
resource_list = resource_client.resources.list_by_resource_group(
    resource_group, expand = "createdTime,changedTime")

# Show the groups in formatted output
column_width = 36

print("Resource".ljust(column_width) + "Type".ljust(column_width)
    + "Create date".ljust(column_width) + "Change date".ljust(column_width))
print("-" * (column_width * 4))

for resource in list(resource_list):
    print(f"{resource.name:{column_width}}{resource.type:{column_width}}"
        f"{str(resource.created_time):{column_width}}{str(resource.changed_time):"
        f"{column_width}}")
```

Authentication in the code

Later in this article, you sign in to Azure with the Azure CLI to run the sample code. If your account has permissions to create and list resource groups in your Azure subscription, the code

will run successfully.

To use such code in a production script, you can set environment variables to use a service principal-based method for authentication. To learn more, see [How to authenticate Python apps with Azure services](#). You need to ensure that the service principal has sufficient permissions to create and list resource groups in your subscription by assigning it an appropriate [role in Azure](#); for example, the *Contributor* role on your subscription.

Reference links for classes used in the code

- [DefaultAzureCredential \(azure.identity\)](#)
- [ResourceManagementClient \(azure.mgmt.resource\)](#)

4: Run the scripts

1. If you haven't already, sign in to Azure using the Azure CLI:

```
Azure CLI
```

```
az login
```

2. Set the `AZURE_SUBSCRIPTION_ID` environment variable to your subscription ID. (You can run the [az account show](#) command and get your subscription ID from the `id` property in the output):

```
Bash
```

```
Bash
```

```
export AZURE_SUBSCRIPTION_ID=00000000-0000-0000-0000-000000000000
```

3. List all resources groups in the subscription:

```
Console
```

```
python list_groups.py
```

4. List all resources in a resource group:

```
Console
```

```
python list_resources.py
```

By default, the code lists resources in "myResourceGroup". To use a different resource group, set the `RESOURCE_GROUP_NAME` environment variable to the desired group name.

For reference: equivalent Azure CLI commands

The following Azure CLI command lists resource groups in a subscription:

```
Azure CLI
```

```
az group list
```

The following command lists resources within the "myResourceGroup" in the centralus region (the `location` argument is necessary to identify a specific data center):

```
Azure CLI
```

```
az resource list --resource-group myResourceGroup --location centralus
```

See also

- [Example: Provision a resource group](#)
- [Example: Provision Azure Storage](#)
- [Example: Use Azure Storage](#)
- [Example: Provision a web app and deploy code](#)
- [Example: Provision and query a database](#)
- [Example: Provision a virtual machine](#)
- [Use Azure Managed Disks with virtual machines](#)
- [Complete a short survey about the Azure SDK for Python ↗](#)

Example: Create Azure Storage using the Azure libraries for Python

05/30/2025

In this article, you learn how to use the Azure management libraries for Python to create a resource group, along with an Azure Storage account and a Blob storage container.

After provisioning these resources, refer to the section [Example: Use Azure Storage](#) to see how to use the Azure client libraries in Python to upload a file to the Blob container.

The [Equivalent Azure CLI commands](#) for bash and PowerShell are listed later in this article. If you prefer to use the Azure portal, see [Create an Azure storage account](#) and [Create a blob container](#).

1: Set up your local development environment

If you haven't already, set up an environment where you can run the code. Here are some options:

- Configure a Python virtual environment using `venv` or your tool of choice. To start using the virtual environment, be sure to activate it. To install python, see [Install Python](#).

Bash

Azure CLI

```
#!/bin/bash
# Create a virtual environment
python -m venv .venv
# Activate the virtual environment
source .venv/Scripts/activate # only required for Windows (Git Bash)
```

- Use a [conda environment](#). To install Conda, see [Install Miniconda](#).
- Use a [Dev Container](#) in [Visual Studio Code](#) or [GitHub Codespaces](#).

2: Install the needed Azure library packages

- In your console, create a `requirements.txt` file that lists the management libraries used in this example:

Azure CLI

```
azure-mgmt-resource  
azure-mgmt-storage  
azure-identity
```

2. In your console with the virtual environment activated, install the requirements:

Console

```
pip install -r requirements.txt
```

3. Set environment variables

In this step, you set environment variables for use in the code in this article. The code uses the `os.environ` method to retrieve the values.

Bash

Azure CLI

```
#!/bin/bash  
export AZURE_RESOURCE_GROUP_NAME=<ResourceGroupName> # Change to your  
preferred resource group name  
export LOCATION=<Location> # Change to your preferred region  
export AZURE_SUBSCRIPTION_ID=$(az account show --query id --output tsv)  
export STORAGE_ACCOUNT_NAME=<StorageAccountName> # Change to your preferred  
storage account name  
export CONTAINER_NAME=<ContainerName> # Change to your preferred container  
name
```

4: Write code to create a storage account and blob container

In this step, you create a Python file named `provision_blob.py` with the following code. This Python script uses the Azure SDK for Python management libraries to create a resource group, Azure Storage account, and Blob container using the Azure SDK for Python.

Python

```
import os, random

# Import the needed management objects from the libraries. The azure.common
library
# is installed automatically with the other libraries.
from azure.identity import DefaultAzureCredential
from azure.mgmt.resource import ResourceManagementClient
from azure.mgmt.storage import StorageManagementClient
from azure.mgmt.storage.models import BlobContainer

# Acquire a credential object.
credential = DefaultAzureCredential()

# Retrieve subscription ID from environment variable.
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]

# Retrieve resource group name and location from environment variables
RESOURCE_GROUP_NAME = os.environ["AZURE_RESOURCE_GROUP_NAME"]
LOCATION = os.environ["LOCATION"]

# Step 1: Provision the resource group.
resource_client = ResourceManagementClient(credential, subscription_id)

rg_result = resource_client.resource_groups.create_or_update(RESOURCE_GROUP_NAME,
    { "location": LOCATION })

print(f"Provisioned resource group {rg_result.name}")

# For details on the previous code, see Example: Provision a resource group
# at https://docs.microsoft.com/azure/developer/python/azure-sdk-example-resource-
group

# Step 2: Provision the storage account, starting with a management object.

storage_client = StorageManagementClient(credential, subscription_id)

STORAGE_ACCOUNT_NAME = os.environ["STORAGE_ACCOUNT_NAME"]

# Check if the account name is available. Storage account names must be unique
across
# Azure because they're used in URLs.
availability_result = storage_client.storage_accounts.check_name_availability(
    { "name": STORAGE_ACCOUNT_NAME }
)

if not availability_result.name_available:
    print(f"Storage name {STORAGE_ACCOUNT_NAME} is already in use. Try another
name.")
    exit()

# The name is available, so provision the account
poller = storage_client.storage_accounts.begin_create(RESOURCE_GROUP_NAME,
STORAGE_ACCOUNT_NAME,
```

```

    {
        "location" : LOCATION,
        "kind": "StorageV2",
        "sku": {"name": "Standard_LRS"}
    }
}

# Long-running operations return a poller object; calling poller.result()
# waits for completion.
account_result = poller.result()
print(f"Provisioned storage account {account_result.name}")

# Step 3: Retrieve the account's primary access key and generate a connection
# string.
keys = storage_client.storage_accounts.list_keys(RESOURCE_GROUP_NAME,
STORAGE_ACCOUNT_NAME)

print(f"Primary key for storage account: {keys.keys[0].value}")

conn_string =
f"DefaultEndpointsProtocol=https;EndpointSuffix=core.windows.net;AccountName=
{STORAGE_ACCOUNT_NAME};AccountKey={keys.keys[0].value}"

# print(f"Connection string: {conn_string}")

# Step 4: Provision the blob container in the account (this call is synchronous)
CONTAINER_NAME = os.environ["CONTAINER_NAME"]
container = storage_client.blob_containers.create(RESOURCE_GROUP_NAME,
STORAGE_ACCOUNT_NAME, CONTAINER_NAME, BlobContainer())

print(f"Provisioned blob container {container.name}")

```

Authentication in the code

Later in this article, you sign in to Azure using the Azure CLI to execute the sample code. If your account has sufficient permissions to create resource groups and storage resources in your Azure subscription, the script should run successfully without additional configuration.

To use this code in a production environment, authenticate using a service principal by setting environment variables. This approach enables secure, automated access without relying on interactive login. For detailed guidance, see [How to authenticate Python apps with Azure services](#).

Ensure that the service principal is assigned a role with sufficient permissions to create resource groups and storage accounts. For example, assigning the Contributor role at the subscription level provides the necessary access. To learn more about role assignments, see [Role-based access control \(RBAC\) in Azure](#).

Reference links for classes used in the code

- [DefaultAzureCredential \(azure.identity\)](#)
- [ResourceManagementClient \(azure.mgmt.resource\)](#)
- [StorageManagementClient \(azure.mgmt.storage\)](#)

5. Run the script

1. If you haven't already, sign in to Azure using the Azure CLI:

```
Azure CLI
```

```
az login
```

2. Run the script:

```
Console
```

```
python provision_blob.py
```

The script takes a minute or two to complete.

6: Verify the resources

1. Open the [Azure portal](#) to verify that the resource group and storage account were created as expected. You may need to wait a minute and also select **Show hidden types** in the resource group.

The screenshot shows the Azure portal's Resource Groups page. A resource group named 'PythonAzureExample-Storage-rg' is selected. On the left, a navigation menu includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Events', 'Settings' (with 'Deployments', 'Security', 'Policies', 'Properties', and 'Locks' options), and 'Data storage' (with 'Containers', 'File shares', 'Queues', and 'Tables' options). The main pane displays the 'Essentials' section for the selected resource group, showing the subscription is Primary, the location is Central US, and there are no deployments. Below this is a table listing storage accounts, with one entry for 'pythonazurestorage99737'. A filter bar at the top of the table allows filtering by Type (set to 'all') and Location (set to 'all'). A red box highlights the 'Show hidden types' checkbox, which is checked.

2. Select the storage account, then select **Data storage** > **Containers** in the left-hand menu to verify that the "blob-container-01" appears:

The screenshot shows the Azure Storage account 'pythonazurestorage99737'. The left sidebar has 'Data storage' selected, with 'Containers' highlighted by a red box. The main area shows a table with a single row for 'blob-container-01', which is also highlighted by a red box. The table has columns for 'Name' and 'Type'. The 'Name' column contains the value 'blob-container-01'. The 'Type' column is empty. The top right of the screen shows standard Azure navigation and search controls.

3. If you want to try using these resources from application code, continue with [Example: Use Azure Storage](#).

For another example of using the Azure Storage management library, see the [Manage Python Storage sample](#).

7: Clean up resources

Leave the resources in place if you want to follow the article [Example: Use Azure Storage](#) to use these resources in app code. Otherwise, run the `az group delete` command if you don't need to keep the resource group and storage resources created in this example.

Resource groups don't incur any ongoing charges in your subscription, but resources, like storage accounts, in the resource group might incur charges. It's a good practice to clean up any group that you aren't actively using. The `--no-wait` argument allows the command to return immediately instead of waiting for the operation to finish.

Bash

Azure CLI

```
#!/bin/bash
az group delete -n $AZURE_RESOURCE_GROUP_NAME --no-wait
```

For reference: equivalent Azure CLI commands

The following Azure CLI commands complete the same creation steps as the Python script:

Bash

Azure CLI

```
#!/bin/bash
#!/bin/bash

# Set variables
export LOCATION=<Location> # Change to your preferred region
export AZURE_RESOURCE_GROUP_NAME=<ResourceGroupName> # Change to your
preferred resource group name
export STORAGE_ACCOUNT_NAME=<StorageAccountName> # Change to your preferred
storage account name
export CONTAINER_NAME=<ContainerName> # Change to your preferred container
name

# Provision the resource group
echo "Creating resource group: $AZURE_RESOURCE_GROUP_NAME"
az group create \
    --location "$LOCATION" \
    --name "$AZURE_RESOURCE_GROUP_NAME"

# Provision the storage account
az storage account create -g $AZURE_RESOURCE_GROUP_NAME -l $LOCATION -n
$STORAGE_ACCOUNT_NAME --kind StorageV2 --sku Standard_LRS

echo Storage account name is $STORAGE_ACCOUNT_NAME

# Retrieve the connection string
CONNECTION_STRING=$(az storage account show-connection-string -g
$AZURE_RESOURCE_GROUP_NAME -n $STORAGE_ACCOUNT_NAME --query connectionString)
```

```
# Provision the blob container
az storage container create --name $CONTAINER_NAME --account-name
$STORAGE_ACCOUNT_NAME --connection-string $CONNECTION_STRING
```

See also

- [Example: Use Azure Storage](#)
- [Example: Create a resource group](#)
- [Example: List resource groups in a subscription](#)
- [Example: Create a web app and deploy code](#)
- [Example: Create and query a database](#)
- [Example: Create a virtual machine](#)
- [Use Azure Managed Disks with virtual machines](#)
- [Complete a short survey about the Azure SDK for Python ↗](#)

Example: Access Azure Storage using the Azure libraries for Python

Article • 10/03/2024

In this article, you learn how to use the Azure client libraries in Python application code to upload a file to an Azure Blob storage container. The article assumes you've created the resources shown in [Example: Create Azure Storage](#).

All the commands in this article work the same in Linux/macOS bash and Windows command shells unless noted.

1. Set up your local development environment

If you haven't already, set up an environment where you can run this code. Here are some options:

- Configure a Python virtual environment using `venv` or your tool of choice. You can create the virtual environment locally or in [Azure Cloud Shell](#) and run the code there. Be sure to activate the virtual environment to start using it.
- Use a [conda environment](#).
- Use a [Dev Container](#) in [Visual Studio Code](#) or [GitHub Codespaces](#).

2. Install library packages

In your `requirements.txt` file, add lines for the client library package you need and save the file.

```
txt
azure-storage-blob
azure-identity
```

Then, in your terminal or command prompt, install the requirements.

```
Console
pip install -r requirements.txt
```

3. Create a file to upload

Create a source file named `sample-source.txt`. This file name is what the code expects.

```
txt  
  
Hello there, Azure Storage. I'm a friendly file ready to be stored in a blob.
```

4. Use blob storage from app code

This section demonstrates two ways to access data in the blob container that you created in [Example: Create Azure Storage](#). To access data in the blob container, your app must be able to authenticate with Azure and be authorized to access data in the container. This section presents two ways of doing this:

- The **Passwordless (Recommended)** method authenticates the app by using `DefaultAzureCredential`. `DefaultAzureCredential` is a chained credential that can authenticate an app (or a user) using a sequence of different credentials, including developer tool credentials, application service principals, and managed identities.
- The **Connection string** method uses a connection string to access the storage account directly.

For the following reasons and more, we recommend using the passwordless method whenever possible:

- A connection string authenticates the connecting agent with the Storage *account* rather than with individual resources within that account. As a result, a connection string grants broader authorization than might be needed. With `DefaultAzureCredential` you can grant more granular, least privileged permissions over your storage resources to the identity your app runs under using [Azure RBAC](#).
- A connection string contains access info in plain text and therefore presents potential vulnerabilities if it's not properly constructed or secured. If such a connection string is exposed, it can be used to access a wide range of resources within the Storage account.
- A connection string is usually stored in an environment variable, which makes it vulnerable to compromise if an attacker gains access to your environment. Many of the credential types supported by `DefaultAzureCredential` don't require storing secrets in your environment.

Passwordless (Recommended)

`DefaultAzureCredential` is an opinionated, preconfigured chain of credentials. It's designed to support many environments, along with the most common authentication flows and developer tools. An instance of `DefaultAzureCredential` determines which credential types to try to get a token for based on a combination of its runtime environment, the value of certain well-known environment variables, and, optionally, parameters passed into its constructor.

In the following steps, you configure an application service principal as the application identity. Application service principals are suitable for use both during local development and for apps hosted on-premises. To configure `DefaultAzureCredential` to use the application service principal, you set the following environment variables: `AZURE_CLIENT_ID`, `AZURE_TENANT_ID`, and `AZURE_CLIENT_SECRET`.

Notice that a client secret is configured. This is necessary for an application service principal, but, depending on your scenario, you can also configure `DefaultAzureCredential` to use credentials that don't require setting a secret or password in an environment variable.

For example, in local development, if `DefaultAzureCredential` can't get a token using configured environment variables, it tries to get one using the user (already signed into development tools like Azure CLI); for an app hosted in Azure, `DefaultAzureCredential` can be configured to use a managed identity. In all cases, the code in your app remains the same, only the configuration and/or the runtime environment changes.

1. Create a file named `use_blob_auth.py` with the following code. The comments explain the steps.

Python

```
import os
import uuid

from azure.identity import DefaultAzureCredential

# Import the client object from the SDK library
from azure.storage.blob import BlobClient

credential = DefaultAzureCredential()

# Retrieve the storage blob service URL, which is of the form
# https://<your-storage-account-name>.blob.core.windows.net/
```

```
storage_url = os.environ["AZURE_STORAGE_BLOB_URL"]

# Create the client object using the storage URL and the credential
blob_client = BlobClient(
    storage_url,
    container_name="blob-container-01",
    blob_name=f"sample-blob-{str(uuid.uuid4())[0:5]}.txt",
    credential=credential,
)

# Open a local file and upload its contents to Blob Storage
with open("./sample-source.txt", "rb") as data:
    blob_client.upload_blob(data)
print(f"Uploaded sample-source.txt to {blob_client.url}")
```

Reference links:

- [DefaultAzureCredential \(azure.identity\)](#)
- [BlobClient \(azure.storage.blob\)](#)

2. Create an environment variable named `AZURE_STORAGE_BLOB_URL`:

```
cmd
Windows Command Prompt
set
AZURE_STORAGE_BLOB_URL=https://pythonazurestorage12345.blob.core.windows.net
```

Replace "pythonazurestorage12345" with the name of your storage account.

The `AZURE_STORAGE_BLOB_URL` environment variable is used only by this example. It isn't used by the Azure libraries.

3. Use the `az ad sp create-for-rbac` command to create a new service principal for the app. The command creates the app registration for the app at the same time. Give the service principal a name of your choosing.

```
Azure CLI
az ad sp create-for-rbac --name <service-principal-name>
```

The output of this command will look like the following. Make note of these values or keep this window open as you'll need these values in the next step and won't be able to view the password (client secret) value again. You can,

however, add a new password later without invalidating the service principal or existing passwords if needed.

JSON

```
{  
    "appId": "00001111-aaaa-2222-bbbb-3333cccc4444",  
    "displayName": "<service-principal-name>",  
    "password": "Aa1Bb~2Cc3.-Dd4Ee5Ff6Gg7Hh8Ii9_Jj0Kk1Ll2",  
    "tenant": "aaaabbbb-0000-cccc-1111-dddd2222eeee"  
}
```

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI installed](#).

4. Create environment variables for the application service principal:

Create the following environment variables with the values from the output of the previous command. These variables tell `DefaultAzureCredential` to use the application service principal.

- `AZURE_CLIENT_ID` → The app ID value.
- `AZURE_TENANT_ID` → The tenant ID value.
- `AZURE_CLIENT_SECRET` → The password/credential generated for the app.

cmd

Windows Command Prompt

```
set AZURE_CLIENT_ID=00001111-aaaa-2222-bbbb-3333cccc4444  
set AZURE_TENANT_ID=aaaabbbb-0000-cccc-1111-dddd2222eeee  
set AZURE_CLIENT_SECRET=Aa1Bb~2Cc3.-  
Dd4Ee5Ff6Gg7Hh8Ii9_Jj0Kk1Ll2
```

5. Attempt to run the code (which fails intentionally):

Console

```
python use_blob_auth.py
```

6. Observe the error "This request is not authorized to perform this operation using this permission." The error is expected because the local service principal that you're using doesn't yet have permission to access the blob container.

7. Grant [Storage Blob Data Contributor](#) permissions on the blob container to the service principal using the `az role assignment create` Azure CLI command:

```
Azure CLI
```

```
az role assignment create --assignee <AZURE_CLIENT_ID> \
    --role "Storage Blob Data Contributor" \
    --scope
"/subscriptions/<AZURE_SUBSCRIPTION_ID>/resourceGroups/PythonAzureExample-Storage-rg/providers/Microsoft.Storage/storageAccounts/pythonazurestorage12345/blobServices/default/containers/blob-container-01"
```

The `--assignee` argument identifies the service principal. Replace `<AZURE_CLIENT_ID>` placeholder with the app ID of your service principal.

The `--scope` argument identifies where this role assignment applies. In this example, you grant the "Storage Blob Data Contributor" role to the service principal for the container named "blob-container-01".

- Replace `PythonAzureExample-Storage-rg` and `pythonazurestorage12345` with the resource group that contains your storage account and the exact name of your storage account. Also, adjust the name of the blob container, if necessary. If you use the wrong name, you see the error, "Cannot perform requested operation on nested resource. Parent resource 'pythonazurestorage12345' not found."
- Replace the `<AZURE_SUBSCRIPTION_ID>` place holder with your Azure subscription ID. (You can run the `az account show` command and get your subscription ID from the `id` property in the output.)

💡 Tip

If the role assignment command returns an error "No connection adapters were found" when using bash shell, try setting `export MSYS_NO_PATHCONV=1` to avoid path translation. For more information, see this [issue](#).

8. Wait a minute or two for the permissions to propagate, then run the code again to verify that it now works. If you see the permissions error again, wait a little longer, then try the code again.

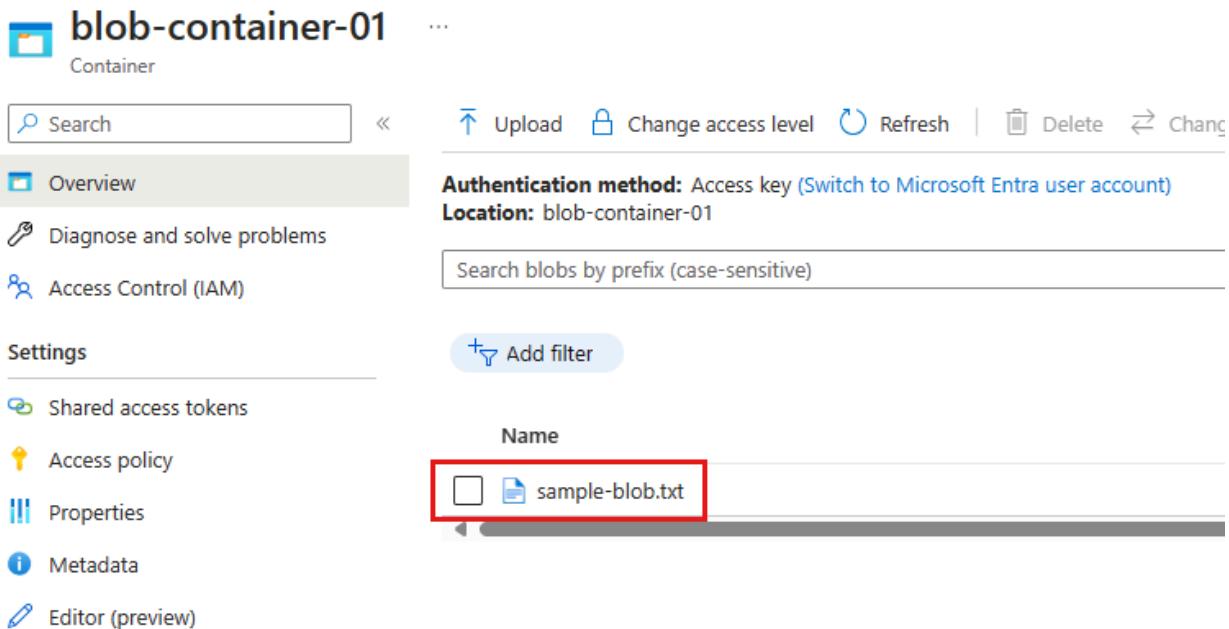
For more information on role assignments, see [How to assign role permissions using the Azure CLI](#).

ⓘ Important

In the preceding steps, your app ran under an application service principal. An application service principal requires a client secret in its configuration. However, you can use the same code to run the app under different credential types that don't require you to explicitly configure a password or secret in the environment. For example, during development, `DefaultAzureCredential` can use developer tool credentials like the credentials you use to sign in via the Azure CLI; or, for apps hosted in Azure, it can use a [managed identity](#). To learn more, see [Authenticate Python apps to Azure services by using the Azure SDK for Python](#).

5. Verify blob creation

After running the code of either method, go to the [Azure portal](#), navigate into the blob container to verify that a new blob exists named `sample-blob-{random}.txt` with the same contents as the `sample-source.txt` file:



The screenshot shows the Azure Storage Blob Container Overview page for 'blob-container-01'. The left sidebar includes options like Overview, Diagnose and solve problems, Access Control (IAM), Settings, Shared access tokens, Access policy, Properties, Metadata, and Editor (preview). The main area displays a table with one row for a blob named 'sample-blob.txt'. A red box highlights this row. The table columns are 'Name' and 'Type'. The 'Name' column contains 'sample-blob.txt' with a checkbox icon to its left. The 'Type' column contains a blue document icon.

If you created an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, you can also use the Azure CLI to verify that the blob exists using the `az storage blob list` command:

```
az storage blob list --container-name blob-container-01
```

If you followed the instructions to use passwordless authentication, you can add the `--connection-string` parameter to the preceding command with the connection string for your storage account. To get the connection string, use the [az storage account show-connection-string](#) command.

Azure CLI

```
az storage account show-connection-string --resource-group PythonAzureExample-Storage-rg --name pythonazurestorage12345 --output tsv
```

Use the entire connection string as the value for the `--connection-string` parameter.

ⓘ Note

If your Azure user account has the "Storage Blob Data Contributor" role on the container, you can use the following command to list the blobs in the container:

Azure CLI

```
az storage blob list --container-name blob-container-01 --account-name pythonazurestorage12345 --auth-mode login
```

6. Clean up resources

Run the [az group delete](#) command if you don't need to keep the resource group and storage resources used in this example. Resource groups don't incur any ongoing charges in your subscription, but resources, like storage accounts, in the resource group might continue to incur charges. It's a good practice to clean up any group that you aren't actively using. The `--no-wait` argument allows the command to return immediately instead of waiting for the operation to finish.

Azure CLI

```
az group delete -n PythonAzureExample-Storage-rg --no-wait
```

You can also use the [ResourceManagementClient.resource_groups.begin_delete](#) method to delete a resource group from code. The code in [Example: Create a resource group](#) demonstrates usage.

If you followed the instructions to use passwordless authentication, it's a good idea to delete the application service principal you created. You can use the [az ad app delete](#) command. Replace the <AZURE_CLIENT_ID> placeholder with the app ID of your service principal.

Azure CLI

```
az ad app delete --id <AZURE_CLIENT_ID>
```

See also

- [Quickstart: Azure Blob Storage client library for Python](#)
- [Example: Create a resource group](#)
- [Example: List resource groups in a subscription](#)
- [Example: Create a web app and deploy code](#)
- [Example: Create Azure Storage](#)
- [Example: Create and query a database](#)
- [Example: Create a virtual machine](#)
- [Use Azure Managed Disks with virtual machines](#)
- [Complete a short survey about the Azure SDK for Python ↗](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Example: Use the Azure libraries to create and deploy a web app

Article • 04/21/2025

This example shows how to use the Azure SDK management libraries in a Python script to create and deploy a web app to Azure App Service, with the app code pulled from a GitHub repository.

The Azure SDK for Python includes management libraries (namespaces beginning with `azure-mgmt`) that let you automate resource configuration and deployment — similar to what you can do with the Azure portal, Azure CLI, or ARM templates. For examples, see [Quickstart: Deploy a Python \(Django or Flask\) web app to Azure App Service](#).

1: Set up your local development environment

If you haven't already, set up an environment where you can run this code. Here are some options:

- Configure a Python virtual environment using `venv` or your tool of choice. You can create the virtual environment locally or in [Azure Cloud Shell](#) and run the code there. Be sure to activate the virtual environment to start using it. To install python, see [Install Python](#).

Bash

```
python -m venv .venv
source .venv/bin/activate # Linux or macOS
.venv\Scripts\activate # Windows
```

- Use a [conda environment](#). To install Conda, see [Install Miniconda](#).
- Use a [Dev Container](#) in [Visual Studio Code](#) or [GitHub Codespaces](#).

2: Install the required Azure library packages

Create a file named `requirements.txt` with the following contents:

txt

```
azure-mgmt-resource
azure-mgmt-web
azure-identity
```

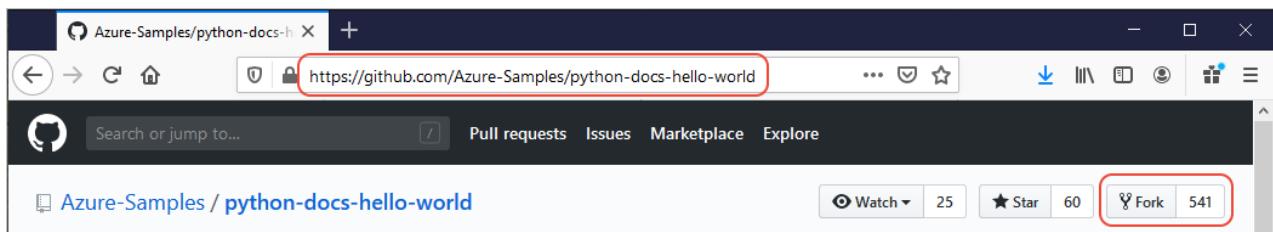
In your local development environment, install the requirements using the following code:

```
Console
```

```
pip install -r requirements.txt
```

3: Fork the sample repository

1. Visit <https://github.com/Azure-Samples/python-docs-hello-world> and fork the repository into your own GitHub account. Using a fork ensures that you have the necessary permissions to deploy the app to Azure.



2. Next, create an environment variable named `REPO_URL` and set it to the URL of your forked repository. This variable is required by the example code in the next section.

```
bash
```

```
Bash
```

```
export REPO_URL=<url_of_your_fork>
export AZURE_SUBSCRIPTION_ID=<subscription_id>
```

A screenshot of a terminal window. The title bar says 'bash'. The main pane is titled 'Bash'. Inside the bash shell, two commands are entered: 'export REPO_URL=<url_of_your_fork>' and 'export AZURE_SUBSCRIPTION_ID=<subscription_id>'. Both commands are preceded by a blue 'export' keyword.

4: Write code to create and deploy a web app

Create a Python file named `provision_deploy_web_app.py` and add the following code. The in-line comments explain what each part of the script does. The `REPO_URL` and `AZURE_SUBSCRIPTION_ID` environment variables should already be set in the previous step.

```
Python
```

```
import random, os
from azure.identity import AzureCliCredential
from azure.mgmt.resource import ResourceManagementClient
from azure.mgmt.web import WebSiteManagementClient

# Acquire a credential object using CLI-based authentication.
```

A screenshot of a code editor window. The title bar says 'Python'. The main pane contains Python code. The code imports random, os, AzureCliCredential, ResourceManagementClient, and WebSiteManagementClient from the azure module. It then defines a single comment line starting with '# Acquire a credential object using CLI-based authentication.'

```
credential = AzureCliCredential()

# Retrieve subscription ID from environment variable
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]

# Constants we need in multiple places: the resource group name and the region
# in which we provision resources. You can change these values however you want.
RESOURCE_GROUP_NAME = 'PythonAzureExample-WebApp-rg'
LOCATION = "centralus"

# Step 1: Provision the resource group.
resource_client = ResourceManagementClient(credential, subscription_id)

rg_result = resource_client.resource_groups.create_or_update(RESOURCE_GROUP_NAME,
    { "location": LOCATION })

print(f"Provisioned resource group {rg_result.name}")

# For details on the previous code, see Example: Provision a resource group
# at https://docs.microsoft.com/azure/developer/python/azure-sdk-example-resource-group

#Step 2: Provision the App Service plan, which defines the underlying VM for the
# web app.

# Names for the App Service plan and App Service. We use a random number with the
# latter to create a reasonably unique name. If you've already provisioned a
# web app and need to re-run the script, set the WEB_APP_NAME environment
# variable to that name instead.
SERVICE_PLAN_NAME = 'PythonAzureExample-WebApp-plan'
WEB_APP_NAME = os.environ.get("WEB_APP_NAME", f"PythonAzureExample-WebApp-{random.randint(1,100000):05}")

# Obtain the client object
app_service_client = WebSiteManagementClient(credential, subscription_id)

# Provision the plan; Linux is the default
poller =
app_service_client.app_service_plans.begin_create_or_update(RESOURCE_GROUP_NAME,
    SERVICE_PLAN_NAME,
    {
        "location": LOCATION,
        "reserved": True,
        "sku" : {"name" : "B1"}
    }
)

plan_result = poller.result()

print(f"Provisioned App Service plan {plan_result.name}")

# Step 3: With the plan in place, provision the web app itself, which is the
# process that can host
```

```

# whatever code we want to deploy to it.

poller = app_service_client.web_apps.begin_create_or_update(RESOURCE_GROUP_NAME,
    WEB_APP_NAME,
    {
        "location": LOCATION,
        "server_farm_id": plan_result.id,
        "site_config": {
            "linux_fx_version": "python|3.8"
        }
    }
)

web_app_result = poller.result()

print(f"Provisioned web app {web_app_result.name} at
{web_app_result.default_host_name}")

# Step 4: deploy code from a GitHub repository. For Python code, App Service on
# Linux runs
# the code inside a container that makes certain assumptions about the structure
# of the code.
# For more information, see How to configure Python apps,
# https://docs.microsoft.com/azure/app-service/containers/how-to-configure-python.
#
# The create_or_update_source_control method doesn't provision a web app. It only
# sets the
# source control configuration for the app. In this case we're simply pointing to
# a GitHub repository.
#
# You can call this method again to change the repo.

REPO_URL = os.environ["REPO_URL"]

poller =
app_service_client.web_apps.begin_create_or_update_source_control(RESOURCE_GROUP_N
AME,
    WEB_APP_NAME,
    {
        "location": "GitHub",
        "repo_url": REPO_URL,
        "branch": "master",
        "is_manual_integration": True
    }
)

sc_result = poller.result()

print(f"Set source control on web app to {sc_result.branch} branch of
{sc_result.repo_url}")

# Step 5: Deploy the code using the repository and branch configured in the
# previous step.
#
# If you push subsequent code changes to the repo and branch, you must call this

```

```
method again
# or use another Azure tool like the Azure CLI or Azure portal to redeploy.
# Note: By default, the method returns None.

app_service_client.web_apps.sync_repository(RESOURCE_GROUP_NAME, WEB_APP_NAME)

print(f"Deploy code")
```

This code uses CLI-based authentication (using `AzureCliCredential`) because it demonstrates actions that you might otherwise do with the Azure CLI directly. In both cases, you're using the same identity for authentication. Depending on your environment, you might need to run `az login` first to authenticate.

To use such code in a production script (for example, to automate VM management), use `DefaultAzureCredential` (recommended) with a service principal based method as described in [How to authenticate Python apps with Azure services](#).

Reference links for classes used in the code

- [AzureCliCredential \(azure.identity\)](#)
- [ResourceManagementClient \(azure.mgmt.resource\)](#)
- [WebSiteManagementClient \(azure.mgmt.web import\)](#)

5: Run the script

Console

```
python provision_deploy_web_app.py
```

6: Verify the web app deployment

To view the deployed website, run the following command:

Azure CLI

```
az webapp browse --name <PythonAzureExample-WebApp-12345> --resource-group
PythonAzureExample-WebApp-rg
```

Replace the web app name (`--name`) with the value generated by the script. You don't need to change the resource group name (`--resource-group`) unless you changed it in the script. When you open the site, you should see "Hello, World!" in your browser.

💡 Tip

If you don't see the expected output, wait a few minutes and try again.

If you're still not seeing the expected output:

1. Go to the [Azure portal ↗](#).
2. Navigate to **Resource groups**, and locate the resource group you created.
3. Select the resource group to view its resources. Make sure it includes both an App Service Plan and an App Service.
4. Select the **App Service**, and then go to **Deployment Center**.
5. Open the **logs** tab to check the deployment logs for any errors or status updates.

7: Redeploy the web app code (optional)

The script provisions all the necessary resources to host your web app and configures the deployment source to use your forked repository using manual integration. With manual integration, you need to manually trigger the web app to pull updates from the specified repository and branch.

The script uses the `WebSiteManagementClient.web_apps.sync_repository` method to trigger the web app to pull code from your repository. If you make further changes to your code, you can redeploy by calling this API again, or by using other Azure tools such as the Azure CLI or the Azure portal.

You can redeploy your code using the Azure CLI by running the [az webapp deployment source sync](#) command:

Azure CLI

```
az webapp deployment source sync --name <PythonAzureExample-WebApp-12345> --resource-group PythonAzureExample-WebApp-rg
```

You don't need to change the resource group name (`--resource-group`) unless you changed it in the script.

To deploy your code from Azure portal:

1. Go to the [Azure portal ↗](#).
2. Navigate to **Resource groups**, and locate the resource group you created.
3. Select the resource group name to view its resources. Make sure it includes both an App Service Plan and an App Service.

4. Select the App Service, and then go to Deployment Center.
5. On the top menu, select Sync to trigger the deployment of your code.

8: Clean up resources

Azure CLI

```
az group delete --name PythonAzureExample-WebApp-rg --no-wait
```

You do not need to change resource group name (`--resource-group` option) unless you changed it in the script.

If you no longer need the resource group created in this example, you can delete it by running the `az group delete` command. While resource groups don't incur ongoing charges, it's a good practice to clean up any unused resources. Use the `--no-wait` argument to immediately return control to the command line without waiting for the deletion to complete.

You can also delete a resource group programmatically using the [ResourceManagementClient.resource_groups.begin_delete](#) method.

See also

- [Example: Create a resource group](#)
- [Example: List resource groups in a subscription](#)
- [Example: Create Azure Storage](#)
- [Example: Use Azure Storage](#)
- [Example: Create and query a MySQL database](#)
- [Example: Create a virtual machine](#)
- [Use Azure Managed Disks with virtual machines](#)
- [Complete a short survey about the Azure SDK for Python ↗](#)

Example: Use the Azure libraries to create a database

05/30/2025

This example demonstrates how to use the Azure SDK for Python management libraries to programmatically create an Azure Database for MySQL flexible server and a corresponding database. It also includes a basic script that uses the mysql-connector-python library (not part of the Azure SDK) to connect to and query the database.

You can adapt this example to create an Azure Database for PostgreSQL flexible server by modifying the relevant SDK imports and API calls.

If you prefer to use the Azure CLI, [Equivalent Azure CLI commands](#) are provided later in this article. For a graphical experience, refer to the Azure portal documentation:

- [Create a MySQL server](#)
- [Create a PostgreSQL server](#)

Unless otherwise specified, all examples and commands work consistently across Linux/macOS bash and Windows command shells.

1: Set up your local development environment

If you haven't already, set up an environment where you can run the code. Here are some options:

- Configure a Python virtual environment using `venv` or your tool of choice. To start using the virtual environment, be sure to activate it. To install python, see [Install Python](#).



The screenshot shows a terminal window with two tabs: "Bash" and "Azure CLI". The "Azure CLI" tab is active and contains the following command-line session:

```
#!/bin/bash
# Create a virtual environment
python -m venv .venv
# Activate the virtual environment
source .venv/Scripts/activate # only required for Windows (Git Bash)
```

- Use a [conda environment](#). To install Conda, see [Install Miniconda](#).
- Use a [Dev Container](#) in [Visual Studio Code](#) or [GitHub Codespaces](#).

2: Install the needed Azure library packages

In this step, you install the Azure SDK libraries needed to create the database.

1. In your console, create a `requirements.txt` file that lists the management libraries used in this example:

```
Azure CLI  
  
azure-mgmt-resource  
azure-mgmt-rdbms  
azure-identity  
mysql-connector-python
```

ⓘ Note

The `mysql-connector-python` library isn't part of the Azure SDK. It's a third-party library that you can use to connect to MySQL databases. You can also use other libraries, such as `PyMySQL` or `SQLAlchemy`, to connect to MySQL databases.

2. In your console with the virtual environment activated, install the requirements:

```
Console  
  
pip install -r requirements.txt
```

ⓘ Note

On Windows, attempting to install the mysql library into a 32-bit Python library produces an error about the `mysql.h` file. In this case, install a 64-bit version of Python and try again.

3. Set environment variables

In this step, you set environment variables for use in the code in this article. The code uses the `os.environ` method to retrieve the values.

Bash

```
Azure CLI
```

```

#!/bin/bash
export AZURE_RESOURCE_GROUP_NAME=<ResourceGroupName> # Change to your
preferred resource group name
export LOCATION=<Location> # Change to your preferred region
export AZURE_SUBSCRIPTION_ID=$(az account show --query id --output tsv)
export PUBLIC_IP_ADDRESS=$(curl -s https://api.ipify.org)
export DB_SERVER_NAME=<DB_Server_Name> # Change to your preferred DB server
name
export DB_ADMIN_NAME=<DB_Admin_Name> # Change to your preferred admin name
export DB_ADMIN_PASSWORD=<DB_Admin_Passwrod> # Change to your preferred admin
password
export DB_NAME=<DB_Name> # Change to your preferred database name
export DB_PORT=3306
export version=ServerVersion.EIGHT0_21

```

4: Write code to create and configure a MySQL Flexible Server with a database

In this step, you create a Python file named *provision_blob.py* with the following code. This Python script uses the Azure SDK for Python management libraries to create a resource group, a MySQL flexible server, and a database on that server.

Python

```

import random, os
from azure.identity import DefaultAzureCredential
from azure.mgmt.resource import ResourceManagementClient
from azure.mgmt.rdbms.mysql_flexibleServers import MySQLManagementClient
from azure.mgmt.rdbms.mysql_flexibleServers.models import Server, ServerVersion

# Acquire a credential object using CLI-based authentication.
credential = DefaultAzureCredential()

# Retrieve subscription ID from environment variable
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]

# Retrieve resource group name and location from environment variables
RESOURCE_GROUP_NAME = os.environ["AZURE_RESOURCE_GROUP_NAME"]
LOCATION = os.environ["LOCATION"]

# Step 1: Provision the resource group.
resource_client = ResourceManagementClient(credential, subscription_id)

rg_result = resource_client.resource_groups.create_or_update(RESOURCE_GROUP_NAME,
    { "location": LOCATION })

print(f"Provisioned resource group {rg_result.name}")

```

```
# For details on the previous code, see Example: Provision a resource group
# at https://docs.microsoft.com/azure/developer/python/azure-sdk-example-resource-
group

# Step 2: Provision the database server

# Retrieve server name, admin name, and admin password from environment variables

db_server_name = os.environ.get("DB_SERVER_NAME")
db_admin_name = os.environ.get("DB_ADMIN_NAME")
db_admin_password = os.environ.get("DB_ADMIN_PASSWORD")

# Obtain the management client object
mysql_client = MySQLManagementClient(credential, subscription_id)

# Provision the server and wait for the result
server_version = os.environ.get("DB_SERVER_VERSION")

poller = mysql_client.servers.begin_create(RESOURCE_GROUP_NAME,
    db_server_name,
    Server(
        location=LOCATION,
        administrator_login=db_admin_name,
        administrator_login_password=db_admin_password,
        version=ServerVersion[server_version] # Note: dictionary-style enum
access
    )
)

server = poller.result()

print(f"Provisioned MySQL server {server.name}")

# Step 3: Provision a firewall rule to allow the local workstation to connect

RULE_NAME = "allow_ip"
ip_address = os.environ["PUBLIC_IP_ADDRESS"]

# Provision the rule and wait for completion
poller = mysql_client.firewall_rules.begin_create_or_update(RESOURCE_GROUP_NAME,
    db_server_name, RULE_NAME,
    { "start_ip_address": ip_address, "end_ip_address": ip_address }
)

firewall_rule = poller.result()

print(f"Provisioned firewall rule {firewall_rule.name}")

# Step 4: Provision a database on the server

db_name = os.environ.get("DB_NAME", "example-db1")
```

```
poller = mysql_client.databases.begin_create_or_update(RESOURCE_GROUP_NAME,
    db_server_name, db_name, {})

db_result = poller.result()

print(f"Provisioned MySQL database {db_result.name} with ID {db_result.id}")
```

Authentication in the code

Later in this article, you sign in to Azure using the Azure CLI to execute the sample code. If your account has sufficient permissions to create resource groups and storage resources in your Azure subscription, the script should run successfully without additional configuration.

For use in production environments, we recommend that you authenticate with a service principal by setting the appropriate environment variables. This approach enables secure, non-interactive access suitable for automation. For setup instructions, see [How to authenticate Python apps with Azure services](#).

Ensure the service principal is assigned a role with adequate permissions—such as the Contributor role at the subscription or resource group level. For details on assigning roles, refer to [Role-based access control \(RBAC\) in Azure](#).

Reference links for classes used in the code

- [ResourceManagementClient \(azure.mgmt.resource\)](#)
- [MySQLManagementClient \(azure.mgmt.rdbms.mysql_flexibleServers\)](#)
- [Server \(azure.mgmt.rdbms.mysql_flexibleServers.models\)](#)
- [ServerVersion \(azure.mgmt.rdbms.mysql_flexibleServers.models\)](#)

For PostgreSQL database server, see:

- [PostgreSQLManagementClient \(azure.mgmt.rdbms.postgresql_flexibleServers\)](#)

5: Run the script

1. If you haven't already, sign in to Azure using the Azure CLI:

```
Azure CLI
```

```
az login
```

2. Run the script:

Console

```
python provision_db.py
```

The script takes a minute or two to complete.

6: Insert a record and query the database

In this step, you create a table in the database and insert a record. You can use the mysql-connector library to connect to the database and run SQL commands.

1. Create a file named *use_db.py* with the following code.

This code works only for MySQL; you use different libraries for PostgreSQL.

Python

```
import os
import mysql.connector

db_server_name = os.environ["DB_SERVER_NAME"]
db_admin_name = os.getenv("DB_ADMIN_NAME")
db_admin_password = os.getenv("DB_ADMIN_PASSWORD")

db_name = os.getenv("DB_NAME")
db_port = os.getenv("DB_PORT")

connection = mysql.connector.connect(user=db_admin_name,
                                      password=db_admin_password, host=f"{db_server_name}.mysql.database.azure.com",
                                      port=db_port, database=db_name,
                                      ssl_ca='./BaltimoreCyberTrustRoot.crt.pem')

cursor = connection.cursor()

"""

# Alternate pyodbc connection; include pyodbc in requirements.txt
import pyodbc

driver = "{MySQL ODBC 5.3 UNICODE Driver}"

connect_string = f"DRIVER={driver};PORT=3306;SERVER={db_server_name}.mysql.database.azure.com; \
                  f"DATABASE={DB_NAME};UID={db_admin_name};PWD={db_admin_password}"

connection = pyodbc.connect(connect_string)
"""

table_name = "ExampleTable1"
```

```

sql_create = f"CREATE TABLE {table_name} (name varchar(255), code int)"

cursor.execute(sql_create)
print(f"Successfully created table {table_name}")

sql_insert = f"INSERT INTO {table_name} (name, code) VALUES ('Azure', 1)"
insert_data = "('Azure', 1)"

cursor.execute(sql_insert)
print("Successfully inserted data into table")

sql_select_values= f"SELECT * FROM {table_name}"

cursor.execute(sql_select_values)
row = cursor.fetchone()

while row:
    print(str(row[0]) + " " + str(row[1]))
    row = cursor.fetchone()

connection.commit()

```

All of this code uses the mysql.connector API. The only Azure-specific part is the full host domain for MySQL server (mysql.database.azure.com).

2. Next, download the certificate needed to communicate over TSL/SSL with your Azure Database for MySQL server. For more information, see [Obtain an SSL Certificate](#) in the Azure Database for MySQL documentation.

Bash

Azure CLI

```

#!/bin/bash
# Download Baltimore CyberTrust Root certificate required for Azure MySQL
SSL connections
CERT_URL="https://www.digicert.com/CACerts/BaltimoreCyberTrustRoot.crt.pem"
CERT_FILE="BaltimoreCyberTrustRoot.crt.pem"
echo "Downloading SSL certificate..."
curl -o "$CERT_FILE" "$CERT_URL"

```

3. Finally, run the code:

Console

```
python use_db.py
```

If you see an error that your client IP address isn't allowed, check that you defined the environment variable `PUBLIC_IP_ADDRESS` correctly. If you already created the MySQL server with the wrong IP address, you can add another in the [Azure portal](#). In the portal, select the MySQL server, and then select **Connection security**. Add the IP address of your workstation to the list of allowed IP addresses.

7: Clean up resources

Run the `az group delete` command if you don't need to keep the resource group and storage resources created in this example.

Resource groups don't incur any ongoing charges in your subscription, but resources, like storage accounts, in the resource group might continue to incur charges. It's a good practice to clean up any group that you aren't actively using. The `--no-wait` argument allows the command to return immediately instead of waiting for the operation to finish.

Bash

Azure CLI

```
#!/bin/bash
az group delete -n $AZURE_RESOURCE_GROUP_NAME --no-wait
```

You can also use the `ResourceManagementClient.resource_groups.begin_delete` method to delete a resource group from code. The code in [Example: Create a resource group](#) demonstrates usage.

For reference: equivalent Azure CLI commands

The following Azure CLI commands complete the same provisioning steps as the Python script. For a PostgreSQL database, use `az postgres flexible-server` commands.

Bash

Azure CLI

```
#!/bin/bash
#!/bin/bash

# Set variables
export LOCATION=<Location> # Change to your preferred region
export AZURE_RESOURCE_GROUP_NAME=<ResourceGroupName> # Change to your
```

```

preferred resource group name
export DB_SERVER_NAME=<DB_Server_Name> # Change to your preferred DB server
name
export DB_ADMIN_NAME=<DB_Admin_Name> # Change to your preferred admin name
export DB_ADMIN_PASSWORD=<DB_Admin_Password> # Change to your preferred admin
password
export DB_NAME=<DB_Name> # Change to your preferred database name
export DB_SERVER_VERSION="5.7"

# Get public IP address
export PUBLIC_IP_ADDRESS=$(curl -s https://api.ipify.org)

# Provision the resource group
echo "Creating resource group: $AZURE_RESOURCE_GROUP_NAME"
az group create \
    --location "$LOCATION" \
    --name "$AZURE_RESOURCE_GROUP_NAME"

# Provision the MySQL Flexible Server
echo "Creating MySQL Flexible Server: $DB_SERVER_NAME"
az mysql flexible-server create \
    --location "$LOCATION" \
    --resource-group "$AZURE_RESOURCE_GROUP_NAME" \
    --name "$DB_SERVER_NAME" \
    --admin-user "$DB_ADMIN_NAME" \
    --admin-password "$DB_ADMIN_PASSWORD" \
    --sku-name Standard_B1ms \
    --version "$DB_SERVER_VERSION" \
    --yes

# Provision a firewall rule to allow access from the public IP address
echo "Creating firewall rule for public IP: $PUBLIC_IP_ADDRESS"
az mysql flexible-server firewall-rule create \
    --resource-group "$AZURE_RESOURCE_GROUP_NAME" \
    --name "$DB_SERVER_NAME" \
    --rule-name allow_ip \
    --start-ip-address "$PUBLIC_IP_ADDRESS" \
    --end-ip-address "$PUBLIC_IP_ADDRESS"

# Provision the database
echo "Creating database: $DB_NAME"
az mysql flexible-server db create \
    --resource-group "$AZURE_RESOURCE_GROUP_NAME" \
    --server-name "$DB_SERVER_NAME" \
    --database-name "$DB_NAME"

echo "MySQL Flexible Server and database created successfully."

```

See also

- Example: Create a resource group
- Example: List resource groups in a subscription
- Example: Create Azure Storage
- Example: Use Azure Storage
- Example: Create and deploy a web app
- Example: Create a virtual machine
- Use Azure Managed Disks with virtual machines
- Complete a short survey about the Azure SDK for Python ↗

Example: Use the Azure libraries to create a virtual machine

06/11/2025

In this article, you learn how to use the Azure SDK management libraries in a Python script to create a resource group that contains a Linux virtual machine.

The [Equivalent Azure CLI commands](#) are listed later in this article. If you prefer to use the Azure portal, see [Create a Linux VM](#) and [Create a Windows VM](#).

(!) Note

Creating a virtual machine through code is a multi-step process that involves provisioning a number of other resources that the virtual machine requires. If you're simply running such code from the command line, it's much easier to use the [az vm create](#) command, which automatically provisions these secondary resources with defaults for any setting you choose to omit. The only required arguments are a resource group, VM name, image name, and login credentials. For more information, see [Quick Create a virtual machine with the Azure CLI](#).

1: Set up your local development environment

If you haven't already, set up an environment where you can run this code. Here are some options:

Azure CLI

```
#!/bin/bash
# Create a virtual environment
python -m venv .venv
# Activate the virtual environment
source .venv/Scripts/activate # only required for Windows (Git Bash)
```

- Use a [conda environment](#). To install Conda, see [Install Miniconda](#).
- Use a [Dev Container](#) in [Visual Studio Code](#) or [GitHub Codespaces](#).

2: Install the needed Azure library packages

Create a *requirements.txt* file specifying the Azure SDK management packages required by this script.

```
txt  
  
azure-mgmt-resource  
azure-mgmt-compute  
azure-mgmt-network  
azure-identity
```

Next, install the management libraries specified in *requirements.txt*:

```
Console  
  
pip install -r requirements.txt
```

3: Write code to create a virtual machine

Create a Python file named *provision_vm.py* with the following code. The comments explain the details:

```
Python  
  
# Import the needed credential and management objects from the libraries.  
import os  
  
from azure.identity import DefaultAzureCredential  
from azure.mgmt.compute import ComputeManagementClient  
from azure.mgmt.network import NetworkManagementClient  
from azure.mgmt.resource import ResourceManagementClient  
  
print(  
    "Provisioning a virtual machine...some operations might take a \  
    minute or two."  
)  
  
# Acquire a credential object.  
credential = DefaultAzureCredential()  
  
# Retrieve subscription ID from environment variable.  
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]  
  
# Step 1: Provision a resource group  
  
# Obtain the management object for resources.  
resource_client = ResourceManagementClient(credential, subscription_id)  
  
# Constants we need in multiple places: the resource group name and
```

```
# the region in which we provision resources. You can change these
# values however you want.
RESOURCE_GROUP_NAME = "PythonAzureExample-VM-rg"
LOCATION = "westus2"

# Provision the resource group.
rg_result = resource_client.resource_groups.create_or_update(
    RESOURCE_GROUP_NAME, {"location": LOCATION})
)

print(
    f"Provisioned resource group {rg_result.name} in the \
{rg_result.location} region"
)

# For details on the previous code, see Example: Provision a resource
# group at https://learn.microsoft.com/azure/developer/python/
# azure-sdk-example-resource-group

# Step 2: provision a virtual network

# A virtual machine requires a network interface client (NIC). A NIC
# requires a virtual network and subnet along with an IP address.
# Therefore we must provision these downstream components first, then
# provision the NIC, after which we can provision the VM.

# Network and IP address names
VNET_NAME = "python-example-vnet"
SUBNET_NAME = "python-example-subnet"
IP_NAME = "python-example-ip"
IP_CONFIG_NAME = "python-example-ip-config"
NIC_NAME = "python-example-nic"

# Obtain the management object for networks
network_client = NetworkManagementClient(credential, subscription_id)

# Provision the virtual network and wait for completion
poller = network_client.virtual_networks.begin_create_or_update(
    RESOURCE_GROUP_NAME,
    VNET_NAME,
    {
        "location": LOCATION,
        "address_space": {"address_prefixes": ["10.0.0.0/16"]},
    },
)
vnet_result = poller.result()

print(
    f"Provisioned virtual network {vnet_result.name} with address \
prefixes {vnet_result.address_space.address_prefixes}"
)

# Step 3: Provision the subnet and wait for completion
poller = network_client.subnets.begin_create_or_update(
```

```
RESOURCE_GROUP_NAME,
VNET_NAME,
SUBNET_NAME,
{"address_prefix": "10.0.0.0/24"},
)
subnet_result = poller.result()

print(
    f"Provisioned virtual subnet {subnet_result.name} with address \
prefix {subnet_result.address_prefix}"
)

# Step 4: Provision an IP address and wait for completion
poller = network_client.public_ip_addresses.begin_create_or_update(
    RESOURCE_GROUP_NAME,
    IP_NAME,
    {
        "location": LOCATION,
        "sku": {"name": "Standard"},
        "public_ip_allocation_method": "Static",
        "public_ip_address_version": "IPv4",
    },
)
ip_address_result = poller.result()

print(
    f"Provisioned public IP address {ip_address_result.name} \
with address {ip_address_result.ip_address}"
)

# Step 5: Provision the network interface client
poller = network_client.network_interfaces.begin_create_or_update(
    RESOURCE_GROUP_NAME,
    NIC_NAME,
    {
        "location": LOCATION,
        "ip_configurations": [
            {
                "name": IP_CONFIG_NAME,
                "subnet": {"id": subnet_result.id},
                "public_ip_address": {"id": ip_address_result.id},
            }
        ],
    },
)
nic_result = poller.result()

print(f"Provisioned network interface client {nic_result.name}")

# Step 6: Provision the virtual machine

# Obtain the management object for virtual machines
compute_client = ComputeManagementClient(credential, subscription_id)
```

```

VM_NAME = "ExampleVM"
USERNAME = "azureuser"
PASSWORD = "ChangePa$$w0rd24"

print(
    f"Provisioning virtual machine {VM_NAME}; this operation might \
take a few minutes."
)

# Provision the VM specifying only minimal arguments, which defaults
# to an Ubuntu 18.04 VM on a Standard DS1 v2 plan with a public IP address
# and a default virtual network/subnet.

poller = compute_client.virtual_machines.begin_create_or_update(
    RESOURCE_GROUP_NAME,
    VM_NAME,
    {
        "location": LOCATION,
        "storage_profile": {
            "image_reference": {
                "publisher": "Canonical",
                "offer": "UbuntuServer",
                "sku": "16.04.0-LTS",
                "version": "latest",
            }
        },
        "hardware_profile": {"vm_size": "Standard_DS1_v2"},
        "os_profile": {
            "computer_name": VM_NAME,
            "admin_username": USERNAME,
            "admin_password": PASSWORD,
        },
        "network_profile": {
            "network_interfaces": [
                {
                    "id": nic_result.id,
                }
            ]
        },
    },
)
vm_result = poller.result()

print(f"Provisioned virtual machine {vm_result.name}")

```

Authentication in the code

Later in this article, you sign in to Azure using the Azure CLI to execute the sample code. If your account has sufficient permissions to create resource groups and storage resources in your Azure subscription, the script should run successfully without additional configuration.

To use this code in a production environment, authenticate using a service principal by setting environment variables. This approach enables secure, automated access without relying on interactive login. For detailed guidance, see [How to authenticate Python apps with Azure services](#).

Ensure that the service principal is assigned a role with sufficient permissions to create resource groups and storage accounts. For example, assigning the Contributor role at the subscription level provides the necessary access. To learn more about role assignments, see [Role-based access control \(RBAC\) in Azure](#).

Reference links for classes used in the code

- [DefaultCredential \(azure.identity\)](#)
- [ResourceManagementClient \(azure.mgmt.resource\)](#)
- [NetworkManagementClient \(azure.mgmt.network\)](#)
- [ComputeManagementClient \(azure.mgmt.compute\)](#)

4. Run the script

1. If you haven't already, sign in to Azure using the Azure CLI:

```
Azure CLI
```

```
az login
```

2. Set the `AZURE_SUBSCRIPTION_ID` environment variable to your subscription ID. (You can run the `az account show` command and get your subscription ID from the `id` property in the output):

```
Azure CLI
```

```
export AZURE_SUBSCRIPTION_ID=$(az account show --query id -o tsv)
```

3. Run the script:

```
Console
```

```
python provision_vm.py
```

The provisioning process takes a few minutes to complete.

5. Verify the resources

Open the [Azure portal](#), navigate to the "PythonAzureExample-VM-rg" resource group, and note the virtual machine, virtual disk, network security group, public IP address, network interface, and virtual network.

The screenshot shows the Azure portal's resource group overview page for "PythonAzureExample-VM-rg". The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Events, Settings (Deployments, Security, Policies, Properties, Locks), and Cost Management (Cost analysis, Cost alerts (preview)). The main area displays resource details under the "Essentials" section, including Subscription (change) to Primary, Deployment ID, Tags (change), and Location (West US 2). Below this is a search bar and filter options (Type == all, Location == all, Add filter). A table lists five resources: ExampleVM (Virtual machine), ExampleVM_disk1_f1eda1ebf984a11983a1b550f37b865 (Disk), python-example-ip (Public IP address), python-example-nic (Network interface), and python-example-vnet (Virtual network). The last four resources are highlighted with a red box.

You can also use the Azure CLI to verify that the VM exists with the `az vm list` command:

The screenshot shows the Azure CLI interface with the command `az vm list --resource-group PythonAzureExample-VM-rg` entered in the terminal window.

Equivalent Azure CLI commands

The screenshot shows the Azure CLI interface with the following commands entered:

```
# Provision the resource group
az group create -n PythonAzureExample-VM-rg -l westus2

# Provision a virtual network and subnet
az network vnet create -g PythonAzureExample-VM-rg -n python-example-vnet \
--address-prefix 10.0.0.0/16 --subnet-name python-example-subnet \
--subnet-prefix 10.0.0.0/24

# Provision a virtual machine
az vm create -n ExampleVM -g PythonAzureExample-VM-rg --image UbuntuLTS \
--size Standard_B1ms --admin-username exampleuser --admin-password !ExampleUser@123
```

```
# Provision a public IP address

az network public-ip create -g PythonAzureExample-VM-rg -n python-example-ip \
    --allocation-method Dynamic --version IPv4

# Provision a network interface client

az network nic create -g PythonAzureExample-VM-rg --vnet-name python-example-vnet \
    --subnet python-example-subnet -n python-example-nic \
    --public-ip-address python-example-ip

# Provision the virtual machine

az vm create -g PythonAzureExample-VM-rg -n ExampleVM -l "westus2" \
    --nics python-example-nic --image UbuntuLTS --public-ip-sku Standard \
    --admin-username azureuser --admin-password ChangePa$$w0rd24
```

If you get an error about capacity restrictions, you can try a different size or region. For more information, see [Resolve errors for SKU not available](#).

6: Clean up resources

Leave the resources in place if you want to continue to use the virtual machine and network you created in this article. Otherwise, run the [az group delete](#) command to delete the resource group.

Resource groups don't incur any ongoing charges in your subscription, but resources contained in the group, like virtual machines, might continue to incur charges. It's a good practice to clean up any group that you aren't actively using. The `--no-wait` argument allows the command to return immediately instead of waiting for the operation to finish.

Azure CLI

```
az group delete -n PythonAzureExample-VM-rg --no-wait
```

You can also use the [ResourceManagementClient.resource_groups.begin_delete](#) method to delete a resource group from code. The code in [Example: Create a resource group](#) demonstrates usage.

See also

- [Example: Create a resource group](#)
- [Example: List resource groups in a subscription](#)
- [Example: Create Azure Storage](#)

- Example: Use Azure Storage
- Example: Create a web app and deploy code
- Example: Create and query a database
- Use Azure Managed Disks with virtual machines
- Complete a short survey about the Azure SDK for Python ↗

The following resources contain more comprehensive examples using Python to create a virtual machine:

- [Azure Virtual Machines Management Samples - Python ↗](#) (GitHub). The sample demonstrates more management operations like starting and restarting a VM, stopping and deleting a VM, increasing the disk size, and managing data disks.

Use Azure Managed Disks with the Azure libraries (SDK) for Python

06/12/2025

Azure Managed Disks are high-performance, durable block storage designed for use with Azure Virtual Machines and Azure VMware Solution. They simplify disk management, offer greater scalability, enhance security, and eliminate the need to manage storage accounts directly. For more information, see [Azure Managed Disks](#).

For operations on Managed Disks associated with an existing VM, use the `azure-mgmt-compute` library.

The code examples in this article demonstrate common operations with Managed Disks using the `azure-mgmt-compute` library. These examples are not meant to be run as standalone scripts, but rather to be integrated into your own code. To learn how to create a `ComputeManagementClient` instance from `azure.mgmt.compute` in your script, see [Example - Create a virtual machine](#).

For more complete examples of how to use the `azure-mgmt-compute` library, see [Azure SDK for Python samples for compute](#) ↗ in GitHub.

Standalone Managed Disks

The following examples show different ways to provision standalone Managed Disks.

Create an empty Managed Disk

This example shows how to create a new empty Managed Disk. You can use it as a blank disk to attach to a virtual machine or as a starting point for creating snapshots or images.

Python

```
from azure.mgmt.compute.models import DiskCreateOption

poller = compute_client.disks.begin_create_or_update(
    'my_resource_group',
    'my_disk_name',
    {
        'location': 'eastus',
        'disk_size_gb': 20,
        'creation_data': {
            'create_option': DiskCreateOption.empty
    }
}
```

```
        }
    )
disk_resource = poller.result()
```

Create a Managed Disk from blob storage

This example shows how to create a Managed Disk from a VHD file stored in Azure Blob Storage. This is helpful when you want to reuse or move an existing virtual hard disk into Azure.

Python

```
from azure.mgmt.compute.models import DiskCreateOption

poller = compute_client.disks.begin_create_or_update(
    'my_resource_group',
    'my_disk_name',
    {
        'location': 'eastus',
        'creation_data': {
            'create_option': DiskCreateOption.IMPORT,
            'storage_account_id': '/subscriptions/<subscription-
id>/resourceGroups/<resource-group-
name>/providers/Microsoft.Storage/storageAccounts/<storage-account-name>',
            'source_uri': 'https://<storage-account-
name>.blob.core.windows.net/vm-images/test.vhd'
        }
    }
)
disk_resource = poller.result()
```

Create a Managed Disk image from blob storage

This example shows how to create a Managed Disk image from a VHD file stored in Azure Blob Storage. This is useful when you want to make a reusable image that can be used to create new virtual machines.

Python

```
from azure.mgmt.compute.models import OperatingSystemStateTypes, HyperVGeneration

poller = compute_client.images.begin_create_or_update(
    'my_resource_group',
    'my_image_name',
    {
        'location': 'eastus',
        'storage_profile': {
            'os_disk': {
                'os_type': 'Linux',

```

```
        'os_state': OperatingSystemStateTypes.GENERALIZED,
        'blob_uri': 'https://<storage-account-
name>.blob.core.windows.net/vm-images/test.vhd',
        'caching': "ReadWrite",
    },
},
{
'hyper_v_generation': HyperVGeneration.V2,
}
)
image_resource = poller.result()
```

Create a Managed Disk from your own image

This example shows how to create a new Managed Disk by copying an existing one. This is helpful when you want to make a backup or use the same disk setup on another virtual machine.

Python

```
from azure.mgmt.compute.models import DiskCreateOption

# If you don't know the id, do a 'get' like this to obtain it
managed_disk = compute_client.disks.get(self.group_name, 'myImageDisk')

poller = compute_client.disks.begin_create_or_update(
    'my_resource_group',
    'my_disk_name',
    {
        'location': 'eastus',
        'creation_data': {
            'create_option': DiskCreateOption.COPY,
            'source_resource_id': managed_disk.id
        }
    }
)

disk_resource = poller.result()
```

Virtual machine with Managed Disks

You can create a virtual machine with an implicitly created Managed Disk based on a specific disk image, eliminating the need to manually define all disk details.

A Managed Disk is created implicitly when creating a VM from an OS image in Azure. Azure automatically handles the storage account, so you don't need to specify `storage_profile.os_disk` or create a storage account manually.

Python

```
storage_profile = azure.mgmt.compute.models.StorageProfile(
    image_reference = azure.mgmt.compute.models.ImageReference(
        publisher='Canonical',
        offer='UbuntuServer',
        sku='16.04-LTS',
        version='latest'
    )
)
```

For a complete example showing how to create a virtual machine using the Azure management libraries for Python, see [Example - Create a virtual machine](#). This example demonstrates how to use the `storage_profile` parameter.

You can also create a `storage_profile` from your own image:

Python

```
# If you don't know the id, do a 'get' like this to obtain it
image = compute_client.images.get(self.group_name, 'myImageDisk')

storage_profile = azure.mgmt.compute.models.StorageProfile(
    image_reference = azure.mgmt.compute.models.ImageReference(
        id = image.id
    )
)
```

You can easily attach a previously provisioned Managed Disk:

Python

```
vm = compute_client.virtual_machines.get(
    'my_resource_group',
    'my_vm'
)
managed_disk = compute_client.disks.get('my_resource_group', 'myDisk')

vm.storage_profile.data_disks.append({
    'lun': 12, # You choose the value, depending of what is available for you
    'name': managed_disk.name,
    'create_option': DiskCreateOptionTypes.attach,
    'managed_disk': {
        'id': managed_disk.id
    }
})

async_update = compute_client.virtual_machines.begin_create_or_update(
    'my_resource_group',
    vm.name,
```

```
    vm,
)
async_update.wait()
```

Virtual Machine Scale Sets with Managed Disks

Before Azure Managed Disks, you had to manually create a storage account for each VM in your Virtual Machine Scale Set and use the `vhd_containers` parameter to specify those storage accounts in the Scale Set REST API.

With Azure Managed Disks, storage account management is no longer required. As a result, the `storage_profile` for [Virtual Machine Scale Sets](#) used for Virtual Machine Scale Sets can now match the one used for individual VM creation:

Python

```
'storage_profile': {
    'image_reference': {
        "publisher": "Canonical",
        "offer": "UbuntuServer",
        "sku": "16.04-LTS",
        "version": "latest"
    }
},
```

The full sample is as follows:

Python

```
naming_infix = "PyTestInfix"

vmss_parameters = {
    'location': self.region,
    "overprovision": True,
    "upgrade_policy": {
        "mode": "Manual"
    },
    'sku': {
        'name': 'Standard_A1',
        'tier': 'Standard',
        'capacity': 5
    },
    'virtual_machine_profile': {
        'storage_profile': {
            'image_reference': {
                "publisher": "Canonical",
                "offer": "UbuntuServer",
                "sku": "16.04-LTS",
                "version": "latest"
            }
        }
    }
},
```

```

        "version": "latest"
    }
},
'os_profile': {
    'computer_name_prefix': naming_infix,
    'admin_username': 'Foo12',
    'admin_password': 'BaR@123!!!!',
},
'network_profile': {
    'network_interface_configurations' : [
        {
            'name': naming_infix + 'nic',
            "primary": True,
            'ip_configurations': [
                {
                    'name': naming_infix + 'ipconfig',
                    'subnet': {
                        'id': subnet.id
                    }
                }
            ]
        }
    ]
}
}

# Create VMSS test
result_create = compute_client.virtual_machine_scale_sets.begin_create_or_update(
    'my_resource_group',
    'my_scale_set',
    vmss_parameters,
)
vmss_result = result_create.result()

```

Other operations with Managed Disks

Resizing a Managed Disk

This example shows how to make an existing Managed Disk larger. This is useful when you need more space for your data or applications.

Python

```

managed_disk = compute_client.disks.get('my_resource_group', 'myDisk')
managed_disk.disk_size_gb = 25

async_update = self.compute_client.disks.begin_create_or_update(
    'my_resource_group',
    'myDisk',
    managed_disk
)
async_update.wait()

```

Update the storage account type of the Managed Disks

This example shows how to change the storage type of a Managed Disk and make it larger. This is helpful when you need more space or better performance for your data or applications.

Python

```
from azure.mgmt.compute.models import StorageAccountTypes

managed_disk = compute_client.disks.get('my_resource_group', 'myDisk')
managed_disk.account_type = StorageAccountTypes.STANDARD_LRS

async_update = self.compute_client.disks.begin_create_or_update(
    'my_resource_group',
    'myDisk',
    managed_disk
)
async_update.wait()
```

Create an image from blob storage

This example shows how to create a Managed Disk image from a VHD file stored in Azure Blob Storage. This is useful when you want to make a reusable image that you can use to create new virtual machines.

Python

```
async_create_image = compute_client.images.create_or_update(
    'my_resource_group',
    'myImage',
    {
        'location': 'eastus',
        'storage_profile': {
            'os_disk': {
                'os_type': 'Linux',
                'os_state': "Generalized",
                'blob_uri': 'https://<storage-account-
name>.blob.core.windows.net/vm-images/test.vhd',
                'caching': "ReadWrite",
            }
        }
    }
)
image = async_create_image.result()
```

Create a snapshot of a Managed Disk that is currently attached to a virtual machine

This example shows how to take a snapshot of a Managed Disk that's attached to a virtual machine. You can use the snapshot to back up the disk or restore it later if needed.

Python

```
managed_disk = compute_client.disks.get('my_resource_group', 'myDisk')

async_snapshot_creation = self.compute_client.snapshots.begin_create_or_update(
    'my_resource_group',
    'mySnapshot',
    {
        'location': 'eastus',
        'creation_data': {
            'create_option': 'Copy',
            'source_uri': managed_disk.id
        }
    }
)
snapshot = async_snapshot_creation.result()
```

See also

- [Example: Create a virtual machine](#)
- [Example: Create a resource group](#)
- [Example: List resource groups in a subscription](#)
- [Example: Create Azure Storage](#)
- [Example: Use Azure Storage](#)
- [Example: Create and use a MySQL database](#)
- [Complete a short survey about the Azure SDK for Python ↗](#)

Configure logging in the Azure libraries for Python

Article • 11/01/2024

Azure Libraries for Python that are [based on azure.core](#) provide logging output using the standard Python [logging](#) library.

The general process to work with logging is as follows:

1. Acquire the logging object for the desired library and set the logging level.
2. Register a handler for the logging stream.
3. To include HTTP information, pass a `logging_enable=True` parameter to a client object constructor, a credential object constructor, or to a specific method.

Details are provided in the remaining sections of this article.

As a general rule, the best resource for understanding logging usage within the libraries is to browse the SDK source code at [github.com/Azure/azure-sdk-for-python](#). We encourage you to clone this repository locally so you can easily search for details when needed, as the following sections suggest.

Set logging levels

Python

```
import logging

# ...

# Acquire the logger for a library (azure.mgmt.resource in this example)
logger = logging.getLogger('azure.mgmt.resource')

# Set the desired logging level
logger.setLevel(logging.DEBUG)
```

- This example acquires the logger for the `azure.mgmt.resource` library, then sets the logging level to `logging.DEBUG`.
- You can call `logger.setLevel` at any time to change the logging level for different segments of code.

To set a level for a different library, use that library's name in the `logging.getLogger` call. For example, the `azure-eventhubs` library provides a logger named `azure.eventhubs`, the

azure-storage-queue library provides a logger named `azure.storage.queue`, and so on. (The SDK source code frequently uses the statement `logging.getLogger(__name__)`, which acquires a logger using the name of the containing module.)

You can also use more general namespaces. For example,

Python

```
import logging

# Set the logging level for all azure-storage-* libraries
logger = logging.getLogger('azure.storage')
logger.setLevel(logging.INFO)

# Set the logging level for all azure-* libraries
logger = logging.getLogger('azure')
logger.setLevel(logging.ERROR)
```

The `azure` logger is used by some libraries instead of a specific logger. For example, the `azure-storage-blob` library uses the `azure` logger.

You can use the `logger.isEnabledFor` method to check whether any given logging level is enabled:

Python

```
print(
    f"Logger enabled for ERROR={logger.isEnabledFor(logging.ERROR)}, "
    f"WARNING={logger.isEnabledFor(logging.WARNING)}, "
    f"INFO={logger.isEnabledFor(logging.INFO)}, "
    f"DEBUG={logger.isEnabledFor(logging.DEBUG)}"
)
```

Logging levels are the same as the [standard logging library levels](#). The following table describes the general use of these logging levels in the Azure libraries for Python:

[+] Expand table

Logging level	Typical use
<code>logging.ERROR</code>	Failures where the application is unlikely to recover (such as out of memory).
<code>logging.WARNING</code> (default)	A function fails to perform its intended task (but not when the function can recover, such as retrying a REST API call). Functions typically log a warning when raising exceptions. The warning level automatically enables the error level.

Logging level	Typical use
logging.INFO	Function operates normally or a service call is canceled. Info events typically include requests, responses, and headers. The info level automatically enables the error and warning levels.
logging.DEBUG	Detailed information that is commonly used for troubleshooting and includes a stack trace for exceptions. The debug level automatically enables the info, warning, and error levels. CAUTION: If you also set <code>logging_enable=True</code> , the debug level includes sensitive information such as account keys in headers and other credentials. Be sure to protect these logs to avoid compromising security.
logging.NOTSET	Disable all logging.

Library-specific logging level behavior

The exact logging behavior at each level depends on the library in question. Some libraries, such as `azure.eventhub`, perform extensive logging whereas other libraries do little.

The best way to examine the exact logging for a library is to search for the logging levels in the [Azure SDK for Python source code](#):

1. In the repository folder, navigate into the `sdk` folder, then navigate into the folder for the specific service of interest.
2. In that folder, search for any of the following strings:
 - `_LOGGER.error`
 - `_LOGGER.warning`
 - `_LOGGER.info`
 - `_LOGGER.debug`

Register a log stream handler

To capture logging output, you must register at least one log stream handler in your code:

Python

```
import logging
# Direct logging output to stdout. Without adding a handler,
# no logging output is visible.
```

```
handler = logging.StreamHandler(stream=sys.stdout)
logger.addHandler(handler)
```

This example registers a handler that directs log output to stdout. You can use other types of handlers as described on [logging.handlers](#) in the Python documentation or use the standard [logging.basicConfig](#) method.

Enable HTTP logging for a client object or operation

By default, logging within the Azure libraries doesn't include any HTTP information. To include HTTP information in log output, you must explicitly pass `logging_enable=True` to a client or credential object constructor or to a specific method.

⊗ Caution

HTTP logging can include sensitive information such as account keys in headers and other credentials. Be sure to protect these logs to avoid compromising security.

Enable HTTP logging for a client object

Python

```
from azure.storage.blob import BlobClient
from azure.identity import DefaultAzureCredential

# Enable HTTP logging on the client object when using DEBUG level
# endpoint is the Blob storage URL.
client = BlobClient(endpoint, DefaultAzureCredential(), logging_enable=True)
```

Enabling HTTP logging for a client object enables logging for all operations invoked through that object.

Enable HTTP logging for a credential object

Python

```
from azure.storage.blob import BlobClient
from azure.identity import DefaultAzureCredential

# Enable HTTP logging on the credential object when using DEBUG level
credential = DefaultAzureCredential(logging_enable=True)
```

```
# endpoint is the Blob storage URL.  
client = BlobClient(endpoint, credential)
```

Enabling HTTP logging for a credential object enables logging for all operations invoked through that object, but not for operations in a client object that don't involve authentication.

Enable logging for an individual method

Python

```
from azure.storage.blob import BlobClient  
from azure.identity import DefaultAzureCredential  
  
# endpoint is the Blob storage URL.  
client = BlobClient(endpoint, DefaultAzureCredential())  
  
# Enable HTTP logging for only this operation when using DEBUG level  
client.create_container("container01", logging_enable=True)
```

Example logging output

The following code is that shown in [Example: Use a storage account](#) with the addition of enabling DEBUG and HTTP logging:

Python

```
import logging  
import os  
import sys  
import uuid  
  
from azure.core import exceptions  
from azure.identity import DefaultAzureCredential  
from azure.storage.blob import BlobClient  
  
logger = logging.getLogger("azure")  
logger.setLevel(logging.DEBUG)  
  
# Set the logging level for the azure.storage.blob library  
logger = logging.getLogger("azure.storage.blob")  
logger.setLevel(logging.DEBUG)  
  
# Direct logging output to stdout. Without adding a handler,  
# no logging output is visible.  
handler = logging.StreamHandler(stream=sys.stdout)  
logger.addHandler(handler)
```

```

print(
    f"Logger enabled for ERROR={logger.isEnabledFor(logging.ERROR)}, "
    f"WARNING={logger.isEnabledFor(logging.WARNING)}, "
    f"INFO={logger.isEnabledFor(logging.INFO)}, "
    f"DEBUG={logger.isEnabledFor(logging.DEBUG)}"
)

try:
    credential = DefaultAzureCredential()
    storage_url = os.environ["AZURE_STORAGE_BLOB_URL"]
    unique_str = str(uuid.uuid4())[0:5]

    # Enable logging on the client object
    blob_client = BlobClient(
        storage_url,
        container_name="blob-container-01",
        blob_name=f"sample-blob-{unique_str}.txt",
        credential=credential,
    )

    with open("./sample-source.txt", "rb") as data:
        blob_client.upload_blob(data, logging_body=True,
logging_enable=True)

except (
    exceptions.ClientAuthenticationError,
    exceptions.HttpResponseError
) as e:
    print(e.message)

```

The output is as follows:

Output

```

Logger enabled for ERROR=True, WARNING=True, INFO=True, DEBUG=True
Request URL: 'https://pythonazurestorage12345.blob.core.windows.net/blob-
container-01/sample-blob-5588e.txt'
Request method: 'PUT'
Request headers:
    'Content-Length': '77'
    'x-ms-blob-type': 'BlockBlob'
    'If-None-Match': '*'
    'x-ms-version': '2023-11-03'
    'Content-Type': 'application/octet-stream'
    'Accept': 'application/xml'
    'User-Agent': 'azsdk-python-storage-blob/12.19.0 Python/3.10.11
(Windows-10-10.0.22631-SP0)'
    'x-ms-date': 'Fri, 19 Jan 2024 19:25:53 GMT'
    'x-ms-client-request-id': '8f7b1b0b-b700-11ee-b391-782b46f5c56b'
    'Authorization': '*****'
Request body:
b"Hello there, Azure Storage. I'm a friendly file ready to be stored in a

```

```
blob."  
Response status: 201  
Response headers:  
    'Content-Length': '0'  
    'Content-MD5': 'SUutm0872jZh+KYqtgjbTA=='  
    'Last-Modified': 'Fri, 19 Jan 2024 19:25:54 GMT'  
    'ETag': '"0x8DC1924749AE3C3"'  
    'Server': 'Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0'  
    'x-ms-request-id': '7ac499fa-601e-006d-3f0d-4bdf28000000'  
    'x-ms-client-request-id': '8f7b1b0b-b700-11ee-b391-782b46f5c56b'  
    'x-ms-version': '2023-11-03'  
    'x-ms-content-crc64': 'rtHLUlztgxc='  
    'x-ms-request-server-encrypted': 'true'  
    'Date': 'Fri, 19 Jan 2024 19:25:53 GMT'  
Response content:  
b''
```

Note

If you get an authorization error, make sure the identity you're running under is assigned the "Storage Blob Data Contributor" role on your blob container. To learn more, see [Use blob storage from app code \(Passwordless tab\)](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

How to configure proxies for the Azure SDK for Python

05/21/2025

A proxy is often needed if:

- You're behind a corporate firewall
- Your network traffic needs to go through a security appliance
- You want to use a custom proxy for debugging or routing

If your organization requires the use of a proxy server to access internet resources, you need to set an environment variable with the proxy server information to use the Azure SDK for Python. Setting the environment variables (`HTTP_PROXY` and `HTTPS_PROXY`) causes the Azure SDK for Python to use the proxy server at run time.

A proxy server URL has of the form `http[s]://[username:password@]<ip_address_or_domain>:<port>/` where the username and password combination is optional.

You can obtain your proxy information from your IT/network team, from your browser, or from network utilities.

You can then configure a proxy globally by using environment variables, or you can specify a proxy by passing an argument named `proxies` to an individual client constructor or operation method.

Global configuration

To configure a proxy globally for your script or app, define `HTTP_PROXY` or `HTTPS_PROXY` environment variables with the server URL. These variables work with any version of the Azure libraries. Note that `HTTPS_PROXY` doesn't mean `HTTPS` proxy, but the proxy for `https://` requests.

These environment variables are ignored if you pass the parameter `use_env_settings=False` to a client object constructor or operation method.

Set from the command line

cmd

Windows Command Prompt

```
rem Non-authenticated HTTP server:  
set HTTP_PROXY=http://10.10.1.10:1180  
  
rem Authenticated HTTP server:  
set HTTP_PROXY=http://username:password@10.10.1.10:1180  
  
rem Non-authenticated HTTPS server:  
set HTTPS_PROXY=http://10.10.1.10:1180  
  
rem Authenticated HTTPS server:  
set HTTPS_PROXY=http://username:password@10.10.1.10:1180
```

Set in Python code

You can set proxy settings using environment variables, with no custom configuration necessary.

Python

```
import os  
os.environ["HTTP_PROXY"] = "http://10.10.1.10:1180"  
  
# Alternate URL and variable forms:  
# os.environ["HTTP_PROXY"] = "http://username:password@10.10.1.10:1180"  
# os.environ["HTTPS_PROXY"] = "http://10.10.1.10:1180"  
# os.environ["HTTPS_PROXY"] = "http://username:password@10.10.1.10:1180"
```

Custom configuration

Set in Python code per-client or per-method

For custom configuration, you can specify a proxy for a specific client object or operation method. Specify a proxy server with an argument named `proxies`.

For example, the following code from the article [Example: use Azure storage](#) specifies an HTTPS proxy with user credentials with the `BlobClient` constructor. In this case, the object comes from the `azure.storage.blob` library, which is based on `azure.core`.

Python

```
from azure.identity import DefaultAzureCredential  
  
# Import the client object from the SDK library  
from azure.storage.blob import BlobClient
```

```
credential = DefaultAzureCredential()

storage_url = "https://<storageaccountname>.blob.core.windows.net"

blob_client = BlobClient(storage_url, container_name="blob-container-01",
    blob_name="sample-blob.txt", credential=credential,
    proxies={"https": "https://username:password@10.10.1.10:1180" })
)

# Other forms that the proxy URL might take:
# proxies={"http": "http://10.10.1.10:1180" }
# proxies={"http": "http://username:password@10.10.1.10:1180" }
# proxies={"https": "https://10.10.1.10:1180" }
```

Multicloud: Connect to all regions with the Azure libraries for Python

08/25/2025

You can use the Azure libraries for Python to connect to all regions where Azure is [available](#).

By default, the Azure libraries are configured to connect to the global Azure cloud.

Using pre-defined sovereign cloud constants

Pre-defined sovereign cloud constants are provided by the `AzureAuthorityHosts` module of the `azure.identity` library:

- `AZURE_CHINA`
- `AZURE_GOVERNMENT`
- `AZURE_PUBLIC_CLOUD`

To use a definition, import the appropriate constant from `azure.identity.AzureAuthorityHosts` and apply it when creating client objects.

When using `DefaultAzureCredential`, as shown in the following example, you can specify the cloud by using the appropriate value from `azure.identity.AzureAuthorityHosts`.

Python

```
import os
from azure.mgmt.resource import ResourceManagementClient, SubscriptionClient
from azure.identity import DefaultAzureCredential, AzureAuthorityHosts
from azure.core import AzureClouds

authority = AzureAuthorityHosts.AZURE_CHINA

# Set environment variable AZURE_SUBSCRIPTION_ID as well as environment variables
# for DefaultAzureCredential. For combinations of environment variables, see
# https://github.com/Azure/azure-sdk-for-python/tree/main/sdk/identity/azure-
# identity#environment-variables
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]

# When using sovereign domains (that is, any cloud other than AZURE_PUBLIC_CLOUD),
# you must use an authority with DefaultAzureCredential.
credential = DefaultAzureCredential(authority=authority)

resource_client = ResourceManagementClient(
    credential, subscription_id, cloud_setting=AzureClouds.AZURE_CHINA_CLOUD
)
```

```
subscription_client = SubscriptionClient(  
    credential, cloud_setting=AzureClouds.AZURE_CHINA_CLOUD  
)
```

➊ Note

The `cloud_setting` feature is newly added and is rolling out across Azure SDK management libraries. During this period, some clients support it while others do not. To check support, look for a `cloud_setting` parameter on the client constructor. If your service's client doesn't expose `cloud_setting` yet, you can still target sovereign clouds using the previous approach shown in the examples below.

Python

```
import os  
from azure.mgmt.resource import ResourceManagementClient, SubscriptionClient  
from azure.identity import DefaultAzureCredential, AzureAuthorityHosts  
  
authority = AzureAuthorityHosts.AZURE_CHINA  
resource_manager = "https://management.chinacloudapi.cn"  
  
# Set environment variable AZURE_SUBSCRIPTION_ID as well as environment variables  
# for DefaultAzureCredential. For combinations of environment variables, see  
# https://github.com/Azure/azure-sdk-for-python/tree/main/sdk/identity/azure-  
# identity#environment-variables  
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]  
  
# When using sovereign domains (that is, any cloud other than AZURE_PUBLIC_CLOUD),  
# you must use an authority with DefaultAzureCredential.  
credential = DefaultAzureCredential(authority=authority)  
  
resource_client = ResourceManagementClient(  
    credential,  
    subscription_id,  
    base_url=resource_manager,  
    credential_scopes=[resource_manager + "/.default"],  
)  
  
subscription_client = SubscriptionClient(  
    credential,  
    base_url=resource_manager,  
    credential_scopes=[resource_manager + "/.default"],  
)
```

Using your own cloud definition

In the following code, replace the values of the `authority`, `endpoint`, and `audience` variables with values appropriate for your private cloud.

Python

```
import os
from azure.mgmt.resource import ResourceManagementClient, SubscriptionClient
from azure.identity import DefaultAzureCredential
from azure.profiles import KnownProfiles

# Set environment variable AZURE_SUBSCRIPTION_ID as well as environment variables
# for DefaultAzureCredential. For combinations of environment variables, see
# https://github.com/Azure/azure-sdk-for-python/tree/main/sdk/identity/azure-
# identity#environment-variables
subscription_id = os.environ["AZURE_SUBSCRIPTION_ID"]

authority = "<your authority>"
endpoint = "<your endpoint>"
audience = "<your audience>

# When using a private cloud, you must use an authority with
DefaultAzureCredential.
# The active_directory endpoint should be a URL like
https://login.microsoftonline.com.
credential = DefaultAzureCredential(authority=authority)

resource_client = ResourceManagementClient(
    credential, subscription_id,
    base_url=endpoint,
    profile=KnownProfiles.v2019_03_01_hybrid,
    credential_scopes=[audience])

subscription_client = SubscriptionClient(
    credential,
    base_url=endpoint,
    profile=KnownProfiles.v2019_03_01_hybrid,
    credential_scopes=[audience])
```

For example, for Azure Stack, you can use the `az cloud show` CLI command to return the details of a registered cloud. The following output shows the values returned for the Azure public cloud, but the output for an Azure Stack private cloud should be similar.

Output

```
{
  "endpoints": {
    "activeDirectory": "https://login.microsoftonline.com",
    "activeDirectoryDataLakeResourceId": "https://datalake.azure.net/",
    "activeDirectoryGraphResourceId": "https://graph.windows.net/",
    "activeDirectoryResourceId": "https://management.core.windows.net/",
    "appInsightsResourceId": "https://api.applicationinsights.io",
```

```

    "appInsightsTelemetryChannelResourceId":  

    "https://dc.applicationinsights.azure.com/v2/track",  

    "attestationResourceId": "https://attest.azure.net",  

    "azmirrorStorageAccountResourceId": null,  

    "batchResourceId": "https://batch.core.windows.net/",  

    "gallery": "https://gallery.azure.com/",  

    "logAnalyticsResourceId": "https://api.loganalytics.io",  

    "management": "https://management.core.windows.net/",  

    "mediaResourceId": "https://rest.media.azure.net",  

    "microsoftGraphResourceId": "https://graph.microsoft.com/",  

    "osssrdbmsResourceId": "https://osssrdbms-aad.database.windows.net",  

    "portal": "https://portal.azure.com",  

    "resourceManager": "https://management.azure.com/",  

    "sqlManagement": "https://management.core.windows.net:8443/",  

    "synapseAnalyticsResourceId": "https://dev.azuresynapse.net",  

    "vmImageAliasDoc": "https://raw.githubusercontent.com/Azure/azure-rest-api-specs/main/arm-compute/quickstart-templates/aliases.json"
},  

"isActive": true,  

"name": "AzureCloud",  

"profile": "latest",  

"suffixes": {  

    "acrLoginServerEndpoint": ".azurecr.io",  

    "attestationEndpoint": ".attest.azure.net",  

    "azureDatalakeAnalyticsCatalogAndJobEndpoint": "azuredatalakeanalytics.net",  

    "azureDatalakeStoreFileSystemEndpoint": "azuredatalakestore.net",  

    "keyvaultDns": ".vault.azure.net",  

    "mariadbServerEndpoint": ".mariadb.database.azure.com",  

    "mhsmDns": ".managedhsm.azure.net",  

    "mysqlServerEndpoint": ".mysql.database.azure.com",  

    "postgresqlServerEndpoint": ".postgres.database.azure.com",  

    "sqlServerHostname": ".database.windows.net",  

    "storageEndpoint": "core.windows.net",  

    "storageSyncEndpoint": "afs.azure.net",  

    "synapseAnalyticsEndpoint": ".dev.azuresynapse.net"
}  

}
}

```

In the preceding code, you can set `authority` to the value of the `endpoints.activeDirectory` property, `endpoint` to the value of the `endpoints.resourceManager` property, and `audience` to the value of `endpoints.activeDirectoryResourceId` property + `".default"`.

For more information, see [Use Azure CLI with Azure Stack Hub](#) and [Get authentication information for Azure Stack Hub](#).

Overview of Azure SDK for Python fundamentals

08/26/2025

The Fundamentals of the Python SDK for Azure section equips developers with the foundational concepts and core behaviors that underpin every client library in the Azure SDK for Python. These topics are considered "fundamentals" because they establish the essential building blocks for effective, idiomatic, and resilient application development across all Azure services.

Whether you're working with HTTP retries, handling errors, understanding SDK response types, or following consistent language design guidelines, these articles provide the baseline knowledge required to confidently navigate and extend your use of the Azure SDK. Mastering these fundamentals ensures that you're not only writing functional code but also writing code that's maintainable, robust, and aligned with best practices across the Azure ecosystem.

[] [Expand table](#)

Article Title	Purpose
Handling errors produced by the Azure SDK for Python	Describes the SDK's comprehensive error model, including best practices for handling specific exception types and implementing resilient error-handling strategies.
HTTP pipeline and retries in the Azure SDK libraries for Python	Provides a deep dive into the SDK's internal HTTP pipeline, showing how policies like retries, logging, and authentication are layered to manage requests and responses.
Understanding common response types in the Azure SDK for Python	Explains how SDK methods return intuitive, strongly typed Python objects, simplifying how you work with Azure responses and long-running operations.
Azure SDK Language Design Guidelines for Python	Outlines the conventions and design patterns used across the SDK to ensure consistency, usability, and alignment with Python best practices.

Handling errors produced by the Azure SDK for Python

08/26/2025

Building reliable cloud applications requires more than just implementing features—it demands robust error handling strategies. When working with distributed systems and cloud services, your application must be prepared to handle various failure scenarios gracefully.

The Azure SDK for Python provides a comprehensive error model designed to help developers build resilient applications. Understanding this error model is crucial for:

- Improving application reliability by anticipating and handling common failure scenarios
- Enhancing user experience through meaningful error messages and graceful degradation
- Simplifying troubleshooting by capturing and logging relevant diagnostic information

This article explores the Azure SDK for Python's error architecture and provides practical guidance for implementing effective error handling in your applications.

How the Azure SDK for Python models errors

The Azure SDK for Python uses a hierarchical exception model that provides both general and specific error handling capabilities. At the core of this model is `AzureError`, which serves as the base exception class for all Azure SDK-related errors.

Exception hierarchy

AzureError	
	ClientAuthenticationError
	ResourceNotFoundError
	ResourceExistsError
	ResourceModifiedError
	ResourceNotModifiedError
	ServiceRequestError
	ServiceResponseError
	HttpErrorResponse

Key exception types

 Expand table

Error	Description
AzureError	The base exception class for all Azure SDK errors. Use this as a catch-all when you need to handle any Azure-related error.
ClientAuthenticationError	Raised when authentication fails. Common causes include invalid credentials, expired tokens, and misconfigured authentication settings.
ResourceNotFoundError	Raised when attempting to access a resource that doesn't exist. This typically corresponds to HTTP 404 responses.
ResourceExistsError	Raised when attempting to create a resource that already exists. This helps prevent accidental overwrites.
ServiceRequestError	Raised when the SDK can't send a request to the service. Common causes include network connectivity issues, DNS resolution failures, and invalid service endpoints.
ServiceResponseError	Raised when the service returns an unexpected response that the SDK can't process.
HttpErrorResponse	Raised for HTTP error responses (4xx and 5xx status codes). This exception provides access to the underlying HTTP response details.

Common error scenarios

Understanding typical error scenarios helps you implement appropriate handling strategies for each situation.

Authentication and authorization errors

Authentication failures occur when the SDK can't verify your identity:

Python

```
from azure.core.exceptions import ClientAuthenticationError
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient

try:
    credential = DefaultAzureCredential()
    blob_service = BlobServiceClient(
        account_url="https://myaccount.blob.core.windows.net",
        credential=credential
    )
    # Attempt to list containers
    containers = blob_service.list_containers()
except ClientAuthenticationError as e:
```

```
print(f"Authentication failed: {e.message}")
# Don't retry - fix credentials first
```

Authorization errors (typically `HttpResponseError` with 403 status) occur when you lack permissions:

Python

```
from azure.core.exceptions import HttpResponseError

try:
    blob_client.upload_blob(data)
except HttpResponseError as e:
    if e.status_code == 403:
        print("Access denied. Check your permissions.")
    else:
        raise
```

Resource errors

Handle missing resources gracefully:

Python

```
from azure.core.exceptions import ResourceNotFoundError

try:
    blob_client = container_client.get_blob_client("myblob.txt")
    content = blob_client.download_blob().readall()
except ResourceNotFoundError:
    print("Blob not found. Using default content.")
    content = b"default"
```

Prevent duplicate resource creation:

Python

```
from azure.core.exceptions import ResourceExistsError

try:
    container_client.create_container()
except ResourceExistsError:
    print("Container already exists.")
    # Continue with existing container
```

Server errors

Handle server-side failures appropriately:

Python

```
from azure.core.exceptions import HttpResponseError

try:
    result = client.process_data(large_dataset)
except HttpResponseError as e:
    if 500 <= e.status_code < 600:
        print(f"Server error ({e.status_code}). The service may be temporarily unavailable.")
        # Consider retry logic here
    else:
        raise
```

Best practices for error handling

- Use specific exception handling - Always catch specific exceptions before falling back to general ones:

Python

```
from azure.core.exceptions import (
    AzureError,
    ClientAuthenticationError,
    ResourceNotFoundError,
    HttpResponseError
)

try:
    # Azure SDK operation
    result = client.get_resource()
except ClientAuthenticationError:
    # Handle authentication issues
    print("Please check your credentials")
except ResourceNotFoundError:
    # Handle missing resources
    print("Resource not found")
except HttpResponseError as e:
    # Handle specific HTTP errors
    if e.status_code == 429:
        print("Rate limited. Please retry later.")
    else:
        print(f"HTTP error {e.status_code}: {e.message}")
except AzureError as e:
    # Catch-all for other Azure errors
    print(f"Azure operation failed: {e}")
```

- **Implement appropriate retry strategies** - Some errors warrant retry attempts, while others don't.

Don't retry on:

- 401 Unauthorized (authentication failures)
- 403 Forbidden (authorization failures)
- 400 Bad Request (client errors)
- 404 Not Found (unless you expect the resource to appear)

Consider retrying on:

- 408 Request Timeout
- 429 Too Many Requests (with appropriate backoff)
- 500 Internal Server Error
- 502 Bad Gateway
- 503 Service Unavailable
- 504 Gateway Timeout

- **Extract meaningful error information**

Python

```
from azure.core.exceptions import HttpResponseError

try:
    client.perform_operation()
except HttpResponseError as e:
    # Extract detailed error information
    print(f"Status code: {e.status_code}")
    print(f"Error message: {e.message}")
    print(f"Error code: {e.error.code if e.error else 'N/A'}")

    # Request ID is crucial for Azure support
    if hasattr(e, 'response') and e.response:
        request_id = e.response.headers.get('x-ms-request-id')
        print(f"Request ID: {request_id}")
```

Retry policies and resilience

The Azure SDK includes built-in retry mechanisms that handle transient failures automatically.

Default retry behavior

Most Azure SDK clients include default retry policies that:

- Retry on connection errors and specific HTTP status codes

- Use exponential backoff with jitter
- Limit the number of retry attempts

Customize retry policies

If the default behavior doesn't suit your use case, you can customize the retry policy as in the following example:

Python

```
from azure.storage.blob import BlobServiceClient
from azure.core.pipeline.policies import RetryPolicy

# Create a custom retry policy
retry_policy = RetryPolicy(
    retry_total=5, # Maximum retry attempts
    retry_backoff_factor=2, # Exponential backoff factor
    retry_backoff_max=60, # Maximum backoff time in seconds
    retry_on_status_codes=[408, 429, 500, 502, 503, 504]
)

# Apply to client
blob_service = BlobServiceClient(
    account_url="https://myaccount.blob.core.windows.net",
    credential=credential,
    retry_policy=retry_policy
)
```

Avoid handling network and timeout errors with custom loops

You should try to use built-in retries for network and timeout errors before implementing your own custom logic.

Python

```
from azure.core.exceptions import ServiceRequestError
import time

# Avoid this approach if possible
max_retries = 3
retry_count = 0

while retry_count < max_retries:
    try:
        response = client.get_secret("mysecret")
        break
    except ServiceRequestError as e:
        retry_count += 1
        if retry_count >= max_retries:
```

```
        raise
    print(f"Network error. Retrying... ({retry_count}/{max_retries})")
    time.sleep(2 ** retry_count) # Exponential backoff
```

Implementing circuit breaker patterns

For critical operations, consider implementing circuit breaker patterns:

Python

```
class CircuitBreaker:
    def __init__(self, failure_threshold=5, recovery_timeout=60):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.failure_count = 0
        self.last_failure_time = None
        self.state = 'closed' # closed, open, half-open

    def call(self, func, *args, **kwargs):
        if self.state == 'open':
            if time.time() - self.last_failure_time > self.recovery_timeout:
                self.state = 'half-open'
            else:
                raise Exception("Circuit breaker is open")

        try:
            result = func(*args, **kwargs)
            if self.state == 'half-open':
                self.state = 'closed'
                self.failure_count = 0
            return result
        except Exception as e:
            self.failure_count += 1
            self.last_failure_time = time.time()

            if self.failure_count >= self.failure_threshold:
                self.state = 'open'

            raise e
```

Understanding error messages and codes

Azure services return structured error responses that provide valuable debugging information.

- Parsing error responses

Python

```

from azure.core.exceptions import HttpResponseError
import json

try:
    client.create_resource(resource_data)
except HttpResponseError as e:
    # Many Azure services return JSON error details
    if e.response and e.response.text():
        try:
            error_detail = json.loads(e.response.text())
            print(f"Error code: {error_detail.get('error', {}).get('code')}")
            print(f"Error message: {error_detail.get('error',
{}).get('message')")

            # Some services provide additional details
            if 'details' in error_detail.get('error', {}):
                for detail in error_detail['error']['details']:
                    print(f" - {detail.get('code')}:
{detail.get('message')}")
        except json.JSONDecodeError:
            print(f"Raw error: {e.response.text()}")

```

- **Capturing diagnostic information** - Always capture key diagnostic information for troubleshooting:

Python

```

import logging
from azure.core.exceptions import AzureError

logger = logging.getLogger(__name__)

try:
    result = client.perform_operation()
except AzureError as e:
    # Log comprehensive error information
    logger.error(
        "Azure operation failed",
        extra={
            'error_type': type(e).__name__,
            'error_message': str(e),
            'operation': 'perform_operation',
            'timestamp': datetime.utcnow().isoformat(),
            'request_id': getattr(e.response, 'headers', {}).get('x-ms-request-id') if hasattr(e, 'response') else None
        }
    )
    raise

```

- **Logging and diagnostics** - Enable SDK-level logging for detailed troubleshooting:

Python

```
import logging
import sys

# Configure logging for Azure SDKs
logging.basicConfig(level=logging.DEBUG)

# Enable HTTP request/response logging
logging.getLogger('azure.core.pipeline.policies.http_logging_policy').setLevel(logging.DEBUG)

# For specific services
logging.getLogger('azure.storage.blob').setLevel(logging.DEBUG)
logging.getLogger('azure.identity').setLevel(logging.DEBUG)
```

For more information about logging, see [Configure logging in the Azure libraries for Python](#).

- **Using network tracing** - For deep debugging, enable network-level tracing:

 **Important**

HTTP logging can include sensitive information such as account keys in headers and other credentials. Be sure to protect these logs to avoid compromising security.

Python

```
from azure.storage.blob import BlobServiceClient

# Enable network tracing
blob_service = BlobServiceClient(
    account_url="https://myaccount.blob.core.windows.net",
    credential=credential,
    logging_enable=True, # Enable logging
    logging_body=True     # Log request/response bodies (careful with
sensitive data)
)
```

Special considerations for async programming

When using async clients, error handling requires special attention.

- **Basic async error handling**

Python

```

import asyncio
from azure.core.exceptions import AzureError

async def get_secret_async(client, secret_name):
    try:
        secret = await client.get_secret(secret_name)
        return secret.value
    except ResourceNotFoundError:
        print(f"Secret '{secret_name}' not found")
        return None
    except AzureError as e:
        print(f"Error retrieving secret: {e}")
        raise

```

- Handling cancellations

Python

```

async def long_running_operation(client):
    try:
        result = await client.start_long_operation()
        # Wait for completion
        final_result = await result.result()
        return final_result
    except asyncio.CancelledError:
        print("Operation cancelled")
        # Cleanup if necessary
        if hasattr(result, 'cancel'):
            await result.cancel()
        raise
    except AzureError as e:
        print(f"Operation failed: {e}")
        raise

```

- Concurrent error handling

Python

```

async def process_multiple_resources(client, resource_ids):
    tasks = []
    for resource_id in resource_ids:
        task = client.get_resource(resource_id)
        tasks.append(task)

    results = []
    errors = []

    # Use gather with return_exceptions to handle partial failures
    outcomes = await asyncio.gather(*tasks, return_exceptions=True)

    for resource_id, outcome in zip(resource_ids, outcomes):

```

```
if isinstance(outcome, Exception):
    errors.append((resource_id, outcome))
else:
    results.append(outcome)

# Process successful results and errors appropriately
if errors:
    print(f"Failed to process {len(errors)} resources")
    for resource_id, error in errors:
        print(f" - {resource_id}: {error}")

return results
```

Summary of best practices

Effective error handling in Azure SDK for Python applications requires:

- **Anticipate failures:** Cloud applications must expect and handle partial failures gracefully.
- **Use specific exception handling:** Catch specific exceptions like ResourceNotFoundError and ClientAuthenticationError before falling back to general AzureError handling.
- **Implement smart retry logic:** Use built-in retry policies or customize them based on your needs. Remember that not all errors should trigger retries.
- **Capture diagnostic information:** Always log request IDs, error codes, and timestamps for effective troubleshooting.
- **Provide meaningful user feedback:** Transform technical errors into user-friendly messages while preserving technical details for support.
- **Test error scenarios:** Include error handling in your test coverage to ensure your application behaves correctly under failure conditions.

Next steps

- [Azure Core exceptions Module reference](#)
- Learn about [troubleshooting authentication and authorization issues](#)
- Explore [Azure Monitor OpenTelemetry](#) for comprehensive application monitoring

Understanding HTTP pipeline and retries in the Azure SDK for Python

08/26/2025

When you make a call to any Azure service using the Azure SDK for Python—whether it's Blob Storage, Key Vault, Cosmos DB, or any other HTTP based service—your request doesn't go directly to the Azure service. Instead, it flows through a sophisticated HTTP pipeline that handles critical cross-cutting concerns automatically.

Understanding how the HTTP pipeline works is essential for building robust, performant applications. The pipeline manages retries for transient failures, handles authentication, provides logging capabilities, and enables you to add custom behavior when needed. This knowledge helps you debug performance issues, optimize resiliency, and customize your application's interaction with Azure services.

What is the HTTP pipeline?

The Azure SDK for Python uses an internal HTTP pipeline architecture to process all requests and responses. This pipeline consists of a series of policies that execute in sequence, each responsible for a specific aspect of the HTTP communication. Think of the pipeline as a chain of processing steps:



Each policy in the pipeline can:

- Modify the request before it's sent
- Process the response after it's received
- Perform actions like retrying failed requests
- Add headers, log information, or implement custom logic

Key policies in the pipeline

The Azure SDK for Python includes several built-in policies that handle common scenarios:

- **RetryPolicy:** Automatically retries requests that fail due to transient errors. This policy implements intelligent retry logic with exponential backoff to avoid overwhelming services during outages.
- **BearerTokenCredentialPolicy:** Manages authentication by automatically acquiring and refreshing access tokens. This policy ensures your requests include valid authentication credentials without manual token management.
- **NetworkTraceLoggingPolicy:** Captures detailed information about HTTP requests and responses for debugging purposes. This policy is invaluable when troubleshooting communication issues.
- **HttpTransport:** The lowest layer of the pipeline that actually sends HTTP requests over the network. In the Azure SDK for Python, this is typically implemented using requests or aiohttp for asynchronous operations.

Additional policies

- **RedirectPolicy:** Handles HTTP redirects automatically
- **DistributedTracingPolicy:** Integrates with distributed tracing systems for monitoring
- **ProxyPolicy:** Routes requests through HTTP proxies when configured
- **UserAgentPolicy:** Adds SDK version information to request headers

Retry behavior

The Azure SDK for Python implements intelligent retry logic to handle transient failures automatically. Understanding this behavior helps you build more resilient applications.

Automatically retried conditions

The SDK retries requests for these HTTP status codes:

- **408 Request Timeout:** The server timed out waiting for the request
- **429 Too Many Requests:** Rate limiting is in effect
- **500 Internal Server Error:** Temporary server issue
- **502 Bad Gateway:** Temporary network issue
- **503 Service Unavailable:** Service temporarily unavailable
- **504 Gateway Timeout:** Gateway or proxy timeout

Default retry configuration

The default retry settings provide a good balance between resilience and performance:

- Maximum retry attempts: 3

- Retry mode: Exponential backoff
- Base delay: 0.8 seconds
- Maximum delay: 60 seconds
- Maximum total retry time: 120 seconds

The exponential backoff calculation follows this pattern:

```
Python
```

```
delay = min(base_delay * (2 ** retry_attempt), max_delay)
```

Customizing retries

You can customize retry behavior when creating SDK clients to match your application's specific requirements.

```
Python
```

```
from azure.storage.blob import BlobServiceClient
from azure.core.pipeline.policies import RetryPolicy

# Custom retry configuration
retry_policy = RetryPolicy(
    retry_total=5,                      # Maximum number of retry attempts
    retry_backoff_factor=0.5,            # Base backoff time in seconds
    retry_backoff_max=120,              # Maximum backoff time in seconds
    retry_on_status_codes=[429, 500, 502, 503, 504] # HTTP status codes to retry
)

# Apply custom retry policy to client
client = BlobServiceClient(
    account_url="https://myaccount.blob.core.windows.net",
    credential=credential,
    retry_policy=retry_policy
)
```

Disabling retries

For scenarios where retries aren't appropriate:

```
Python
```

```
from azure.core.pipeline.policies import RetryPolicy

# Disable retries completely
no_retry_policy = RetryPolicy(retry_total=0)
```

```
client = BlobServiceClient(  
    account_url="https://myaccount.blob.core.windows.net",  
    credential=credential,  
    retry_policy=no_retry_policy  
)
```

Diagnosing and debugging retry behavior

Understanding when and why retries occur is crucial for troubleshooting performance issues.

Enable SDK logging

The Azure SDK for Python uses Python's standard logging framework:

Python

```
import logging  
import sys  
  
# Configure logging to see retry attempts  
logging.basicConfig(  
    level=logging.DEBUG,  
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',  
    stream=sys.stdout  
)  
  
# Enable specific Azure SDK loggers  
azure_logger = logging.getLogger('azure')  
azure_logger.setLevel(logging.DEBUG)  
  
# Now SDK operations will log retry attempts
```

Identifying retry patterns

Look for log entries like:

Output

```
Retry attempt 1 for request [GET]  
https://myaccount.blob.core.windows.net/container/blob  
Waiting 0.8 seconds before retry
```

Common retry pitfalls

- **Retrying non-transient errors:** The SDK doesn't retry client errors (4xx) except for 408 and 429
- **Ignoring retry latency:** Remember that retries add latency to failed operations
- **Insufficient timeout:** Ensure your overall operation timeout accounts for retry delays

Advanced: Adding custom policies

You can extend the pipeline with custom policies for specialized scenarios.

Creating a custom policy

Python

```
from azure.core.pipeline import PipelineRequest, PipelineResponse
from azure.core.pipeline.policies import HTTPPolicy
from typing import Any, Optional

class CustomTelemetryPolicy(HTTPPolicy):
    """Custom policy to add telemetry headers"""

    def send(self, request: PipelineRequest) -> PipelineResponse:
        # Add custom header before sending request
        request.http_request.headers['X-Custom-Telemetry'] = 'my-app-v1.0'

        # Continue with the pipeline
        response = self.next.send(request)

        # Log response time
        print(f"Request to {request.http_request.url} completed")

        return response
```

Applying custom policies

Python

```
from azure.storage.blob import BlobServiceClient

# Create client with custom policy
client = BlobServiceClient(
    account_url="https://myaccount.blob.core.windows.net",
    credential=credential,
    per_call_policies=[CustomTelemetryPolicy()], # Policies that run per request
    per_retry_policies=[] # Policies that run per retry attempt
)
```

Policy ordering

Policies execute in a specific order:

- Per-call policies (execute once per operation)
- Retry policy
- Per-retry policies (execute on each attempt)
- Authentication policy
- HTTP transport

Best practices

Use default settings when possible

The default retry configuration works well for most scenarios. Only customize when you have specific requirements.

Customization guidelines

When customizing retry behavior:

- **Use exponential backoff:** Prevents overwhelming services during recovery
- **Set reasonable limits:** Cap total retry time to prevent indefinite waiting
- **Monitor retry metrics:** Track retry rates in production to identify issues
- **Consider circuit breakers:** For high-volume scenarios, implement circuit breaker patterns

What not to retry

Avoid retrying these types of errors:

- **Authentication failures (401, 403):** Authentication errors require fixing credentials, not retrying
- **Client errors (400, 404):** Client errors indicate problems with the request itself
- **Business logic errors:** Application-specific errors that don't resolve with retries

Operational excellence

- **Log correlation IDs:** Include `x-ms-client-request-id` in logs for Azure support
- **Set appropriate timeouts:** Balance between reliability and user experience
- **Test retry behavior:** Verify your application handles retries gracefully

- **Monitor performance:** Track P95/P99 latencies (percentile-based latency metrics) including retry overhead

Next steps

- [Implementing resilient applications](#)

Understanding common response types in the Azure SDK for Python

08/26/2025

The Azure SDK for Python abstracts calls to the underlying Azure service communication protocol, whether that protocol is HTTP or AMQP (which is used for messaging SDKs like ServiceBus, EventHubs, etc.). For example, if you use one of the libraries that utilizes HTTP, then the Azure SDK for Python is making HTTP requests and receiving HTTP responses under the hood. The SDK abstracts away this complexity, allowing you to work with intuitive Python objects instead of raw HTTP responses or JSON payloads.

Understanding the types of objects you receive from SDK operations is essential for writing effective Azure applications. This article explains the common response types you encounter and how they relate to the underlying HTTP communication.

ⓘ Note

This article only examines the HTTP scenario, not the AMQP scenario.

Deserialized Python objects

The Azure SDK for Python prioritizes developer productivity by returning strongly typed Python objects from service operations. Instead of parsing JSON or handling HTTP status codes directly, you work with resource models that represent Azure resources as Python objects.

For example, when you retrieve a blob from Azure Storage, you receive a `BlobProperties` object with attributes like `name`, `size`, and `last_modified`, rather than a raw JSON dictionary:

Python

```
from azure.storage.blob import BlobServiceClient

# Connect to storage account
blob_service_client = BlobServiceClient.from_connection_string(connection_string)
container_client = blob_service_client.get_container_client("mycontainer")

# Get blob properties - returns a BlobProperties object
blob_client = container_client.get_blob_client("myblob.txt")
properties = blob_client.get_blob_properties()

# Access properties as Python attributes
print(f"Blob name: {properties.name}")
```

```
print(f"Blob size: {properties.size} bytes")
print(f"Last modified: {properties.last_modified}")
```

Where the data comes from

Understanding the data flow helps you appreciate what the SDK does behind the scenes:

- Your code calls an SDK method - You invoke a method like `get_blob_properties()`
- The SDK constructs an HTTP request - The SDK builds the appropriate HTTP request with headers, authentication, and query parameters
- Azure service responds - The service returns an HTTP response, typically with a JSON payload in the response body
- The SDK processes the response - The SDK:
 - Checks the HTTP status code
 - Parses the response body (usually JSON)
 - Validates the data against expected schemas
 - Maps the data to Python model objects
- Your code receives Python objects - You work with the deserialized objects, not raw HTTP data

This abstraction allows you to focus on your application logic rather than HTTP protocol details.

Common response types

The Azure SDK for Python uses several standard response types across all services. Understanding these types helps you work effectively with any Azure service.

Resource models

Most SDK operations return resource models—Python objects that represent Azure resources. These models are service-specific but follow consistent patterns:

Python

```
# Azure Key Vault example
from azure.keyvault.secrets import SecretClient

secret_client = SecretClient(vault_url=vault_url, credential=credential)
secret = secret_client.get_secret("mysecret") # Returns KeyVaultSecret

print(f"Secret name: {secret.name}")
print(f"Secret value: {secret.value}")
```

```
print(f"Secret version: {secret.properties.version}")

# Azure Cosmos DB example
from azure.cosmos import CosmosClient

cosmos_client = CosmosClient(url=cosmos_url, credential=credential)
database = cosmos_client.get_database_client("mydatabase")
container = database.get_container_client("mycontainer")
item = container.read_item(item="item-id", partition_key="partition-value") # Returns dict

print(f"Item ID: {item['id']}
```

ItemPaged for collection results

When listing resources, the SDK returns `ItemPaged` objects that handle pagination transparently:

Python

```
from azure.storage.blob import BlobServiceClient
from azure.core.paging import ItemPaged

blob_service_client = BlobServiceClient.from_connection_string(connection_string)
container_client = blob_service_client.get_container_client("mycontainer")

# list_blobs returns ItemPaged[BlobProperties]
blobs: ItemPaged[BlobProperties] = container_client.list_blobs()

# Iterate naturally - SDK handles pagination
for blob in blobs:
    print(f"Blob: {blob.name}, Size: {blob.size}")
```

Accessing the raw HTTP response

While the SDK's high-level abstractions meet most needs, you sometimes need access to the underlying HTTP response. Common scenarios include:

- Debugging failed requests
- Accessing custom response headers
- Implementing custom retry logic
- Working with nonstandard response formats

Most SDK methods accept a `raw_response_hook` parameter:

Python

```
from azure.keyvault.secrets import SecretClient

secret_client = SecretClient(vault_url=vault_url, credential=credential)

def inspect_response(response):
    # Access the raw HTTP response
    print(f"Request URL: {response.http_request.url}")
    print(f"Status code: {response.http_response.status_code}")
    print(f"Response headers: {dict(response.http_response.headers)}")

    # Access custom headers
    request_id = response.http_response.headers.get('x-ms-request-id')
    print(f"Request ID: {request_id}")

    # Must return the response
    return response

# Hook is called before deserialization
secret = secret_client.get_secret("mysecret", raw_response_hook=inspect_response)
```

Paging and iterators

Azure services often return large collections of resources. The SDK uses ItemPaged to handle these collections efficiently without loading everything into memory at once.

Automatic pagination

The SDK automatically fetches new pages as you iterate:

```
Python

# List all blobs - could be thousands
blobs = container_client.list_blobs()

# SDK fetches pages as needed during iteration
for blob in blobs:
    process_blob(blob)  # Pages loaded on-demand
```

Working with pages explicitly

You can also work with pages directly when needed:

```
Python

blobs = container_client.list_blobs()
```

```
# Process by page
for page in blobs.by_page():
    print(f"Processing page with {len(list(page))} items")
    for blob in page:
        process_blob(blob)
```

Controlling page size

Many list operations accept a results_per_page parameter:

Python

```
# Fetch 100 items per page instead of the default
blobs = container_client.list_blobs(results_per_page=100)
```

Some methods for some Azure services have other mechanisms for controlling page size. For example, KeyVault and Azure Search use the `top` kwarg to limit results per call. See the [source code](#) for Azure Search's `search()` method as an example.

Special case: Long-running operations and pollers

Some Azure operations can't complete immediately. Examples include:

- Creating or deleting virtual machines
- Deploying ARM templates
- Training machine learning models
- Copying large blobs

These operations return poller objects that track the operation's progress.

Working with pollers

Python

```
from azure.mgmt.storage import StorageManagementClient

storage_client = StorageManagementClient(credential, subscription_id)

# Start storage account creation
poller = storage_client.storage_accounts.begin_create(
    resource_group_name="myresourcegroup",
    account_name="mystorageaccount",
    parameters=storage_parameters
)
```

```

# Option 1: Wait for completion (blocking)
storage_account = poller.result()

# Option 2: Check status periodically
while not poller.done():
    print(f"Status: {poller.status()}")
    time.sleep(5)

storage_account = poller.result()

```

Asynchronous pollers

When using `async/await` patterns, you work with `AsyncLROPoller`:

Python

```

from azure.storage.blob.aio import BlobServiceClient

async with BlobServiceClient.from_connection_string(connection_string) as client:
    container_client = client.get_container_client("mycontainer")

    # Start async copy operation
    blob_client = container_client.get_blob_client("large-blob.vhd")
    poller = await blob_client.begin_copy_from_url(source_url)

    # Wait for async completion
    copy_properties = await poller.result()

```

Polling objects for long-running operations example: Virtual Machines

Deploying Virtual Machines is an example of an operation that takes time to complete and handles it by returning poller objects (LROPoller for synchronous code, AsyncLROPoller for asynchronous code):

Python

```

from azure.mgmt.compute import ComputeManagementClient
from azure.core.polling import LROPoller

compute_client = ComputeManagementClient(credential, subscription_id)

# Start VM creation - returns immediately with a poller
poller: LROPoller = compute_client.virtual_machines.begin_create_or_update(
    resource_group_name="myresourcegroup",
    vm_name="myvm",
    parameters=vm_parameters
)

```

```
# Wait for completion and get the result
vm = poller.result() # Blocks until operation completes
print(f"VM {vm.name} provisioned successfully")
```

Accessing response for paged results

For paged results, use the `by_page()` method with `raw_response_hook`:

Python

```
def page_response_hook(response):
    continuation_token = response.http_response.headers.get('x-ms-continuation')
    print(f"Continuation token: {continuation_token}")
    return response

blobs = container_client.list_blobs()
for page in blobs.by_page(raw_response_hook=page_response_hook):
    for blob in page:
        print(blob.name)
```

Best practices

- Prefer high-level abstractions
- Work with the SDK's resource models rather than raw responses whenever possible, and avoid accessing any method prefixed with an underscore `_` since, by convention, those are private in Python. There are no guarantees about breaking changes, etc. compared to public APIs:

Python

```
# Preferred: Work with typed objects
secret = secret_client.get_secret("mysecret")
if secret.properties.enabled:
    use_secret(secret.value)

# Avoid: Manual JSON parsing (unless necessary) ...
# AND avoid accessing any objects or methods that start with `_
response = secret_client._client.get(...) # Don't access internal clients
data = json.loads(response.text)
if data['attributes']['enabled']:
    use_secret(data['value'])
```

- Handle pagination properly - Always iterate over paged results instead of converting to a list:

Python

```
# Good: Memory-efficient iteration
for blob in container_client.list_blobs():
    process_blob(blob)

# Avoid: Loading everything into memory
all_blobs = list(container_client.list_blobs()) # Could consume excessive
memory
```

- Use `poller.result()` for long-running operations - Always use the `result()` method to ensure operations complete successfully:

Python

```
# Correct: Wait for operation completion
poller = compute_client.virtual_machines.begin_delete(
    resource_group_name="myresourcegroup",
    vm_name="myvm"
)
poller.result() # Ensures deletion completes
print("VM deleted successfully")

# Wrong: Assuming immediate completion
poller = compute_client.virtual_machines.begin_delete(...)
print("VM deleted successfully") # Deletion might still be in progress!
```

- Access raw responses only when needed - Use raw response access sparingly and only for specific requirements:

Python

```
# Good use case: Debugging or logging
def log_request_id(response):
    request_id = response.http_response.headers.get('x-ms-request-id')
    logger.info(f"Operation request ID: {request_id}")
    return response

blob_client.upload_blob(data, raw_response_hook=log_request_id)

# Good use case: Custom error handling
def check_custom_header(response):
    if response.http_response.headers.get('x-custom-error'):
        raise CustomApplicationError("Custom error condition detected")
    return response
```

Azure SDK Language Design Guidelines for Python

08/26/2025

Azure SDK Design Guidelines are comprehensive standards that ensure consistency, predictability, and ease of use across all Azure SDKs. These guidelines help developers work efficiently with Azure services by providing familiar patterns and behaviors across different services and programming languages.

The guidelines consist of two categories:

- **General Guidelines:** Core principles that apply to all Azure SDKs regardless of programming language
- **Language-Specific Guidelines:** Implementation details optimized for each supported language, including [Python](#), [.NET](#), [Java](#), [TypeScript](#), and many more (see the Table of Contents starting on the [General Guidelines: Introduction](#) page).

These guidelines are developed openly on GitHub, allowing community review and contribution.

General design principles

All Azure SDKs follow these fundamental principles:

 [Expand table](#)

Principle	Description
Idiomatic usage	SDKs follow language-specific conventions and patterns
Consistency	Uniform behaviors across different Azure services
Simplicity	Common tasks require minimal code
Progressive disclosure	Advanced features are available but don't complicate basic usage
Robustness	Built-in handling for errors, retries, and timeouts

Python-specific guidelines

The rest of this document will focus on the [Python guidelines](#).

Naming conventions

Azure SDKs for Python follow standard Python naming conventions:

- **Methods** - Use snake_case.

```
Python
```

```
list_containers()  
get_secret()  
create_database()
```

- **Variables** - Use snake_case.

```
Python
```

```
connection_string = "..."  
retry_count = 3
```

- **Classes** - Use PascalCase.

```
Python
```

```
BlobServiceClient  
SecretClient  
CosmosClient
```

- **Constants** - Use UPPER_CASE.

```
Python
```

```
DEFAULT_CHUNK_SIZE  
MAX_RETRIES
```

Package structure

Azure SDK packages follow a consistent structure:

```
ascii
```

```
azure-<service>-<feature>  
└── azure/  
    └── <service>/  
        ├── __init__.py  
        ├── _client.py  
        └── _models.py
```

```
|   └── aio/           # Async implementations  
|       └── __init__.py
```

Client instantiation

Clients provide multiple instantiation methods:

Python

```
from azure.storage.blob import BlobServiceClient  
from azure.identity import DefaultAzureCredential  
  
# Note: do not use connection string if you can possibly avoid it!  
  
# Using account URL and credential  
credential = DefaultAzureCredential()  
client = BlobServiceClient(account_url="https://account.blob.core.windows.net",  
                           credential=credential)
```

Authentication

Azure SDKs use consistent authentication patterns:

Python

```
from azure.identity import DefaultAzureCredential, ClientSecretCredential  
  
# Default credential chain  
credential = DefaultAzureCredential()  
  
# Explicit credential  
credential = ClientSecretCredential(  
    tenant_id="tenant-id",  
    client_id="client-id",  
    client_secret="secret"  
)
```

Context managers

Most Azure SDK clients implement context manager protocols for automatic resource cleanup:

Python

```
from azure.storage.blob import BlobServiceClient
```

```
# Automatic cleanup with context manager
with BlobServiceClient.from_connection_string(conn_str) as client:
    container_client = client.get_container_client("mycontainer")
    blob_list = container_client.list_blobs()
```

! Note

While most clients support context managers, verify specific client documentation for availability.

Asynchronous operations

Async clients are provided in separate `.aio` modules:

Python

```
from azure.storage.blob.aio import BlobServiceClient
import asyncio

async def list_blobs_async():
    async with BlobServiceClient.from_connection_string(conn_str) as client:
        container_client = client.get_container_client("mycontainer")
        async for blob in container_client.list_blobs():
            print(blob.name)

# Run async function
asyncio.run(list_blobs_async())
```

Long-running operations

Long-running operations use the `begin_` prefix and return poller objects:

Python

```
from azure.storage.blob import BlobServiceClient

client = BlobServiceClient.from_connection_string(conn_str)
container_client = client.get_container_client("mycontainer")

# Start long-running operation
poller = container_client.begin_copy_blob_from_url(source_url)

# Wait for completion
result = poller.result()

# Or check status
print(result.status)
```

```
if poller.done():
    result = poller.result()
```

Pagination

List operations return iterables that handle pagination automatically:

Python

```
from azure.storage.blob import BlobServiceClient

client = BlobServiceClient.from_connection_string(conn_str)

# Automatic pagination
for container in client.list_containers():
    print(container.name)

# Manual pagination control
containers = client.list_containers(results_per_page=10).by_page()
for page in containers:
    for container in page:
        print(container.name)
```

Return types

Methods return strongly typed model objects rather than dictionaries:

Python

```
from azure.keyvault.secrets import SecretClient

client = SecretClient(vault_url="...", credential=credential)

# Returns KeyVaultSecret object, not dict
secret = client.get_secret("my-secret")
print(secret.value)
print(secret.properties.created_on)
```

Error handling

Azure SDK exceptions inherit from AzureError and provide specific exception types:

Python

```
from azure.core.exceptions import (
    AzureError,
```

```
        ResourceNotFoundError,  
        ResourceExistsError,  
        ClientAuthenticationError,  
        HttpResponseError  
)  
  
try:  
    blob_client.download_blob()  
except ResourceNotFoundError:  
    # Handle missing resource  
    print("Blob not found")  
except ClientAuthenticationError:  
    # Handle authentication failure  
    print("Authentication failed")  
except HttpResponseError as e:  
    # Handle HTTP errors  
    print(f"HTTP {e.status_code}: {e.message}")  
except AzureError as e:  
    # Handle any other Azure SDK error  
    print(f"Azure SDK error: {e}")
```

Configuration options

Clients accept configuration through keyword arguments:

Python

```
from azure.storage.blob import BlobServiceClient  
  
client = BlobServiceClient(  
    account_url="...",  
    credential=credential,  
    # Configuration options  
    max_single_put_size=64 * 1024 * 1024,  
    max_block_size=4 * 1024 * 1024,  
    retry_total=3,  
    logging_enable=True  
)
```

Common SDK patterns

Service client hierarchy

Azure SDKs typically follow a three-level hierarchy:

- **Service Client:** Entry point for service operations

Python

```
service_client = BlobServiceClient(...)
```

- **Resource Client:** Operations on specific resources

Python

```
container_client = service_client.get_container_client("container")
```

- **Operation Methods:** Actions on resources

Python

```
blob_client = container_client.get_blob_client("blob.txt")
blob_client.upload_blob(data)
```

Consistent method naming

Method names follow predictable patterns:

[] [Expand table](#)

Operation	Method Pattern	Example
Create	<code>create_<resource></code>	<code>create_container()</code>
Read	<code>get_<resource></code>	<code>get_blob()</code>
Update	<code>update_<resource></code>	<code>update_secret()</code>
Delete	<code>delete_<resource></code>	<code>delete_container()</code>
List	<code>list_<resources></code>	<code>list_blobs()</code>
Exists	<code>exists()</code>	<code>blob_client.exists()</code>

Working with Azure SDKs

The following example demonstrates how design guidelines create consistency across different Azure services:

Python

```

from azure.storage.blob import BlobServiceClient
from azure.keyvault.secrets import SecretClient
from azure.cosmos import CosmosClient
from azure.identity import DefaultAzureCredential

# Consistent authentication
credential = DefaultAzureCredential()

# Consistent client instantiation
blob_service = BlobServiceClient(
    account_url="https://account.blob.core.windows.net",
    credential=credential
)
secret_client = SecretClient(
    vault_url="https://vault.vault.azure.net",
    credential=credential
)
cosmos_client = CosmosClient(
    url="https://account.documents.azure.com",
    credential=credential
)

# Consistent method patterns
containers = blob_service.list_containers()
secrets = secret_client.list_properties_of_secrets()
databases = cosmos_client.list_databases()

# Consistent error handling
from azure.core.exceptions import ResourceNotFoundError

try:
    blob_service.get_container_client("container").get_container_properties()
    secret_client.get_secret("secret")
    cosmos_client.get_database_client("database").read()
except ResourceNotFoundError as e:
    print(f"Resource not found: {e}")

```

Contributing to Azure SDKs

When extending or contributing to Azure SDKs:

- **Follow Python idioms** - Use Pythonic patterns and conventions
- **Maintain consistency** - Align with existing SDK patterns
- **Write comprehensive tests** - Include unit and integration tests
- **Document thoroughly** - Provide docstrings and examples
- **Review guidelines** - Consult the Azure SDK Contribution Guide

Here's an example of implementing a custom client method that follows the Language Design Guidelines.

Python

```
def list_items_with_prefix(self, prefix: str, **kwargs) ->
    ItemPaged[ItemProperties]:
    """List items that start with the specified prefix.

    :param str prefix: The prefix to filter items
    :return: An iterable of ItemProperties
    :rtype: ~azure.core.paging.ItemPaged[ItemProperties]

    :example:
        items = client.list_items_with_prefix("test-")
        for item in items:
            print(item.name)
    """
    return self.list_items(name_starts_with=prefix, **kwargs)
```

Next steps

- Review the complete [Azure SDK Design Guidelines](#)
- Read the [Azure SDK Releases page](#)

Azure libraries package index

05/21/2025

Azure Python SDK packages are published to [PyPI](#), including beta releases marked with a "b" in the version number (such as 1.0.0b1). For the latest versions and release history, see the [Azure SDK Releases: Python](#).

If you're looking for help with using a specific SDK package, you have several options:

- **Source code and samples:** In the Azure SDK for Python GitHub repository, each package has its own folder. To view its code, select the GitHub link in the "Source" column of the package index. Most repos include a README.md file with example usage.
- **API Reference documentation:** To view the API reference, select the Docs link in the same table. These pages provide an overview of the package, its classes, and their available methods.
- **Tutorials and developer guidance:** To explore tutorials, how-to guides, and installation instructions, Visit the [Azure for Python developers](#) documentation. For example, to learn how to install SDK packages, see: [How to install Azure library packages for Python](#).

! Note

The **Name** column contains a friendly name for each package. To find the name you need to use to install the package with [pip](#), use the links in the **Package**, **Docs**, or **Source** columns. For example, the **Name** column for the Azure Blob Storage package is "Blobs" while the package name is *azure-storage-blob*.

For Conda libraries, see the [Microsoft channel on anaconda.org](#).

Libraries using azure.core

[Expand table](#)

Name	Package	Docs	Source
AI Agents	PyPI 1.0.0	docs	GitHub 1.0.0
	PyPI 1.1.0b1		GitHub 1.1.0b1
AI Evaluation	PyPI 1.8.0	docs	GitHub 1.8.0
AI Generative	PyPI 1.0.0b11		GitHub 1.0.0b11

Name	Package	Docs	Source
AI Model Inference	PyPI 1.0.0b9	docs	GitHub 1.0.0b9
AI Projects	PyPI 1.0.0b11	docs	GitHub 1.0.0b11
AI Resources	PyPI 1.0.0b9		GitHub 1.0.0b9
Anomaly Detector	PyPI 3.0.0b6	docs	GitHub 3.0.0b6
App Configuration	PyPI 1.7.1	docs	GitHub 1.7.1
App Configuration Provider	PyPI 2.1.0	docs	GitHub 2.1.0
Attestation	PyPI 1.0.0	docs	GitHub 1.0.0
Azure AI Search	PyPI 11.4.0 PyPI 11.6.0b12	docs	GitHub 11.4.0 GitHub 11.6.0b12
Azure AI Vision SDK	PyPI 0.15.1b1		GitHub 0.15.1b1
Azure Blob Storage Checkpoint Store	PyPI 1.2.0	docs	GitHub 1.2.0
Azure Blob Storage Checkpoint Store AIO	PyPI 1.2.0	docs	GitHub 1.2.0
Azure Monitor OpenTelemetry	PyPI 1.6.10	docs	GitHub 1.6.10
Azure Remote Rendering	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Communication Call Automation	PyPI 1.3.0 PyPI 1.4.0b1	docs	GitHub 1.3.0 GitHub 1.4.0b1
Communication Chat	PyPI 1.3.0	docs	GitHub 1.3.0
Communication Email	PyPI 1.0.0 PyPI 1.0.1b1	docs	GitHub 1.0.0 GitHub 1.0.1b1
Communication Identity	PyPI 1.5.0	docs	GitHub 1.5.0
Communication JobRouter	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
Communication Messages	PyPI 1.1.0 PyPI 1.2.0b1	docs	GitHub 1.1.0 GitHub 1.2.0b1
Communication Network Traversal	PyPI 1.1.0b2	docs	GitHub 1.1.0b2
Communication Phone Numbers	PyPI 1.2.0 PyPI 1.3.0b1	docs	GitHub 1.2.0 GitHub 1.3.0b1
Communication Rooms	PyPI 1.2.0	docs	GitHub 1.2.0
Communication Sms	PyPI 1.1.0	docs	GitHub 1.1.0

Name	Package	Docs	Source
Confidential Ledger	PyPI 1.1.1 ↗ PyPI 1.2.0b1 ↗	docs	GitHub 1.1.1 ↗ GitHub 1.2.0b1 ↗
Container Registry	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Content Safety	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Conversational Language Understanding	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Core - Client - Core	PyPI 1.34.0 ↗	docs	GitHub 1.34.0 ↗
Core - Client - Core HTTP	PyPI 1.0.0b6 ↗	docs	GitHub 1.0.0b6 ↗
Core - Client - Experimental	PyPI 1.0.0b4 ↗	docs	GitHub 1.0.0b4 ↗
Core - Client - Tracing Opentelemetry	PyPI 1.0.0b12 ↗	docs	GitHub 1.0.0b12 ↗
Core Tracing Opencensus	PyPI 1.0.0b10 ↗	docs	GitHub 1.0.0b10 ↗
Cosmos DB	PyPI 4.9.0 ↗ PyPI 4.12.0b1 ↗	docs	GitHub 4.9.0 ↗ GitHub 4.12.0b1 ↗
Defender EASM	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Dev Center	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Device Update	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Digital Twins	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Document Intelligence	PyPI 1.0.2 ↗	docs	GitHub 1.0.2 ↗
Document Translation	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Event Grid	PyPI 4.22.0 ↗	docs	GitHub 4.22.0 ↗
Event Hubs	PyPI 5.15.0 ↗	docs	GitHub 5.15.0 ↗
Face	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Face	PyPI 0.6.1 ↗	docs	GitHub 0.6.1 ↗
FarmBeats	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Form Recognizer	PyPI 3.3.3 ↗	docs	GitHub 3.3.3 ↗
Health Deidentification	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Health Insights Cancer Profiling	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Health Insights Clinical Matching	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗

Name	Package	Docs	Source
Health Insights Radiology Insights	PyPI 1.0.0	docs	GitHub 1.0.0
Identity	PyPI 1.23.0	docs	GitHub 1.23.0
Identity Broker	PyPI 1.2.0 PyPI 1.3.0b1	docs	GitHub 1.2.0 GitHub 1.3.0b1
Image Analysis	PyPI 1.0.0	docs	GitHub 1.0.0
Key Vault - Administration	PyPI 4.5.0 PyPI 4.6.0b1	docs	GitHub 4.5.0 GitHub 4.6.0b1
Key Vault - Certificates	PyPI 4.9.0 PyPI 4.10.0b1	docs	GitHub 4.9.0 GitHub 4.10.0b1
Key Vault - Keys	PyPI 4.10.0 PyPI 4.11.0b1	docs	GitHub 4.10.0 GitHub 4.11.0b1
Key Vault - Secrets	PyPI 4.9.0 PyPI 4.10.0b1	docs	GitHub 4.9.0 GitHub 4.10.0b1
Key Vault - Security Domain	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Load Testing	PyPI 1.0.1 PyPI 1.1.0b1	docs	GitHub 1.0.1 GitHub 1.1.0b1
Machine Learning	PyPI 1.27.1	docs	GitHub 1.27.1
Machine Learning - Feature Store	PyPI 1.0.1		GitHub 1.0.1
Managed Private Endpoints	PyPI 0.4.0	docs	GitHub 0.4.0
Maps Geolocation	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
Maps Render	PyPI 2.0.0b2	docs	GitHub 2.0.0b2
Maps Route	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
Maps Search	PyPI 2.0.0b2	docs	GitHub 2.0.0b2
Media Analytics Edge	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Metrics Advisor	PyPI 1.0.1	docs	GitHub 1.0.1
Mixed Reality Authentication	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Models Repository	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Monitor Ingestion	PyPI 1.0.4	docs	GitHub 1.0.4
Monitor Query	PyPI 1.4.1	docs	GitHub 1.4.1

Name	Package	Docs	Source
OpenTelemetry Exporter	PyPI 1.0.0b37	docs	GitHub 1.0.0b37
Personalizer	PyPI 1.0.0b1		GitHub 1.0.0b1
Purview Account	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Purview Administration	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Purview Catalog	PyPI 1.0.0b4	docs	GitHub 1.0.0b4
Purview Data Map	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Purview Scanning	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Purview Sharing	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
Purview Workflow	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Question Answering	PyPI 1.1.0	docs	GitHub 1.1.0
Schema Registry	PyPI 1.3.0	docs	GitHub 1.3.0
Schema Registry - Avro	PyPI 1.0.0	docs	GitHub 1.0.0
Schema Registry - Avro	PyPI 1.0.0b4	docs	GitHub 1.0.0b4
Service Bus	PyPI 7.14.2	docs	GitHub 7.14.2
Spark	PyPI 0.7.0	docs	GitHub 0.7.0
Storage - Blobs	PyPI 12.25.1	docs	GitHub 12.25.1
	PyPI 12.26.0b1		GitHub 12.26.0b1
Storage - Blobs Changefeed	PyPI 12.0.0b5	docs	GitHub 12.0.0b5
Storage - Files Data Lake	PyPI 12.20.0	docs	GitHub 12.20.0
	PyPI 12.21.0b1		GitHub 12.21.0b1
Storage - Files Share	PyPI 12.21.0	docs	GitHub 12.21.0
	PyPI 12.22.0b1		GitHub 12.22.0b1
Storage - Queues	PyPI 12.12.0	docs	GitHub 12.12.0
	PyPI 12.13.0b1		GitHub 12.13.0b1
Synapse - AccessControl	PyPI 0.7.0	docs	GitHub 0.7.0
Synapse - Artifacts	PyPI 0.20.0	docs	GitHub 0.20.0
Synapse - Monitoring	PyPI 0.2.0	docs	GitHub 0.2.0
Tables	PyPI 12.7.0	docs	GitHub 12.7.0

Name	Package	Docs	Source
Text Analytics	PyPI 5.3.0 ↗	docs	GitHub 5.3.0 ↗
Text Translation	PyPI 1.0.1 ↗	docs	GitHub 1.0.1 ↗
TimeZones	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Video Analyzer Edge	PyPI 1.0.0b4 ↗	docs	GitHub 1.0.0b4 ↗
Web PubSub	PyPI 1.2.2 ↗	docs	GitHub 1.2.2 ↗
Web PubSub Client	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Core - Management - Core	PyPI 1.5.0 ↗	docs	GitHub 1.5.0 ↗
Resource Management - Astro	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Dev Center	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Elastic SAN	PyPI 1.1.0 ↗ PyPI 1.2.0b2 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b2 ↗
Resource Management - Security DevOps	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Advisor	PyPI 9.0.0 ↗ PyPI 10.0.0b1 ↗	docs	GitHub 9.0.0 ↗ GitHub 10.0.0b1 ↗
Resource Management - Agrifood	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Agrifood	PyPI 1.0.0b3 ↗	docs	GitHub 1.0.0b3 ↗
Resource Management - AKS Developer Hub	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Alerts Management	PyPI 1.0.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - API Center	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - API Management	PyPI 5.0.0 ↗	docs	GitHub 5.0.0 ↗
Resource Management - App Compliance Automation	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - App Configuration	PyPI 4.0.0 ↗	docs	GitHub 4.0.0 ↗
Resource Management - App Platform	PyPI 10.0.1 ↗	docs	GitHub 10.0.1 ↗
Resource Management - App Service	PyPI 8.0.0 ↗	docs	GitHub 8.0.0 ↗
Resource Management - Application Insights	PyPI 4.1.0 ↗	docs	GitHub 4.1.0 ↗
Resource Management - Arc Data	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗

Name	Package	Docs	Source
Resource Management - Arize AI Observability Eval	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Attestation	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Authorization	PyPI 4.0.0 ↗	docs	GitHub 4.0.0 ↗
Resource Management - Automanage	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Automation	PyPI 1.0.0 ↗ PyPI 1.1.0b4 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b4 ↗
Resource Management - Azure AD B2C	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Azure AI Search	PyPI 9.1.0 ↗ PyPI 9.2.0b3 ↗	docs	GitHub 9.1.0 ↗ GitHub 9.2.0b3 ↗
Resource Management - Azure Stack	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Azure Stack HCI	PyPI 7.0.0 ↗ PyPI 8.0.0b4 ↗	docs	GitHub 7.0.0 ↗ GitHub 8.0.0b4 ↗
Resource Management - Azure VMware Solution	PyPI 9.0.0 ↗	docs	GitHub 9.0.0 ↗
Resource Management - BareMetal Infrastructure	PyPI 1.0.0 ↗ PyPI 1.1.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b2 ↗
Resource Management - Batch	PyPI 18.0.0 ↗	docs	GitHub 18.0.0 ↗
Resource Management - Batch AI	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
Resource Management - Billing	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
Resource Management - Billing Benefits	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Bot Service	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Carbonoptimization	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Change Analysis	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Chaos	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Cognitive Services	PyPI 13.6.0 ↗ PyPI 13.7.0b1 ↗	docs	GitHub 13.6.0 ↗ GitHub 13.7.0b1 ↗
Resource Management - Commerce	PyPI 6.0.0 ↗ PyPI 6.1.0b1 ↗	docs	GitHub 6.0.0 ↗ GitHub 6.1.0b1 ↗

Name	Package	Docs	Source
Resource Management - Communication	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Compute	PyPI 34.1.0 ↗	docs	GitHub 34.1.0 ↗
Resource Management - Compute Fleet	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Compute Schedule	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Confidential Ledger	PyPI 1.0.0 ↗ PyPI 2.0.0b5 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b5 ↗
Resource Management - Confluent	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Connected Cache	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Connected VMware	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Consumption	PyPI 10.0.0 ↗ PyPI 11.0.0b1 ↗	docs	GitHub 10.0.0 ↗ GitHub 11.0.0b1 ↗
Resource Management - Container Apps	PyPI 3.2.0 ↗	docs	GitHub 3.2.0 ↗
Resource Management - Container Instances	PyPI 10.1.0 ↗ PyPI 10.2.0b1 ↗	docs	GitHub 10.1.0 ↗ GitHub 10.2.0b1 ↗
Resource Management - Container Orchestrator Runtime	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Container Registry	PyPI 14.0.0 ↗ PyPI 14.1.0b1 ↗	docs	GitHub 14.0.0 ↗ GitHub 14.1.0b1 ↗
Resource Management - Container Service	PyPI 36.0.0 ↗	docs	GitHub 36.0.0 ↗
Resource Management - Container Service Fleet	PyPI 3.1.0 ↗	docs	GitHub 3.1.0 ↗
Resource Management - Content Delivery Network	PyPI 13.1.1 ↗	docs	GitHub 13.1.1 ↗
Resource Management - Cosmos DB	PyPI 9.8.0 ↗ PyPI 10.0.0b5 ↗	docs	GitHub 9.8.0 ↗ GitHub 10.0.0b5 ↗
Resource Management - Cosmos DB	PyPI 0.1.4 ↗		GitHub 0.1.4 ↗
Resource Management - Cosmos DB for PostgreSQL	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Cost Management	PyPI 4.0.1 ↗	docs	GitHub 4.0.1 ↗
Resource Management - Custom Providers	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Data Box	PyPI 3.1.0 ↗	docs	GitHub 3.1.0 ↗

Name	Package	Docs	Source
Resource Management - Data Box Edge	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Data Factory	PyPI 9.2.0	docs	GitHub 9.2.0
Resource Management - Data Lake Analytics	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Data Lake Store	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
Resource Management - Data Migration	PyPI 10.0.0 PyPI 10.1.0b1	docs	GitHub 10.0.0 GitHub 10.1.0b1
Resource Management - Data Protection	PyPI 1.4.0	docs	GitHub 1.4.0
Resource Management - Data Share	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
Resource Management - Database Watcher	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Databricks	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Datadog	PyPI 2.1.0	docs	GitHub 2.1.0
Resource Management - Defender EASM	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Dependencymap	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Deployment Manager	PyPI 1.0.0 PyPI 2.0.0b1	docs	GitHub 1.0.0 GitHub 2.0.0b1
Resource Management - Desktop Virtualization	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Dev Spaces	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
Resource Management - Device Provisioning Services	PyPI 1.1.0 PyPI 1.2.0b2	docs	GitHub 1.1.0
Resource Management - Device Registry	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Device Update	PyPI 1.1.0	docs	GitHub 1.1.0
Resource Management - DevOps Infrastructure	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - DevTest Labs	PyPI 9.0.0 PyPI 10.0.0b2	docs	GitHub 9.0.0 GitHub 10.0.0b2
Resource Management - Digital Twins	PyPI 7.0.0	docs	GitHub 7.0.0
Resource Management - DNS	PyPI 8.2.0	docs	GitHub 8.2.0

Name	Package	Docs	Source
Resource Management - DNS Resolver	PyPI 1.0.0 ↗ PyPI 1.1.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b2 ↗
Resource Management - Durable Task	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Dynatrace	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Edge Order	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Edge Zones	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Education	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Elastic	PyPI 1.0.0 ↗ PyPI 1.1.0b4 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b4 ↗
Resource Management - Event Grid	PyPI 10.4.0 ↗	docs	GitHub 10.4.0 ↗
Resource Management - Event Hubs	PyPI 11.2.0 ↗	docs	GitHub 11.2.0 ↗
Resource Management - Extended Location	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Fabric	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Fluid Relay	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Front Door	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Resource Management - Graph Services	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Guest Configuration	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - HANA on Azure	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Hardware Security Modules	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - HDInsight	PyPI 9.0.0 ↗ PyPI 9.1.0b1 ↗	docs	GitHub 9.0.0 ↗ GitHub 9.1.0b1 ↗
Resource Management - HDInsight Containers	PyPI 1.0.0b3 ↗	docs	GitHub 1.0.0b3 ↗
Resource Management - Health Bot	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Health Data AI Services	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Healthcare APIs	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Hybrid Compute	PyPI 9.0.0 ↗ PyPI 9.1.0b1 ↗	docs	GitHub 9.0.0 ↗ GitHub 9.1.0b1 ↗

Name	Package	Docs	Source
Resource Management - Hybrid Connectivity	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Hybrid Container Service	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Hybrid Kubernetes	PyPI 1.1.0 ↗ PyPI 1.2.0b2 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b2 ↗
Resource Management - Hybrid Network	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Image Builder	PyPI 1.4.0 ↗	docs	GitHub 1.4.0 ↗
Resource Management - Impact Reporting	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Informatica Data Management	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - IoT Central	PyPI 9.0.0 ↗ PyPI 10.0.0b2 ↗	docs	GitHub 9.0.0 ↗ GitHub 10.0.0b2 ↗
Resource Management - IoT Firmware Defense	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - IoT Hub	PyPI 4.0.0 ↗	docs	GitHub 4.0.0 ↗
Resource Management - IoT Operations	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Key Vault	PyPI 11.0.0 ↗	docs	GitHub 11.0.0 ↗
Resource Management - Kubernetes Configuration	PyPI 3.1.0 ↗	docs	GitHub 3.1.0 ↗
Resource Management - Kubernetesconfiguration-Extensions	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Kubernetesconfiguration-Extensiontypes	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Kubernetesconfiguration-Fluxconfigurations	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Kusto	PyPI 3.4.0 ↗	docs	GitHub 3.4.0 ↗
Resource Management - Lab Services	PyPI 2.0.0 ↗ PyPI 2.1.0b1 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b1 ↗
Resource Management - Lambdatesthyperexecute	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Large Instance	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Load Testing	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Log Analytics	PyPI 12.0.0 ↗ PyPI 13.0.0b7 ↗	docs	GitHub 12.0.0 ↗ GitHub 13.0.0b7 ↗

Name	Package	Docs	Source
Resource Management - Logic Apps	PyPI 10.0.0 ↗ PyPI 10.1.0b1 ↗	docs	GitHub 10.0.0 ↗ GitHub 10.1.0b1 ↗
Resource Management - Logz	PyPI 1.1.1 ↗	docs	GitHub 1.1.1 ↗
Resource Management - Machine Learning Compute	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Machine Learning Services	PyPI 1.0.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Maintenance	PyPI 2.1.0 ↗ PyPI 2.2.0b2 ↗	docs	GitHub 2.1.0 ↗ GitHub 2.2.0b2 ↗
Resource Management - Managed Applications	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Managed Grafana	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Managed Network Fabric	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Managed Service Identity	PyPI 7.0.0 ↗ PyPI 7.1.0b1 ↗	docs	GitHub 7.0.0 ↗ GitHub 7.1.0b1 ↗
Resource Management - Managed Services	PyPI 6.0.0 ↗ PyPI 7.0.0b2 ↗	docs	GitHub 6.0.0 ↗ GitHub 7.0.0b2 ↗
Resource Management - Management Groups	PyPI 1.0.0 ↗ PyPI 1.1.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b2 ↗
Resource Management - Management Partner	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Maps	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Marketplace Ordering	PyPI 1.1.0 ↗ PyPI 1.2.0b2 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b2 ↗
Resource Management - Media Services	PyPI 10.2.1 ↗	docs	GitHub 10.2.1 ↗
Resource Management - Migration Assessment	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Migration Discovery SAP	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Mixed Reality	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Mobile Network	PyPI 3.3.0 ↗	docs	GitHub 3.3.0 ↗
Resource Management - Mongo Cluster	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Mongodbatlas	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗

Name	Package	Docs	Source
Resource Management - Monitor	PyPI 6.0.2 ↗ PyPI 7.0.0b1 ↗	docs	GitHub 6.0.2 ↗ GitHub 7.0.0b1 ↗
Resource Management - MySQL Flexible Servers	PyPI 1.0.0b3 ↗	docs	GitHub 1.0.0b3 ↗
Resource Management - Neon Postgres	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - NetApp Files	PyPI 13.5.0 ↗ PyPI 14.0.0b1 ↗	docs	GitHub 13.5.0 ↗ GitHub 14.0.0b1 ↗
Resource Management - Network	PyPI 29.0.0 ↗	docs	GitHub 29.0.0 ↗
Resource Management - Network Analytics	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Network Function	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Networkcloud	PyPI 2.0.0 ↗ PyPI 2.1.0b1 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b1 ↗
Resource Management - New Relic Observability	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Nginx	PyPI 3.0.0 ↗ PyPI 3.1.0b2 ↗	docs	GitHub 3.0.0 ↗ GitHub 3.1.0b2 ↗
Resource Management - Notification Hubs	PyPI 8.0.0 ↗ PyPI 8.1.0b2 ↗	docs	GitHub 8.0.0 ↗ GitHub 8.1.0b2 ↗
Resource Management - Oep	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Operations Management	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Oracle Database	PyPI 1.0.0 ↗ PyPI 1.0.0.post2 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.0.0.post2 ↗
Resource Management - Orbital	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Palo Alto Networks - Next Generation Firewall	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Peering	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Pinecone Vector DB	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Playwright Testing	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Policy Insights	PyPI 1.0.0 ↗ PyPI 1.1.0b4 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b4 ↗

Name	Package	Docs	Source
Resource Management - Portal	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Portalservicescopilot	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - PostgreSQL	PyPI 10.1.0 ↗ PyPI 10.2.0b18 ↗	docs	GitHub 10.1.0 ↗ GitHub 10.2.0b18 ↗
Resource Management - PostgreSQL Flexible Servers	PyPI 1.1.0 ↗ PyPI 1.2.0b1 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b1 ↗
Resource Management - Power BI Dedicated	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Private DNS	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Resource Management - Purestorageblock	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Purview	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Quantum	PyPI 1.0.0b5 ↗	docs	GitHub 1.0.0b5 ↗
Resource Management - Qumulo	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Quota	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Recovery Services	PyPI 3.1.0 ↗	docs	GitHub 3.1.0 ↗
Resource Management - Recovery Services Backup	PyPI 9.2.0 ↗	docs	GitHub 9.2.0 ↗
Resource Management - Recovery Services Data Replication	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Recovery Services Site Recovery	PyPI 1.3.0 ↗	docs	GitHub 1.3.0 ↗
Resource Management - Red Hat OpenShift	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Redis	PyPI 14.5.0 ↗	docs	GitHub 14.5.0 ↗
Resource Management - Redis Enterprise	PyPI 3.0.0 ↗ PyPI 3.1.0b4 ↗	docs	GitHub 3.0.0 ↗ GitHub 3.1.0b4 ↗
Resource Management - Region Move	PyPI 1.0.0b1 ↗		GitHub 1.0.0b1 ↗
Resource Management - Relay	PyPI 1.1.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.1.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Reservations	PyPI 2.3.0 ↗	docs	GitHub 2.3.0 ↗
Resource Management - Resource Connector	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗

Name	Package	Docs	Source
Resource Management - Resource Graph	PyPI 8.0.0 ↗ PyPI 8.1.0b3 ↗	docs	GitHub 8.0.0 ↗ GitHub 8.1.0b3 ↗
Resource Management - Resource Health	PyPI 1.0.0b6 ↗	docs	GitHub 1.0.0b6 ↗
Resource Management - Resource Mover	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Resources	PyPI 23.4.0 ↗	docs	GitHub 23.4.0 ↗
Resource Management - Resources	PyPI 23.4.0 ↗	docs	GitHub 23.4.0 ↗
Resource Management - Scheduler	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
Resource Management - Scvmm	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Secretsstoreextension	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Security	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
Resource Management - Security Insights	PyPI 1.0.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Self Help	PyPI 1.0.0 ↗ PyPI 2.0.0b4 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b4 ↗
Resource Management - Serial Console	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Server Management	PyPI 2.0.1 ↗		GitHub 2.0.1 ↗
Resource Management - Service Bus	PyPI 9.0.0 ↗	docs	GitHub 9.0.0 ↗
Resource Management - Service Fabric	PyPI 2.1.0 ↗ PyPI 2.2.0b1 ↗	docs	GitHub 2.1.0 ↗ GitHub 2.2.0b1 ↗
Resource Management - Service Fabric Managed Clusters	PyPI 2.0.0 ↗ PyPI 2.1.0b2 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b2 ↗
Resource Management - Service Linker	PyPI 1.1.0 ↗ PyPI 1.2.0b3 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b3 ↗
Resource Management - Service Networking	PyPI 2.0.0 ↗ PyPI 2.1.0b1 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b1 ↗
Resource Management - SignalR	PyPI 1.2.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.2.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Sitemanager	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Sphere	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗

Name	Package	Docs	Source
Resource Management - Spring App Discovery	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - SQL	PyPI 3.0.1	docs	GitHub 3.0.1
	PyPI 4.0.0b21		GitHub 4.0.0b21
Resource Management - SQL Virtual Machine	PyPI 1.0.0b6	docs	GitHub 1.0.0b6
Resource Management - Standby Pool	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Storage	PyPI 23.0.0	docs	GitHub 23.0.0
Resource Management - Storage Actions	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Storage Cache	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Storage Import/Export	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Storage Mover	PyPI 2.1.0	docs	GitHub 2.1.0
Resource Management - Storage Pool	PyPI 1.0.0	docs	GitHub 1.0.0
	PyPI 1.1.0b1		GitHub 1.1.0b1
Resource Management - Storage Sync	PyPI 1.0.0	docs	GitHub 1.0.0
	PyPI 2.0.0b1		GitHub 2.0.0b1
Resource Management - Stream Analytics	PyPI 1.0.0	docs	GitHub 1.0.0
	PyPI 2.0.0b2		GitHub 2.0.0b2
Resource Management - Subscriptions	PyPI 3.1.1	docs	GitHub 3.1.1
	PyPI 3.2.0b1		GitHub 3.2.0b1
Resource Management - Support	PyPI 7.0.0	docs	GitHub 7.0.0
Resource Management - Synapse	PyPI 2.0.0	docs	GitHub 2.0.0
	PyPI 2.1.0b7		GitHub 2.1.0b7
Resource Management - Terraform	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Test Base	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Time Series Insights	PyPI 1.0.0	docs	GitHub 1.0.0
	PyPI 2.0.0b1		GitHub 2.0.0b1
Resource Management - Traffic Manager	PyPI 1.1.0	docs	GitHub 1.1.0
Resource Management - Trusted Signing	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - VMware Solution by CloudSimple	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Voice Services	PyPI 1.0.0	docs	GitHub 1.0.0

Name	Package	Docs	Source
Resource Management - Web PubSub	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Weights & Biases	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Workload Monitor	PyPI 1.0.0b4	docs	GitHub 1.0.0b4
Resource Management - Workloads	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Workloads SAP Virtual Instance	PyPI 1.0.0	docs	GitHub 1.0.0

All libraries

[\[\]](#) [Expand table](#)

Name	Package	Docs	Source
AI Agents	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
AI Evaluation	PyPI 1.8.0	docs	GitHub 1.8.0
AI Model Inference	PyPI 1.0.0b9	docs	GitHub 1.0.0b9
AI Projects	PyPI 1.0.0b11	docs	GitHub 1.0.0b11
Anomaly Detector	PyPI 3.0.0b6	docs	GitHub 3.0.0b6
App Configuration	PyPI 1.7.1	docs	GitHub 1.7.1
App Configuration Provider	PyPI 2.1.0	docs	GitHub 2.1.0
Attestation	PyPI 1.0.0	docs	GitHub 1.0.0
Azure AI Search	PyPI 11.4.0 PyPI 11.6.0b12	docs	GitHub 11.4.0 GitHub 11.6.0b12
Azure Blob Storage Checkpoint Store	PyPI 1.2.0	docs	GitHub 1.2.0
Azure Blob Storage Checkpoint Store AIO	PyPI 1.2.0	docs	GitHub 1.2.0
Azure Monitor OpenTelemetry	PyPI 1.6.10	docs	GitHub 1.6.10
Azure Remote Rendering	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Communication Call Automation	PyPI 1.3.0 PyPI 1.4.0b1	docs	GitHub 1.3.0 GitHub 1.4.0b1
Communication Chat	PyPI 1.3.0	docs	GitHub 1.3.0

Name	Package	Docs	Source
Communication Email	PyPI 1.0.0 ↗ PyPI 1.0.1b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.0.1b1 ↗
Communication Identity	PyPI 1.5.0 ↗	docs	GitHub 1.5.0 ↗
Communication JobRouter	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Communication Messages	PyPI 1.1.0 ↗ PyPI 1.2.0b1 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b1 ↗
Communication Phone Numbers	PyPI 1.2.0 ↗ PyPI 1.3.0b1 ↗	docs	GitHub 1.2.0 ↗ GitHub 1.3.0b1 ↗
Communication Rooms	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Communication Sms	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Confidential Ledger	PyPI 1.1.1 ↗ PyPI 1.2.0b1 ↗	docs	GitHub 1.1.1 ↗ GitHub 1.2.0b1 ↗
Container Registry	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Content Safety	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Conversational Language Understanding	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Core - Client - Core	PyPI 1.34.0 ↗	docs	GitHub 1.34.0 ↗
Core - Client - Core HTTP	PyPI 1.0.0b6 ↗	docs	GitHub 1.0.0b6 ↗
Core - Client - Experimental	PyPI 1.0.0b4 ↗	docs	GitHub 1.0.0b4 ↗
Core - Client - Tracing Opentelemetry	PyPI 1.0.0b12 ↗	docs	GitHub 1.0.0b12 ↗
Core Tracing Opencensus	PyPI 1.0.0b10 ↗	docs	GitHub 1.0.0b10 ↗
Cosmos DB	PyPI 4.9.0 ↗ PyPI 4.12.0b1 ↗	docs	GitHub 4.9.0 ↗ GitHub 4.12.0b1 ↗
Defender EASM	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Dev Center	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Device Update	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Digital Twins	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Document Intelligence	PyPI 1.0.2 ↗	docs	GitHub 1.0.2 ↗
Document Translation	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗

Name	Package	Docs	Source
Event Grid	PyPI 4.22.0	docs	GitHub 4.22.0
Event Hubs	PyPI 5.15.0	docs	GitHub 5.15.0
Face	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
FarmBeats	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Form Recognizer	PyPI 3.3.3	docs	GitHub 3.3.3
Health Deidentification	PyPI 1.0.0	docs	GitHub 1.0.0
Health Insights Cancer Profiling	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Health Insights Clinical Matching	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Health Insights Radiology Insights	PyPI 1.0.0	docs	GitHub 1.0.0
Identity	PyPI 1.23.0	docs	GitHub 1.23.0
Identity Broker	PyPI 1.2.0 PyPI 1.3.0b1	docs	GitHub 1.2.0 GitHub 1.3.0b1
Image Analysis	PyPI 1.0.0	docs	GitHub 1.0.0
Key Vault - Administration	PyPI 4.5.0 PyPI 4.6.0b1	docs	GitHub 4.5.0 GitHub 4.6.0b1
Key Vault - Certificates	PyPI 4.9.0 PyPI 4.10.0b1	docs	GitHub 4.9.0 GitHub 4.10.0b1
Key Vault - Keys	PyPI 4.10.0 PyPI 4.11.0b1	docs	GitHub 4.10.0 GitHub 4.11.0b1
Key Vault - Secrets	PyPI 4.9.0 PyPI 4.10.0b1	docs	GitHub 4.9.0 GitHub 4.10.0b1
Key Vault - Security Domain	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Load Testing	PyPI 1.0.1 PyPI 1.1.0b1	docs	GitHub 1.0.1 GitHub 1.1.0b1
Machine Learning	PyPI 1.27.1	docs	GitHub 1.27.1
Machine Learning - Feature Store	PyPI 1.0.1		GitHub 1.0.1
Managed Private Endpoints	PyPI 0.4.0	docs	GitHub 0.4.0
Maps Geolocation	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
Maps Render	PyPI 2.0.0b2	docs	GitHub 2.0.0b2

Name	Package	Docs	Source
Maps Route	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
Maps Search	PyPI 2.0.0b2	docs	GitHub 2.0.0b2
Media Analytics Edge	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Mixed Reality Authentication	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Monitor Ingestion	PyPI 1.0.4	docs	GitHub 1.0.4
Monitor Query	PyPI 1.4.1	docs	GitHub 1.4.1
OpenTelemetry Exporter	PyPI 1.0.0b37	docs	GitHub 1.0.0b37
Personalizer	PyPI 1.0.0b1		GitHub 1.0.0b1
Purview Account	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Purview Administration	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Purview Data Map	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Purview Scanning	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Purview Sharing	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
Purview Workflow	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Question Answering	PyPI 1.1.0	docs	GitHub 1.1.0
Schema Registry	PyPI 1.3.0	docs	GitHub 1.3.0
Schema Registry - Avro	PyPI 1.0.0	docs	GitHub 1.0.0
Service Bus	PyPI 7.14.2	docs	GitHub 7.14.2
Spark	PyPI 0.7.0	docs	GitHub 0.7.0
Storage - Blobs	PyPI 12.25.1 PyPI 12.26.0b1	docs	GitHub 12.25.1 GitHub 12.26.0b1
Storage - Blobs Changefeed	PyPI 12.0.0b5	docs	GitHub 12.0.0b5
Storage - Files Data Lake	PyPI 12.20.0 PyPI 12.21.0b1	docs	GitHub 12.20.0 GitHub 12.21.0b1
Storage - Files Share	PyPI 12.21.0 PyPI 12.22.0b1	docs	GitHub 12.21.0 GitHub 12.22.0b1
Storage - Queues	PyPI 12.12.0 PyPI 12.13.0b1	docs	GitHub 12.12.0 GitHub 12.13.0b1

Name	Package	Docs	Source
Synapse - AccessControl	PyPI 0.7.0 ↗	docs	GitHub 0.7.0 ↗
Synapse - Artifacts	PyPI 0.20.0 ↗	docs	GitHub 0.20.0 ↗
Synapse - Monitoring	PyPI 0.2.0 ↗	docs	GitHub 0.2.0 ↗
Tables	PyPI 12.7.0 ↗	docs	GitHub 12.7.0 ↗
Text Analytics	PyPI 5.3.0 ↗	docs	GitHub 5.3.0 ↗
Text Translation	PyPI 1.0.1 ↗	docs	GitHub 1.0.1 ↗
TimeZones	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Web PubSub	PyPI 1.2.2 ↗	docs	GitHub 1.2.2 ↗
Web PubSub Client	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Core - Management - Core	PyPI 1.5.0 ↗	docs	GitHub 1.5.0 ↗
Resource Management - Astro	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Dev Center	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Elastic SAN	PyPI 1.1.0 ↗ PyPI 1.2.0b2 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b2 ↗
Resource Management - Security DevOps	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Advisor	PyPI 9.0.0 ↗ PyPI 10.0.0b1 ↗	docs	GitHub 9.0.0 ↗ GitHub 10.0.0b1 ↗
Resource Management - Agrifood	PyPI 1.0.0b3 ↗	docs	GitHub 1.0.0b3 ↗
Resource Management - AKS Developer Hub	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Alerts Management	PyPI 1.0.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - API Center	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - API Management	PyPI 5.0.0 ↗	docs	GitHub 5.0.0 ↗
Resource Management - App Compliance Automation	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - App Configuration	PyPI 4.0.0 ↗	docs	GitHub 4.0.0 ↗
Resource Management - App Platform	PyPI 10.0.1 ↗	docs	GitHub 10.0.1 ↗
Resource Management - App Service	PyPI 8.0.0 ↗	docs	GitHub 8.0.0 ↗

Name	Package	Docs	Source
Resource Management - Application Insights	PyPI 4.1.0 ↗	docs	GitHub 4.1.0 ↗
Resource Management - Arc Data	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
	PyPI 2.0.0b1 ↗		GitHub 2.0.0b1 ↗
Resource Management - Arize AI Observability Eval	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Attestation	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
	PyPI 2.0.0b1 ↗		GitHub 2.0.0b1 ↗
Resource Management - Authorization	PyPI 4.0.0 ↗	docs	GitHub 4.0.0 ↗
Resource Management - Automanage	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
	PyPI 2.0.0b1 ↗		GitHub 2.0.0b1 ↗
Resource Management - Automation	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
	PyPI 1.1.0b4 ↗		GitHub 1.1.0b4 ↗
Resource Management - Azure AI Search	PyPI 9.1.0 ↗	docs	GitHub 9.1.0 ↗
	PyPI 9.2.0b3 ↗		GitHub 9.2.0b3 ↗
Resource Management - Azure Stack	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
	PyPI 2.0.0b1 ↗		GitHub 2.0.0b1 ↗
Resource Management - Azure Stack HCI	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
	PyPI 8.0.0b4 ↗		GitHub 8.0.0b4 ↗
Resource Management - Azure VMware Solution	PyPI 9.0.0 ↗	docs	GitHub 9.0.0 ↗
Resource Management - BareMetal Infrastructure	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
	PyPI 1.1.0b2 ↗		GitHub 1.1.0b2 ↗
Resource Management - Batch	PyPI 18.0.0 ↗	docs	GitHub 18.0.0 ↗
Resource Management - Billing	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
Resource Management - Billing Benefits	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Bot Service	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Carbonoptimization	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Change Analysis	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Chaos	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Cognitive Services	PyPI 13.6.0 ↗	docs	GitHub 13.6.0 ↗
	PyPI 13.7.0b1 ↗		GitHub 13.7.0b1 ↗
Resource Management - Commerce	PyPI 6.0.0 ↗	docs	GitHub 6.0.0 ↗
	PyPI 6.1.0b1 ↗		GitHub 6.1.0b1 ↗

Name	Package	Docs	Source
Resource Management - Communication	PyPI 2.1.0	docs	GitHub 2.1.0
Resource Management - Compute	PyPI 34.1.0	docs	GitHub 34.1.0
Resource Management - Compute Fleet	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Compute Schedule	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Confidential Ledger	PyPI 1.0.0 PyPI 2.0.0b5	docs	GitHub 1.0.0 GitHub 2.0.0b5
Resource Management - Confluent	PyPI 2.1.0	docs	GitHub 2.1.0
Resource Management - Connected Cache	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Connected VMware	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Consumption	PyPI 10.0.0 PyPI 11.0.0b1	docs	GitHub 10.0.0 GitHub 11.0.0b1
Resource Management - Container Apps	PyPI 3.2.0	docs	GitHub 3.2.0
Resource Management - Container Instances	PyPI 10.1.0 PyPI 10.2.0b1	docs	GitHub 10.1.0 GitHub 10.2.0b1
Resource Management - Container Orchestrator Runtime	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Container Registry	PyPI 14.0.0 PyPI 14.1.0b1	docs	GitHub 14.0.0 GitHub 14.1.0b1
Resource Management - Container Service	PyPI 36.0.0	docs	GitHub 36.0.0
Resource Management - Container Service Fleet	PyPI 3.1.0	docs	GitHub 3.1.0
Resource Management - Content Delivery Network	PyPI 13.1.1	docs	GitHub 13.1.1
Resource Management - Cosmos DB	PyPI 9.8.0 PyPI 10.0.0b5	docs	GitHub 9.8.0 GitHub 10.0.0b5
Resource Management - Cosmos DB for PostgreSQL	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
Resource Management - Cost Management	PyPI 4.0.1	docs	GitHub 4.0.1
Resource Management - Custom Providers	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
Resource Management - Data Box	PyPI 3.1.0	docs	GitHub 3.1.0
Resource Management - Data Box Edge	PyPI 2.0.0	docs	GitHub 2.0.0

Name	Package	Docs	Source
Resource Management - Data Factory	PyPI 9.2.0 ↗	docs	GitHub 9.2.0 ↗
Resource Management - Data Lake Analytics	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Data Lake Store	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Data Migration	PyPI 10.0.0 ↗ PyPI 10.1.0b1 ↗	docs	GitHub 10.0.0 ↗ GitHub 10.1.0b1 ↗
Resource Management - Data Protection	PyPI 1.4.0 ↗	docs	GitHub 1.4.0 ↗
Resource Management - Data Share	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Database Watcher	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Databricks	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Datadog	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Defender EASM	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Dependencymap	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Deployment Manager	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Desktop Virtualization	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Dev Spaces	PyPI 1.0.0b3 ↗	docs	GitHub 1.0.0b3 ↗
Resource Management - Device Provisioning Services	PyPI 1.1.0 ↗ PyPI 1.2.0b2 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Device Registry	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Device Update	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - DevOps Infrastructure	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - DevTest Labs	PyPI 9.0.0 ↗ PyPI 10.0.0b2 ↗	docs	GitHub 9.0.0 ↗ GitHub 10.0.0b2 ↗
Resource Management - Digital Twins	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
Resource Management - DNS	PyPI 8.2.0 ↗	docs	GitHub 8.2.0 ↗
Resource Management - DNS Resolver	PyPI 1.0.0 ↗ PyPI 1.1.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b2 ↗

Name	Package	Docs	Source
Resource Management - Durable Task	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Dynatrace	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Edge Order	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Edge Zones	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Education	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Elastic	PyPI 1.0.0 ↗ PyPI 1.1.0b4 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b4 ↗
Resource Management - Event Grid	PyPI 10.4.0 ↗	docs	GitHub 10.4.0 ↗
Resource Management - Event Hubs	PyPI 11.2.0 ↗	docs	GitHub 11.2.0 ↗
Resource Management - Extended Location	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Fabric	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Fluid Relay	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Front Door	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Resource Management - Graph Services	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Guest Configuration	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - HANA on Azure	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Hardware Security Modules	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - HDInsight	PyPI 9.0.0 ↗ PyPI 9.1.0b1 ↗	docs	GitHub 9.0.0 ↗ GitHub 9.1.0b1 ↗
Resource Management - HDInsight Containers	PyPI 1.0.0b3 ↗	docs	GitHub 1.0.0b3 ↗
Resource Management - Health Bot	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Health Data AI Services	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Healthcare APIs	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Hybrid Compute	PyPI 9.0.0 ↗ PyPI 9.1.0b1 ↗	docs	GitHub 9.0.0 ↗ GitHub 9.1.0b1 ↗
Resource Management - Hybrid Connectivity	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗

Name	Package	Docs	Source
Resource Management - Hybrid Container Service	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Hybrid Kubernetes	PyPI 1.1.0	docs	GitHub 1.1.0
	PyPI 1.2.0b2		GitHub 1.2.0b2
Resource Management - Hybrid Network	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Image Builder	PyPI 1.4.0	docs	GitHub 1.4.0
Resource Management - Impact Reporting	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Informatica Data Management	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - IoT Central	PyPI 9.0.0	docs	GitHub 9.0.0
	PyPI 10.0.0b2		GitHub 10.0.0b2
Resource Management - IoT Firmware Defense	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - IoT Hub	PyPI 4.0.0	docs	GitHub 4.0.0
Resource Management - IoT Operations	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Key Vault	PyPI 11.0.0	docs	GitHub 11.0.0
Resource Management - Kubernetes Configuration	PyPI 3.1.0	docs	GitHub 3.1.0
Resource Management - Kubernetesconfiguration-Extensions	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Kubernetesconfiguration-Extensiontypes	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Kubernetesconfiguration-Fluxconfigurations	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Kusto	PyPI 3.4.0	docs	GitHub 3.4.0
Resource Management - Lab Services	PyPI 2.0.0	docs	GitHub 2.0.0
	PyPI 2.1.0b1		GitHub 2.1.0b1
Resource Management - Lambdatesthyperexecute	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Large Instance	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Load Testing	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Log Analytics	PyPI 12.0.0	docs	GitHub 12.0.0
	PyPI 13.0.0b7		GitHub 13.0.0b7
Resource Management - Logic Apps	PyPI 10.0.0	docs	GitHub 10.0.0
	PyPI 10.1.0b1		GitHub 10.1.0b1

Name	Package	Docs	Source
Resource Management - Machine Learning Compute	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Machine Learning Services	PyPI 1.0.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Maintenance	PyPI 2.1.0 ↗ PyPI 2.2.0b2 ↗	docs	GitHub 2.1.0 ↗ GitHub 2.2.0b2 ↗
Resource Management - Managed Applications	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Managed Grafana	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Managed Network Fabric	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Managed Service Identity	PyPI 7.0.0 ↗ PyPI 7.1.0b1 ↗	docs	GitHub 7.0.0 ↗ GitHub 7.1.0b1 ↗
Resource Management - Managed Services	PyPI 6.0.0 ↗ PyPI 7.0.0b2 ↗	docs	GitHub 6.0.0 ↗ GitHub 7.0.0b2 ↗
Resource Management - Management Groups	PyPI 1.0.0 ↗ PyPI 1.1.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b2 ↗
Resource Management - Management Partner	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Maps	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Marketplace Ordering	PyPI 1.1.0 ↗ PyPI 1.2.0b2 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b2 ↗
Resource Management - Media Services	PyPI 10.2.1 ↗	docs	GitHub 10.2.1 ↗
Resource Management - Migration Assessment	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Migration Discovery SAP	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Mixed Reality	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Mobile Network	PyPI 3.3.0 ↗	docs	GitHub 3.3.0 ↗
Resource Management - Mongo Cluster	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Mongodbatlas	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Monitor	PyPI 6.0.2 ↗ PyPI 7.0.0b1 ↗	docs	GitHub 6.0.2 ↗ GitHub 7.0.0b1 ↗
Resource Management - MySQL Flexible Servers	PyPI 1.0.0b3 ↗	docs	GitHub 1.0.0b3 ↗

Name	Package	Docs	Source
Resource Management - Neon Postgres	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - NetApp Files	PyPI 13.5.0 PyPI 14.0.0b1	docs	GitHub 13.5.0 GitHub 14.0.0b1
Resource Management - Network	PyPI 29.0.0	docs	GitHub 29.0.0
Resource Management - Network Function	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Networkcloud	PyPI 2.0.0 PyPI 2.1.0b1	docs	GitHub 2.0.0 GitHub 2.1.0b1
Resource Management - New Relic Observability	PyPI 1.1.0	docs	GitHub 1.1.0
Resource Management - Nginx	PyPI 3.0.0 PyPI 3.1.0b2	docs	GitHub 3.0.0 GitHub 3.1.0b2
Resource Management - Notification Hubs	PyPI 8.0.0 PyPI 8.1.0b2	docs	GitHub 8.0.0 GitHub 8.1.0b2
Resource Management - Oep	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Operations Management	PyPI 1.0.0 PyPI 2.0.0b1	docs	GitHub 1.0.0 GitHub 2.0.0b1
Resource Management - Oracle Database	PyPI 1.0.0 PyPI 1.0.0.post2	docs	GitHub 1.0.0 GitHub 1.0.0.post2
Resource Management - Orbital	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Palo Alto Networks - Next Generation Firewall	PyPI 1.0.0 PyPI 2.0.0b1	docs	GitHub 1.0.0 GitHub 2.0.0b1
Resource Management - Peering	PyPI 1.0.0 PyPI 2.0.0b1	docs	GitHub 1.0.0 GitHub 2.0.0b1
Resource Management - Pinecone Vector DB	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Playwright Testing	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Policy Insights	PyPI 1.0.0 PyPI 1.1.0b4	docs	GitHub 1.0.0 GitHub 1.1.0b4
Resource Management - Portal	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
Resource Management - Portalservicescopilot	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - PostgreSQL	PyPI 10.1.0 PyPI 10.2.0b18	docs	GitHub 10.1.0 GitHub 10.2.0b18

Name	Package	Docs	Source
Resource Management - PostgreSQL Flexible Servers	PyPI 1.1.0 ↗ PyPI 1.2.0b1 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b1 ↗
Resource Management - Power BI Dedicated	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Private DNS	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Resource Management - Purestorageblock	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Purview	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Quantum	PyPI 1.0.0b5 ↗	docs	GitHub 1.0.0b5 ↗
Resource Management - Qumulo	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Quota	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Recovery Services	PyPI 3.1.0 ↗	docs	GitHub 3.1.0 ↗
Resource Management - Recovery Services Backup	PyPI 9.2.0 ↗	docs	GitHub 9.2.0 ↗
Resource Management - Recovery Services Data Replication	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Recovery Services Site Recovery	PyPI 1.3.0 ↗	docs	GitHub 1.3.0 ↗
Resource Management - Red Hat OpenShift	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Redis	PyPI 14.5.0 ↗	docs	GitHub 14.5.0 ↗
Resource Management - Redis Enterprise	PyPI 3.0.0 ↗ PyPI 3.1.0b4 ↗	docs	GitHub 3.0.0 ↗ GitHub 3.1.0b4 ↗
Resource Management - Relay	PyPI 1.1.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.1.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Reservations	PyPI 2.3.0 ↗	docs	GitHub 2.3.0 ↗
Resource Management - Resource Connector	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Resource Graph	PyPI 8.0.0 ↗ PyPI 8.1.0b3 ↗	docs	GitHub 8.0.0 ↗ GitHub 8.1.0b3 ↗
Resource Management - Resource Health	PyPI 1.0.0b6 ↗	docs	GitHub 1.0.0b6 ↗
Resource Management - Resource Mover	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Resources	PyPI 23.4.0 ↗	docs	GitHub 23.4.0 ↗

Name	Package	Docs	Source
Resource Management - Resources	PyPI 23.4.0 ↗	docs	GitHub 23.4.0 ↗
Resource Management - Scvmm	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Secretsstoreextension	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Security	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
Resource Management - Security Insights	PyPI 1.0.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Self Help	PyPI 1.0.0 ↗ PyPI 2.0.0b4 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b4 ↗
Resource Management - Serial Console	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Service Bus	PyPI 9.0.0 ↗	docs	GitHub 9.0.0 ↗
Resource Management - Service Fabric	PyPI 2.1.0 ↗ PyPI 2.2.0b1 ↗	docs	GitHub 2.1.0 ↗ GitHub 2.2.0b1 ↗
Resource Management - Service Fabric Managed Clusters	PyPI 2.0.0 ↗ PyPI 2.1.0b2 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b2 ↗
Resource Management - Service Linker	PyPI 1.1.0 ↗ PyPI 1.2.0b3 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b3 ↗
Resource Management - Service Networking	PyPI 2.0.0 ↗ PyPI 2.1.0b1 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b1 ↗
Resource Management - SignalR	PyPI 1.2.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.2.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Sitemanager	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Sphere	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Spring App Discovery	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - SQL	PyPI 3.0.1 ↗ PyPI 4.0.0b21 ↗	docs	GitHub 3.0.1 ↗ GitHub 4.0.0b21 ↗
Resource Management - SQL Virtual Machine	PyPI 1.0.0b6 ↗	docs	GitHub 1.0.0b6 ↗
Resource Management - Standby Pool	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Storage	PyPI 23.0.0 ↗	docs	GitHub 23.0.0 ↗
Resource Management - Storage Actions	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗

Name	Package	Docs	Source
Resource Management - Storage Cache	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Storage Mover	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Storage Pool	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Storage Sync	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Stream Analytics	PyPI 1.0.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Subscriptions	PyPI 3.1.1 ↗ PyPI 3.2.0b1 ↗	docs	GitHub 3.1.1 ↗ GitHub 3.2.0b1 ↗
Resource Management - Support	PyPI 7.0.0 ↗	docs	GitHub 7.0.0 ↗
Resource Management - Synapse	PyPI 2.0.0 ↗ PyPI 2.1.0b7 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b7 ↗
Resource Management - Terraform	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Time Series Insights	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Traffic Manager	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Trusted Signing	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - VMware Solution by CloudSimple	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Voice Services	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Web PubSub	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Weights & Biases	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Workloads	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Workloads SAP Virtual Instance	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
azure-communication-administration	PyPI 1.0.0b4 ↗		
azureml-fsspec	PyPI 1.0.0 ↗		
Batch	PyPI 14.2.0 ↗ PyPI 15.0.0b2 ↗	docs	GitHub 14.2.0 ↗ GitHub 15.0.0b2 ↗
Core - Client - Tracing Opencensus	PyPI 1.0.0b10 ↗	docs	GitHub 1.0.0b10 ↗

Name	Package	Docs	Source
Device Provisioning Services	PyPI 1.0.0b1 ↗		
Device Provisioning Services	PyPI 1.2.0 ↗		
Ink Recognizer	PyPI 1.0.0b1 ↗		GitHub 1.0.0b1 ↗
IoT Device	PyPI 2.12.0 ↗ PyPI 3.0.0b2 ↗		
IoT Hub	PyPI 2.6.1 ↗		
iotedgedev	PyPI 3.3.7 ↗		
iotedgehubdev	PyPI 0.14.18 ↗		
Key Vault	PyPI 4.2.0 ↗		GitHub 4.2.0 ↗
Kusto Data	PyPI 2.0.0 ↗		
Machine Learning	PyPI 1.2.0 ↗		
Machine Learning - Table	PyPI 1.3.0 ↗		
Machine Learning Monitoring	PyPI 0.1.0a3 ↗		
Personalizer	PyPI 0.1.0 ↗		GitHub 0.1.0 ↗
Purview Administration	PyPI 1.0.0b1 ↗		GitHub 1.0.0b1 ↗
Service Fabric	PyPI 8.2.0.0 ↗	docs	GitHub 8.2.0.0 ↗
Speech	PyPI 1.14.0 ↗		
Storage	PyPI 0.37.0 ↗		GitHub 0.37.0 ↗
Storage - Files Data Lake	PyPI 0.0.51 ↗		
Text Analytics	PyPI 1.0.2 ↗		
Uamqp	PyPI 1.6.11 ↗		GitHub 1.6.11 ↗
azure-agrifood-nspkg	PyPI 1.0.0 ↗		
azure-ai-language-nspkg	PyPI 1.0.0 ↗		
azure-ai-translation-nspkg	PyPI 1.0.0 ↗		
azure-iot-nspkg	PyPI 1.0.1 ↗		
azure-media-nspkg	PyPI 1.0.0 ↗		
azure-messaging-nspkg	PyPI 1.0.0 ↗		

Name	Package	Docs	Source
azure-mixedreality-nspkg	PyPI 1.0.0 ↗		
azure-monitor-nspkg	PyPI 1.0.0 ↗		
azure-purview-nspkg	PyPI 2.0.0 ↗		
azure-security-nspkg	PyPI 1.0.0 ↗		
Cognitive Services Knowledge Namespace Package	PyPI 3.0.0 ↗		GitHub 3.0.0 ↗
Cognitive Services Language Namespace Package	PyPI 3.0.1 ↗		GitHub 3.0.1 ↗
Cognitive Services Namespace Package	PyPI 3.0.1 ↗		GitHub 3.0.1 ↗
Cognitive Services Search Namespace Package	PyPI 3.0.1 ↗		GitHub 3.0.1 ↗
Cognitive Services Vision Namespace Package	PyPI 3.0.1 ↗		GitHub 3.0.1 ↗
Communication Namespace Package	PyPI 0.0.0b1 ↗	docs	
Core Namespace Package	PyPI 3.0.2 ↗		GitHub 3.0.2 ↗
Data Namespace Package	PyPI 1.0.0 ↗	docs	
Digital Twins Namespace Package	PyPI 1.0.0 ↗		
Key Vault Namespace Package	PyPI 1.0.0 ↗		GitHub 1.0.0 ↗
Search Namespace Package	PyPI 1.0.0 ↗		GitHub 1.0.0 ↗
Storage Namespace Package	PyPI 3.1.0 ↗		GitHub 3.1.0 ↗
Synapse Namespace Package	PyPI 1.0.0 ↗		GitHub 1.0.0 ↗
Text Analytics Namespace Package	PyPI 1.0.0 ↗		GitHub 1.0.0 ↗
Resource Management	PyPI 5.0.0 ↗		GitHub 5.0.0 ↗
Resource Management - Common	PyPI 0.20.0 ↗		
apiview-stub-generator	PyPI 0.3.7 ↗		
azure-pylint-guidelines-checker	PyPI 0.0.8 ↗		
Dev Tools	PyPI 1.2.0 ↗		GitHub 1.2.0 ↗
Doc Warden	PyPI 0.7.2 ↗		GitHub 0.7.2 ↗
Tox Monorepo	PyPI 0.1.2 ↗		GitHub 0.1.2 ↗

Reference

Services

Advisor	Container Instances	Fabric
Alerts Management	Container Registry	Fluid Relay
API Center	Container Service	Front Door
API Management	Container Service Fleet	Functions
App Compliance Automation	Content Delivery Network	Grafana
App Configuration	Cosmos DB	Graph Services
App Service	Cosmos DB for PostgreSQL	HANA on Azure
Application Insights	Cost Management	Hardware Security Modules
Arc Data	Custom Providers	HDInsight
Arize AI	Data Box	Health Data AI Services
Attestation	Data Box Edge	Health Deidentification
Authorization	Data Explorer	Healthcare APIs
Automanage	Data Factory	Hybrid Compute
Automation	Data Protection	Hybrid Connectivity
Azure Stack	Data Share	Hybrid Container Service
Azure Stack HCI	Database Migration Service	Hybrid Kubernetes
Azure VMware Solution	Databricks	Hybrid Network
BareMetal Infrastructure	Datadog	Identity
Batch	Deployment Manager	Image Builder
Billing	Desktop Virtualization	Image Search
Bot Service	Dev Center	Informatica Data
Carbonoptimization	Device Registry	Management
Change Analysis	DevOps Infrastructure	IoT
Chaos	DevTest Labs	Key Vault
Cognitive Services	DNS	Kubernetes Configuration
Commerce	DNS Resolver	Lab Services
Communication	Dynatrace	Lambdatesthyperexecute
Compute	Edge Order	Load Testing
Compute Fleet	Edge Zones	Log Analytics
Compute Schedule	Elastic	Logic Apps
Confidential Ledger	Elastic SAN	Machine Learning
Confluent	Entity Search	Maintenance
Connected VMware	Event Grid	Managed Network Fabric
Consumption	Event Hubs	Managed Service Identity
Container Apps	Extended Location	Managed Services

Management Groups	Policy Insights	Self Help
Management Partner	Portal	Serial Console
Maps	PostgreSQL	Service Bus
Marketplace Ordering	PostgreSQL Flexible Servers	Service Fabric
Mixed Reality	Power BI Dedicated	Service Linker
Mobile Network	Private DNS	Service Networking
Mongo Cluster	Purestorageblock	Sphere
Mongodbatlas	Purview	SQL
Monitor	Qumulo	Standby Pool
Neon Postgres	Quota	Storage
NetApp Files	Recovery Services	Stream Analytics
Network	Red Hat OpenShift (ARO)	Subscriptions
New Relic Observability	Redis	Support
Nginx	Relay	Synapse
Notification Hubs	Resource Connector	Tables
Operations Management	Resource Graph	Traffic Manager
Operator Nexus - Network	Resource Mover	Video Search
Cloud	Resources	Voice Services
Oracle Database	Schema Registry	Web PubSub
Orbital	Scvmm	Web Search
Palo Alto Networks	Search	Workloads
Peering	Security	Other
Playwright Testing	Security Insights	

Hosting Python apps on Azure

05/21/2025

Azure offers several options for hosting your application, each suited to different levels of control and responsibility. For an overview of these options, see [Hosting applications on Azure](#).

In general, selecting a hosting option involves balancing control with management responsibility. The more control you require over the infrastructure, the more responsibility you take on for management of one or more resources.

We recommend starting with Azure App Service, which provides a highly managed environment with minimal administrative overhead. As your needs evolve, you can explore other options that offer increased flexibility and control, such as Azure Container Apps, Azure Kubernetes Service (AKS), or ultimately Azure Virtual Machines, which provide the greatest control but also require the most maintenance.

The hosting options in this article are presented in order from more managed (less responsibility on your part) to less managed (more control and responsibility).

- **Web app hosting with Azure App Service:**
 - [Quickstart: Deploy a Python \(Django or Flask\) web app to Azure App Service](#)
 - [Deploy a Python \(Django or Flask\) web app with PostgreSQL in Azure](#)
 - [Create and deploy a Flask web app to Azure with a system-assigned managed identity](#)
 - [Configure a Python app for Azure App Service](#)
- **Content delivery network with Azure Static web apps**
 - [Static website hosting in Azure Storage](#)
 - [Quickstart: Building your first static site with Azure Static Web Apps](#)
- **Serverless hosting with Azure Functions:**
 - [Quickstart: Create a Python function in Azure from the command line](#)
 - [Quickstart: Create a function in Azure with Python using Visual Studio Code](#)
 - [Connect Azure Functions to Azure Storage using command line tools](#)
 - [Connect Azure Functions to Azure Storage using Visual Studio Code](#)
- **Container hosting with Azure:**
 - [Overview of Python Container Apps in Azure](#)
 - [Deploy a container to App Service](#)
 - [Deploy a container to Azure Container Apps](#)
 - [Quickstart: Deploy an Azure Kubernetes Service cluster using the Azure CLI](#)
 - [Deploy a container in Azure Container Instances using the Azure CLI](#)

- Create your first Service Fabric container application on Linux
- Compute intensive and long running operations with Azure Batch:
 - Use Python to create and run an Azure Batch job
 - Tutorial: Run a parallel file processing workload with Azure Batch using Python
 - Tutorial: Run Python scripts through Azure Data Factory using Azure Batch
- On-demand, scalable computing resources with Azure Virtual Machines:
 - Quickstart: Use the Azure CLI to deploy a Linux virtual machine (VM) in Azure

Data solutions for Python apps on Azure

06/05/2025

Azure offers a wide range of fully managed database and storage solutions, including relational, NoSQL, and in-memory databases, with support for both proprietary and open-source technologies. You can also choose from object, block, and file storage services. The following articles can help you get started using these options with Python on Azure.

Databases

- **PostgreSQL:** Build scalable, secure, and fully managed enterprise apps using open-source PostgreSQL. You can scale single-node PostgreSQL for high performance or migrate existing PostgreSQL and Oracle workloads to the cloud.
 - [Quickstart: Use Python to connect and query data in Azure Database for PostgreSQL - Flexible Server](#)
 - [Quickstart: Use Python to connect and query data in Azure Database for PostgreSQL - Single Server](#)
 - [Deploy a Python \(Django or Flask\) web app with PostgreSQL in Azure App Service](#)
- **MySQL:** Build scalable applications using a fully managed, intelligent MySQL database in the cloud.
 - [Quickstart: Use Python to connect and query data in Azure Database for MySQL - Flexible Server](#)
 - [Quickstart: Use Python to connect and query data in Azure Database for MySQL](#)
- **Azure SQL:** Build scalable applications with a fully managed and intelligent SQL database platform in the cloud.
 - [Quickstart: Use Python to query a database in Azure SQL Database or Azure SQL Managed Instance](#)

NoSQL, blobs, tables, files, graphs, and caches

- **Cosmos DB:** Build low-latency, high-availability apps at global scale, or migrate Cassandra, MongoDB, and other NoSQL workloads to the cloud.
 - [Quickstart: Azure Cosmos DB for NoSQL client library for Python](#)
 - [Quickstart: Azure Cosmos DB for MongoDB for Python with MongoDB driver](#)
 - [Quickstart: Build a Cassandra app with Python SDK and Azure Cosmos DB](#)
 - [Quickstart: Build an API for Table app with Python SDK and Azure Cosmos DB](#)
 - [Quickstart: Azure Cosmos DB for Apache Gremlin library for Python](#)

- **Blob storage:** Secure, massively scalable object storage for cloud-native apps, data lakes, archives, high-performance computing (HPC), and machine learning.
 - [Quickstart: Azure Blob Storage client library for Python](#)
 - [Azure Storage samples using v12 Python client libraries](#)
- **Azure Data Lake Storage Gen2:** Scalable, secure data lake optimized for high-performance analytics.
 - [Use Python to manage directories and files in Azure Data Lake Storage Gen2](#)
 - [Use Python to manage ACLs in Azure Data Lake Storage Gen2](#)
- **File storage:** Simple, secure, and serverless enterprise-grade cloud file shares.
 - [Develop for Azure Files with Python](#)
- **Redis Cache:** Accelerate application performance with a scalable, in-memory data store compatible with open source.
 - [Quickstart: Use Azure Cache for Redis in Python](#)

Big data and analytics

- **Azure Data Lake analytics:** Fully managed, pay-per-job analytics service that delivers powerful parallel data processing with built-in enterprise-grade security, auditing, and support.
 - [Manage Azure Data Lake Analytics using Python](#)
 - [Develop U-SQL with Python for Azure Data Lake Analytics in Visual Studio Code](#)
- **Azure Data Factory:** A fully managed data integration service that lets you visually build, orchestrate, and automate data movement and transformation across various data sources.
 - [Quickstart: Create a data factory and pipeline using Python](#)
 - [Transform data by running a Python activity in Azure Databricks](#)
- **Azure Event Hubs:** A fully managed, hyper-scale telemetry ingestion service designed to collect, transform, and store millions of events per second from connected devices and applications.
 - [Send events to or receive events from event hubs by using Python](#)
 - [Capture Event Hubs data in Azure Storage and read it by using Python \(azure-eventhub\)](#)
- **HDInsight:** A fully managed cloud service that runs popular open-source frameworks like Hadoop and Spark, backed by a 99.9% SLA for enterprise-grade big data analytics.
 - [Use Spark & Hive Tools for Visual Studio Code](#)

- **Azure Databricks:** A fully managed, fast, easy and collaborative Apache® Spark™ based analytics platform optimized for big data and AI workloads on Azure.
 - [Connect to Azure Databricks from Excel, Python, or R](#)
 - [Get Started with Azure Databricks](#)
 - [Tutorial: Azure Data Lake Storage Gen2, Azure Databricks & Spark](#)
- **Azure Synapse Analytics:** A fully managed analytics service that unifies data integration, enterprise data warehousing, and big data analytics into a single platform.
 - [Quickstart: Use Python to query a database in Azure SQL Database or Azure SQL Managed Instance \(includes Azure Synapse Analytics\)](#)

Identity and access management for Python apps on Azure

06/05/2025

In Azure, identity and access management (IAM) for Python applications involves two key concepts:

- **Authentication:** Verifying the identity of a user, group, service, or application
- **Authorization:** Determining what actions that identity is allowed to perform on Azure resources

Azure provides multiple IAM options to fit your application's security requirements. This article includes links to essential resources to help you get started.

To learn more, see [Recommendations for identity and access management](#).

Passwordless connections

Whenever possible, we recommend using managed identities to simplify identity management and enhance security. Managed identities support passwordless authentication, eliminating the need to embed sensitive credentials—such as passwords or client secrets—in code or environment variables. Managed identities are available for Azure services like App Service, Azure Functions, and Azure Container Apps. They allow your applications to authenticate to Azure services without needing to manage credentials.

The following resources demonstrate how to use the Azure SDK for Python with passwordless authentication via [DefaultAzureCredential](#). `DefaultAzureCredential` is ideal for most applications running in Azure, as it seamlessly supports both local development and production environments by chaining multiple credential types in a secure and intelligent order.

- [Overview: Passwordless connection for Azure services](#)
- [Authenticate Python Apps to Azure services using the Azure SDK for Python](#)
- [Use DefaultAzureCredential in an application](#)
- [Quickstart: Azure Blob Storage client library for Python with passwordless connections](#)
- [Quickstart: Send messages to and receive message from Azure Service Bus queues with passwordless connections](#)

- [Create and deploy a Flask web app to Azure with a system-assigned managed identity](#)
- [Create and deploy a Django web app to Azure with a user-assigned managed identity](#)

Service Connector

Many Azure resources commonly used in Python applications support the [Service Connector](#). The Service Connector streamlines the process of configuring secure connections between Azure services. It automates the setup of authentication, network access, and connection strings between compute services (like App Service or Container Apps) and dependent services (such as Azure Storage, Azure SQL, or Cosmos DB). This reduces manual steps, helps enforce best practices (like using managed identities and private endpoints), and improves deployment consistency and security.

- [Quickstart: Create a service connection in App Service from the Azure portal](#)
- [Tutorial: Using Service Connector to build a Django app with Postgres on Azure App Service](#)

Key Vault

Using a key management solution such as [Azure Key Vault](#) offers greater control over your secrets and credentials, though it comes with added management complexity.

- [Quickstart: Azure Key Vault certificate client library for Python](#)
- [Quickstart: Azure Key Vault keys client library for Python](#)
- [Quickstart: Azure Key Vault secret client library for Python](#)

Authentication and identity for signing in users in apps

You can develop Python applications that allow users to sign in with Microsoft identities (like Azure AD accounts) or external social accounts (such as Google or Facebook). Once authenticated, your app can authorize users to access its own APIs or Microsoft APIs, such as Microsoft Graph, to interact with resources like user profiles, calendars, and emails.

- [Quickstart: Sign in users and call the Microsoft Graph API from a Python web app](#)
- [Web app authentication topics](#)

- Quickstart: Acquire a token and call Microsoft Graph from a Python daemon app
- Back-end service, daemon, and script authentication topics

Machine learning for Python apps on Azure

06/12/2025

The following articles help you get started with Azure Machine Learning. Azure Machine Learning v2 REST APIs, Azure CLI extension, and Python SDK are designed to streamline the entire machine learning lifecycle and accelerate production workflows. The links in this article target v2, which is recommended if you're starting a new machine learning project.

Getting started

In Azure Machine Learning, the workspace is the main resource that organizes and manages everything you create, such as datasets, models, and experiments.

- [Quickstart: Get started with Azure Machine Learning](#)
- [Manage Azure Machine Learning workspaces in the portal or with the Python SDK \(v2\)](#)
- [Run Jupyter notebooks in your workspace](#)
- [Tutorial: Model development on a cloud workstation](#)

Deploy models

Deploy models for low-latency, real-time machine learning predictions.

- [Tutorial: Designer - deploy a machine learning model](#)
- [Deploy and score a machine learning model by using an online endpoint](#)

Automated machine learning

Automated ML (AutoML) refers to the process of streamlining machine learning model development by automating its repetitive and time-consuming tasks.

- [Train a regression model with AutoML and Python \(SDK v1\)](#)
- [Set up AutoML training for tabular data with the Azure Machine Learning CLI and Python SDK \(v2\)](#)

Data access

With Azure Machine Learning, you can import data from your local computer or connect to existing cloud storage services.

- [Create and manage data assets](#)

- [Tutorial: Upload, access and explore your data in Azure Machine Learning](#)
- [Access data in a job](#)

Machine learning pipelines

Use machine learning pipelines to build workflows that connect different stages of the ML process.

- [Use Azure Pipelines with Azure Machine Learning](#)
- [Create and run machine learning pipelines using components with the Azure Machine Learning SDK v2](#)
- [Tutorial: Create production ML pipelines with Python SDK v2 in a Jupyter notebook](#)

AI services for Python apps on Azure

06/02/2025

Azure AI services are cloud-based artificial intelligence (AI) services that help developers build cognitive intelligence into applications without having direct AI or data science skills or knowledge. There are ready-made AI services for computer vision and image processing, language analysis and translation, speech, decision-making, search, and Azure OpenAI that you can use in your Python applications.

Because of the dynamic nature of Azure AI services, the best way to find getting started material for Python is to begin on the [Azure AI services hub page](#), and then find the specific service you're looking for.

1. On the hub page, select a service to go its documentation landing page. For example, for [Azure AI Vision](#).
2. On the service's landing page, select a category of the service. For example, in Computer Vision, select [Image Analysis](#).
3. In the documentation, look for **Quickstarts** in the table of contents. For example, in the Image Analysis documentation, under Quickstarts, there's a [Version 4.0 quickstart \(preview\)](#).
4. In quickstart articles, choose the Python programming language if it exists or the REST API.

If you don't see a quickstart, in the table of contents search box enter *Python* to find Python-related articles.

Also, you can go to the [Azure Cognitive Services modules for Python](#) overview to learn about the available Python SDK modules. (Azure Cognitive Services is the previous name of Azure AI services. The documentation is currently being updated to reflect the change.)

[Go to the Azure AI services hub page >>>](#)

The documentation for Azure AI Search is in a separate part of the documentation:

- [Quickstart: Full text search using the Azure SDKs](#)
- [Use Python and AI to generate searchable content from Azure blobs](#)

Messaging, Events, and IoT for Python apps on Azure

06/04/2025

The following articles help you get started with messaging, event ingestion and processing, and Internet of Things (IoT) services in Azure.

Messaging

Azure messaging services let different components and apps communicate easily, no matter what language they use or where they're hosted—whether in the same cloud, across multiple clouds, or on-premises.

- **Notifications**
 - [How to use Notification Hubs from Python](#)
- **Queues**
 - [Quickstart: Azure Queue Storage client library for Python](#)
 - [Quickstart: Send messages to and receive messages from Azure Service Bus queues \(Python\)](#)
 - [Send messages to an Azure Service Bus topic and receive messages from subscriptions to the topic \(Python\)](#)
- **Real-time web functionality (SignalR)**
 - [Quickstart: Create a serverless app with Azure Functions and Azure SignalR Service in Python](#)
- **Azure Web PubSub**
 - [How to create a WebPubSubServiceClient with Python and Azure Identity](#)

Events

Azure Event Hubs and Azure Event Grid are two key services for handling events in Azure. They provide capabilities for ingesting, processing, and routing events across various applications and services.

These services allow you to build event-driven architectures and process events in real time.

- **Event Hubs**
 - [Quickstart: Send events to or receive events from event hubs by using Python](#)

- [Quickstart: Capture Event Hubs data in Azure Storage and read it by using Python \(azure-eventhub\)](#)
- **Event Grid**
 - [Quickstart: Route custom events to web endpoint with Azure CLI and Event Grid](#)
 - [Azure Event Grid Client Library Python Samples](#)

Internet of Things (IoT)

Internet of Things (IoT) refers to a set of managed and platform services across edge and cloud that connect, monitor, and control IoT assets. IoT also encompasses device security, operating systems, and data analytics tools to help you build, deploy, and manage IoT applications effectively.

- **IoT Hub**
 - [Quickstart: Send telemetry from an IoT Plug and Play device to Azure IoT Hub](#)
 - [Send cloud-to-device messages with IoT Hub](#)
 - [Upload files from your device to the cloud with IoT Hub](#)
 - [Schedule and broadcast jobs](#)
 - [Quickstart: Control a device connected to an IoT hub](#)
- **Device provisioning**
 - [Quickstart: Provision an X.509 certificate simulated device](#)
 - [Tutorial: Provision devices using symmetric key enrollment groups](#)
 - [Tutorial: Provision multiple X.509 devices using enrollment groups](#)
- **IoT Central/IoT Edge**
 - [Tutorial: Create and connect a client application to your Azure IoT Central application](#)
 - [Tutorial: Develop IoT Edge modules using Visual Studio Code](#)

Other services for Python apps on Azure

06/12/2025

The services referenced in this article for Python are specialized, each designed to address a specific set of problems. The term *other services* includes Azure offerings beyond the foundational categories of compute, networking, storage, and databases. This article features examples from areas like management and governance, media, genomics, and the Internet of Things (IoT). For a full list of available services [Azure products](#).

- **Media streaming:**
 - [Connect to Azure Media Services v3 API](#)
- **Automation:**
 - [Tutorial: Create a Python runbook](#)
- **DevOps:**
 - [Use CI/CD with GitHub Actions to deploy a Python web app to Azure App Service on Linux](#)
 - [Build and deploy a Python cloud app with the Azure Developer CLI \(azd\) open-source tool](#)
 - [Build, test, and deploy Python apps with Azure Pipelines](#)
- **Internet of Things and geographical mapping:**
 - [Tutorial: Route electric vehicles by using Azure Notebooks](#)
 - [Tutorial: Join sensor data with weather forecast data by using Azure Notebooks](#)
- **Burrows-Wheeler Aligner (BWA) and the Genome Analysis Toolkit (GATK):**
 - [Quickstart: Run a workflow through the Microsoft Genomics service](#)
- **Resource management:**
 - [Quickstart: Run your first Resource Graph query using Python](#)
- **Virtual machine management:**
 - [Example: Use the Azure libraries to create a virtual machine](#)