

Azure for Python Developers

Deploy your Python code to Azure for web apps, serverless apps, containers, and machine learning models. Take advantage of the Azure libraries (SDK) for Python to programmatically access the full range of Azure services including storage, databases, pre-built AI capabilities, and much more.

Azure libraries (SDK)

GET STARTED

[Get started](#)

[Set up your local dev environment](#)

[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 • 11/13/2023

Use this document as a checklist and a guide as you begin developing Python applications that will be hosted in the cloud or utilize cloud services. If you follow the links and instructions in this document, you'll:

- have a fundamental understanding of what the cloud is and how you design your application with the cloud in mind.
- setup your local development environment including the tools and libraries you'll need to build cloud-based applications.
- understand the workflow when developing cloud-based applications.

Phase 1: Learn concepts

If you are new to developing applications for the cloud, this short series of articles with videos will help you get up to speed quickly.

- 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](#)

Once you understand the basics of developing applications for the cloud, you will want to set up your development environment and follow a Quickstart or Tutorial to build your first app.

Phase 2: Configure your local Python environment for Azure development

To develop Python applications using Azure, you first want to configure your local development environment. Configuration includes creating an Azure account, installing tools for Azure development, and connecting those tools to your Azure account.

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.

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](#)

Use the Azure portal

Once you have your credentials, you can sign in to the [Azure portal](#) at <https://portal.azure.com>. The Azure portal is typically easiest way to get started with Azure, especially if you're new to Azure and cloud development. In the Azure portal, you can do various management tasks such as creating and deleting resources.

If you're already experienced with Azure and cloud development, you'll probably start off using tools as well such as Visual Studio Code and Azure CLI. Articles in the Python developer center show how to work with the Azure portal, Visual Studio Code, and Azure CLI.

[Sign in to the Azure portal](#)

Use Visual Studio Code

You can use any editor or IDE to write Python code when developing for Azure. However, you may want to consider using [Visual Studio Code](#) for Azure and Python development. Visual Studio Code provides many extensions and customizations for Azure and Python, which make your development cycle and the deployment from a local environment to Azure easier.

For Python development using Visual Studio Code, install:

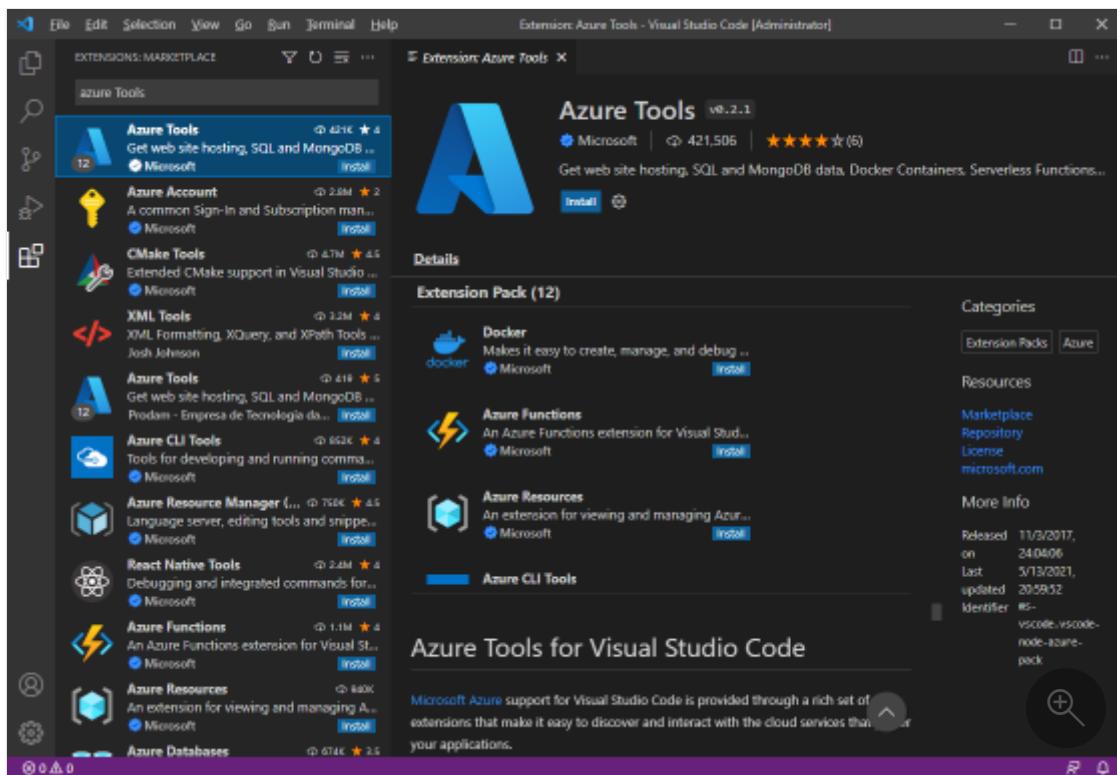
- [Python extension](#). This extension includes IntelliSense (Pylance), Linting, Debugging (multi-threaded, remote), Jupyter Notebooks, code formatting,

refactoring, unit tests, and more.

- [Azure Tools extension pack](#). The extension pack contains extensions for working with Azure App Service, Azure Functions, Azure Storage, Azure Cosmos DB, and Azure Virtual Machines in one convenient package. The Azure extensions make it easy to discover and interact with the Azure.

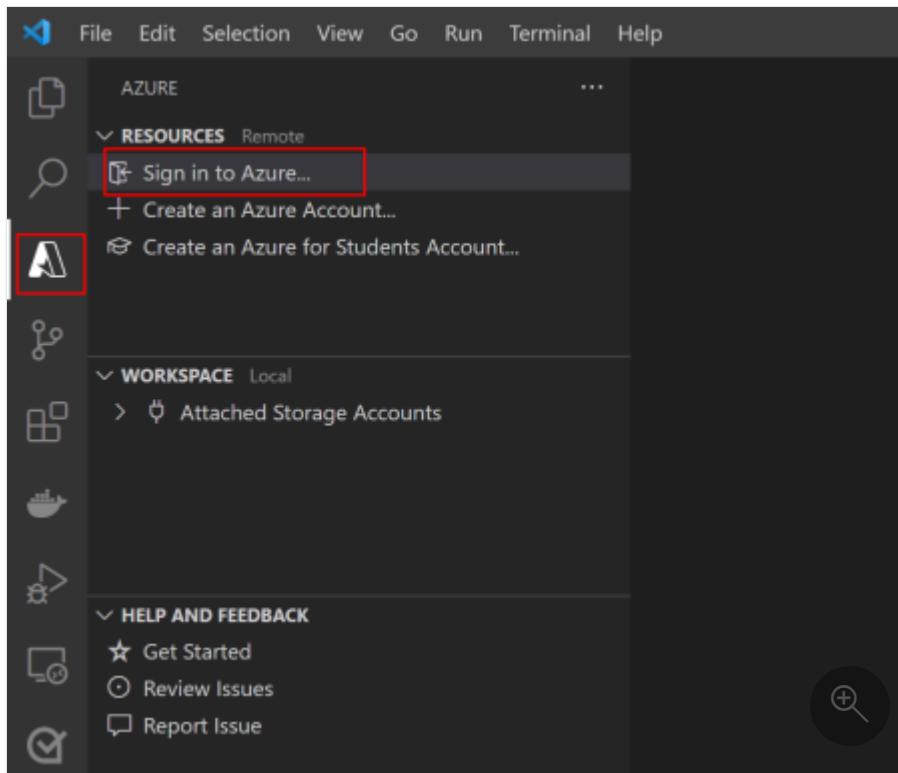
To install extensions from Visual Studio Code:

1. Press **Ctrl+Shift+X** to open the **Extensions** window.
2. Search for the *Azure Tools* extension.
3. Select the **Install** button.



To learn more about installing extensions in Visual Studio Code, refer to the [Extension Marketplace](#) document on the Visual Studio Code website.

After installing the Azure Tools extension, sign in with your Azure account. On the left-hand panel, you'll see an Azure icon. Select this icon, and a control panel for Azure services will appear. Choose **Sign in to Azure...** to complete the authentication process.



ⓘ Note

If you see the error "Cannot find subscription with name [subscription ID]", this may be because you are behind a proxy and unable to reach the Azure API. Configure `HTTP_PROXY` and `HTTPS_PROXY` environment variables with your proxy information in your terminal:

Windows Command Prompt

```
# Windows
set HTTPS_PROXY=https://username:password@proxy:8080
set HTTP_PROXY=http://username:password@proxy:8080
```

Bash

```
# macOS/Linux
export HTTPS_PROXY=https://username:password@proxy:8080
export HTTP_PROXY=http://username:password@proxy:8080
```

Use the Azure CLI

In addition to the Azure portal and Visual Studio Code, Azure also offers the [Azure CLI](#) command-line tool to create and manage Azure resources. The Azure CLI offers the

benefits of efficiency, repeatability, and the ability to script recurring tasks. In practice, most developers use both the Azure portal and the Azure CLI.

After [installing the Azure CLI](#), sign-in to your Azure account from the Azure CLI by typing the command `az login` in a terminal window on your workstation.



Azure CLI

```
az login
```

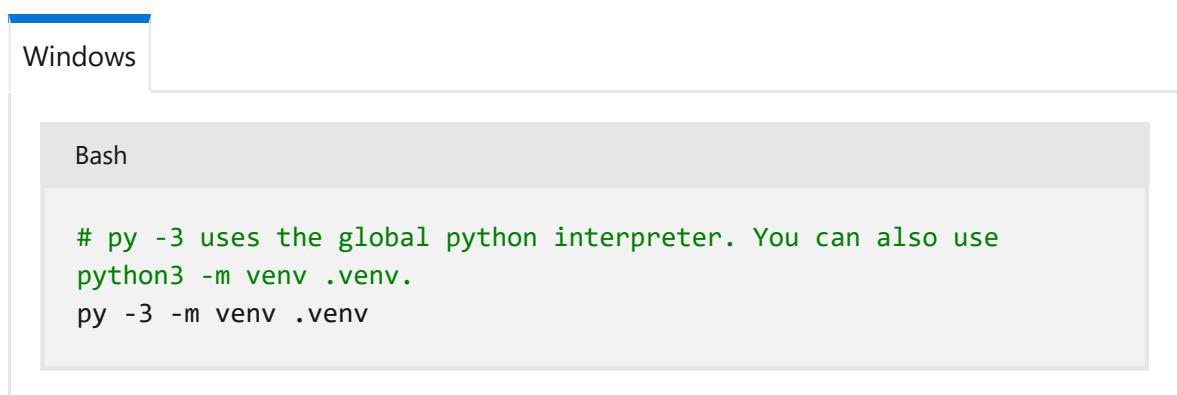
The Azure CLI will open your default browser to complete the sign-in process.

Configure Python virtual environment

When creating Python applications for Azure, it's recommended to create a [virtual environment](#) for each application. A virtual environment is a self-contained directory for a particular version of Python plus the other packages needed for that application.

To create a virtual environment, follow these steps.

1. Open a terminal or command prompt.
2. Create a folder for your project.
3. Create the virtual environment:



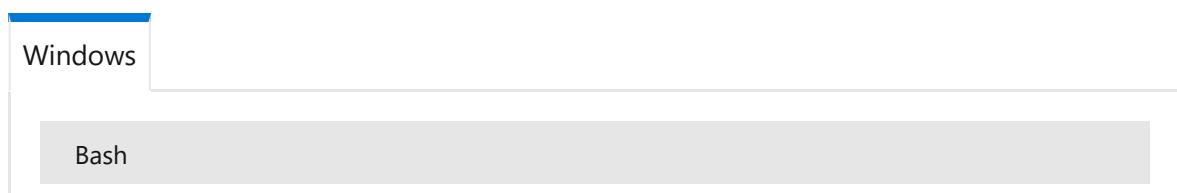
Windows

Bash

```
# py -3 uses the global python interpreter. You can also use
python3 -m venv .venv.
py -3 -m venv .venv
```

This command runs the Python `venv` module and creates a virtual environment in a folder ".venv". Typically, [.gitignore](#) files have a ".venv" entry so that the virtual environment doesn't get checked in with your code checkins.

4. Activate the virtual environment:



Windows

Bash

```
.venv\Scripts\activate
```

ⓘ Note

If you're using Windows Command shell, activate the virtual environment with `.venv\Scripts\activate`. If you're using [Git Bash in Visual Studio Code](#) on Windows, use the command `source .venv/Scripts/activate` instead.

Once you activate that environment (which Visual Studio Code does automatically), running `pip install` installs a library into that environment only. Python code running in a virtual environment uses the specific package versions installed into that virtual environment. Using different virtual environments allows different applications to use different versions of a package, which is sometimes required. To learn more about virtual environments, see [Virtual Environments and Packages](#) in the Python docs.

For example, if your [requirements](#) are in a `requirements.txt` file, then inside the activated virtual environment, you can install them with:

```
Bash
```

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

Phase 3: Understand the Azure development workflow

[Previous article: provisioning, accessing, and managing resources](#)

Now that you understand Azure's model of services and resources, you can understand the overall flow of developing cloud applications with Azure: **provision, code, test, deploy, and manage**.

[Expand table](#)

Step	Primary tools	Activities
Provision	Azure CLI, Azure portal, VS Code Azure Tools extensions, Cloud Shell, Python scripts using Azure SDK management libraries	Create resource groups and create resources in those groups; configure resources to be ready for use from app code and/or ready to receive Python code in deployments.

Step	Primary tools	Activities
Code	Code editor (such as Visual Studio Code and PyCharm), Azure SDK client libraries, reference documentation	Write Python code using the Azure SDK client libraries to interact with provisioned resources.
Test	Python runtime, debugger	Run Python code locally against active cloud resources (typically dev or test resources rather than production resources). The code itself isn't yet hosted on Azure, which helps you debug and iterate quickly.
Deploy	VS Code, Azure CLI, GitHub Actions, Azure Pipelines	Once code has been tested locally, deploy it to an appropriate Azure hosting service where the code itself can run in the cloud. Deployed code typically runs against staging or production resources.
Manage	Azure CLI, Azure portal, VS Code, Python scripts, Azure Monitor	Monitor app performance and responsiveness, make adjustments in production environment, migrate improvements back to dev environment for the next round of provisioning and development.

Step 1: Provision and configure resources

As described in the [previous article of this series](#), the first step in developing any application is to provision and configure the resources that make up the target environment for your application.

Provisioning begins by creating a resource group in a suitable Azure region. You can create a resource group through the Azure portal, VS Code with Azure Tools extensions, the Azure CLI, or with a custom script that uses the Azure SDK management libraries (or REST API).

Within that resource group, you then provision and configure the individual resources you need, again using the portal, VS Code, the CLI, or the Azure SDK. (Again, review the [Azure developer's guide](#) for an overview of available resource types.)

Configuration includes setting 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.

For most application scenarios, you typically create provisioning scripts with the Azure CLI and/or Python code using the Azure SDK management libraries. Such scripts describe the totality of your application's resource needs (essentially defining the custom cloud computer to which you're deploying the application). A script enables you to easily recreate the same set of resources within different environment like development, test, staging, and production. When you automate, you can avoid manually performing many repeated steps in Azure portal or VS Code. Such scripts also make it easy to provision an environment in a different region, or to use different resource groups. If you also maintain these scripts in source control repositories, you also have full auditing and change history.

Step 2: Write your app code to use resources

Once you've provisioned the resources you need for your application, you write the application code to work with the run time aspects of those resources.

For example, in the provisioning step you might have created an Azure storage account, created a blob container within that account, and set access policies for the application on that container. This provisioning process is demonstrated in [Example - Provision Azure Storage](#). From your code, you can then authenticate with that storage account and then create, update, or delete blobs within that container. This run time process is demonstrated in [Example - Use Azure Storage](#). Similarly, you might have provisioned a database with a schema and appropriate permissions (as demonstrated in [Example - Provision a database](#)), so that your application code can connect to the database and perform the usual create-read-update-delete queries.

App code typically uses environment variables to identify the names and URLs of the resources to use. Environment variables allow you to easily switch between cloud environments (dev, test, staging, and production) without any changes to the code. The various Azure services that host application code provide a means to define the necessary variables. For example, in Azure App Service (to host web apps) and Azure Functions (serverless compute for Azure), you define *application settings* through the Azure portal, VS Code, or Azure CLI, which then appear to your code as environment variables.

As a Python developer, you'll likely write your application code in Python using the Azure SDK client libraries for Python. That said, any independent part of a cloud application can be written in any supported language. If you're working on a team using multiple programming languages, it's possible that some parts of the application use Python, some JavaScript, some Java, and others C#.

Application code can use the Azure SDK management libraries to perform provisioning and management operations as needed. Provisioning scripts, similarly, can use the SDK client libraries to initialize resources with specific data, or perform housekeeping tasks on cloud resources even when those scripts are run locally.

Step 3: Test and debug your app code locally

Developers typically like to test app code on their local workstations before deploying that code to the cloud. Testing app code locally means that you're typically accessing other resources that you've already provisioned in the cloud, such as storage, databases, and so forth. The difference is that you're not yet running the app code itself within a cloud service.

By running the code locally, you can also take full advantage of debugging features offered by tools such as Visual Studio Code and manage your code in a source control repository.

You don't need to modify your code at all for local testing: Azure fully supports local development and debugging using the same code you deploy to the cloud.

Environment variables are again the key: in the cloud, your code can access the hosting resource's settings as environment variables. When you create those same environment variables locally, the same code runs without modification. This pattern works for authentication credentials, resource URLs, connection strings, and any number of other settings, making it easy to use resources in a development environment when running code locally and production resources once the code is deployed to the cloud.

Step 4: Deploy your app code to Azure

Once you've tested your code locally, you're ready to deploy the code to the Azure resource that you've provisioned to host it. For example, if you're writing a Django web app, you either deploy that code to a virtual machine (where you provide your own web server) or to Azure App Service (which provides the web server for you). Once deployed, that code is running on the server rather than on your local machine, and can access all the Azure resources for which it's authorized.

As noted in the previous section, in typical development processes you first deploy your code to the resources you've provisioned in a development environment. After a round of testing, you deploy your code to resources in a staging environment, making the application available to your test team and perhaps preview customers. Once you're satisfied with the application's performance, you can deploy the code to your production environment. All of these deployments can also be automated through

continuous integration and continuous deployment using Azure Pipelines and GitHub Actions.

However you do it, once the code is deployed to the cloud, it truly becomes a cloud application, running entirely on the server computers in Azure data centers.

Step 5: Manage, monitor, and revise

After deployment, you want to make sure the application is performing as it should, responding to customer requests and using resources efficiently (and at the lowest cost). You can manage how Azure automatically scales your deployment as needed, and you can collect and monitor performance data with Azure portal, VS Code, the Azure CLI, or custom scripts written with the Azure SDK libraries. You can then make real-time adjustments to your provisioned resources to optimize performance, again using any of the same tools.

Monitoring gives you insight about how you might restructure your cloud application. For example, you may find that certain portions of a web app (such as a group of API endpoints) are used only occasionally in comparison to the primary parts. You could then choose to deploy those APIs separately as serverless Azure Functions. As functions, they have their own backing compute resources that don't compete with the main application but cost only pennies per month. Your main application then becomes more responsive to more customers without having to scale up to a higher-cost tier.

Next steps

You're now familiar with the basic structure of Azure and the overall development flow: provision resources, write and test code, deploy the code to Azure, and then monitor and manage those resources.

- [Develop a Python web app](#)
- [Develop a container app](#)
- [Learn to use the Azure libraries for Python](#)

Develop Python apps that use Azure AI services

Article • 11/21/2023

This article provides documentation, samples and other resources for learning how to develop applications that use Azure OpenAI Service and other Azure AI Services.

Azure AI reference templates

Azure AI reference templates provide you with well-maintained, easy to deploy reference implementations. These ensure a high-quality starting point for your intelligent applications. The end-to-end solutions provide popular, comprehensive reference applications. The building blocks are smaller-scale samples that focus on specific scenarios and tasks.

End-to-end solutions

[+] Expand table

Link	Description
Get started with the Python enterprise chat sample using RAG	An article that walks you through deploying and using the Enterprise chat app sample for Python . This sample is a complete end-to-end solution demonstrating the Retrieval-Augmented Generation (RAG) pattern running in Azure, using Azure AI Search for retrieval and Azure OpenAI large language models to power ChatGPT-style and Q&A experiences.

- [Demo video](#)

Building blocks

[+] Expand table

Building Block	Description
Build a chat app with Azure OpenAI in Python	A simple Python Quart app that streams responses from ChatGPT to an HTML/JS frontend using JSON Lines over a ReadableStream.
Build a LangChain with Azure OpenAI in	An Azure Functions sample that shows how to take a human prompt as HTTP Get or Post input, calculates the completions using chains of

Building Block	Description
Python ↗	human input and templates. This is a starting point that can be used for more sophisticated chains.
Build a ChatGPT Plugin with Azure Container Apps in Python ↗	A sample for creating ChatGPT Plugin using GitHub Codespaces, VS Code, and Azure. The sample includes templates to deploy the plugin to Azure Container Apps using the Azure Developer CLI.
Summarize Text using Azure AI Language with Azure Functions ↗	Take text documents as input, summarize using Azure AI Language, and then output to another text document using Azure Functions.
Azure AI Python Template Gallery ↗	For the full list of Azure AI templates, visit our gallery. All app templates in our gallery can be spun up and deployed using a single command: <code>azd up</code> .
Smart load balancing with Azure Container Apps	This sample solution ↗ is built using the high-performance YARP C# reverse-proxy framework ↗ from Microsoft. However, you don't need to understand C# to use it, you can just build the provided Docker image. This is an alternative solution to the API Management OpenAI smart load balancer ↗ , with the same logic.
Smart load balancing with Azure API Management	The enterprise sample solution ↗ shows how to create an Azure API Management Policy to seamlessly expose a single endpoint to your applications while keeping an efficient logic to consume two or more OpenAI or any API backends based on availability and priority.
Evaluate your chat app	Evaluate a chat app's answers against a set of correct or ideal answers (known as ground truth). The evaluation tools ↗ can be used with any Chat API which conforms to the Chat protocol ↗ .
Load test your chat app with Locust	Use a Locust test to validate your chat app can handle the expected load. If your chat app doesn't scale on your App Service due to Azure OpenAI TPM limits, add a load balancer and test your load again. Smart load balancers include Azure API Management and Azure Container Apps .

Azure OpenAI

End-to-end solutions

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

Link	Description
Get started with the Python	An article that walks you through deploying and using the Enterprise chat app sample for Python ↗ . This sample is a complete end-to-end solution

Link	Description
enterprise chat sample using RAG	demonstrating the Retrieval-Augmented Generation (RAG) pattern running in Azure, using Azure AI Search for retrieval and Azure OpenAI large language models to power ChatGPT-style and Q&A experiences.

Building blocks

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

Link	Description
Build a chat app with Azure OpenAI in Python ↗	A simple Python Quart app that streams responses from ChatGPT to an HTML/JS frontend using JSON Lines over a ReadableStream.
Build a LangChain with Azure OpenAI in Python ↗	A sample shows how to take a human prompt as HTTP Get or Post input, calculates the completions using chains of human input and templates. This is a starting point that can be used for more sophisticated chains.
Build a ChatGPT Plugin with Azure Container Apps in Python ↗	A sample for creating ChatGPT Plugin using GitHub Codespaces, VS Code, and Azure. The sample includes templates to deploy the plugin to Azure Container Apps using the Azure Developer CLI.
Vector Similarity Search with Azure Cache for Redis Enterprise ↗	A walkthrough using Azure Cache for Redis as a backend vector store for RAG scenarios.
OpenAI solutions with your own data using PostgreSQL ↗	An article discussing how Azure Database for PostgreSQL Flexible Server and Azure Cosmos DB for PostgreSQL supports the pgvector extension, along with an overview, scenarios, etc.

SDKs and other samples/guidance

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

Link	Description
OpenAI SDK for Python ↗	The GitHub source code version of the OpenAI Python library provides convenient access to the OpenAI API from applications written in the Python language.
Azure OpenAI SDK Releases ↗	Links to all Azure OpenAI SDK library packages, including links for .NET, Java, JavaScript and Go.
openai Python Package ↗	The PyPi version of the OpenAI Python library.

Link	Description
Get started using GPT-35-Turbo and GPT-4	An article that walks you through creating a chat completion sample.
Streaming Chat completions ↗	A notebook containing example of getting chat completions to work using the Azure endpoints. This example focuses on chat completions but also touches on some other operations that are also available using the API.
Switch from OpenAI to Azure OpenAI ↗	Guidance article on the small changes you need to make to your code in order to swap back and forth between OpenAI and the Azure OpenAI Service.
Embeddings ↗	A notebook demonstrating operations how to use embeddings that can be done using the Azure endpoints. This example focuses on embeddings but also touches some other operations that are also available using the API.
Deploy a model and generate text	An article with minimal, straightforward detailing steps to programmatically chat.
OpenAI with Microsoft Entra ID Role based access control	A look at authentication using Microsoft Entra ID.
OpenAI with Managed Identities	An article with more complex security scenarios requires Azure role-based access control (Azure RBAC). This document covers how to authenticate to your OpenAI resource using Microsoft Entra ID.
More samples ↗	A compilation of useful Azure OpenAI Service resources and code samples to help you get started and accelerate your technology adoption journey.
More guidance	The hub page for Azure OpenAI Service documentation.

Other Azure AI services

End-to-end solutions

[Expand table](#)

Link	Description
Captioning and Call Center Transcription ↗	A repo containing samples for captions and transcriptions in a call center scenario.

Link	Description
Use Document Intelligence to automate a paper based process using the New patient registration with Form Recognizer workshop <small>↗</small> (Code ↗)	A workshop style presentation that walks you through how to use Document Intelligence to convert and automate a paper-based process.

Building blocks

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

Link	Description
Use Speech to converse with OpenAI	Use Azure AI Speech to converse with Azure OpenAI Service. The text recognized by the Speech service is sent to Azure OpenAI. The Speech service synthesizes the text response from Azure OpenAI.
Translate documents from and into more than 100 different languages using Document Translation sample apps <small>↗</small>	A repo containing both a Command Line tool and Windows application that serves as a local interface to the Azure Document Translation service for Windows, macOS and Linux.

SDKs and samples/guidance

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

Link	Description
Integrate Speech into your apps with Speech SDK Samples <small>↗</small>	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 <small>↗</small>	Samples for the Azure.AI.FormRecognizer client library.

Link	Description
Extract, classify, and understand text within documents using Text Analytics in Python	The client Library for Text Analytics. This is 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), a cloud-based conversational AI service, which can extract intents and entities in conversations and acts like an orchestrator to select the best candidate to analyze conversations to get 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 ↗	Detects 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.

Feedback

Was this page helpful?



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

Get started with the Python enterprise chat sample using RAG

Article • 05/14/2024

This article shows you how to deploy and run the [Enterprise chat app sample for Python](#). This sample implements a chat app 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 including the employee handbook, a benefits document and a list of company roles and expectations.

- [Demo video](#)

By following the instructions in this article, you will:

- Deploy a chat app to Azure.
- Get answers about employee benefits.
- Change settings to change behavior of responses.

Once you complete this procedure, you can start modifying the new project with your custom code.

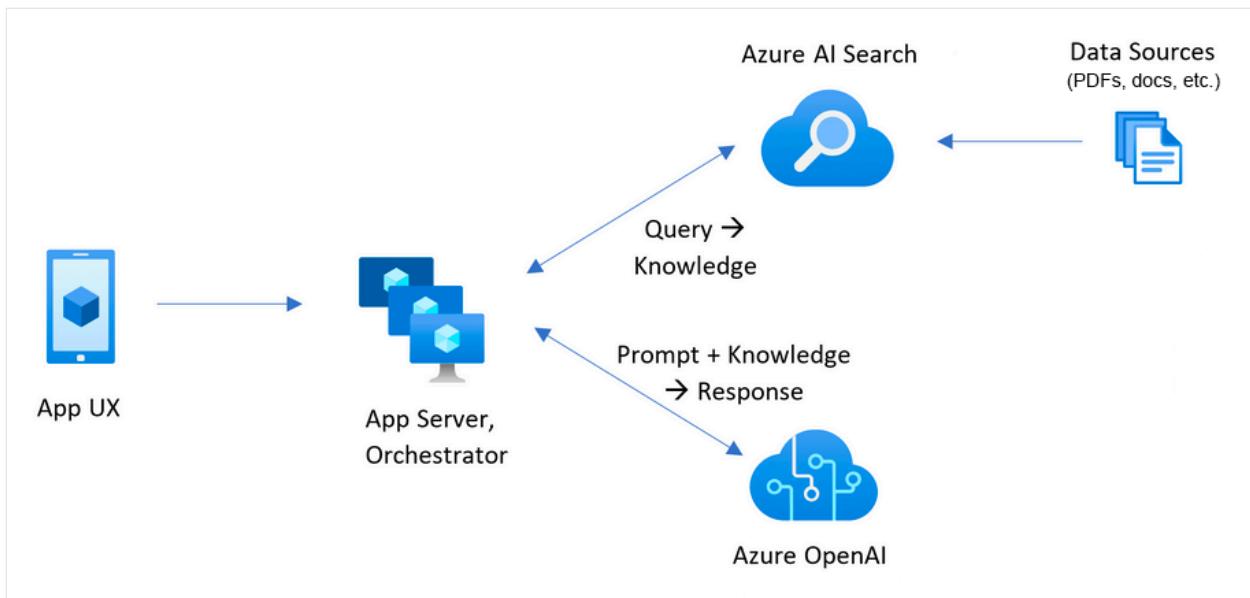
This article is part of a collection of articles that show you how to build a chat app using Azure OpenAI Service and Azure AI Search.

Other articles in the collection include:

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

Architectural overview

A simple architecture of the chat app is shown in the following diagram:



Key components of the architecture include:

- A web application to host the interactive chat experience.
- An Azure AI Search resource to get answers from your own data.
- An Azure OpenAI Service to provide:
 - Keywords to enhance the search over your own data.
 - Answers from the OpenAI model.
 - Embeddings from the ada model

Cost

Most resources in this architecture use a basic or consumption pricing tier. Consumption pricing is based on usage, which means you only pay for what you use. To complete this article, there will be a charge but it will be minimal. When you're done with the article, you can delete the resources to stop incurring charges.

Learn more about [cost in the sample repo ↗](#).

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 using Visual Studio Code.

To use this article, you need the following prerequisites:

Codespaces (recommended)

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

2. Azure account permissions - Your Azure Account must have Microsoft.Authorization/roleAssignments/write permissions, such as [User Access Administrator](#) or [Owner](#).
3. Access granted to Azure OpenAI in the desired Azure subscription. Currently, access to this service is granted only by application. You can apply for access to Azure OpenAI by completing the form at <https://aka.ms/oai/access>. Open an issue on this repo to contact us if you have an issue.
4. 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.

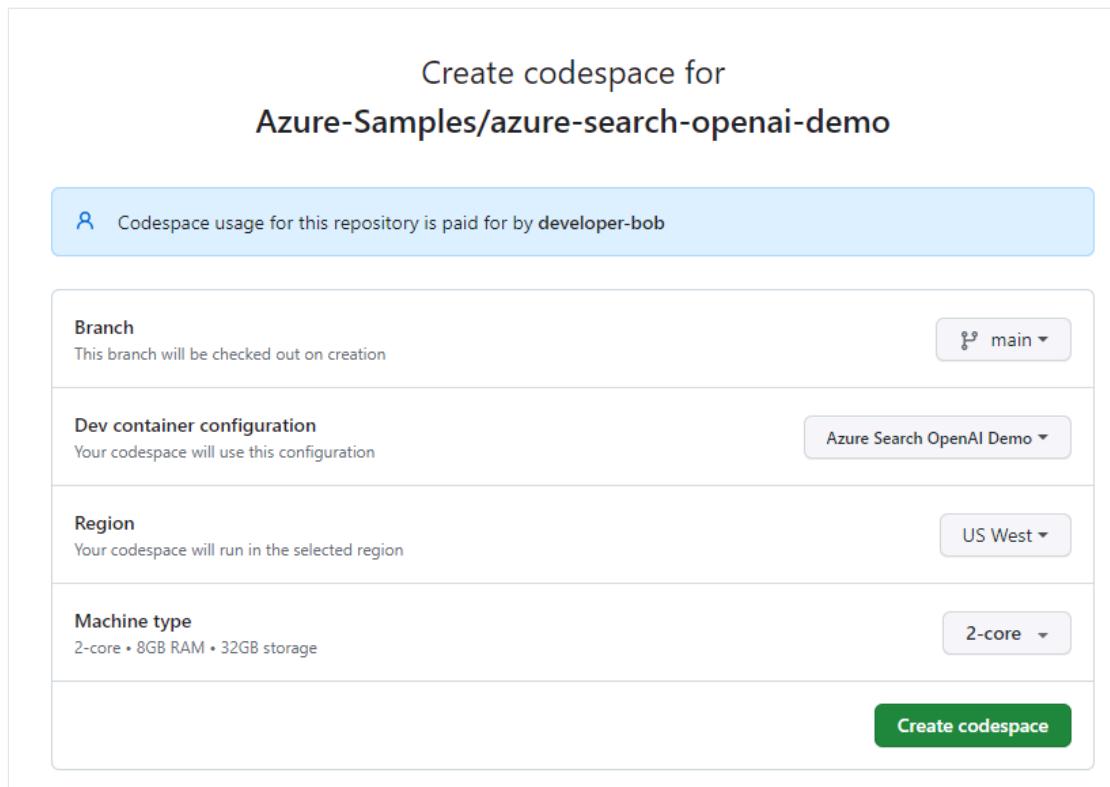
ⓘ Important

All GitHub accounts can use Codespaces for up to 60 hours free each month with 2 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 on the following button, and select *Open link in new windows* in order to have both the development environment and the documentation available at the same time.

[Open this project in GitHub Codespaces](#)

3. On the [Create codespace](#) page, review the codespace configuration settings and then select [Create new 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.
7. The remaining tasks in this article take place in the context of this development container.

Deploy and run

The sample repository contains all the code and configuration files you need to deploy a chat app to Azure. The following steps walk you through the process of deploying the sample to Azure.

Deploy chat app to Azure

ⓘ Important

Azure resources created in this section incur immediate costs, primarily from the Azure AI Search resource. These resources may accrue costs even if you interrupt the command before it is fully executed.

1. Run the following Azure Developer CLI command to provision the Azure resources and deploy the source code:

Bash

```
azd up
```

2. When you're prompted to enter an environment name, keep it short and lowercase. For example, `myenv`. It's used as part of the resource group name.
3. When prompted, select a subscription to create the resources in.
4. When you're prompted to select a location the first time, select a location near you. This location is used for most the resources including hosting.
5. If you're prompted for a location for the OpenAI model or for the Document Intelligence resource, select the location closest to you. If the same location is available as your first location, select that.
6. Wait 5 or 10 minutes after the app is deployed before continuing.
7. After the application has been successfully deployed, you see a URL displayed in the terminal.

Deploying services (azd deploy)

```
(✓) Done: Deploying service backend
- Endpoint: https://app-backend-72xomfpzf3j4o.azurewebsites.net/
SUCCESS: Your Azure app has been deployed!
```

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

GPT + Enterprise data | Sample

Chat Ask a question

Azure OpenAI + Cognitive Search

Clear chat Developer settings

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 may vary as the underlying models are updated.

1. In the browser, select or enter *What happens in a performance review?* in the chat text box.

GPT + Enterprise data | Sample

Chat Ask a question

Azure OpenAI + Cognitive Search

Clear chat Developer settings

What happens in a performance review?

During a performance review, employees will have an opportunity to discuss their successes and challenges in the workplace ¹. The review will include constructive feedback and a written summary that includes a rating of the employee's performance, feedback, and goals and objectives for the upcoming year ¹. The review is a two-way dialogue between managers and employees, and employees are encouraged to be honest and open during the process ¹.

Citations: [1. employee_handbook-3.pdf](#)

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

2. From the answer, select a citation.

The screenshot shows the Azure OpenAI + Cognitive Search chat interface. At the top, there are tabs for "Chat" and "Ask a question" along with a "Clear chat" button and "Developer settings". A search bar at the top right contains the query "What happens in a performance review?". Below the search bar, a detailed answer is provided: "During a performance review, employees will have an opportunity to discuss their successes and challenges in the workplace [1]. The review will include constructive feedback and a written summary that includes a rating of the employee's performance, feedback, and goals and objectives for the upcoming year [1]. The review is a two-way dialogue between managers and employees, and employees are encouraged to be honest and open during the process [1].". A red box highlights the citation link "1. employee_handbook-3.pdf". A "Citations" button is also visible. At the bottom, there is a text input field with placeholder text "Type a new question (e.g. does my plan cover annual eye exams?)" and a "Send" button.

3. In the right-pane, use the tabs to understand how the answer was generated.

[] [Expand table](#)

Tab	Description
Thought process	This is a script of the interactions in chat. You can view the system prompt (content) and your user question (content).
Supporting content	This includes the information to answer your question and the source material. The number of source material citations is noted in the Developer settings . The default value is 3.
Citation	This displays the original page that contains the citation.

4. When you're done, select the selected tab again to close the pane.

Use chat app settings to change behavior of responses

The intelligence of the chat is determined by the OpenAI model and the settings that are used to interact with the model.

Configure answer generation

X

Override prompt template

Temperature



Minimum search score

Minimum reranker score

Retrieve this many search results:

Exclude category

Use semantic ranker for retrieval

Use query-contextual summaries instead of whole documents

Suggest follow-up questions

Retrieval mode *

Stream chat completion responses

Expand table

Setting	Description
Override prompt template	This is the prompt that is used to generate the answer.
Temperature	The temperature used for the final Chat Completion API call, a number between 0 and 1 that controls the "creativity" of the model.
Minimum search score	The minimum score of the search results that are used to generate the answer. Range depends on search mode used .

Setting	Description
Minimum reranker score	The minimum score from the semantic ranker of the search results that are used to generate the answer. Ranges from 0-4.
Retrieve this many search results	This is the number of search results that are used to generate the answer. You can see these sources returned in the <i>Thought process</i> and <i>Supporting content</i> tabs of the citation.
Exclude category	This is the category of documents that are excluded from the search results.
Use semantic ranker for retrieval	This is a feature of Azure AI Search that uses machine learning to improve the relevance of search results.
Use query-contextual summaries instead of whole documents	When both <code>use semantic ranker</code> and <code>use query-contextual summaries</code> are checked, the LLM uses captions extracted from key passages, instead of all the passages, in the highest ranked documents.
Suggest follow-up questions	Have the chat app suggest follow-up questions based on the answer.
Retrieval mode	Vectors + Text means that the search results are based on the text of the documents and the embeddings of the documents. Vectors means that the search results are based on the embeddings of the documents. Text means that the search results are based on the text of the documents.
Stream chat completion responses	Stream response instead of waiting until the complete answer is available for a response.

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

1. In the browser, select the **Developer Settings** tab.
2. Check the **Suggest follow-up questions** checkbox and ask the same question again.

What happens in a performance review?

The chat returned suggested follow-up questions such as the following:

1. What is the frequency of performance reviews?
2. How can employees prepare for a performance review?

3. Can employees dispute the feedback received during the performance review?

3. In the **Settings** tab, deselect **Use semantic ranker for retrieval**.

4. Ask the same question again?

What happens in a performance review?

5. What is the difference in the answers?

With the Semantic ranker: During a performance review at Contoso Electronics, employees will have the opportunity to discuss their successes and challenges in the workplace (1). The review will provide positive and constructive feedback to help employees develop and grow in their roles (1). The employee will receive a written summary of the performance review, which will include a rating of their performance, feedback, and goals and objectives for the upcoming year (1). The performance review is a two-way dialogue between managers and employees (1).

Without the Semantic ranker: During a performance review at Contoso Electronics, employees have the opportunity to discuss their successes and challenges in the workplace. Positive and constructive feedback is provided to help employees develop and grow in their roles. A written summary of the performance review is given, including a rating of performance, feedback, and goals for the upcoming year. The review is a two-way dialogue between managers and employees (1).

Clean up resources

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 --force
```

The switches provide:

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

Clean up GitHub Codespaces

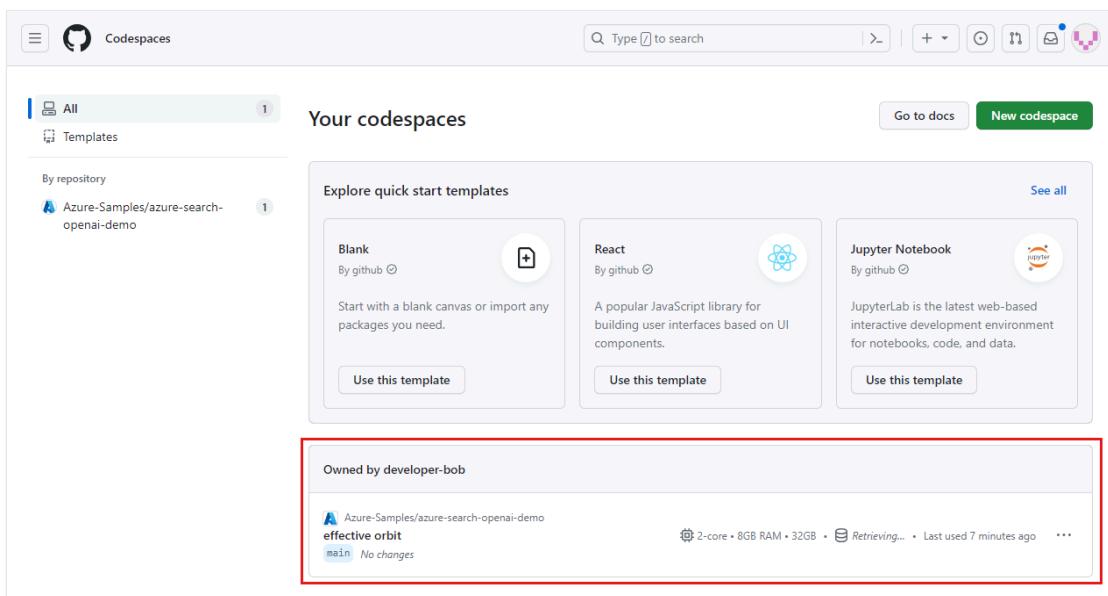
GitHub Codespaces

Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement 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 into the GitHub Codespaces dashboard (<https://github.com/codespaces>).
2. Locate your currently running Codespaces 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) and 'Templates'. A main area titled 'Your codespaces' shows a repository 'Azure-Samples/azure-search-openai-demo'. It features a section for 'Explore quick start templates' with options for 'Blank', 'React', and 'Jupyter Notebook'. Below this is a list of owned codespaces, each with a thumbnail, name, machine type, status, and last used time. A context menu is open for the first listed space, with the 'Delete' option highlighted.

Get help

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

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

Next steps

- [Enterprise chat app GitHub repository](#)
- [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](#)

Feedback

Was this page helpful?

Yes

No

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

Get started with chat document security for Python

Article • 05/14/2024

When you build a [chat application using the 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.

An **authorized user** should have access to answers contained within the documents of the chat app.

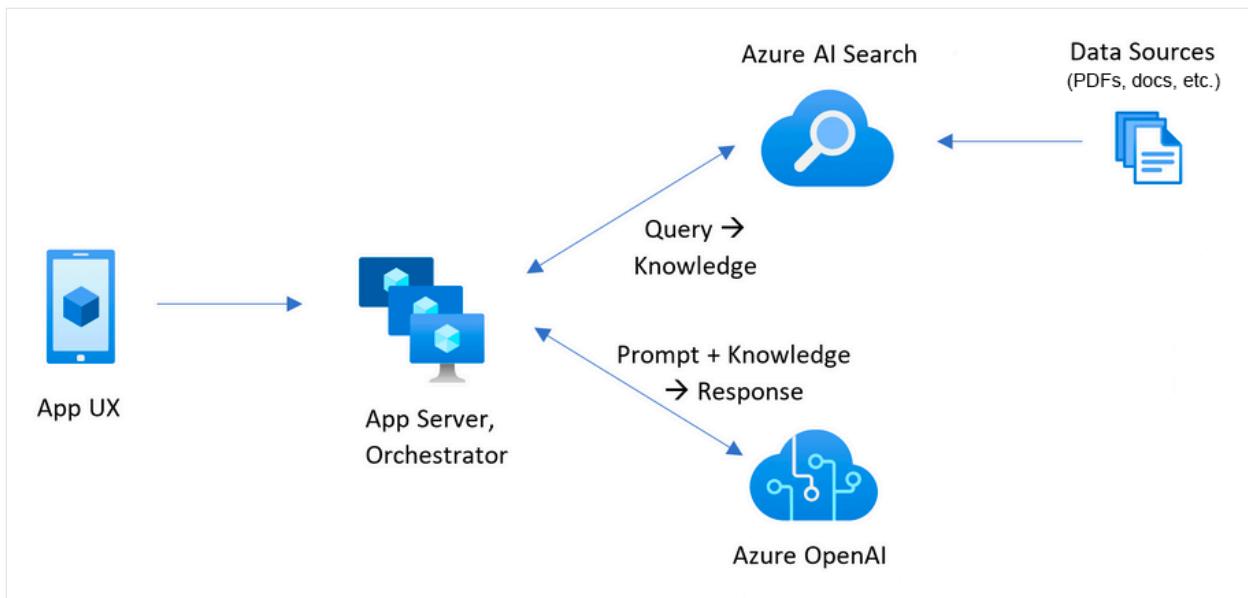
The screenshot shows a web-based AI chat interface. At the top, there is a navigation bar with links for "GPT + Enterprise data | Sample", "Chat", "Ask a question", "Azure OpenAI + Cognitive Search", and "Logout cassie.marks@contoso.com". Below the navigation bar, there is a text input field containing the query "Summarize sales numbers for Q3 2022". The main content area displays a response from the AI, which includes a summary of sales numbers for Q3 2022 and a citation link: "1. q3-2022-sales-final.pdf". There are also icons for a star, a lightbulb, and a clipboard.

An **unauthorized user** shouldn't have access to answers from secured documents they don't have authorization to see.

The screenshot shows a web-based AI chat interface. At the top, there is a navigation bar with links for "GPT + Enterprise data | Sample", "Chat", "Ask a question", "Azure OpenAI + Cognitive Search", and "Logout joe.smith@contoso.com". Below the navigation bar, there is a text input field containing the query "Summarize sales numbers for Q3 2022". The main content area displays a response from the AI stating, "I'm sorry, but I don't have access to any sources that provide information on sales numbers for Q3 2022." There are icons for a star, a lightbulb, and a clipboard.

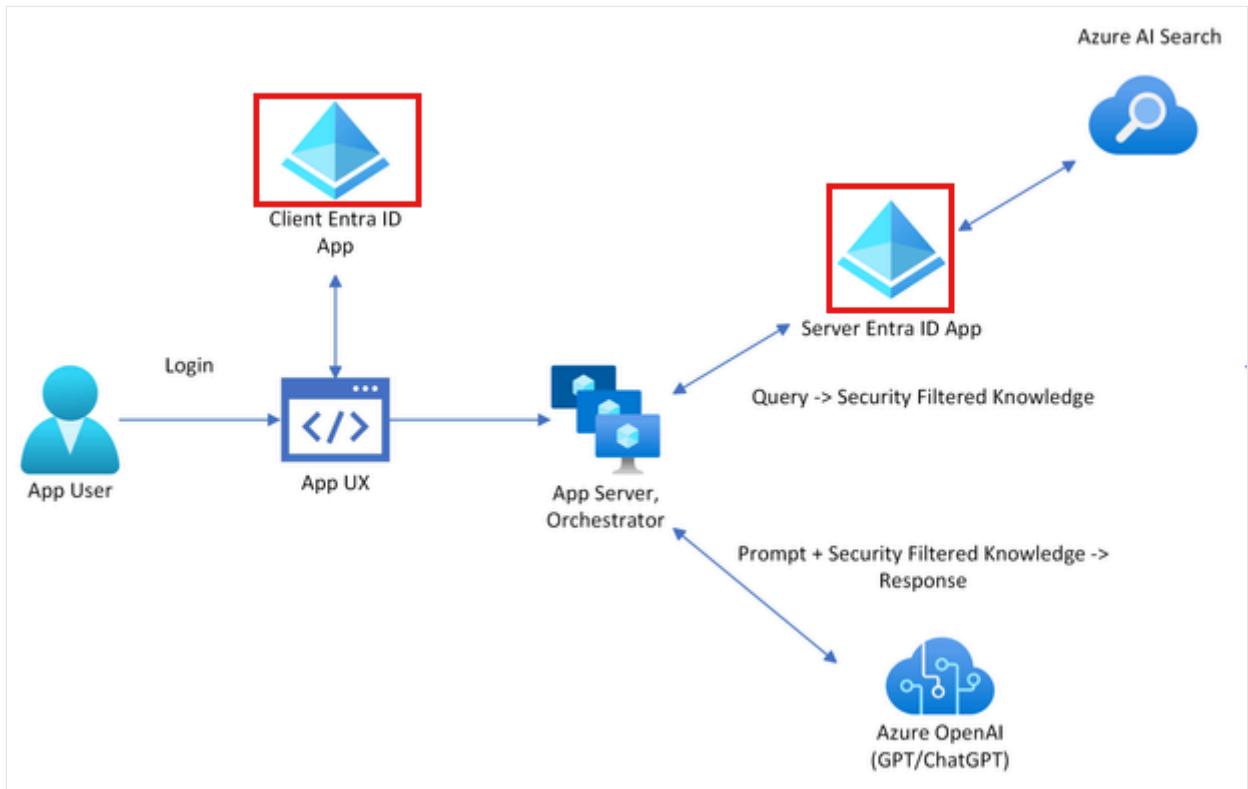
Architectural overview

Without document security feature, the enterprise chat app has a simple architecture 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.

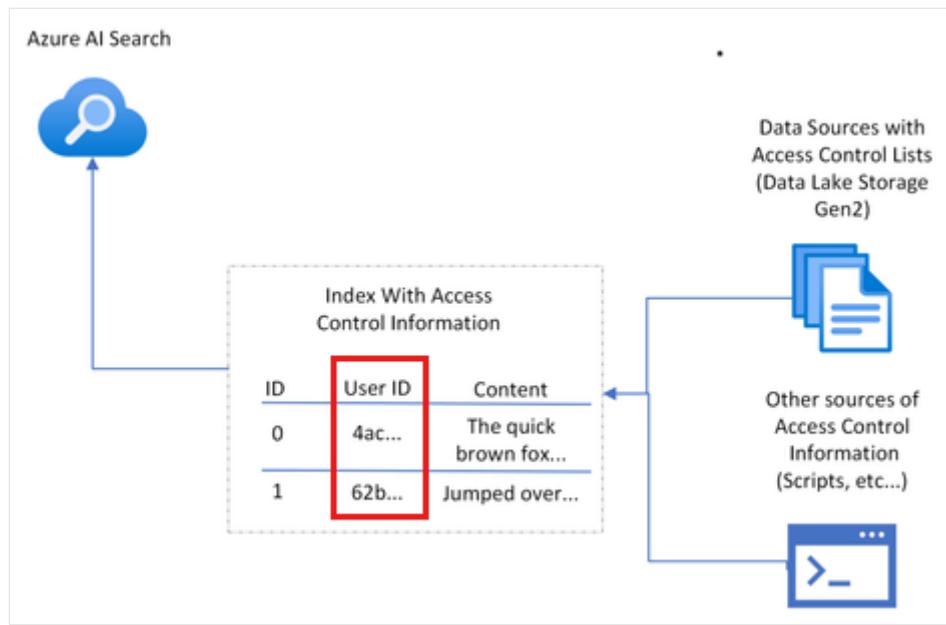


To add security for the documents, you need to update the enterprise chat app:

- Add client authentication to the chat app with Microsoft Entra.
- Add server-side logic to populate a search index which corresponds to the authenticated user's identity that should have access to each document.



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 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, then trim any documents which don't match the user. 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 productionizing requirements to scale to your needs.

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 using Visual Studio Code.

To use this article, you need the following prerequisites:

- 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](#).
- Access granted to Azure OpenAI in the desired Azure subscription. Currently, access to this service is granted only by application. You can apply for access to Azure OpenAI by completing the form at <https://aka.ms/oai/access>.

You need more prerequisites depending on your preferred development environment.

Codespaces (recommended)

1. [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 Codespaces for up to 60 hours free each month with 2 core instances. For more information, see [GitHub Codespaces monthly included storage and core hours](#).

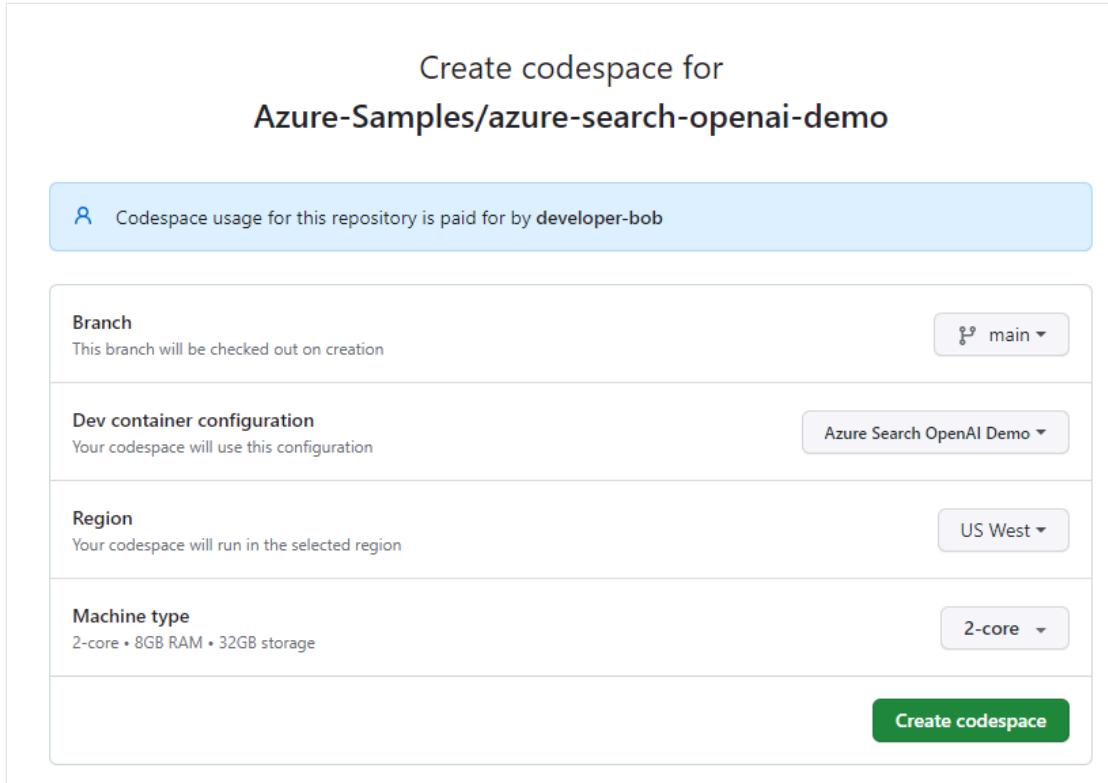
1. Start the process to create a new GitHub Codespace on the `main` branch of the [Azure-Samples/azure-search-openai-demo](#) GitHub repository.
2. Right-click on the following button, and select *Open link in new windows* in order to have both the development environment and the documentation available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



Open in GitHub Codespaces



3. On the **Create codespace** page, review the codespace configuration settings and then select **Create new 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. Complete the authentication process.
7. The remaining tasks in this article take place in the context of this development container.

Get required information with 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`.

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 environment variables for the sample to use authentication.

Console

```
azd env set AZURE_USE_AUTHENTICATION true  
azd env set AZURE_ENFORCE_ACCESS_CONTROL true  
azd env set AZURE_TENANT_ID <REPLACE-WITH-YOUR-TENANT-ID>
```

[+] Expand table

Parameter	Purpose
<code>AZURE_USE_AUTHENTICATION</code>	Enables user sign-in to the chat app. Enables <code>Use oid security filter</code> in the chat app Developer settings .
<code>AZURE_ENFORCE_ACCESS_CONTROL</code>	Requires authentication for any document access. The Developer settings for oid and group security will be turned on and disabled so they can't be disabled from the UI.
<code>AZURE_TENANT_ID</code>	The tenant which authorizes your user sign in.

2. If you need to use `AZURE_AUTH_TENANT_ID` due to a conditional access policy on your user tenant, run the following command to configure the sample to use a second tenant for application hosting.

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.

Deploy chat app to Azure

Deployment includes creating the Azure resources, uploading the documents, creating the Microsoft Entra identity apps (client & server), and turning on identity for the hosting resource.

1. Run the following Azure Developer CLI command to 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: <code>tjones-secure-chat</code> .
Subscription	Select a subscription to create the resources in.
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 is deployed to allow the app to start up.

3. After the application has been successfully deployed, you see a URL displayed in the terminal.
4. Select that 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 is displayed, notice in the top right corner that your user is signed in.
7. Open **Developer settings** and notice both these options are selected and greyed out (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.`

The screenshot shows a chat interface with the following elements:

- Header:** GPT + Enterprise data | Sample, Chat, Ask a question, Azure OpenAI + AI Search, Logout morgan@contoso.com
- Input Field:** What does a Product Manager do?
- Response Card:** The provided sources do not contain specific information about the role of a Product Manager at Contoso Electronics.
- Bottom Input Field:** Type a new question (e.g. does my plan cover annual eye exams?)
- Bottom Right:** A blue arrow icon pointing right.

Open access to a document for a user

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

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

Once 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 chap app, select **Developer settings**.

2. In the ID Token claims section, copy your `objectidentifier`. This is known in the next section as the `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 you have access to it.

Bash

```
./scripts/manageacl.sh \
-v \
--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>remove</code> , <code>remove_all</code> , <code>list</code> .
<code>--acl</code>	Group or user's <code>USER_OBJECT_ID</code>
<code>--url</code>	The file's location in Azure storage, such as <code>https://MYSTORAGENAME.blob.core.windows.net/content/role_library.pdf</code> . Don't surround URL with quotes 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 00000000-0000-0000-0000-000000000000 --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 00000000-0000-0000-0000-000000000000 to 58 search documents
```

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

Bash

```
./scripts/manageacl.sh \  
  -v \  
  --acl-type oids \  
  --acl-action list \  
  --acl <REPLACE_WITH_YOUR_USER_OBJECT_ID> \  
  --url <REPLACE_WITH_YOUR_DOCUMENT_URL>
```

[+] Expand table

Parameter	Purpose
-v	Verbose output.
--acl-type	Group or user (oids): oids
--acl-action	List a Search index field oids. Other options include remove, remove_all, list.
--acl	Group or user's USER_OBJECT_ID
--url	The file's location in Azure storage, such as https://MYSTORAGENAME.blob.core.windows.net/content/role_library.pdf . Don't surround URL with quotes 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 and is used to determine if the document is used in the answer with Azure OpenAI.

Verify Azure AI Search contains your USER_OBJECT_ID

1. Open the [Azure portal](#) and search for your AI Search.
2. Select your search resource from the list.
3. Select Search management -> Indexes.
4. Select the gptkbindex.
5. Select View -> JSON view.
6. Replace the JSON with the following JSON.

```
JSON

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

This 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 did not see the correct answer, verify your USER_OBJECT_ID is set correctly in Azure AI Search for that role_library.pdf.

1. Return to the chat app. You may 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.

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

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

GitHub Codespaces

Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement 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 into the GitHub Codespaces dashboard (<https://github.com/codespaces>).
2. Locate your currently running Codespaces sourced from the [Azure-Samples/azure-search-openai-javascript](#) GitHub repository.

The screenshot shows the GitHub Codespaces dashboard. At the top, there are navigation icons and a search bar. Below the header, there are sections for 'All' and 'Templates'. A sidebar on the left lists repositories, with 'Azure-Samples/azure-search-openai-demo' selected. The main area is titled 'Your codespaces' and contains a section for 'Explore quick start templates' with options for 'Blank', 'React', and 'Jupyter Notebook'. Below this is a section for 'Owned by developer-bob' which lists a single codespace: 'effective orbit' from 'Azure-Samples/azure-search-openai-demo'. This specific codespace entry is highlighted with a red box. It shows details like 'main' branch, 'No changes', and resource usage: '2-core + 8GB RAM + 32GB'. A context menu is open next to the codespace card, with the 'Delete' option highlighted by another red box.

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

This screenshot shows the same GitHub Codespaces dashboard as the previous one, but with a different focus. The context menu for the 'effective orbit' codespace has been fully expanded, showing options like 'Open in...', 'Rename', 'Export changes to a fork', 'Change machine type', 'Keep codespace', and 'Delete'. The 'Delete' option is highlighted with a red box. The rest of the interface is identical to the first screenshot, including the repository sidebar and the 'Owned by developer-bob' section.

Get help

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

Next steps

- [Enterprise chat app GitHub repository](#)
 - [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](#)
-

Feedback

Was this page helpful?



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

Get started with evaluating answers in a chat app

Article • 01/30/2024

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 which affects the answers, run an evaluation to compare the changes. This demo application offers tools you can use today to make it easier to run evaluations.

By following the instructions in this article, you will:

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

Architectural overview

Key components of the architecture include:

- **Azure-hosted chat app:** The chat app runs in Azure App Service. The chat app conforms to the 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:** Can generate a number of questions for each document along with the ground truth answer. The more questions, the longer the evaluation.
- **Evaluator** runs sample questions and prompts against the chat app and returns the results.
- **Review tool** allows you to review the results of the evaluations.
- **Diff tool** allows you to compare the answers between evaluations.

Prerequisites

- Azure subscription. [Create one for free ↗](#)
- Access granted to Azure OpenAI in the desired Azure subscription.

Currently, access to this service is granted only by application. You can apply for access to Azure OpenAI by completing the form at <https://aka.ms/oai/access>.

- Complete the [previous chat App procedure](#) to deploy the chat app to Azure. This procedure loads the data into the Azure AI Search resource. This resource is required for the evaluations app to work. Don't complete the **Clean up resources** section of the previous procedure.

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

- Web API URI: The URI of the deployed chat app API.
- Azure AI Search. The following values are required:
 - Resource name: The name of the Azure AI Search resource name.
 - Index name: The name of the Azure AI Search index where your documents are stored.
 - Query key: The key to query your Search index.
- If you experimented with the chat app authentication, you need to disable user authentication so the evaluation app can access the chat app.

Once you have this information collected, you shouldn't need to use the **chat app** development environment again. It's referred to later in this article several times to indicate how the **chat app** is used by the **Evaluations app**. Don't delete the **chat app** resources until you complete the entire procedure in this article.

- 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 using Visual Studio Code.

Codespaces (recommended)

- GitHub account

Open development environment

Begin now with a development environment that has all the dependencies installed to complete this article. You should arrange your monitor workspace so you can see both this documentation and the development environment at the same time.

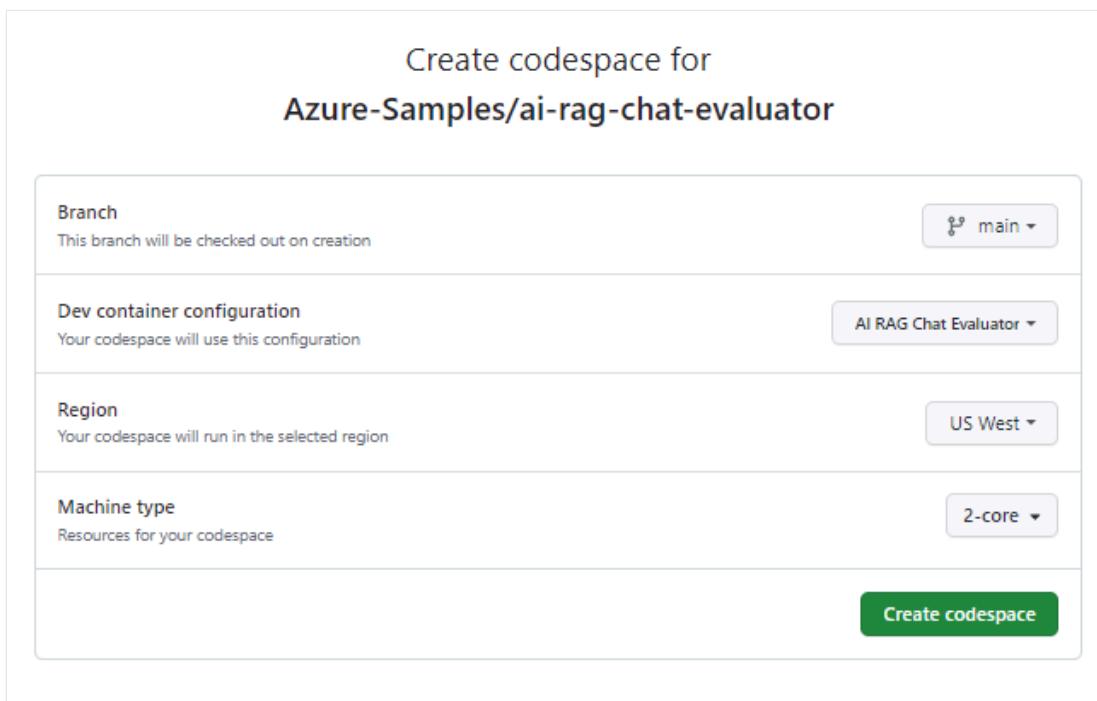
GitHub Codespaces (recommended)

[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 Codespaces for up to 60 hours free each month with 2 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. Right-click on the following button, and select *Open link in new window* in order to have both the development environment and the documentation available at the same time.
[Open this project in GitHub Codespaces](#)
3. On the **Create codespace** page, review the codespace configuration settings and then select **Create new codespace**



The screenshot shows the 'Create codespace for Azure-Samples/ai-rag-chat-evaluator' dialog. It includes fields for Branch (set to main), Dev container configuration (set to AI RAG Chat Evaluator), Region (set to US West), and Machine type (set to 2-core). A large green 'Create codespace' button is at the bottom right.

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

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, for the evaluations app.

```
Bash
```

```
azd up
```

This doesn't deploy the evaluations app, but it does create the **Azure OpenAI** resource with a GPT-4 deployment that's required to run the evaluations locally in the development environment.

8. The remaining tasks in this article take place in the context of this development container.
9. The name of the GitHub repository is shown in the search bar. This helps you distinguish between this 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. Use the following command to get the **Evaluations** app resource information into a `.env` file:

```
Bash
```

```
azd env get-values > .env
```

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

```
Bash
```

```
AZURE_SEARCH_SERVICE="<service-name>"  
AZURE_SEARCH_INDEX="<index-name>"  
AZURE_SEARCH_KEY="<query-key>"
```

The `AZURE_SEARCH_KEY` value is the **query key** for the Azure AI Search instance.

3. Copy the `example_config.json` file at the root of the **Evaluations** app folder into a new file `my_config.json`.
4. Replace the existing content of `my_config.json` with the following content:

```
JSON
```

```
{  
    "testdata_path": "my_input/qa.jsonl",  
    "results_dir": "my_results/experiment<TIMESTAMP>",  
    "target_url": "http://localhost:50505/chat",  
    "target_parameters": {  
        "overrides": {  
            "semantic_ranker": false,  
            "prompt_template": "<READFILE>my_input/prompt_refined.txt"  
        }  
    }  
}
```

5. Change the `target_url` 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

In order 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 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

```
python3 -m scripts generate --output=my_input/qa.jsonl --  
numquestions=14 --persource=2
```

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

Note

The few number of questions and answers per source is meant to allow you to quickly complete this procedure. It isn't meant to be a production evaluation which should have more questions and answers per source.

Run first evaluation with a refined prompt

1. Edit the `my_config.json` config file properties:

- Change `results_dir` to include the name of the prompt:
`my_results/experiment_refined.`
- Change `prompt_template` to: `<READFILE>my_input/experiment_refined.txt` to use the refined prompt template in the evaluation.

The refined prompt is very 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 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
```

```
python3 -m scripts evaluate --config=my_config.json --numquestions=14
```

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

- `eval_results.jsonl`: Each question and answer, along with the GPT metrics for each QA pair.
- `summary.json`: The overall results, like the average GPT metrics.

Run second evaluation with a weak prompt

1. Edit the `my_config.json` config file properties:

- Change `results_dir` to: `my_results/experiment_weak`
- Change `prompt_template` to: `<READFILE>my_input/prompt_weak.txt` to use the weak prompt template in the next evaluation.

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
```

```
python3 -m scripts evaluate --config=my_config.json --numquestions=14
```

Run third evaluation with a specific temperature

Use a prompt which allows for more creativity.

1. Edit the `my_config.json` config file properties:

- Change `results_dir` to: `my_results/experiment_ignoresources_temp09`
- Change `prompt_template` to: `<READFILE>my_input/prompt_ignoresources.txt`
- Add a new override, `"temperature": 0.9` - 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 config object should like the following except use your own `results_dir`:

JSON

```
{  
    "testdata_path": "my_input/qa.jsonl",  
    "results_dir": "my_results/experiment_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

```
python3 -m scripts evaluate --config=my_config.json --numquestions=14
```

Review the evaluation results

You have 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

```
python3 -m review_tools summary my_results
```

2. The results look 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.

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

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

Value	Description
Groundedness	This refers to 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	This 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	This refers to how logically consistent the model's responses are. A coherent response maintains a logical flow and doesn't contradict itself.
Citation	This indicates if the answer was returned in the format requested in the prompt.
Length	This measures the length of the response.

4. The results should indicate all 3 evaluations had high relevance while the `experiment_ignoresources_temp09` had the lowest relevance.

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

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

Compare the answers

Compare the returned answers from the evaluations.

1. Select two of the evaluations to compare, then use the same review tool to compare the answers:

```
Bash
```

```
python3 -m review_tools diff my_results/experiment_refined  
my_results/experiment_ignoresources_temp09
```

2. Review the results.

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

	experiment_refined	experiment_ignoresources_temp09
When choosing an out-of-network provider or services not covered under the Northwind Standard plan, there are a few things you should expect:	<ol style="list-style-type: none">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	<p>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.</p> <p>To minimize your out-of-pocket costs and ensure that you are receiving the best care possible, you should</p>
groundedness	5	5
relevance	5	5
coherence	5	5

3. Enter **Ctrl** + **c** 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 impact 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. For example, `Please answer in about 3 sentences.`

Clean up resources

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

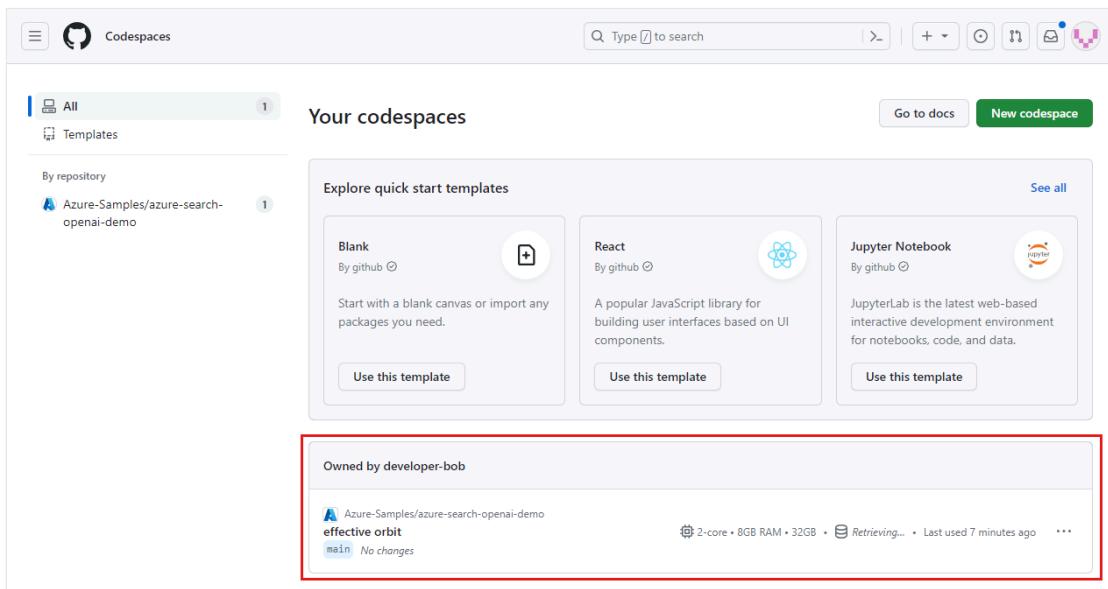
GitHub Codespaces

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

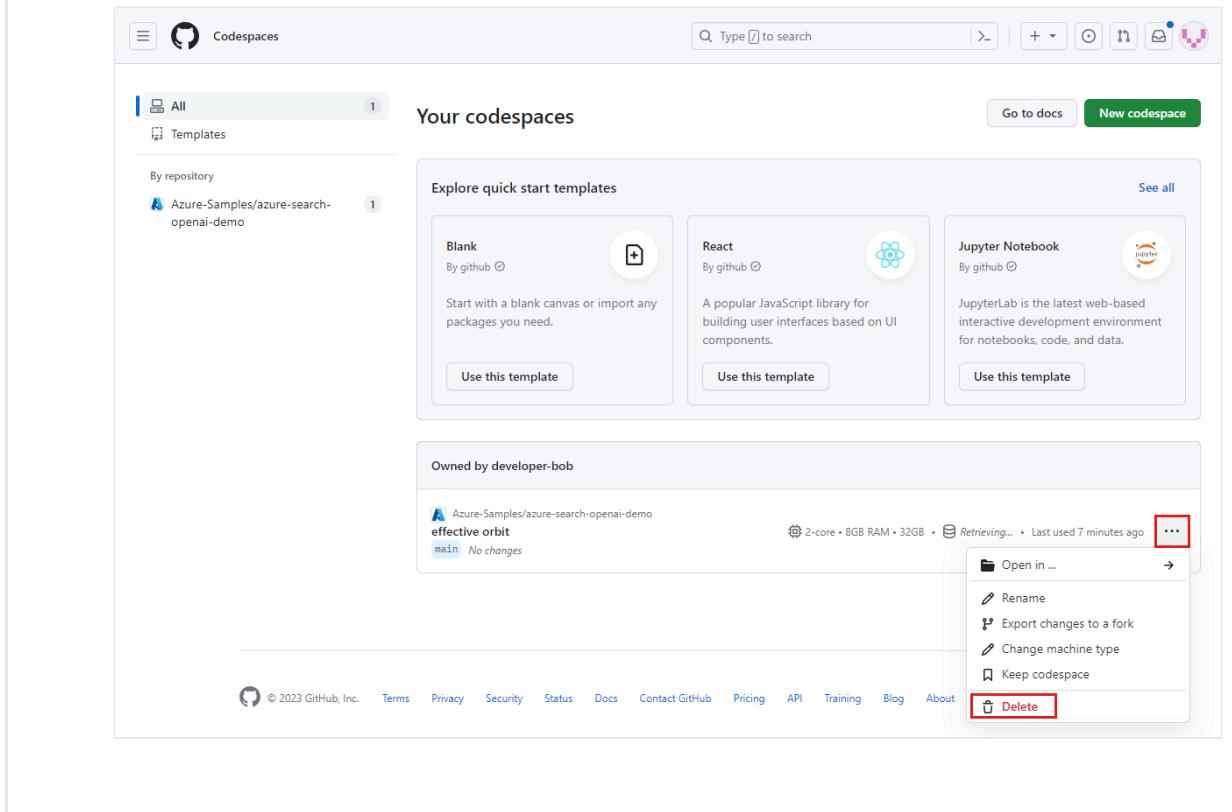
ⓘ Important

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

1. Sign into the GitHub Codespaces dashboard (<https://github.com/codespaces>).
2. Locate your currently running Codespaces sourced from the [Azure-Samples/ai-rag-chat-evaluator](#) GitHub repository.



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



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

- [Javascript](#)
- [Python](#)

Next steps

- [Evaluations repository ↗](#)
- [Enterprise chat app GitHub repository ↗](#)
- [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 ↗](#)

Feedback

Was this page helpful?

Yes

No

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

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

Article • 05/13/2024

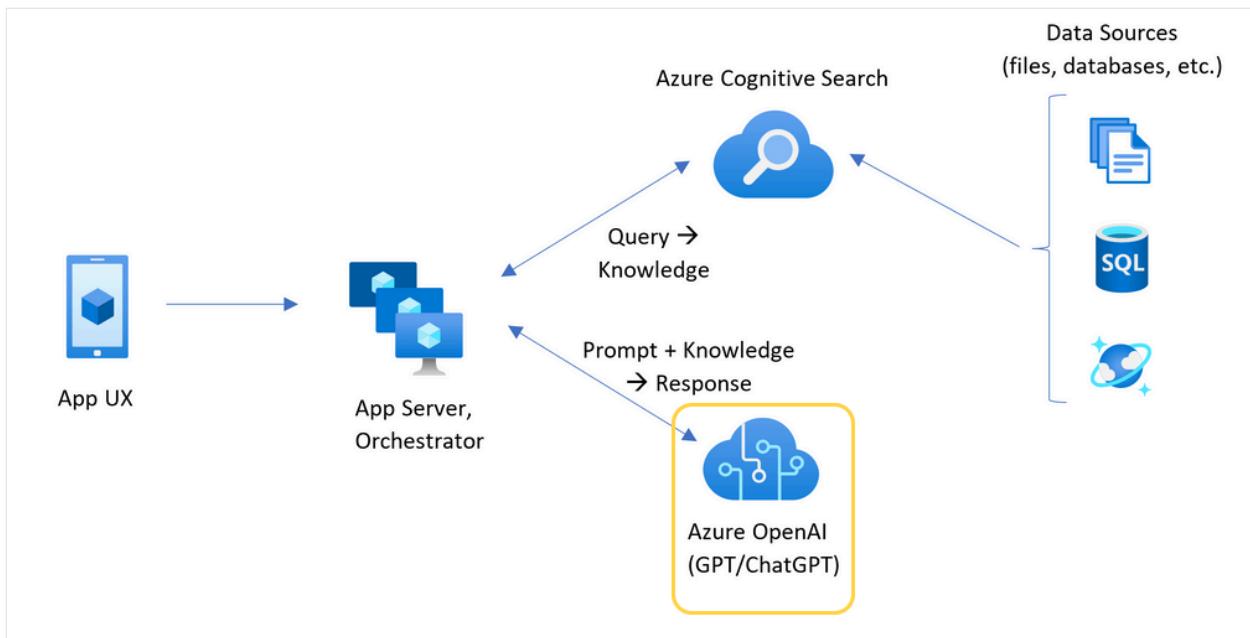
Learn how to add load balancing to your application to extend the chat app beyond the Azure OpenAI token and model quota limits. This approach uses Azure Container Apps to create three Azure OpenAI endpoints, as well as a primary container to direct incoming traffic to one of the three endpoints.

This article requires you to deploy 2 separate samples:

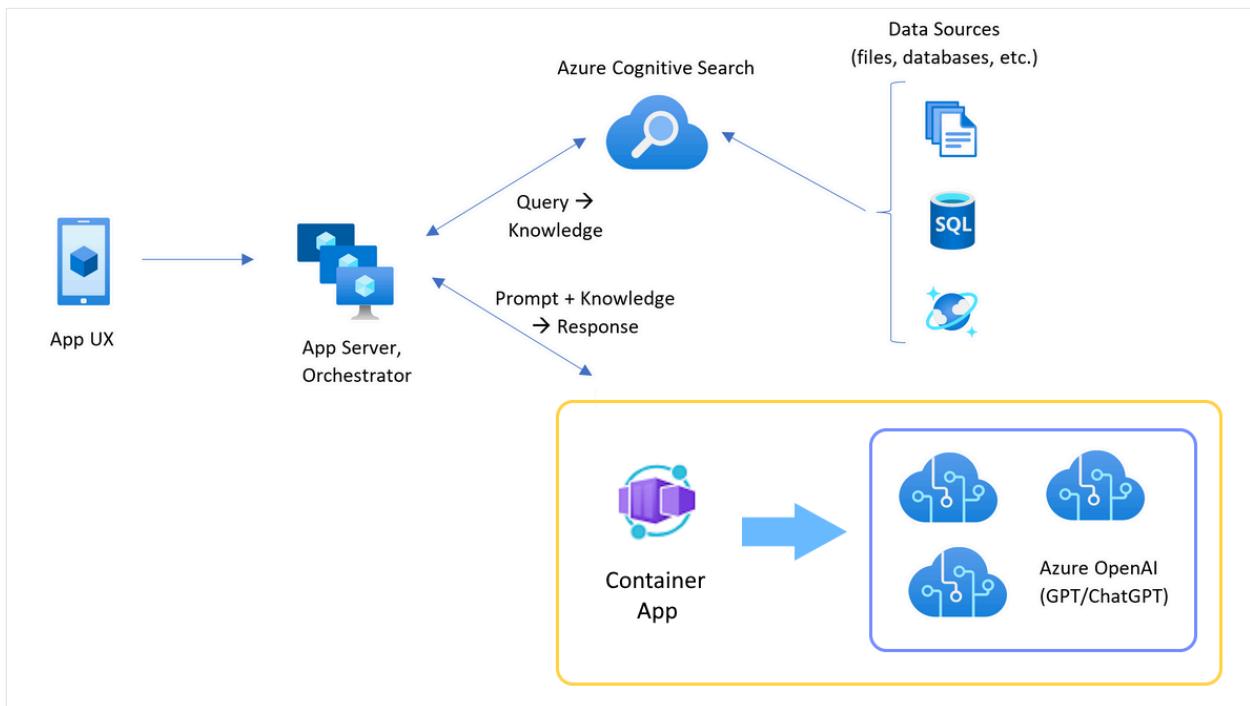
- Chat app
 - If you haven't deployed the chat app yet, wait until after the load balancer sample is deployed.
 - If you have already deployed the chat app once, you'll change the environment variable to support a custom endpoint for the load balancer and redeploy it again.
 - Chat app available in these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

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 using a single Azure OpenAI resource is prone to have conversation failures due to those limits.

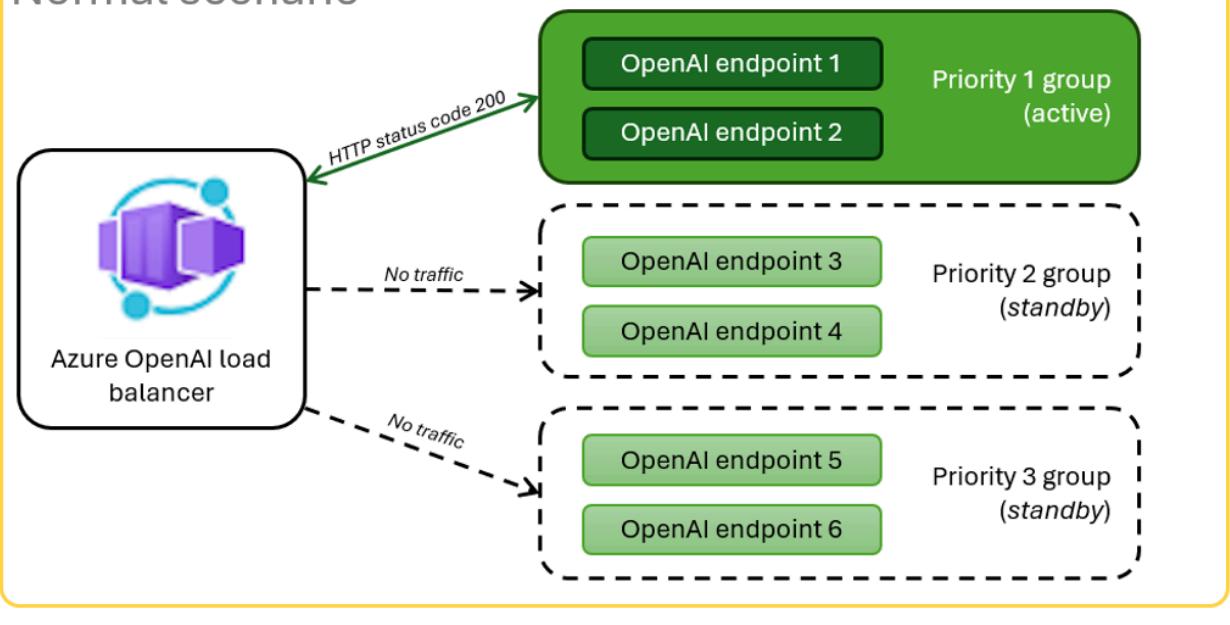


To use the chat app without hitting those limits, use a load balanced solution with Azure Container Apps. This solution seamlessly exposes a single endpoint from Azure Container Apps to your chat app server.



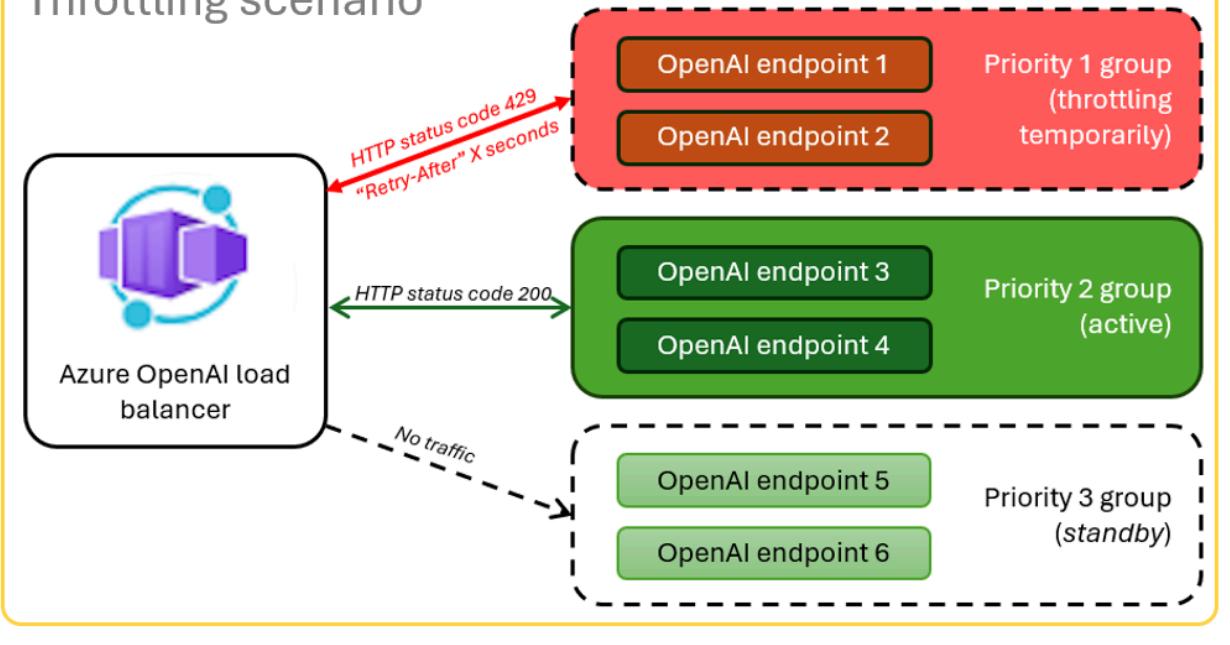
The Azure Container app sits in front of a set of Azure OpenAI resources. The Container app solves two scenarios: normal and throttled. During a **normal scenario** where token and model quota is available, the Azure OpenAI resource returns a 200 back through the Container App and App Server.

Normal scenario



When a resource is in a **throttled scenario** such as due to quota limits, the Azure Container app can retry a different Azure OpenAI resource immediately to fulfill the original chat app request.

Throttling scenario



Prerequisites

- Azure subscription. [Create one for free ↗](#)
- Access granted to Azure OpenAI in the desired Azure subscription.

Currently, access to this service is granted only by application. You can apply for access to Azure OpenAI by completing the form at <https://aka.ms/oai/access>.

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

Codespaces (recommended)

- GitHub account

Open Container apps local balancer sample app

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 Codespaces for up to 60 hours free each month with 2 core instances. For more information, see [GitHub Codespaces monthly included storage and core hours](#).

Deploy 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
```

You'll need to select a subscription and region for the deployment. These don't have to be the same subscription and region as the chat app.

4. Wait for the deployment to complete before continuing.

Get the deployment endpoint

1. Use the following command to display the deployed endpoint for the Azure Container app.

```
Bash
```

```
azd env get-values
```

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

Redeploy Chat app with load balancer endpoint

These are completed on the chat app sample.

Initial deployment

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

 Expand table

Language	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 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 OpenAI requests.

```
Bash
azd env set OPENAI_HOST azure_custom
```

5. Add the following environment variable, substituting `<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 OpenAI request.

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

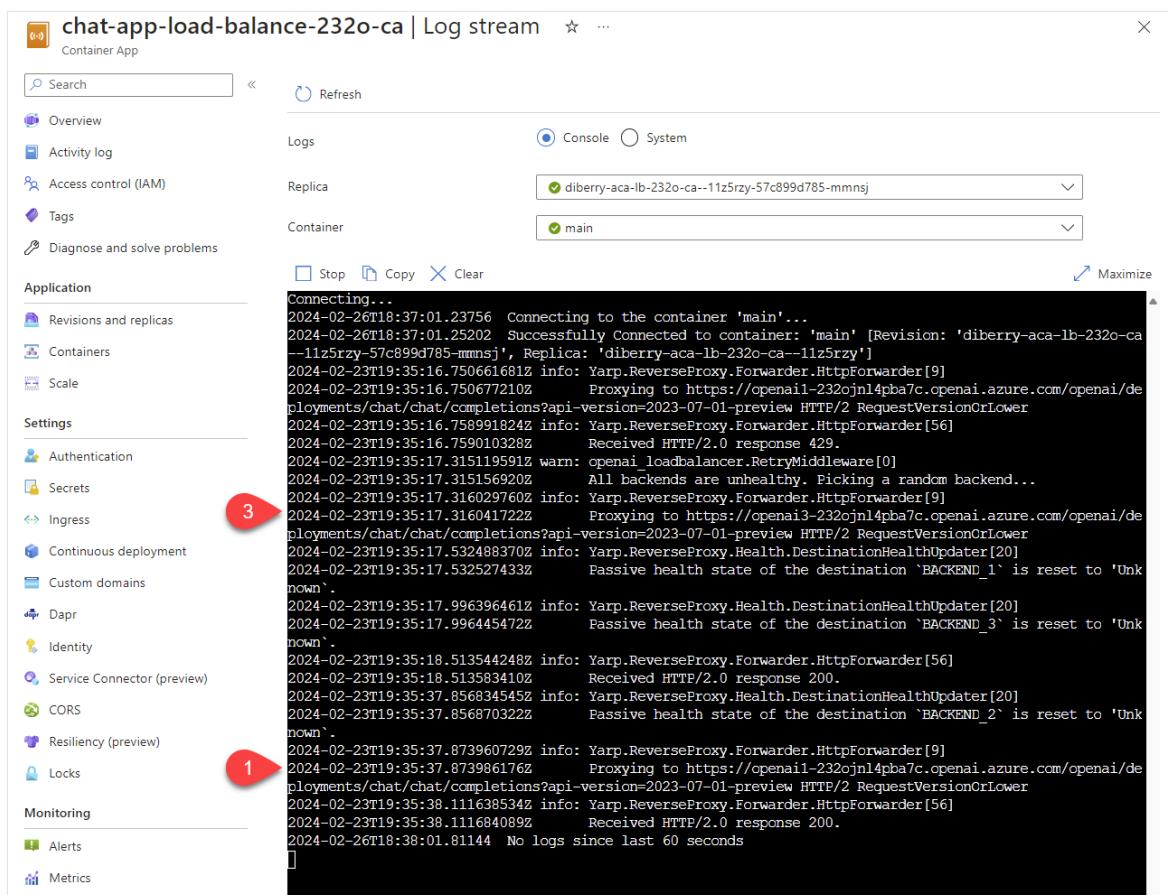
6. Deploy the chat app.

```
Bash
azd up
```

You can now use the chat app with the confidence that it's built to scale 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 Container App 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 beginning with `Proxying to https://openai3`, where `3` indicates the third Azure OpenAI resource.



6. As you use the chat app, when the load balancer receives status that the request has exceeded quota, the load balancer automatically rotates to another resource.

Configure the tokens per minute quota (TPM)

By default, each of the OpenAI instances in the load balancer will be deployed with 30,000 TPM (tokens per minute) capacity. You can use the chat app with the confidence that it's built to scale across many users without running out of quota. Change this value when:

- You get deployment capacity errors: lower that value.
- Planning 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 done with both 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 those resources.

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

Clean upload balancer resources

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

```
Bash
```

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

The switches provide:

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

Clean up GitHub Codespaces

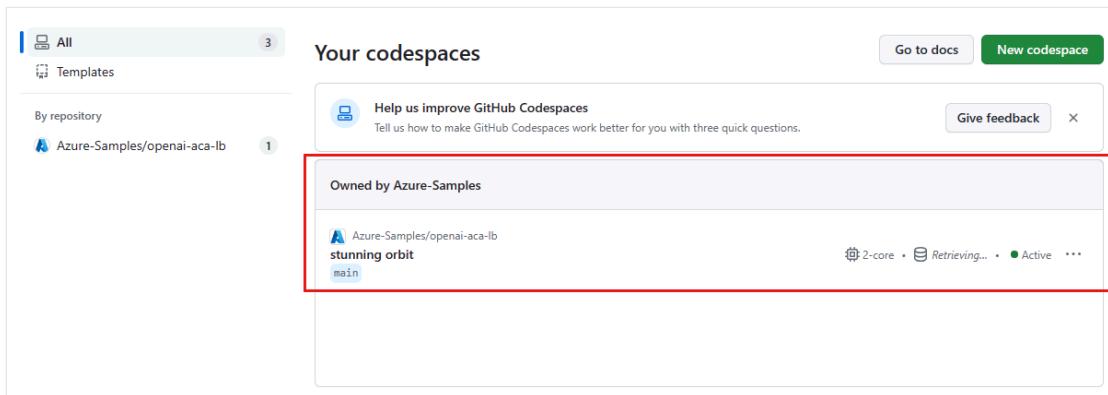
GitHub Codespaces

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

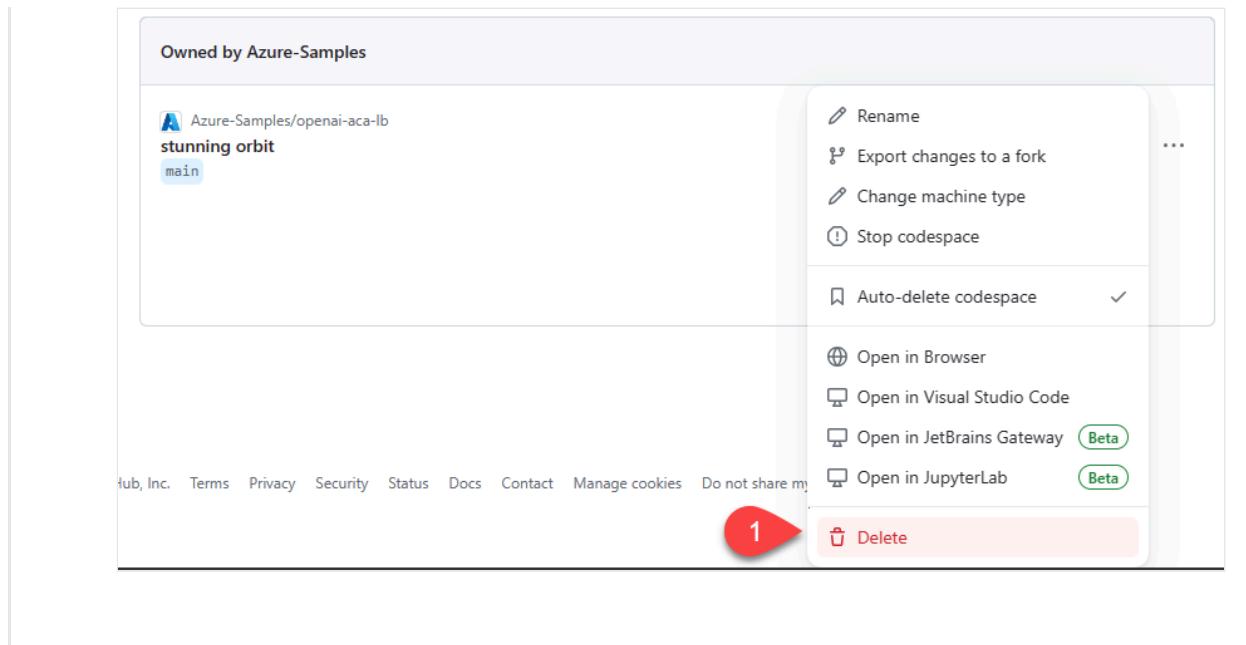
ⓘ Important

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

1. Sign into the GitHub Codespaces dashboard (<https://github.com/codespaces>).
2. Locate your currently running Codespaces sourced from the [azure-samples/openai-aca-lb](#) GitHub repository.



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



Get help

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

Sample code

Samples used in this article include:

- [Python chat app with RAG](#)
- [Load Balancer with Azure Container Apps](#)

Next step

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

Feedback

Was this page helpful?

Yes

No

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

Scale Azure OpenAI for Python with Azure API Management

Article • 04/01/2024

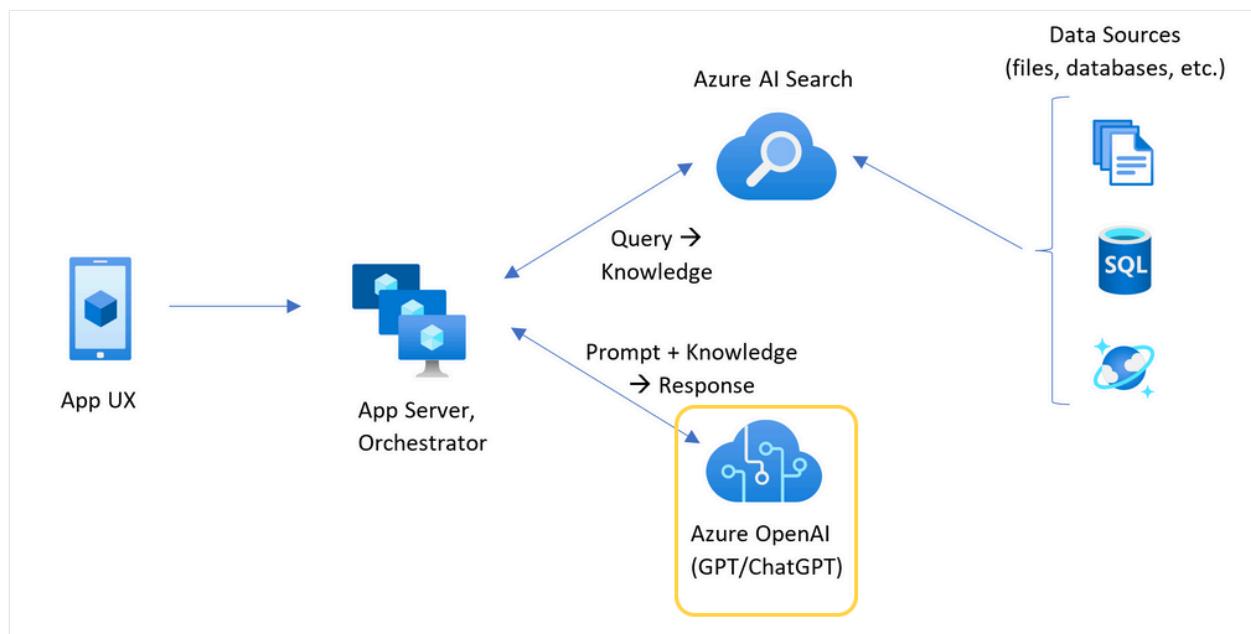
Learn how to add enterprise-grade load balancing to your application to extend the chat app beyond the Azure OpenAI token and model quota limits. This approach uses Azure API Management to intelligently direct traffic between three Azure OpenAI resources.

This article requires you to deploy 2 separate samples:

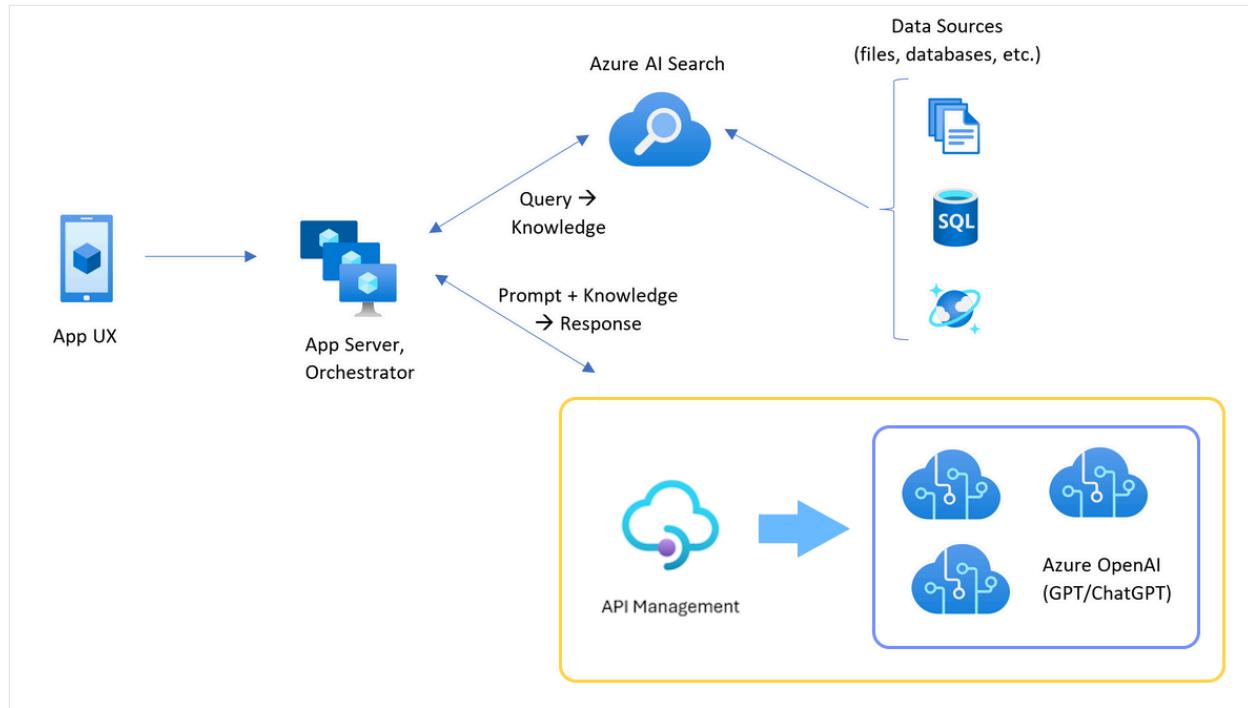
- Chat app
 - If you haven't deployed the chat app yet, wait until after the load balancer sample is deployed.
 - If you have already deployed the chat app once, you'll change the environment variable to support a custom endpoint for the load balancer and redeploy it again.
- Load balancer with Azure API Management

Architecture for load balancing Azure OpenAI with Azure API Management

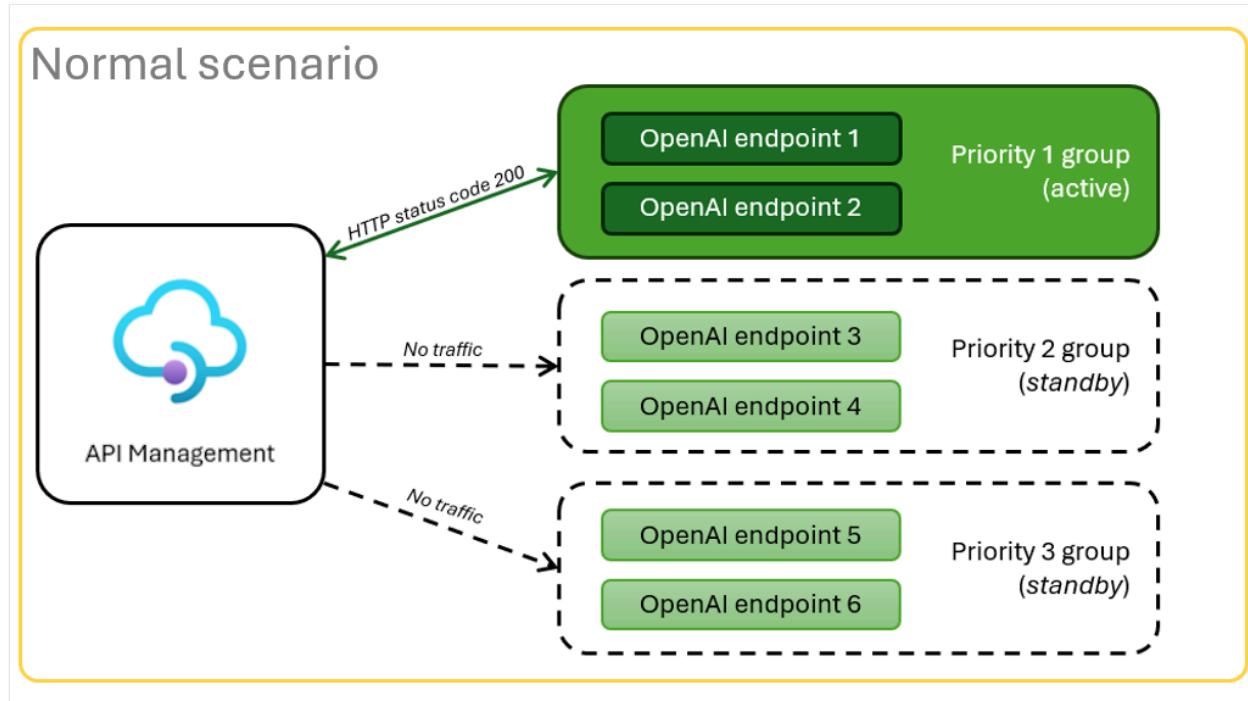
Because the Azure OpenAI resource has specific token and model quota limits, a chat app using a single Azure OpenAI resource is prone to have conversation failures due to those limits.



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

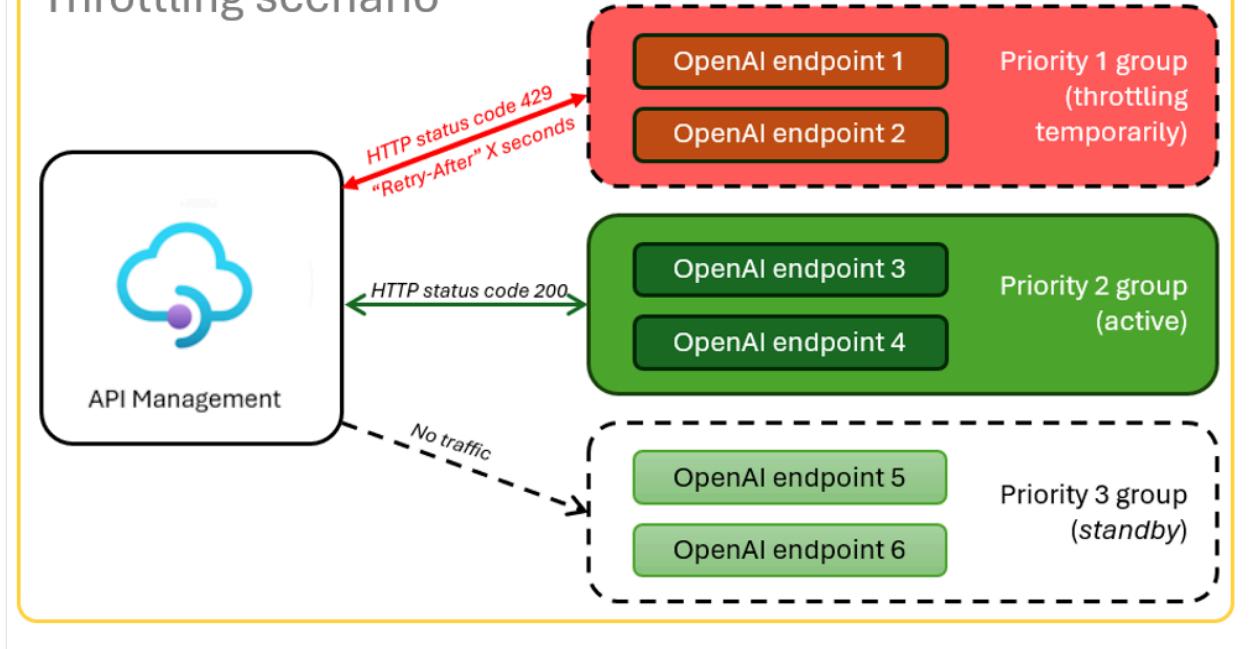


The Azure API Management resource, as an API layer, sits in front of a set of Azure OpenAI resources. The API layer applies to two scenarios: normal and throttled. During a **normal scenario** where token and model quota is available, the Azure OpenAI resource returns a 200 back through the API layer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



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

Throttling scenario



Prerequisites

- Azure subscription. [Create one for free ↗](#)
- Access granted to Azure OpenAI in the desired Azure subscription.

Currently, access to this service is granted only by application. You can apply for access to Azure OpenAI by completing the form at [https://aka.ms/oai/access ↗](https://aka.ms/oai/access).

- [Dev containers ↗](#) are available for both samples, with all dependencies required to complete this article. You can run the dev containers in GitHub Codespaces (in a browser) or locally using Visual Studio Code.

Codespaces (recommended)

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

Open Azure API Management local balancer sample app

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 Codespaces for up to 60 hours free each month with 2 core instances. For more information, see [GitHub Codespaces monthly included storage and core hours](#).

Deploy Azure API Management load balancer

1. To deploy the load balancer to Azure, sign in to Azure Developer CLI (AZD).

Bash

```
azd auth login
```

2. Finish the sign in instructions.

3. Deploy the load balancer app.

Bash

```
azd up
```

You'll need to select a subscription and region for the deployment. These don't have to be the same subscription and region as the chat app.

4. Wait for the deployment to complete before continuing. This may take up to 30 minutes.

Get 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 Chat app with load balancer endpoint

These are completed on the chat app sample.

Initial deployment

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

[Expand table](#)

Language	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 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 OpenAI requests.

Bash

```
azd env set OPENAI_HOST azure_custom
```

5. Add the following environment variable, which tells the Chat app's backend what the value is of the custom URL for the OpenAI request.

```
Bash
```

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

6. Deploy the chat app.

```
Bash
```

```
azd up
```

Configure the tokens per minute quota (TPM)

By default, each of the OpenAI instances in the load balancer will be deployed with 30,000 TPM (tokens per minute) capacity. You can use the chat app with the confidence that it's built to scale across many users without running out of quota. Change this value when:

- You get deployment capacity errors: lower than that value.
- Planning 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 done with both 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 those resources.

- .NET
- Javascript
- Python

Clean up load balancer resources

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

Bash

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

The switches provide:

- `purge`: Deleted resources are immediately purged. This allows you to reuse the Azure OpenAI TPM.
- `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 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 into the GitHub Codespaces dashboard (<https://github.com/codespaces>).

2. Locate your currently running Codespaces sourced from the [azure-samples/openai-apim-lb](#) GitHub repository.

The screenshot shows the GitHub Codespaces interface. On the left, there's a sidebar with 'All' selected, followed by 'Templates'. Below that, it says 'By repository' and lists 'Azure-Samples/openai-apim-lb' with a red notification badge. The main area is titled 'Your codespaces' and contains a single card. The card has a header 'Owned by Azure-Samples' and a sub-header 'Azure-Samples/openai-apim-lb didactic bassoon main No changes'. To the right of the card, it shows resource details: '2-core • 8GB RAM • 32GB • 2.74 GB • Last used 29 minutes ago' and three dots for more options. A red box highlights the entire card area.

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

The screenshot shows the context menu for a specific codespace. The menu items are: 'Rename', 'Export changes to a fork', 'Change machine type', 'Auto-delete codespace' (with a checked checkbox), 'Open in Browser', 'Open in Visual Studio Code', 'Open in JetBrains Gateway' (marked as 'Beta'), and 'Open in JupyterLab' (marked as 'Beta'). At the bottom of the menu, there is a red button labeled 'Delete'. A red callout bubble with the number '1' points to this 'Delete' button.

Get help

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

Sample code

Samples used in this article include:

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

Next step

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

Feedback

Was this page helpful?



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

Load testing Python chat app using RAG with Locust

Article • 03/19/2024

This article provides the process to perform load testing on a Python chat application using the 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 does not exceed the current Azure OpenAI 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.

Watch the demonstration video to understand more about load testing the chat app.

- [Video ↗](#)

Prerequisites

- Azure subscription. [Create one for free ↗](#)
- Access granted to Azure OpenAI in the desired Azure subscription. Currently, access to this service is granted only by application. You can apply for access to Azure OpenAI by completing the form at [https://aka.ms/oai/access ↗](https://aka.ms/oai/access).
- [Dev containers ↗](#) are available for both samples, with all dependencies required to complete this article. You can run the dev containers in GitHub Codespaces (in a browser) or locally using Visual Studio Code.

Codespaces (recommended)

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

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

Open Load test sample app

The load test is in Python chat app repository. You need to return to that dev container to complete these steps.

Run the test

1. Install the dependencies for the load test.

```
Bash
```

```
python3 -m pip install -r requirements-dev.txt
```

2. Start Locust, which uses the Locust test file: [locustfile.py](#) found at the root of the repository.

```
Bash
```

```
locust
```

3. Open the running Locust web site such as <http://localhost:8089>.

4. Enter the following in the Locust web site.

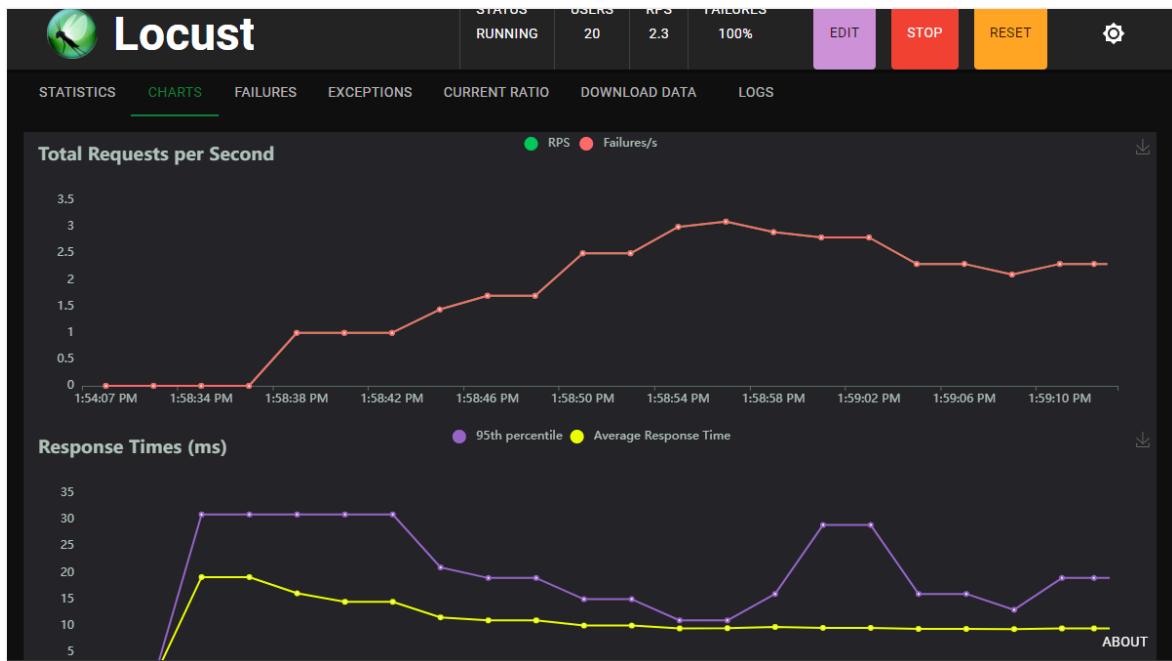
[\[+\] Expand table](#)

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

The image shows the Locust web interface for starting a new load test. At the top, there's a header with the Locust logo and navigation links for HOST, STATUS (READY), RPS (0), FAILURES (0%), and settings. Below the header, there's a form titled "Start new load test" with three main input fields: "Number of users (peak concurrency)" set to 20, "Ramp Up (users started/second)" set to 1, and "Host" set to <https://app-backend-1234.azurewebsites.net>. There's also a "Advanced options" dropdown and a large green "START SWARM" button at the bottom.

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

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



Clean up resources

When you're done 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 resource 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, log your issue to the [repository's Issues ↗](#).

Feedback

Was this page helpful?

Yes

No

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

Configure your local environment for deploying Python web apps on Azure

Article • 07/26/2023

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

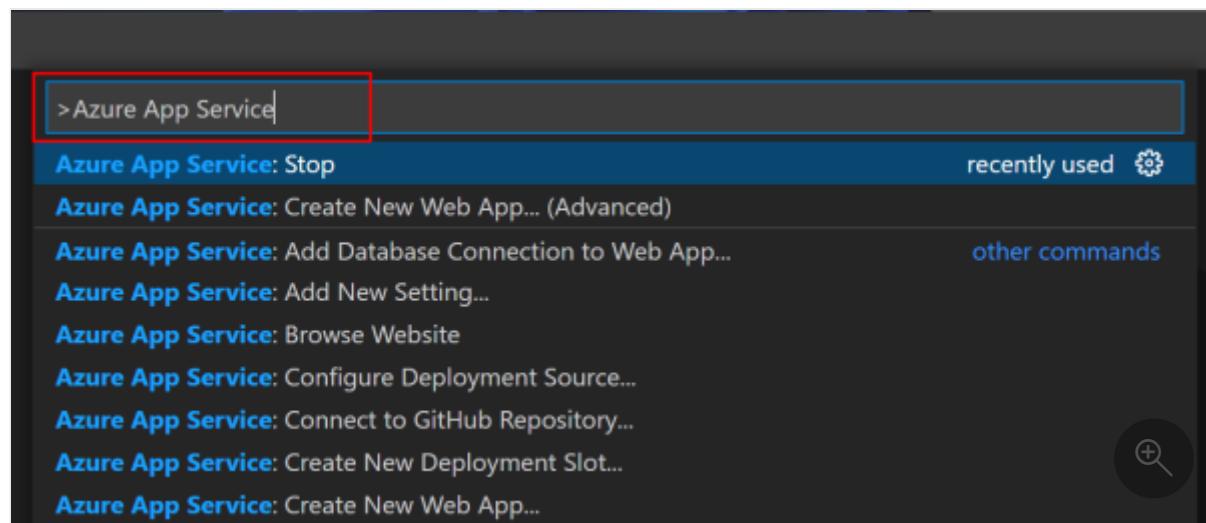
Make sure you have [Python](#) extension installed. For an overview of working with Python in VS Code, see [Getting Started with Python in VS Code](#).

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 IDE interface. The top menu bar includes File, Edit, View, Navigate, Code, Behavior, Run, Tools, VCS, Window, Help. The title bar indicates the current file is 'flask-hello-world' and the path is 'C:\Users\iadanza_gerolamo\PycharmProjects\flask-hello-world\requirements.txt'. The main area shows a 'Project' view with files like 'flask-hello-world', '.venv', 'app.py', and 'requirements.txt'. Below the project tree is a code editor with Python code for a Flask application. The bottom part of the interface is a terminal window with the following text:

```
Terminal: Command Prompt + 
(base) 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-ad8f7232284684a2fb982d4de098fe' doesn't exist
Creating Resource group 'flask-web-app' ...
Resource group creation complete
Creating AppServicePlan 'mango-smoke_asp_7995' ...
Creating webapp 'mango-smoke-ad8f7232284684a2fb982d4de098fe' ...
```

Azure CLI commands

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

Command	Description
az webapp	Manages web apps. Includes the subcommands <code>create</code> to create a web app and the <code>up</code> to create and deploy from a local workspace

Command	Description
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 create to a resource group to put your Azure resources in.
az appservice	Manages App Service plans.
az config	Managed 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-practice 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.

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 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

Overview of the Python Web azd Templates

Article • 09/27/2023

The Python web Azure Developer CLI (`azd`) templates are the fastest and easiest way to get started with building and deploying Python web applications to Azure. This article provides contextual background information as you're getting started.

The best way to get started is to [follow the quickstart](#) to create your first Python web application and deploy it to Azure in minutes with `azd` templates. If you don't want to set up a local development environment, you can still follow the [quickstart using GitHub Codespaces](#) instead.

What are the Python web azd templates?

There are many `azd` templates available on the [Awesome AZD Templates gallery](#). However, this collection of Python web `azd` templates is unique inasmuch as they provide a sample web application with feature parity across many different popular combinations of Azure resources and Python web frameworks.

When you run a Python web `azd` template, you'll:

- **Create a starter application** - Specifically, a website for a fictitious company named Relecloud. The project code features many best practices for the given Python frameworks and packages that are required for that particular stack of technologies. The template is intended to be a starting point for your application. You add or remove application logic and Azure resources as needed.
- **Provision Azure resources** - The template provisions Azure resources for hosting your web app and database using Bicep, a popular infrastructure-as-code tool. Again, you [modify the Bicep templates](#) if you need to add more Azure services.
- **Deploy the starter application to the newly provisioned Azure resources** - The starter application is automatically deployed so you can see it all working in minutes and decide what you want to modify.
- **Optional: Set up a GitHub repository and a CI/CD pipeline** - If you like, the template contains the logic to set up a GitHub repository for you including a GitHub Actions CI/CD pipeline. Within minutes, you're able to make changes to the web project code. When you merge those changes to the *main* branch of your GitHub repo, the CI/CD pipeline publishes them to your new Azure hosting environment.

Who is this for?

The templates are intended to be used by experienced Python web developers who want to start building a new Python web application targeting Azure deployment.

Why would I want to use this?

Using the `azd` templates provides several benefits:

- **Fastest possible start** - With your local development environment and hosting environment setups out of the way, you can focus on building your application within minutes.
- **Easiest possible start** - Execute just a few command line instructions to build out an entire local development, hosting and deployment environment. The workflow is easy to use and easy to remember.
- **Build on Best practices** - Each template is built and maintained by Python on Azure industry veterans. Add your code following their design approaches to build on top of a solid foundation.

Index of templates

The following table lists the available Python web `azd` template monikers to use with the `azd init` command, the technologies implemented in each template, and a link to the GitHub repository if you want to contribute changes.

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	repo ↗

Template	Web Framework	Database	Hosting Platform	GitHub Repo
Apps				
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 ↗

How do the templates work?

You use various `azd` commands to perform tasks defined by an `azd` template. These commands are covered in detail in [Get started using Azure Developer CLI](#).

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

The quickstart walks you through the steps to use a specific `azd` template. It only requires you to execute five command line instructions to production hosting environment, and local development environment:

1. `azd init --template <template name>` - creates a new project from a template and creates a copy of the application code on your local computer. The command prompts you to provide an environment name (like "myapp") that is used as a prefix int the naming of the deployed resources.
2. `azd auth login` - logs you 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 `azd auth login` command is only required the first time you use the Azure Developer CLI (`azd`) per session.
3. `azd up` - provisions the cloud resources and deploys the app to those resources.
4. `azd deploy` - deploys changes to the application source code to resources already provisioned by `azd up`.
5. `azd down` - deletes the Azure resources and the CI/CD pipeline if it was used.

 Tip

Watch the output for `azd` prompts that you need to answer. For example, after executing the `azd up` command, you may be prompted to select a subscription if you belong to more than one. Furthermore, you will be prompted to select a region. You can change the answers to prompts by editing the environment variables stored in the `./azure/` folder of the template.

Once the template has finished, you have a personal copy of the original template where you can modify every file as needed. At a minimum, you can modify the Python project code so that the project has your design and application logic. You can also [modify the infrastructure-as-code configuration](#) if you need to change the Azure resources. See the section titled [What can I edit or delete?](#)

Optional: Modify and reprovision Azure resources

If you want to change the Azure resources that are provisioned, you can [edit the appropriate Bicep files](#) in the template and use:

6. `azd provision` - reprovisions Azure resources to the desired state as defined in the Bicep files.

Set up a CI/CD Pipeline

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

Optional : Automatically set up the GitHub Actions CI/CD pipeline

If you want to implement the GitHub Actions CI/CD pipeline functionality, use the following command:

1. `azd pipeline config` - Allows you to designate a GitHub repository and settings to enable the CI\CD pipeline. Once configured, each time code changes are merged to the *main* branch of the repository, the pipeline deploys the changes to your provisioned Azure services.

What are my other options?

If you don't want to use `azd` templates, you can deploy your Python app to Azure and create Azure resources in many ways.

You can accomplish many of resource creation and deployment steps using one of the following tools:

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

Or if you're looking for an end-to-end tutorial that features Python web development frameworks, check out:

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

Do I have to use Dev Containers?

No. The Python web `azd` templates utilize [Dev Containers ↗](#) by default. Dev Containers provide many benefits, but require some prerequisite knowledge and software. If you don't want to use Dev Containers and prefer to use your local development environment instead, 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 `azd` template can vary depending on the type of project and the underlying technology stack employed. The templates listed in this article follow a common convention:

[+] [Expand table](#)

Folder/Files	Purpose	Description
/	root directory	The root directory contains many different kinds of files and folders for many different purposes.
.azure	<code>azd</code> configuration files	Contains the environment variables that are used by the Azure Developer CLI (<code>azd</code>) commands. This folder is created after you run the <code>azd init</code> command. 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

Folder/Files	Purpose	Description
		resources you need for software development inside of Visual Studio Code.
/.github	GitHub Actions configuration	Contains the configuration settings for the optional GitHub Actions CI/CD pipeline as well as linting and tests. The <code>azure-dev.yaml</code> file can be modified or deleted if you don't want to set up the GitHub Actions pipeline using <code>azd pipeline config</code> command.
/infra	Bicep 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. See Quickstart: Scaling services deployed with the azd Python web templates using Bicep .
/src	starter project code files	Includes any templates required by the web framework, static files, .py files for the code logic and data models, a <code>requirements.txt</code> , and so on. 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	Used internally to generate the <code>azd</code> templates. You can safely delete this file.
/.gitattributes	git attributes	Provides git with important configuration about handling files and folders. You can modify this file as needed.
/.gitignore	git ignore	Tells git to ignore files and folders from being included in the repository. You can modify this file as needed.
/azure.yaml	<code>azd</code> configuration file	Contains the configuration settings for <code>azd up</code> declaring what services and project folders will be deployed. This file MUST NOT be deleted.
/*.md	markdown files	There are several markdown files for different purposes. You can safely delete these files.
/docker-compose.yml	Docker compose	Creates the container package for the application before it's deployed to Azure.
/pyproject.toml	Python build system	Contains the build system requirements of Python projects. You can modify this file to including your preferred tools (for example, to use a linter and unit testing framework).
/requirements-dev.in	pip requirements file	Used to create a development environment version of the requirements using <code>pip install -r</code> command. You can modify this file to include other packages as needed.

💡 Tip

Use good version control practices so you are able to get back to a point in time when the project was working in case you inexplicably break something.

Frequently Asked Questions

Q: I got an error when using an `azd` template. What can I do?

A: See [Troubleshoot Azure Developer CLI](#). You can also report issues on the respective `azd` template's GitHub repository.

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

Article • 09/27/2023

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.eastu
s2.azurecontainerapps.io/

SUCCESS: Your application was provisioned and deployed to Azure in 10 minute
s 45 seconds.
You can view the resources created under the resource group azdtest-rg in Az
ure Portal:
https://portal.azure.com/#@/resource/subscriptions/
/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](#)

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

Article • 12/15/2023

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](#) ↗

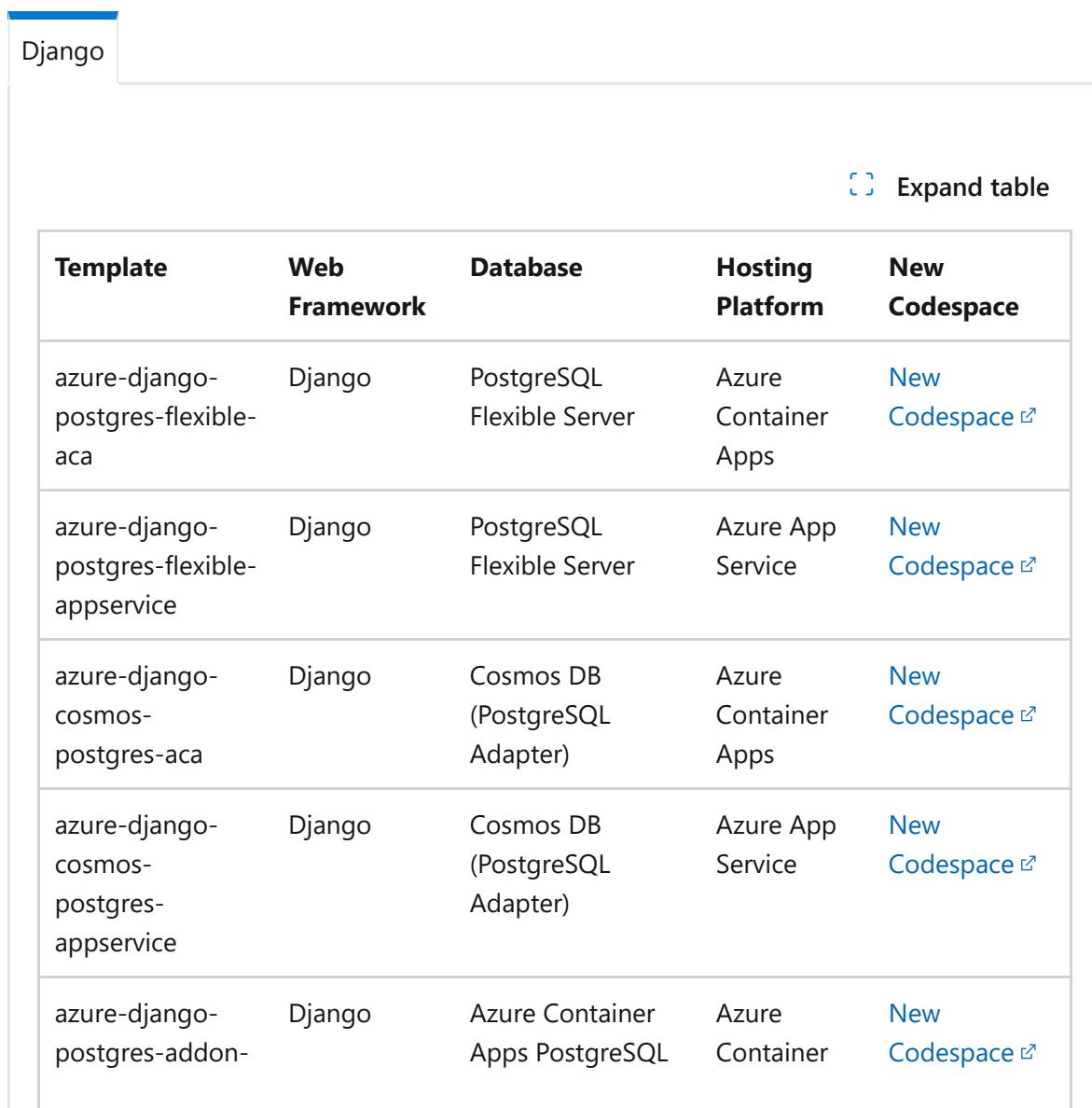
ⓘ Important

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.



The screenshot shows a table of GitHub Codespace templates. The 'Django' template is selected, indicated by a blue border around its row. The table has columns for Template, Web Framework, Database, Hosting Platform, and New Codespace. Each row contains a link to 'New Codespace'.

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.

1. 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.
2. 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](#)

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

Article • 12/19/2023

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).

Add a new service definition with Bicep

You can add a new resource to your deployment by making larger changes to the Bicep in the project. To demonstrate this, you'll add an instance of Azure Cache for Redis to your existing deployment in preparation for a fictitious new feature you plan to add at some future date.

ⓘ Important

As a result of this change, you will be paying for an instance of Azure Cache for Redis until you delete the resource in the Azure portal or using `azd down`. Details about the different service plans and their associated costs can be found on the [Azure Cache for Redis pricing page](#).

1. Create a new file in the `infra` folder named `redis.bicep`. Copy and paste the following code into the new file:

Python

```
param name string
param location string = resourceGroup().location
param tags object = {}
param keyVaultName string
param connStrKeyName string
param passwordKeyName string
param primaryKeyKeyName string

@allowed([
    'Enabled'
    'Disabled'
])
param publicNetworkAccess string = 'Enabled'

@allowed([
    'C'
    'P'
])
param skuFamily string = 'C'

@allowed([
    0
    1
    2
    3
    4
    5
    6
])
param skuCapacity int = 1

@allowed([
    'Basic'
    'Standard'
    'Premium'
])
param skuName string = 'Standard'

param saveKeysToVault bool = true
```

```

resource redis 'Microsoft.Cache/redis@2020-12-01' = {
    name: name
    location: location
    properties: {
        sku: {
            capacity: skuCapacity
            family: skuFamily
            name: skuName
        }
        publicNetworkAccess: publicNetworkAccess
        enableNonSslPort: true
    }
    tags: tags
}

resource keyVault 'Microsoft.KeyVault/vaults@2022-07-01' existing = {
    name: keyVaultName
}

resource redisKey 'Microsoft.KeyVault/vaults/secrets@2022-07-01' = if
(saveKeysToVault) {
    name: primaryKeyKeyName
    parent: keyVault
    properties: {
        value: redis.listKeys().primaryKey
    }
}

resource redisConnStr 'Microsoft.KeyVault/vaults/secrets@2018-02-14' =
if (saveKeysToVault) {
    name: connStrKeyName
    parent: keyVault
    properties: {
        value:
`${name}.redis.cache.windows.net,abortConnect=false,ssl=true,password=${redis.listKeys().primaryKey}`
    }
}

resource redisPassword 'Microsoft.KeyVault/vaults/secrets@2018-02-14' =
if (saveKeysToVault) {
    name: passwordKeyName
    parent: keyVault
    properties: {
        value: redis.listKeys().primaryKey
    }
}

output REDIS_ID string = redis.id
output REDIS_HOST string = redis.properties.hostName

```

2. Modify the *main.bicep* file to create an instance of the "redis" resource.

In the *main.bicep* file, add the following code below the ending curly braces associated with the *Web frontend* section and above the *secrets* section.

```
Python

// Caching server
module redis 'redis.bicep' = {
    name: 'redis'
    scope: resourceGroup
    params: {
        name: replace('${take(prefix, 19)}-rds', '--', '-')
        location: location
        tags: tags
        keyVaultName: keyVault.outputs.name
        connStrKeyName: 'RedisConnectionString'
        passwordKeyName: 'RedisPassword'
        primaryKeyKeyName: 'RedisPrimaryKey'
        publicNetworkAccess: 'Enabled'
        skuFamily: 'C'
        skuCapacity: 1
        skuName: 'Standard'
        saveKeysToVault: true
    }
}
```

3. Add output values to the bottom of the file:

```
Python

output REDIS_ID string = redis.outputs.REDIS_ID
output REDIS_HOST string = redis.outputs.REDIS_HOST
```

4. Confirm that the entire *main.bicep* file is identical to the following code:

```
Python

targetScope = 'subscription'

@minLength(1)
@maxLength(64)
@description('Name which is used to generate a short unique hash for
each resource')
param name string

@minLength(1)
@description('Primary location for all resources')
param location string

@secure()
@description('DBServer administrator password')
param dbserverPassword string
```

```

@secure()
@description('Secret Key')
param secretKey string

@description('Id of the user or app to assign application roles')
param principalId string = ''

var resourceToken = toLower(uniqueString(subscription().id, name,
location))
var prefix = '${name}-${resourceToken}'
var tags = { 'azd-env-name': name }

resource resourceGroup 'Microsoft.Resources/resourceGroups@2021-04-01'
= {
  name: '${name}-rg'
  location: location
  tags: tags
}

// Store secrets in a keyvault
module keyVault './core/security/keyvault.bicep' = {
  name: 'keyvault'
  scope: resourceGroup
  params: {
    name: '${take(replace(prefix, '-', ''), 17)}-vault'
    location: location
    tags: tags
    principalId: principalId
  }
}

module db 'db.bicep' = {
  name: 'db'
  scope: resourceGroup
  params: {
    name: 'dbserver'
    location: location
    tags: tags
    prefix: prefix
    dbserverDatabaseName: 'relecloud'
    dbserverPassword: dbserverPassword
  }
}

// Monitor application with Azure Monitor
module monitoring 'core/monitor/monitoring.bicep' = {
  name: 'monitoring'
  scope: resourceGroup
  params: {
    location: location
    tags: tags
    applicationInsightsDashboardName: '${prefix}-appinsights-dashboard'
    applicationInsightsName: '${prefix}-appinsights'
    logAnalyticsName: '${take(prefix, 50)}-loganalytics' // Max 63
}

```

```

    chars
    }
}

// Web frontend
module web 'web.bicep' = {
    name: 'web'
    scope: resourceGroup
    params: {
        name: replace('${take(prefix, 19)}-appsvc', '--', '-')
        location: location
        tags: tags
        applicationInsightsName: monitoring.outputs.applicationInsightsName
        keyVaultName: keyVault.outputs.name
        appCommandLine: 'entrypoint.sh'
        pythonVersion: '3.12'
        dbserverDomainName: db.outputs.dbserverDomainName
        dbserverUser: db.outputs.dbserverUser
        dbserverDatabaseName: db.outputs.dbserverDatabaseName
    }
}

// Caching server
module redis 'redis.bicep' = {
    name: 'redis'
    scope: resourceGroup
    params: {
        name: replace('${take(prefix, 19)}-rds', '--', '-')
        location: location
        tags: tags
        keyVaultName: keyVault.outputs.name
        connStrKeyName: 'RedisConnectionString'
        passwordKeyName: 'RedisPassword'
        primaryKeyKeyName: 'RedisPrimaryKey'
        publicNetworkAccess: 'Enabled'
        skuFamily: 'C'
        skuCapacity: 1
        skuName: 'Standard'
        saveKeysToVault: true
    }
}

var secrets = [
{
    name: 'DBSERVERPASSWORD'
    value: dbserverPassword
}
{
    name: 'SECRETKEY'
    value: secretKey
}
]

@batchSize(1)
module keyVaultSecrets './core/security/keyvault-secret.bicep' = [for

```

```
secret in secrets: {
  name: 'keyvault-secret-${secret.name}'
  scope: resourceGroup
  params: {
    keyVaultName: keyVault.outputs.name
    name: secret.name
    secretValue: secret.value
  }
}

output AZURE_LOCATION string = location
output AZURE_KEY_VAULT_ENDPOINT string = keyVault.outputs.endpoint
output AZURE_KEY_VAULT_NAME string = keyVault.outputs.name
output APPLICATIONINSIGHTS_NAME string =
monitoring.outputs.applicationInsightsName
output BACKEND_URI string = web.outputs.uri

output REDIS_ID string = redis.outputs.REDIS_ID
output REDIS_HOST string = redis.outputs.REDIS_HOST
```

5. Make sure all your changes are saved, then use the following command to update your provisioned resources on Azure:

```
shell
```

```
azd provision
```

(!) Note

Depending on many factors, adding an instance of Azure Cache for Redis to the existing deployment could take a long time. In testing, we experienced execution times in excess of 20 minutes. As long as you do not see any errors, allow the process to continue until complete.

6. When `azd provision` completes, open the Azure portal, navigate to the Resource Group for your deployment, and in the list of services confirm you now have an instance of Azure Cache for Redis.

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](#)

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

Article • 07/26/2023

In this quickstart, you'll deploy a Python web app (Django or Flask) 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:

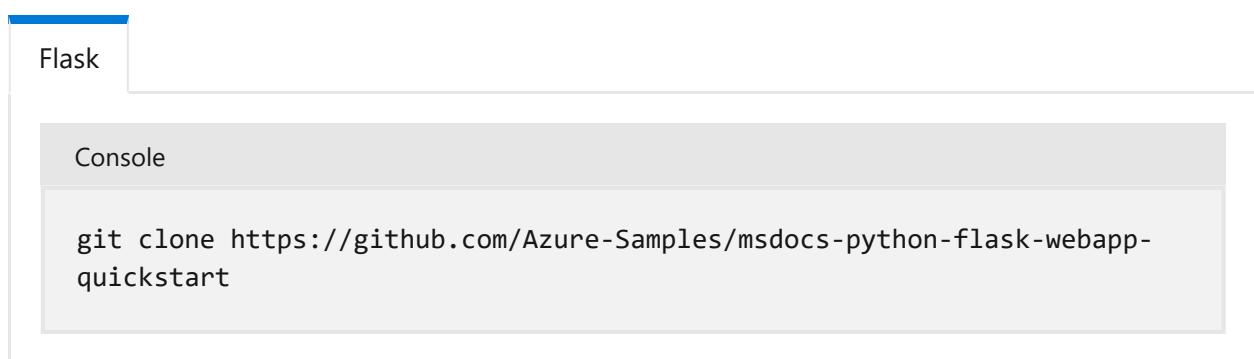
- An Azure account with an active subscription. [Create an account for free ↗](#).
- [Python 3.9 or higher ↗](#) installed locally.

ⓘ Note

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

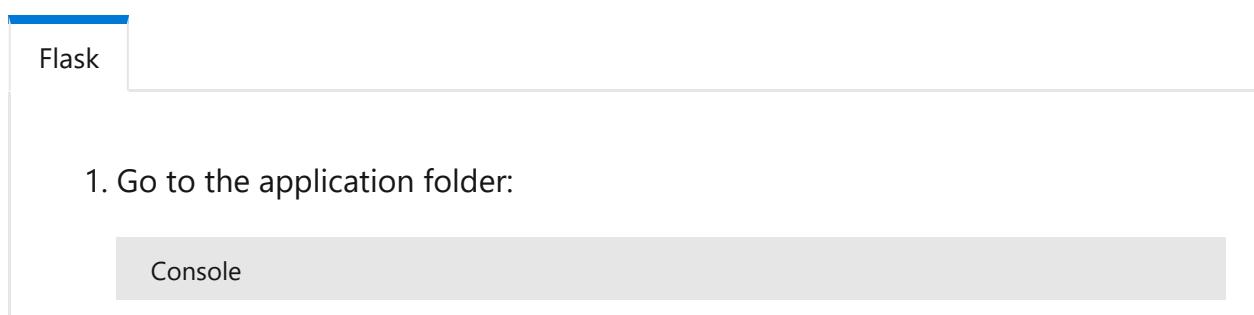
1 - Sample application

This quickstart can be completed using either Flask or Django. 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.



The screenshot shows a terminal window with a blue header bar. On the left, there is a tab labeled "Flask". Below the header, there is a "Console" tab and a code editor area. In the code editor, the command `git clone https://github.com/Azure-Samples/msdocs-python-flask-webapp-quickstart` is displayed.

To run the application locally:



The screenshot shows a terminal window with a blue header bar. On the left, there is a tab labeled "Flask". Below the header, there is a "Console" tab and a code editor area. The text "1. Go to the application folder:" is visible above the code editor.

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

2. Create a virtual environment for the app:

Windows

Cmd

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

3. Install the dependencies:

Console

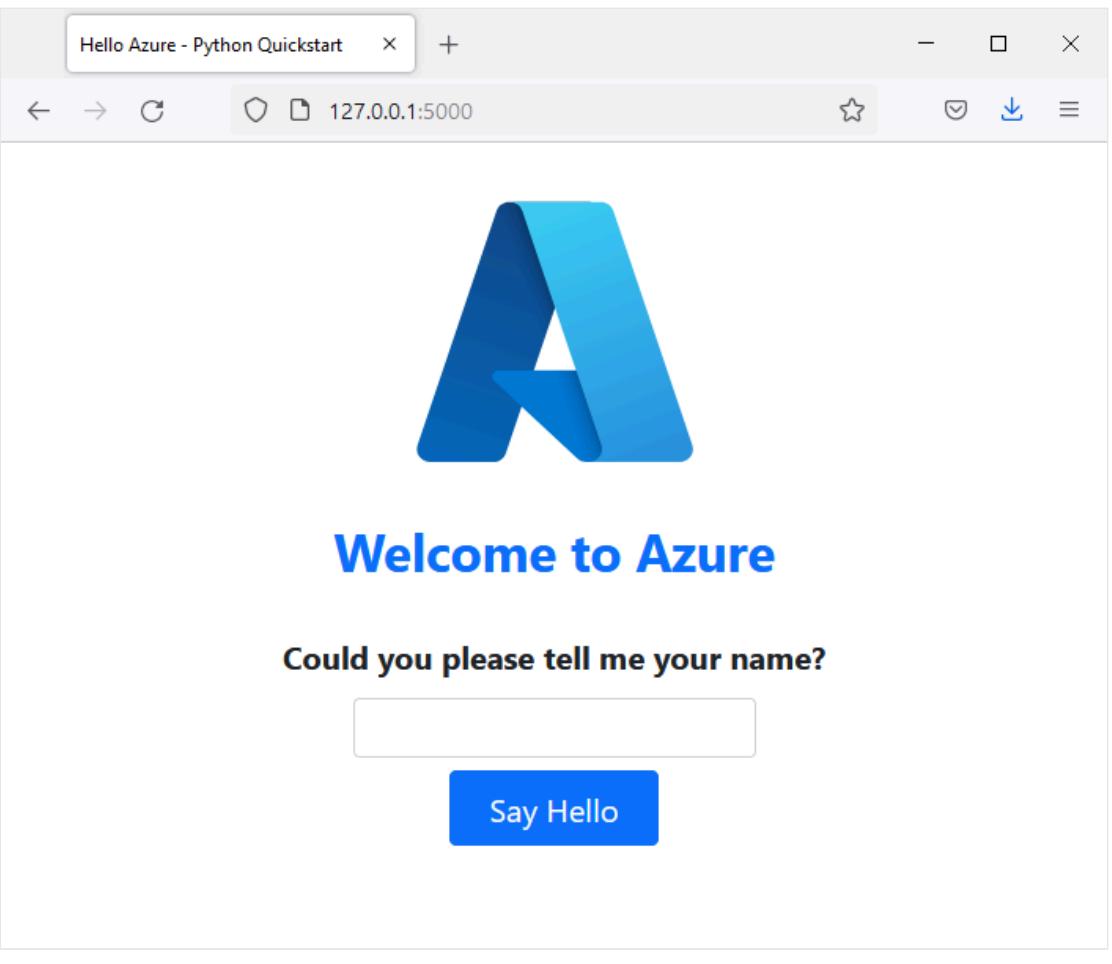
```
pip install -r requirements.txt
```

4. Run the app:

Console

```
flask run
```

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



2 - Create a web app in Azure

To host your application in Azure, you need to create Azure App Service web app in Azure. You can create a web app using the Azure CLI, [VS Code](#), [Azure Tools extension pack](#), or [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, login 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:3.9 --sku B1 --logs
```

- The `--runtime` parameter specifies what version of Python your app is running. This example uses Python 3.9. 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 account 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 gives the message, "You can launch the app at `http://<app-name>.azurewebsites.net`", 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 http://<app-name>.azurewebsites.net
{
  "URL": "http://<app-name>.azurewebsites.net",
  "appserviceplan": "<app-service-plan-name>",
  "location": "centralus",
  "name": "<app-name>",

}
```

```
"os": "<os-type>",
"resourcegroup": "<group-name>",
"runtime_version": "python|3.9",
"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 ↗](#).

3 - Deploy your application code to Azure

Azure App service supports multiple methods to deploy your application code to Azure including support for 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

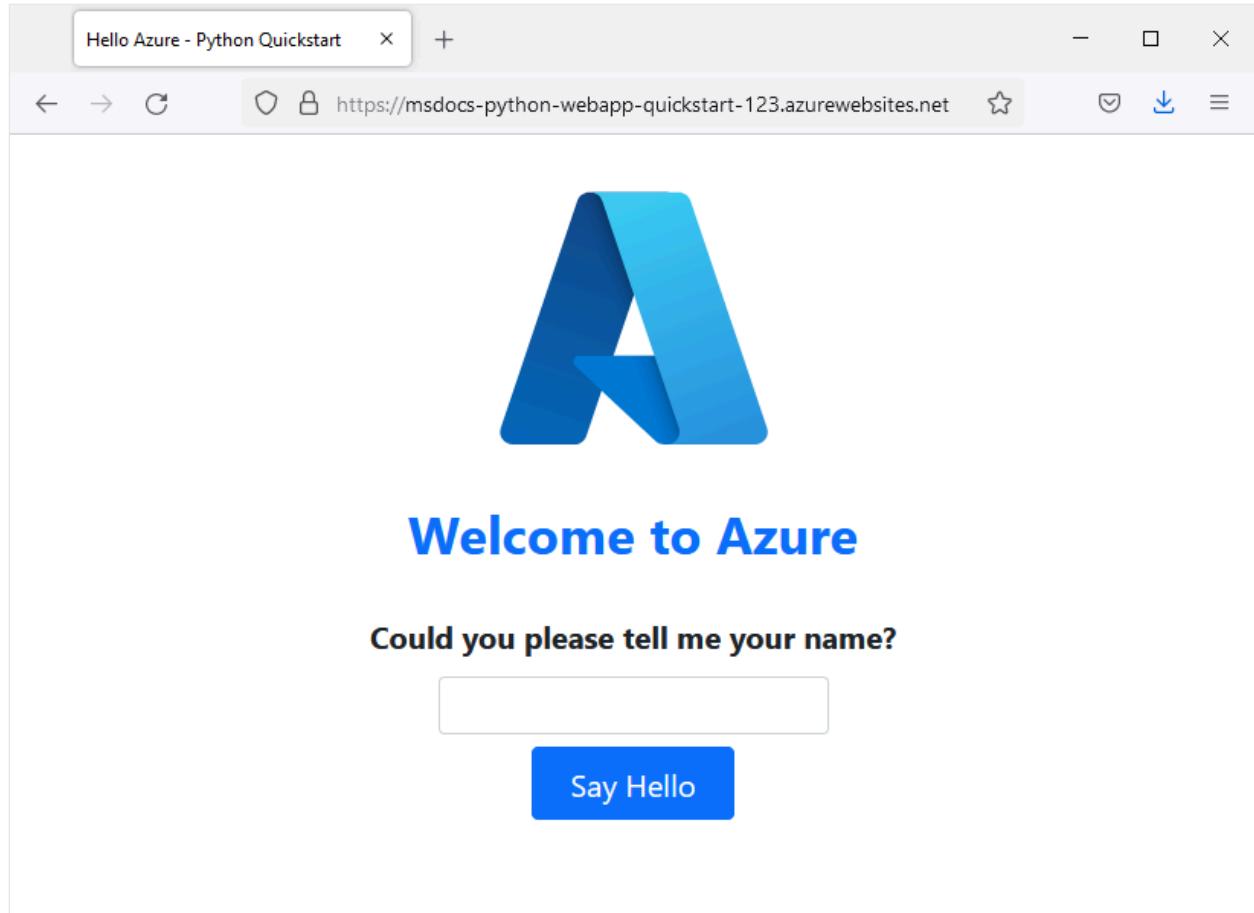
Since the `az webapp up` command created the necessary resources and deployed your application in a single step, you can move on to [4 - Browse to the app](#).

Having issues? Refer first to the [Troubleshooting guide](#), otherwise, [let us know ↗](#).

4 - Browse to the app

Browse to the deployed application in your web browser at the URL `http://<app-name>.azurewebsites.net`. 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.

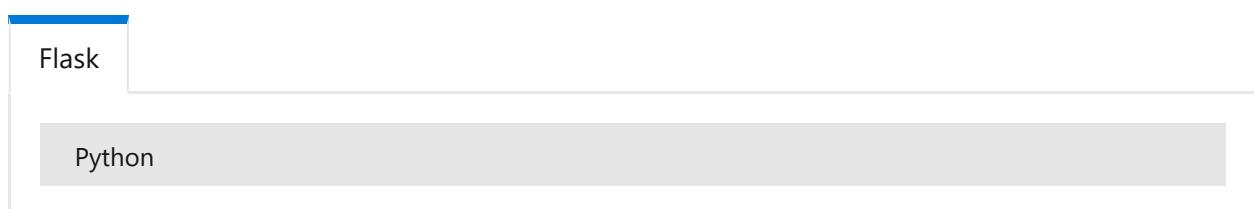


Congratulations! You've deployed your Python app to App Service.

Having issues? Refer first to the [Troubleshooting guide](#), otherwise, [let us know ↗](#).

5 - Stream logs

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



```
@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'))
```

The contents of the App Service diagnostic logs can be reviewed using the Azure CLI, VS Code, or Azure portal.

Azure CLI

First, you need to configure Azure App Service to output logs to the App Service filesystem 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"
```

Clean up resources

When you're finished with the sample app, you can remove all of the resources for the app from Azure. It will not incur extra charges and 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.

Azure CLI

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 \(Django\) web app with PostgreSQL](#)

[Configure Python app](#)

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

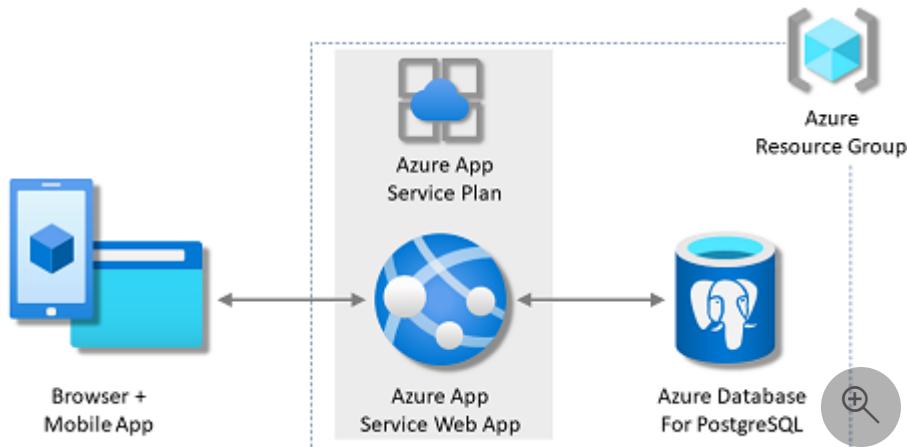
[Tutorial: Run Python app in custom container](#)

[Secure with custom domain and certificate](#)

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

Article • 12/01/2023

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



To complete this tutorial, you'll need:

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

Skip to the end

Flask

With [Azure Developer CLI](#) installed, you can deploy a fully configured sample app shown in this tutorial and see it running in Azure. Just running the following commands in an empty working directory:

Bash

```
azd auth login  
azd init --template msdocs-flask-postgresql-sample-app  
azd up
```

Sample application

Sample Python applications using the Flask and Django framework are provided to help you follow along with this tutorial. To deploy them without running them locally, skip this part.

To run the application locally, make sure you have [Python 3.7 or higher](#) and [PostgreSQL](#) installed locally. Then, clone the sample repository's `starter-no-infra` branch and change to the repository root.

Flask

Bash

```
git clone -b starter-no-infra https://github.com/Azure-Samples/msdocs-flask-postgresql-sample-app  
cd msdocs-flask-postgresql-sample-app
```

Create an `.env` file as shown below using the `.env.sample` file as a guide. Set the value of `DBNAME` to the name of an existing database in your local PostgreSQL instance. Set the values of `DBHOST`, `DBUSER`, and `DBPASS` as appropriate for your local PostgreSQL instance.

```
DBNAME=<database name>  
DBHOST=<database-hostname>  
DBUSER=<db-user-name>  
DBPASS=<db-password>
```

Create a virtual environment for the app:

Windows

Cmd

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

Install the dependencies:

Bash

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

Run the sample application with the following commands:

Flask

Bash

```
# Run database migration
flask db upgrade
# Run the app at http://127.0.0.1:5000
flask run
```

1. Create App Service and PostgreSQL

Flask

Bash

```
git clone https://github.com/Azure-Samples/msdocs-flask-postgresql-
sample-app.git
```

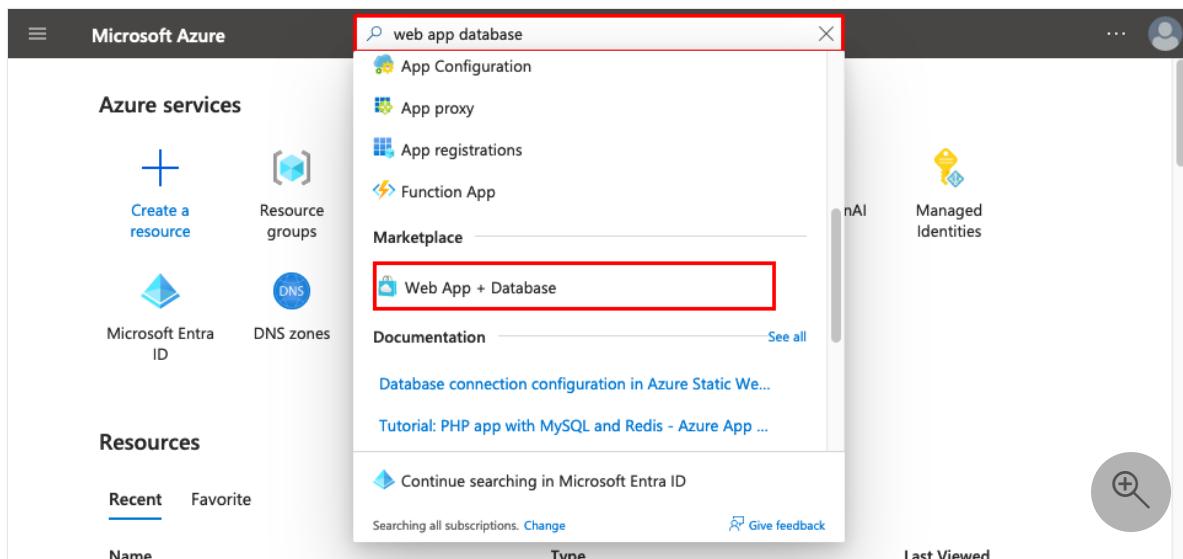
In this step, you create the Azure resources. The steps used in this tutorial create a set of secure-by-default resources that include App Service and Azure Database for PostgreSQL. For the creation process, you'll specify:

- The **Name** for the web app. It's the name used as part of the DNS name for your webapp in the form of `https://<app-name>.azurewebsites.net`.
- The **Region** to run the app physically in the world.
- 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-python-postgres-tutorial**.
2. *Region* → Any Azure region near you.
3. *Name* → **msdocs-python-postgres-XYZ** where **XYZ** is any three random characters. This name must be unique across Azure.
4. *Runtime stack* → **Python 3.10**.
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. *Hosting plan* → **Basic**. When you're ready, you can [scale up](#) to a production pricing tier later.
7. Select **Review + create**.
8. After validation completes, select **Create**.

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 * ⓘ

Antares-Demo

Resource Group * ⓘ

(New) msdocs-python-postgres-tutorial

[Create new](#)

Region *

West Europe

Web App Details

Name *

msdocs-python-postgres-234

.azurewebsites.net

Runtime stack *

Python 3.9

Database

i 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

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.
- **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 zone** → Enables DNS resolution of the PostgreSQL server in the virtual network.

✓ Your deployment is complete

 Deployment name : Microsoft.Web-WebAppDatabase-Portal-afa69d9d-97bc
Subscription : Visual Studio Enterprise Subscription
Resource group : msdocs-python-postgres-tutorial
Start time : 11/29/2023, 10:17:05 AM
Correlation ID : 60262550-6ded-4788-b396-6d96b871d488

- > Deployment details
▼ Next steps

[Go to resource](#)

[Give feedback](#)

 [Tell us about your experience with deployment](#)

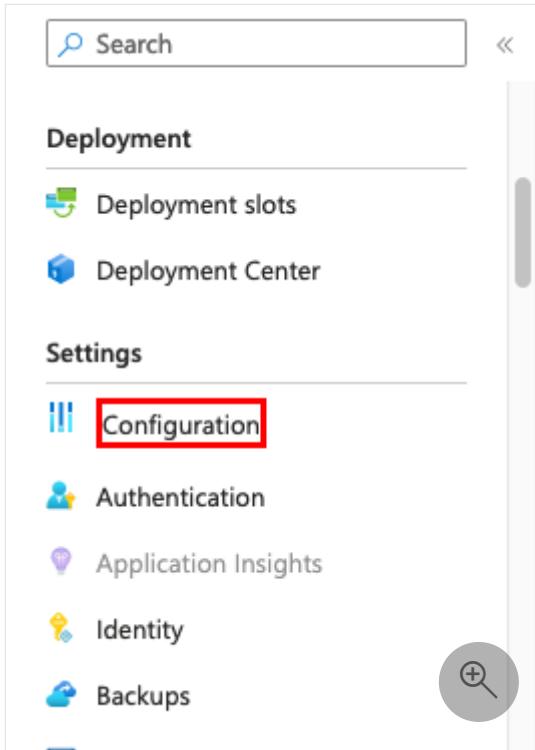


2. Verify connection settings

The creation wizard generated the connectivity variables for you already as [app settings](#). App settings are one way to keep connection secrets out of your code repository. When you're ready to move your secrets to a more secure location, here's an [article on storing in Azure Key Vault](#).

Flask

Step 1: In the App Service page, in the left menu, select Configuration.



Step 2: In the **Application settings** tab of the **Configuration** page, verify that `AZURE_POSTGRESQL_CONNECTIONSTRING` is present. That will be injected into the runtime environment as an environment variable.

Application settings General settings Path mappings Error pages (preview)

Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose the controls below. Application Settings are exposed as environment variables for access by your application.

+ New application setting Show values Advanced edit

Filter application settings

Name	Value	Source
AZURE_POSTGRESQL_CONNECTIONSTRING	(Hidden value. Click to show value)	App Service
FLASK_DEBUG	(Hidden value. Click to show value)	App Service
SCM_DO_BUILD_DURING_DEPLOYMENT	(Hidden value. Click to show value)	App Service
SECRET_KEY	(Hidden value. Click to show value)	App Service

Step 3: In a terminal or command prompt, run the following Python script to generate a unique secret: `python -c 'import secrets; print(secrets.token_hex())'`. Copy the output value to use in the next step.

3. Deploy sample code

In this step, you'll 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 will kick off the build and deploy action.

Flask

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>.
3. Select **Fork**.
4. Select **Create fork**.

The screenshot shows a GitHub fork creation dialog. At the top, there's a progress bar with the text "Forking msdocs-flask-postgresql-sample-app". Below it, there are two buttons: "Cancel" and "Create fork". A note says "This fork will be created in your account". At the bottom, there's a "Create fork" button.

Step 2: In the GitHub page, open Visual Studio Code in the browser by pressing the `. key`.

The screenshot shows the GitHub repository page for `msdocs-flask-postgresql-sample-app`. The repository is public and has 20 forks. The main branch is `main`. The repository has 11 issues, 3 pull requests, and 29 actions. The code tab is selected. On the right, there's an **About** section with no description, website, or topics provided. It includes links to Readme, MIT license, Code of conduct, 8 stars, and 11 watching. There's also a search icon.

Press the '.' key

This branch is up to date with Azure-Samples/msdocs-flask-postgresql-sample-app:main.

No description, website, or topics provided.

Readme
MIT license
Code of conduct
0 stars
0 watching

Step 3: In Visual Studio Code in the browser, open *azureproject/production.py* in the explorer. See the environment variables being used in the production environment, including the app settings that you saw in the configuration page.

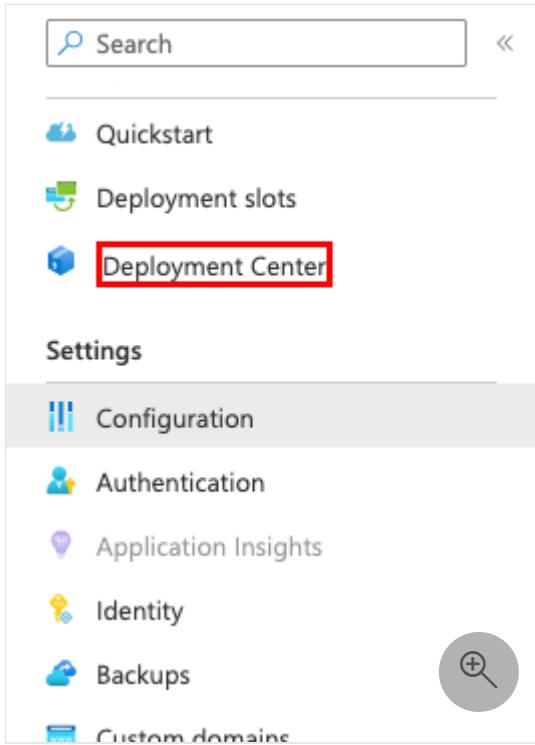
```

1
2
3 : keep the secret key used in production secret!
4 <insecure-7ppocbnx0w71dcuinn*t^_mzal(t@o0iv3fee27g%rg18fc5d0'
5
6
7 os.environ['WEBSITE_HOSTNAME']] if 'WEBSITE_HOSTNAME' in os.environ else []
8 sS = ['https://'+ os.environ['WEBSITE_HOSTNAME']] if 'WEBSITE_HOSTNAME' in os.environ else []
9
10 ss database; the full username for PostgreSQL flexible server is
11 ever-name).
12 stgresql+psycopg2://(dbuser):(dbpass)@(dbhost)/(dbname)".format(
13 con['DBUSER'],
14 con['DBPASS'],
15 con['DBHOST'] + ".postgres.database.azure.com",
16 con['DBNAME']
17

```

Do you want to install the recommended extensions for Python?

Step 4: Back in the App Service page, in the left menu, select Deployment Center.



Step 5: 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 **main**.
6. Keep the default option selected to **Add a workflow**.
7. Under **Authentication type**, select **User-assigned identity**.
8. In the top menu, select **Save**. App Service commits a workflow file into the chosen GitHub repository, in the `.github/workflows` directory.

Screenshot of the Microsoft Azure Deployment Center for the app service 'msdocs-python-postgres-234'. The 'Save' button is highlighted with a red box. The 'Source' dropdown is set to 'GitHub' and is also highlighted with a red box. The 'Organization' dropdown shows '<github-alias>' and the 'Repository' dropdown shows 'msdocs-flask-postgresql-sample-app'. The 'Branch' dropdown is set to 'dev' and is highlighted with a red box. The 'Build' section is visible at the bottom right.

Step 6: In the Deployment Center page:

1. Select **Logs**. A deployment run is already started.
2. In the log item for the deployment run, select **Build/Deploy Logs**.

The screenshot shows the Microsoft Azure Deployment Center for the app service 'msdocs-python-postgres-234'. The 'Logs' tab is highlighted with a red box. On the right, a log entry from Friday, October 7, 2022, at 9:19:3... is shown, with the 'Build/Deploy Log...' link also highlighted with a red box.

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 **Complete**. It takes about 5 minutes.

The screenshot shows a GitHub repository page for '`<github-alias>/msdocs-flask-postgresql-sample-app`'. The 'Actions' tab is selected. A workflow named 'Add or update the Azure App Service build and deployment workflow config' is shown, with the status 'In progress'. The workflow description indicates it's building and deploying a Python app to Azure Web App - msdocs-python-postgres-234 #1. The GitHub run summary shows it was triggered via push 14 seconds ago and is currently in progress.

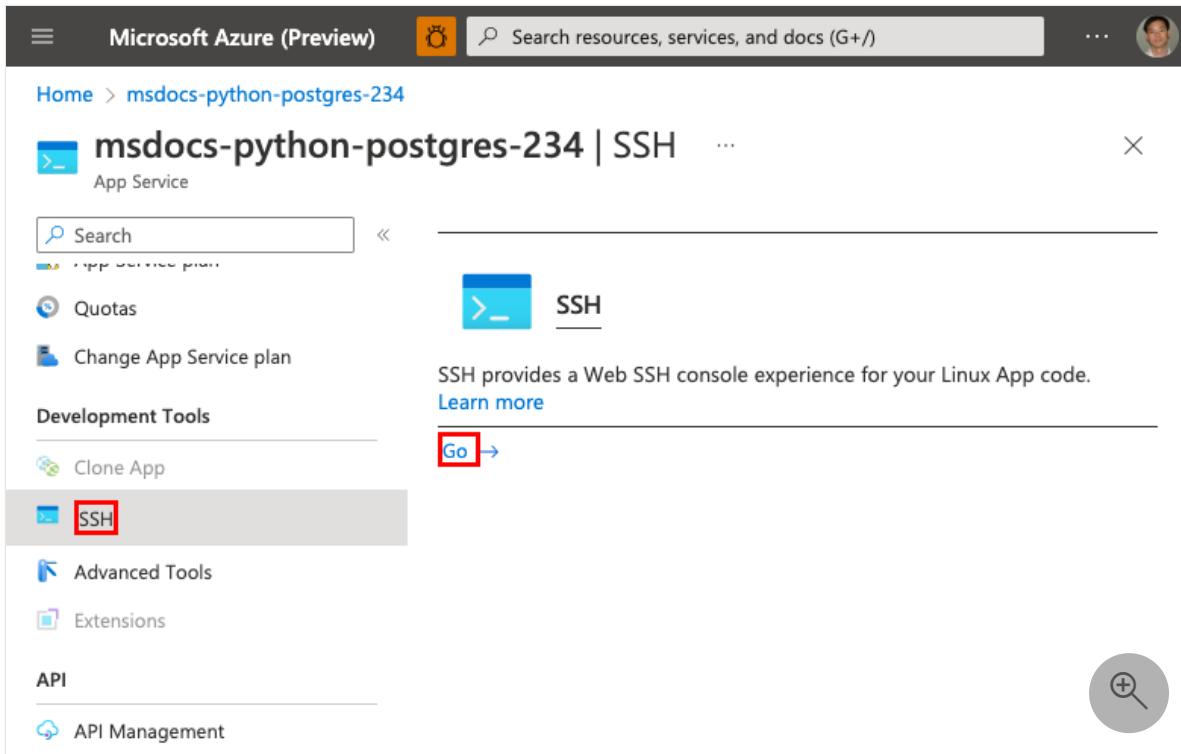
Having issues? Check the [Troubleshooting guide](#).

4. 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 App Service container.

Step 1: Back in the App Service page, in the left menu,

1. Select **SSH**.
2. Select **Go**.



Step 2: In the SSH terminal, run `flask db upgrade`. If it succeeds, App Service is [connecting successfully to the database](#). Only changes to files in `/home` can persist beyond app restarts. Changes outside of `/home` aren't persisted.

```
APP SERVICE ON LINUX

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# 
```

☰ Menu | ssh://root@169.254.130.2:2222 | SSH CONNECTION ESTABLISHED |

5. Browse to the app

Flask

Step 1: In the App Service page:

1. From the left menu, select **Overview**.
2. Select the URL of your app. You can also navigate directly to `https://<app-name>.azurewebsites.net.`

The screenshot shows the Azure App Service Overview page. On the left, there's a navigation menu with 'Overview' selected (highlighted with a red box). Other options include 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Microsoft Defender for Cloud', and 'Events (preview)'. Below this is a 'Deployment' section with 'Deployment slots' and 'Deployment Center'. The main pane displays 'Essentials' information for the app:

Essentials	
Resource group (move)	: msdocs-python-postgres-tutorial
Status	: Running
Location (move)	: East US
Subscription (move)	: Visual Studio Enterprise Subscription
Subscription ID	: 00000000-0000-0000-000000000000
Default domain	: msdocs-python-postgres-125.azurewebsites...
App Service Plan	: ASP-msdocspythonpostgrestutorial-b709 (B1)
Operating System	: Linux
Health Check	: Error fetching health check data. Please try again.

A search bar at the top left and a magnifying glass icon with a plus sign at the bottom right are also visible.

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.

The screenshot shows a web application titled "Azure Restaurant Review". At the top right is a link to "Azure Docs". The main section is titled "Restaurants" and displays two entries in a table:

Name	Rating	Details
Fourth Coffee	★★	2.0 (2 reviews) <button>Details</button>
Contoso Restaurant	★★★★	4.0 (3 reviews) <button>Details</button>

Below the table is a green button labeled "Add new restaurant". In the bottom right corner of the page area, there is a circular icon containing a magnifying glass and a plus sign.

6. Stream diagnostic logs

Azure App Service captures all messages output to the console to help you diagnose issues with your application. The sample app includes `print()` statements to demonstrate this capability as shown below.

The screenshot shows a code editor with a tab labeled "Flask" and a code block labeled "Python". The code contains a single function definition:

```
@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 **App Service logs**.
2. Under **Application logging**, select **File System**.
3. In the top menu, select **Save**.

msdocs-python-postgres-234 | App Service logs

Application logging

- Off
- File System**

Quota (MB) * 35

Retention Period (Days)

Download logs

FTP/deployment username msdocs-python-postgres-234\user234

FTP ftp://waws-prod-am2-603.ftp.azurewebsites.windows.net/site/wwwroot

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.

msdocs-python-postgres-234 | Log stream

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

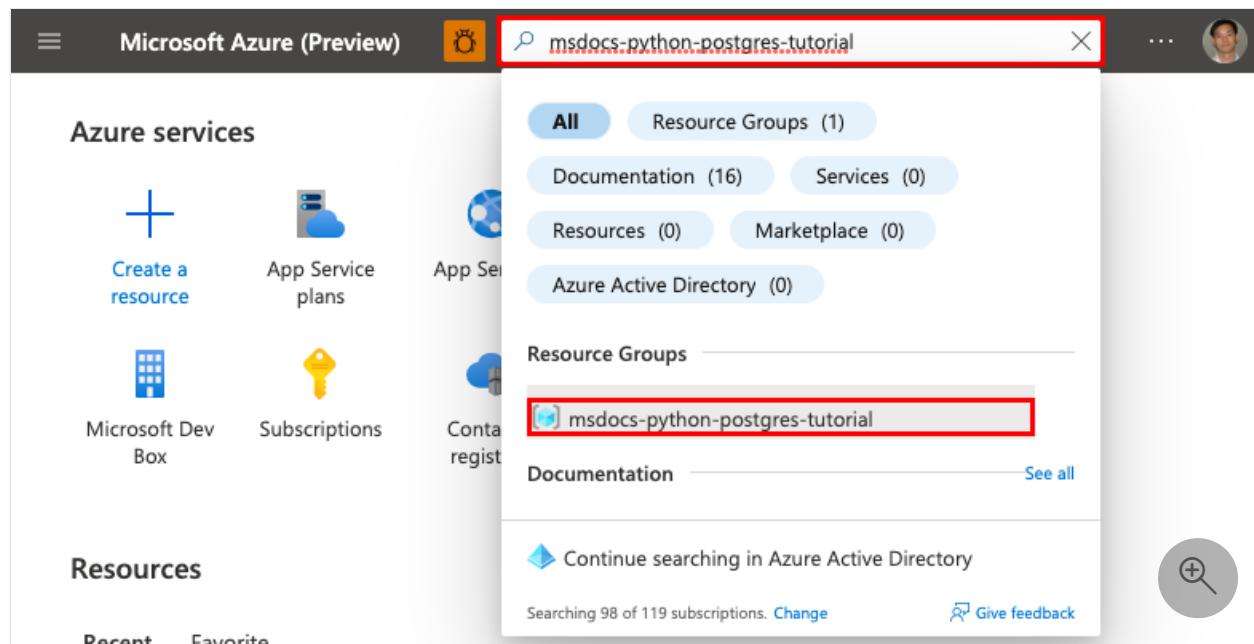
Learn more about logging in Python apps in the series on [setting up Azure Monitor for your Python application](#).

7. 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 (Preview) portal interface. At the top, there's a search bar with the text "msdocs-python-postgres-tutorial". Below the search bar, the main navigation area has sections for "Azure services" and "Resources". Under "Azure services", there are icons for "Create a resource", "App Service plans", "Microsoft Dev Box", and "Subscriptions". Under "Resources", there are "Recent" and "Favorite" links. To the right of the main navigation, a search dropdown is open, showing categories like "All", "Resource Groups (1)", "Documentation (16)", "Services (0)", "Resources (0)", "Marketplace (0)", and "Azure Active Directory (0)". Below these categories, a list of "Resource Groups" is shown, with "msdocs-python-postgres-tutorial" listed and highlighted with a red box. At the bottom of the dropdown, there's a link to "Continue searching in Azure Active Directory" and a "Give feedback" button.

Step 2: In the resource group page, select **Delete resource group**.

[Create](#) [Manage view](#) [Delete resource group](#) ...

^ Essentials View Cost JSON View

Subscription (move)	:	Antares-Demo
Subscription ID	:	00000000-0000-0000-0000-0...
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.

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-postares-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_CONNECTIONSTRING'`, 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_CONNECTIONSTRING`) have been changed. 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 is the Django sample configured to run on Azure App Service?](#)

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 terminal.
- 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 is the Django sample configured to run on Azure App Service?

Note

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

The [Django sample application](#) configures settings in the `azureproject/production.py` file so that it can run in Azure App Service. These changes are common to deploying Django to production, and not specific to App Service.

- Django validates the `HTTP_HOST` header in incoming requests. The sample code uses the [WEBSITE_HOSTNAME](#) environment variable in App Service to add the app's domain name to Django's [ALLOWED_HOSTS](#) setting.

Python

```
# Configure the domain name using the environment variable  
# that Azure automatically creates for us.  
ALLOWED_HOSTS = [os.environ['WEBSITE_HOSTNAME']] if 'WEBSITE_HOSTNAME'  
in os.environ else []
```

- Django doesn't support [serving static files in production](#). For this tutorial, you use [WhiteNoise](#) to enable serving the files. The WhiteNoise package was already installed with requirements.txt, and its middleware is added to the list.

Python

```
# WhiteNoise configuration  
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    # Add whitenoise middleware after the security middleware  
    'whitenoise.middleware.WhiteNoiseMiddleware',
```

Then the static file settings are configured according to the Django documentation.

Python

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"  
STATICFILES_STORAGE =  
    'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

For more information, see [Production settings for Django apps](#).

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 • 04/22/2024

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
```

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)
```

ⓘ 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:

Contoso Café

Street address: 1 Main Street
Description: Nice coffee house.
Rating: ★★★★ 4.5 (2 reviews)

Reviews

Add new review

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?

Yes

No

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 • 04/22/2024

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
```

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="UAManagedIdentityPythonTest$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/$RESOURCE_GROUP_NAME/providers/Microsoft.Storage/storageAccounts/$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 will be 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 have 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:

Azure Restaurant Review

Azure Docs ▾

Contoso Café

Street address: 1 Main Street
Description: Nice coffee house.
Rating: ★★★★ 1 4.5 (2 reviews)

Reviews

Add new review

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?

 Yes

 No

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

Static website hosting in Azure Storage

Article • 09/29/2023

You can serve static content (HTML, CSS, JavaScript, and image files) directly from a storage container named `$web`. Hosting your content in Azure Storage enables you to use serverless architectures that include [Azure Functions](#) and other Platform as a service (PaaS) services. Azure Storage static website hosting is a great option in cases where you don't require a web server to render content.

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.

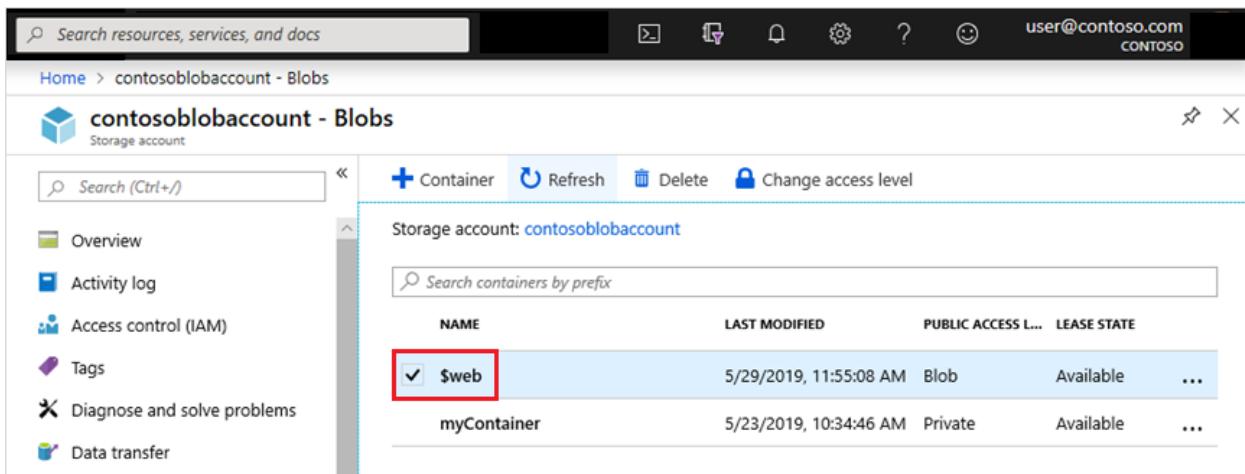
If you need a web server to render content, you can use [Azure App Service](#).

Setting up a static website

Static website hosting is a feature that you have to enable on the storage account.

To enable static website hosting, select the name of your default file, and then optionally provide a path to a custom 404 page. If a blob storage container named `$web` doesn't already exist in the account, one is created for you. Add the files of your site to this container.

For step-by-step guidance, see [Host a static website in Azure Storage](#).



The screenshot shows the Azure Storage Explorer interface. On the left, a sidebar lists options: Home, contosoblobaccount - Blobs, Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and Data transfer. The main area is titled "contosoblobaccount - Blobs" and shows a table of blob containers. The first row, "\$web", has a checked checkbox and is highlighted with a red border. The second row, "myContainer", is regular. The table columns are NAME, LAST MODIFIED, PUBLIC ACCESS L..., and LEASE STATE. The \$web row shows "5/29/2019, 11:55:08 AM", "Blob", "Available", and "...". The myContainer row shows "5/23/2019, 10:34:46 AM", "Private", "Available", and "...".

Files in the **\$web** container are case-sensitive, served through anonymous access requests and are available only through read operations.

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 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://contosoblobaccount.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://contosoblobaccount.z22.web.core.windows.net/index.html` doesn't change.

However, anonymous access to the primary blob service endpoint

`https://contosoblobaccount.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 - Verizon Premium](#). 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.

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.

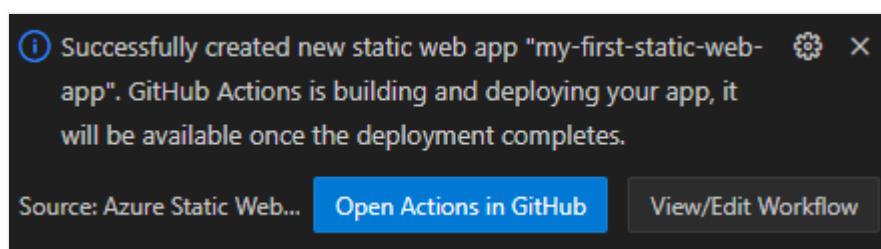
8. Enter the settings values that match your framework choice.

Setting	Value
Framework	Select Custom
Location of application code	Enter <code>/src</code>
Build location	Enter <code>/src</code>

No Framework

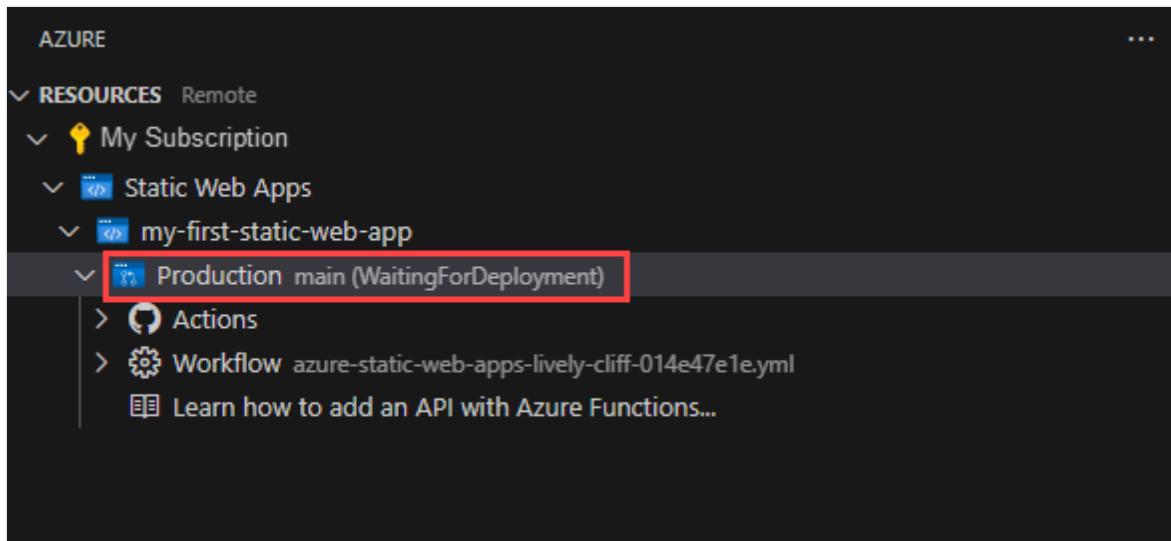
[Expand table](#)

9. Once the app is created, a confirmation notification is shown in Visual Studio Code.



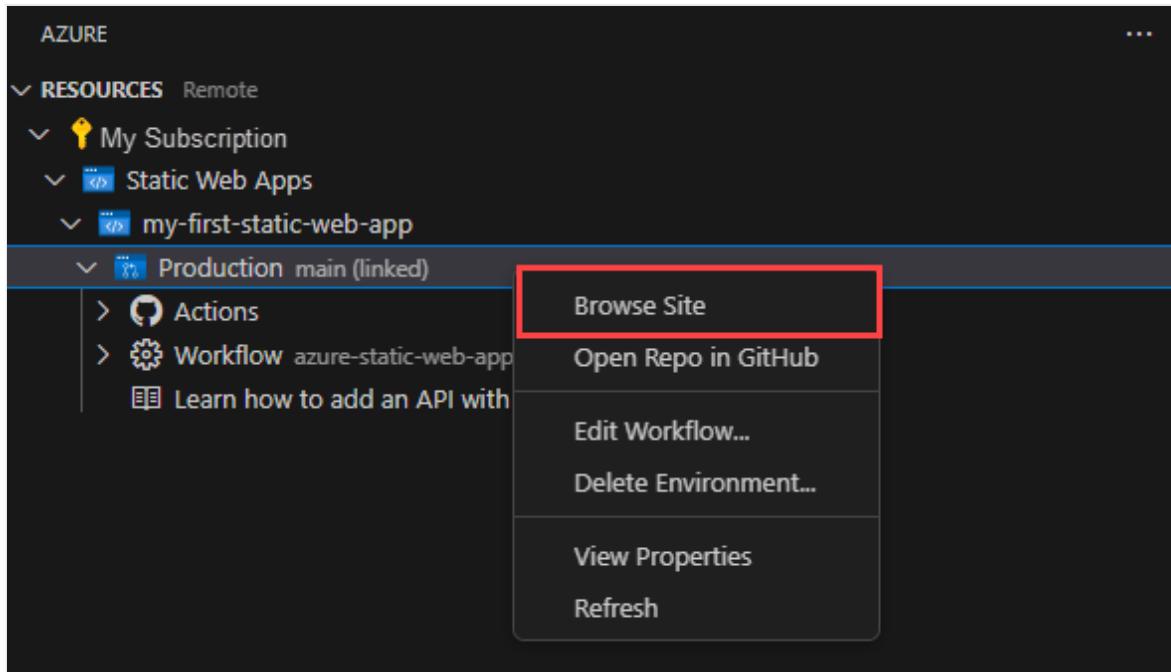
If GitHub presents you with a button labeled **Enable Actions on this repository**, select the button to allow the build action to run on your repository.

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**.

Next steps

Add an API

Create a GitHub Codespaces dev environment with FastAPI and Postgres

Article • 05/23/2023

This article shows you how to run FastAPI and Postgres together in a [GitHub Codespaces](#) environment. Codespaces is a development environment hosted in the cloud. Codespaces enables 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](#).

You can also clone the sample repo locally and when you open the project in Visual Studio Code, you can run using [Dev Containers](#). Dev Containers requires that [Docker Desktop](#) installed locally. If you don't have Docker installed, you can still use VS Code to run the project, but you're using GitHub Codespaces as the 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](#).

With the approach shown in this tutorial, you can start with the sample code and modify it to run other Python frameworks like Django or Flask.

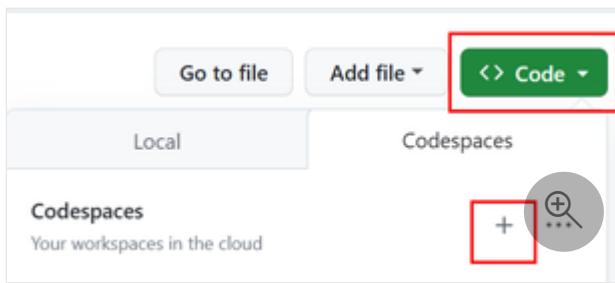
Start the dev environment in Codespaces

There are many possible paths to create and use GitHub Codespaces. This tutorial shows one path you can start with.

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 that sample repo has loaded.

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 missed 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" with version "0.1.0" and "OAS3". Below that is a link to "/openapi.json". The main area is titled "default". It lists four API endpoints:

- GET / Root
- GET /restaurant/{id} Get Restaurant
- POST /restaurant Set Restaurant
- GET /all Get All Restaurants

The "POST /restaurant Set Restaurant" endpoint is highlighted with a green background. To the right of the endpoints is a circular button with a plus sign and a magnifying glass icon.

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 request body is defined as follows:

```
JSON
{
  "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 extension interface in VS Code. On the left, there's a sidebar with icons for connections, schemas, and tables. A red box highlights the 'Tables' icon. Below it, under 'Local database', another red box highlights the 'restaurants' table. A context menu is open over the table, with 'Show Table Records' also highlighted by a red box. The menu includes other options like 'Describe Table', 'Generate Insert Query', 'Add Name(s) To Cursor', and 'Copy Value(s)'. To the right of the menu is a preview window titled 'Local database: 1 records on \'restaurants\' table' showing a single row with 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 • 01/12/2024

In this article, you learn about configuring a custom startup file, if needed, for a Python web app hosted on Azure App Service. For running locally, you don't need a startup file. However, when you deploy a web app to Azure App Service, your code is run in a Docker container that can use any startup commands if they are present.

You need a custom startup file in the following cases:

- You want to start the [Gunicorn](#) default web server with extra arguments beyond the defaults, which are `--bind=0.0.0.0 --timeout 600`.
- Your app is built with a framework other than Flask or Django, or you want to use a different web server besides Gunicorn.
- 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, unless you have an `app.py` or `application.py` in the root folder of your project, *and* the Flask app object is named `app`, then you need a custom startup command.

For more information, see [Configure Python Apps - Container startup process](#).

Create a startup file

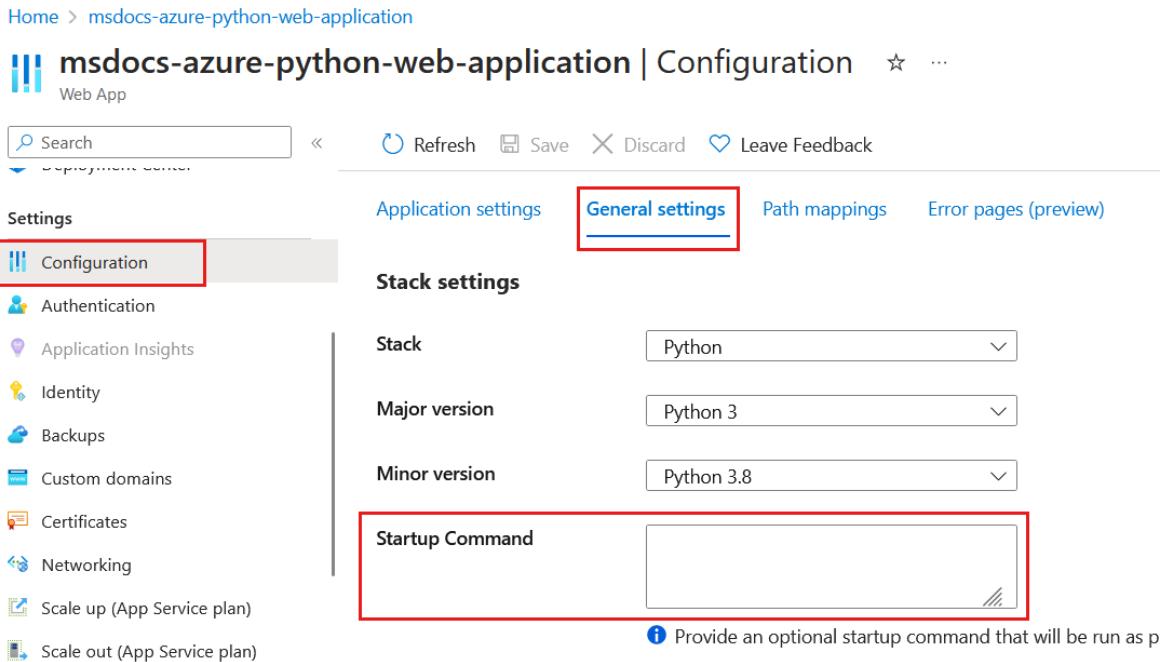
When you need a custom startup file, use the following steps:

1. Create a file in your project named `startup.txt`, `startup.sh`, or another name of your choice that contains your startup command(s). 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**.



The screenshot shows the Azure portal's Configuration page for a Web App named "msdocs-azure-python-web-application". The "General settings" tab is active. On the left, the "Settings" sidebar has "Configuration" selected. The main area shows "Stack settings" with "Stack" set to "Python", "Major version" to "Python 3", and "Minor version" to "Python 3.8". The "Startup Command" field is empty and highlighted with a red box. A note below it says: "Provide an optional startup command that will be run as part of the application's startup process."

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 command file is preferable, however, because this part of your configuration is then in your repository where you can audit changes and redeploy to a different App Service instance altogether.

5. The App Service restarts when you save the changes.

If you haven't deployed your app code, however, visiting the site at this point shows "Application Error." This message indicates that the Gunicorn server started but failed to find the app, and therefore nothing is responding to HTTP requests.

Django startup commands

By default, App Service automatically locates the folder that contains 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 change any of the Gunicorn arguments, such as using `--timeout 1200`, then create a command file with those modifications. For more information, see [Container startup process - Django app](#).

Flask startup commands

By default, the App Service on Linux container assumes that a Flask app's WSGI callable is named `app` and is contained in a file named `application.py` or `app.py` and resides in the app's root folder.

If you use any of the following variations, 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:** for example, 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:** for example, 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.

Use CI/CD with GitHub Actions to deploy a Python web app to Azure App Service on Linux

Article • 08/03/2023

Use the GitHub Actions continuous integration and continuous delivery (CI/CD) platform 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 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 a repository for your app code

If you already have a Python web app to use, make sure it's committed to a GitHub repository.

If you need an app to work with, you can fork and clone the repository at <https://github.com/Microsoft/python-sample-vscode-flask-tutorial>. The code is from the tutorial [Flask in Visual Studio Code](#).

Note

If your app uses Django and a SQLite database, it won't work for this tutorial. If your Django app uses a separate database like PostgreSQL, you can use it with this tutorial. For more information about Django, see [considerations for Django](#) later in this article.

Create the target Azure App Service

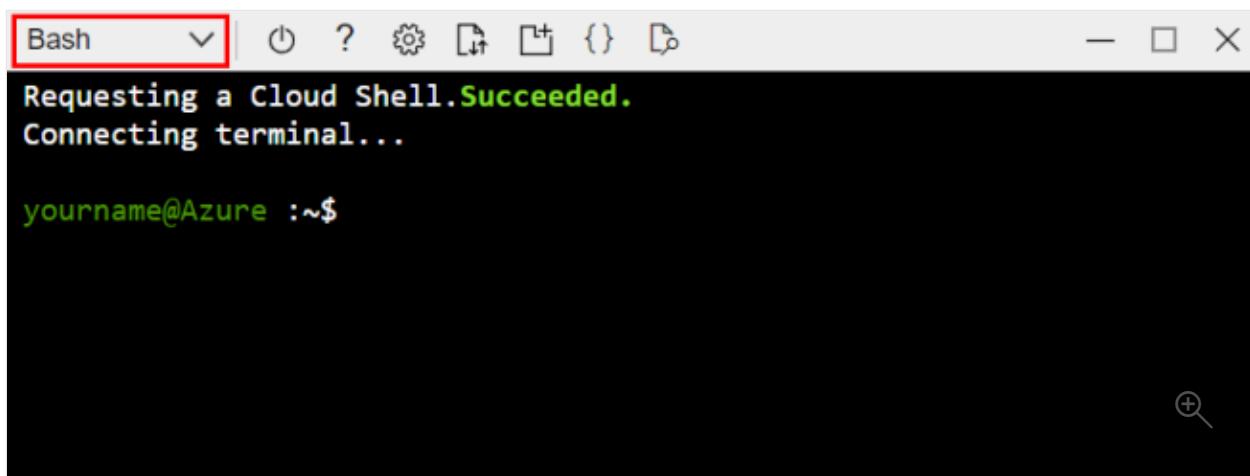
The quickest way to create an App Service instance is to use the [Azure command-line interface](#) (CLI) through the interactive [Azure Cloud Shell](#). The Cloud Shell includes [Git](#) and Azure CLI. In the following steps, you'll use [az webapp up](#) to both create the App Service and do the first deployment of your app.

Step 1. Sign in to the Azure portal at <https://portal.azure.com>.

Step 2. Open the Azure CLI by selecting the Cloud Shell icon on the portal toolbar.



Step 3. In the Cloud Shell, select Bash from the dropdown.



Step 4. In the Cloud Shell, clone your repository using [git clone](#). For example, if you're using the Flask sample app the command is:

```
Bash

git clone https://github.com/<github-user>/python-sample-vscode-flask-tutorial.git
```

Replace <github-user> with the name of the GitHub account where you forked the repo. If you're using a different app repo, this repo is where you'll set up GitHub Actions.

Note

The Cloud Shell is backed by an Azure Storage account in a resource group called *cloud-shell-storage-<your-region>*. That storage account contains an image of the Cloud Shell's file system, which stores the cloned repository. There's a small cost for this storage. You can delete the storage account at the end of this article, along with other resources you create.

Tip

To paste into the Cloud Shell, use **Ctrl+Shift+V**, or right-click and select **Paste** from the context menu.

Step 5. In the Cloud Shell, change directory into the repository folder that has your Python app so the [az webapp up](#) command will recognize the app as Python. For the

example, for the Flask sample app:

```
Bash
```

```
cd python-sample-vscode-flask-tutorial
```

Step 6. In the Cloud Shell, use `az webapp up` to create an App Service and initially deploy your app.

```
Bash
```

```
az webapp up --name <app-service-name> --runtime "PYTHON:3.9"
```

Specify an App Service name that is 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.

Use `az webapp list-runtimes` to get a list of available runtimes. Use the `PYTHON|X.Y` format, where `X.Y` is the Python version.

You can also specify the location of the App Service with the `--location` parameter. Use the `az account list-locations --output table` command to get a list of available locations.

Step 7. If your app uses a custom startup command, then use the `az webapp config` use that command. If your app doesn't have a custom startup command, skip this step.

For example, the *python-sample-vscode-flask-tutorial* app contains a file named *startup.txt* that contains a startup command that you can use as follows:

```
Bash
```

```
az webapp config set \
--resource-group <resource-group-name> \
--name <app-service-name> \
--startup-file startup.txt
```

You can find the resource group name from the output from the previous `az webapp up` command. The resource group name will start with `<azure-account-name>_rg_`.

Step 8. To see the running app, open a browser and go to `http://<app-service-name>.azurewebsites.net`.

If you see a generic page, wait a few seconds for the App Service to start, and refresh the page. If you continue to see the generic page, check that you deployed from the correct folder. For example, if you're using the Flask sample app, the folder is *python-sample-vscode-flask-tutorial*. Also, for the Flask sample app, check that you set the startup command correctly.

Set up continuous deployment in App Service

In the steps below, you'll set up continuous deployment (CD), which means a new code deployment happens when a workflow is triggered. The trigger in this tutorial is any change to the main branch of your repository, such as with a pull request (PR).

Step 1. Add GitHub Action with the [az webapp deployment github-actions add](#) command.

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 to complete the authentication.

If there's an existing workflow file that conflicts with the name App Service uses, you'll be asked to choose whether to overwrite. Use the `--force` parameter to overwrite without asking.

What the add command does:

- Creates new workflow file: `.github/workflows/<workflow-name>.yml` in your repo; the name of the file will contain the name of your App Service.
- Fetches a publish profile with secrets for your App Service and adds it as a GitHub action secret. The name of the secret will start with `AZUREAPPSERVICE_PUBLISHPROFILE_`. This secret is referenced in the workflow file.

Step 2. Get the details of a source control deployment configuration with the [az webapp deployment source show](#) command.

Bash

```
az webapp deployment source show \
--name <app-service-name> \
--resource-group <resource-group-name>
```

In the output from the command, confirm the values for the `repoUrl` and `branch` properties. These values should match the values you specified in the previous step.

GitHub workflow and actions explained

A workflow is defined by 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 that associates 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. Each job in turn is a set of steps. And finally, each step is a shell script or an action.

In terms of the workflow set up with your Python code for deployment to 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 using a publish profile credential to authenticate in Azure. The credential is stored in a GitHub secret .

The workflow template that is 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:

```
yml
```

```
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 YML file in `.github/workflows/<workflow-name>.yml`. You can see your authorized apps and revoke permissions under your GitHub accounts **Settings**, under **Integrations/Applications**.

The screenshot shows the GitHub 'Applications' settings page. On the left, there's a sidebar with options like 'Public profile', 'Account', 'Appearance', 'Accessibility', 'Notifications', 'Access' (with 'Billing and plans', 'Emails', 'Password and authentication', and 'Sessions' listed), and a collapsed 'Actions' section. The main area is titled 'Applications' and has three tabs: 'Installed GitHub Apps', 'Authorized GitHub Apps', and 'Authorized OAuth Apps'. The 'Authorized OAuth Apps' tab is selected and highlighted with a red box. Below it, a message says 'You have granted 5 applications access to your account.' A 'Sort' dropdown and a 'Revoke all' button are on the right. Two apps are listed: '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'). Each app entry has a 'More' button on the right.

Workflow publish profile secret

In the `.github/workflows/<workflow-name>.yml` workflow file that was added to the repo, you'll see a placeholder for publish profile credentials that are needed for the deploy job of the workflow. The publish profile information is stored encrypted in the repository **Settings**, under **Security/Actions**.

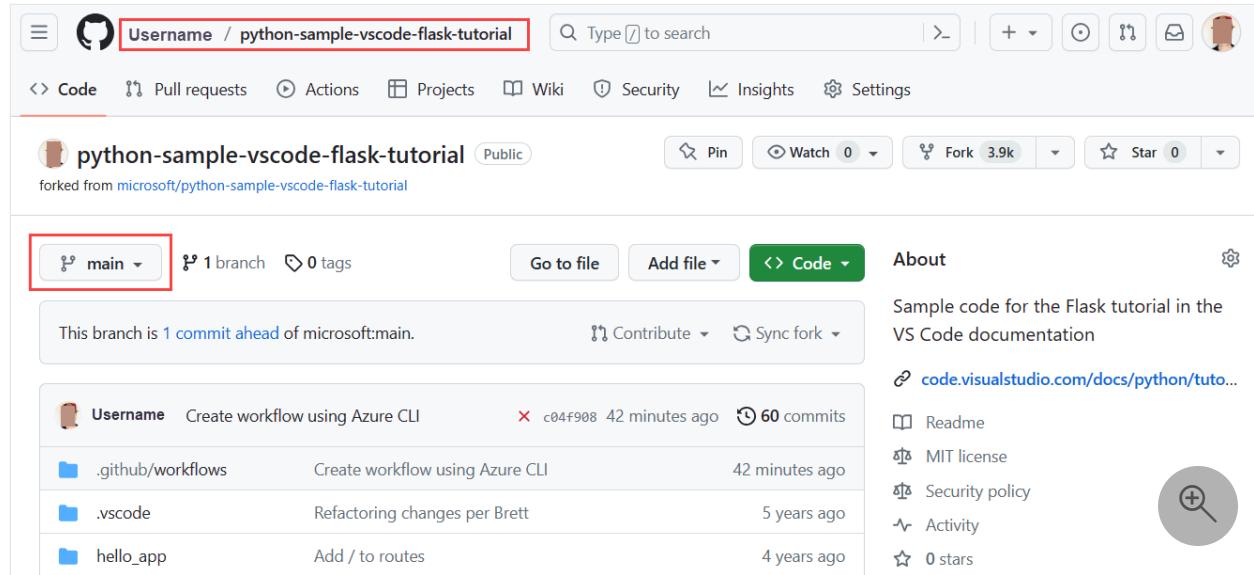
The screenshot shows the GitHub 'Repository secrets' settings page. On the left, there's a sidebar with 'Security' (including 'Code security and analysis', 'Deploy keys', and 'Secrets and variables'), 'Actions' (which is highlighted with a red box), 'Codespaces', and 'Dependabot'. The main area is titled 'Repository secrets' and shows a single secret named 'AZUREAPPSERVICE_PUBLISHPROFILE_913C30ECC24' with the value '24CB29A167A8339E43BFB'. It includes an 'Edit' and a 'Delete' button. A 'More' button is at the bottom 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 the workflow

Now you'll test the workflow by making a change to the repo.

Step 1. Go to your fork of the sample repository (or the repository you used) and select the branch you set as part of the trigger.



The screenshot shows a GitHub repository page for 'python-sample-vscode-flask-tutorial'. The top navigation bar includes links for Code, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. Below the header, the repository name 'python-sample-vscode-flask-tutorial' is displayed, along with a note that it's a fork from 'microsoft/python-sample-vscode-flask-tutorial'. The main content area shows a summary: 'main' branch, 1 branch, 0 tags. It also displays a commit message: 'This branch is 1 commit ahead of microsoft:main.' Below this, a list of commits is shown:

Author	Commit Message	Date
Username	Create workflow using Azure CLI	c04f908 42 minutes ago
.github/workflows	Create workflow using Azure CLI	42 minutes ago
.vscode	Refactoring changes per Brett	5 years ago
hello_app	Add / to routes	4 years ago

On the right side, there's an 'About' section with details like 'Sample code for the Flask tutorial in the VS Code documentation', a link to 'code.visualstudio.com/docs/python/tuto...', and icons for Readme, MIT license, Security policy, Activity, and 0 stars. A magnifying glass icon is also present.

Step 2. Make a small change.

For example, if you used the VS Code Flask tutorial, you can

- Go to the `/hello-app/templates/home.html` file of the trigger branch.
- Select **Edit** and add the text "Redeployed!".

Step 3. Commit the change directly to the branch you're working in.

- On the upper-right of the page you're editing, select the **Commit changes ...** button. The **Commit changes** window opens. In the **Commit changes** window, modify the commit message if desired and then select the **Commit changes** button.
- The commit kicks off the GitHub Actions workflow.

You can also kick off the workflow manually.

Step 1. Go to the **Actions** tab of the repo set up for continuous deployment.

Step 2. Select the workflow in the list of workflows and then select **Run workflow**.

Troubleshooting a failed workflow

To check a workflow's status, go to the Actions tab of the repo. When you drill into the workflow file created in this tutorial, you'll see two jobs "build", and "deploy". For a failed job, look at the output of job tasks for an indication of the failure. Some common issues are:

- 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 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 may not have been set. This setting must be set to `true`. The `az webapp up` command sets the build action automatically.
- If you see an error message with "TLS handshake timeout", run the workflow manually by selecting Trigger auto deployment under the Actions tab of the repo 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 initially created automatically for you. If you modified it, remove the modifications to see if they're causing the failure.

Run a post-deployment script

A post-deployment script can, for example, define environment variables expected by the app code. Add the script as part of the app code and execute it using startup command.

To avoid hard-coding variable values in your workflow YML file, you can instead them in the GitHub web interface and then refer to the variable name in the script. You can create encrypted secrets for a repository or for an environment (account repository). For more information, see [Encrypted secrets in GitHub Docs](#).

Considerations for Django

As noted earlier in this article, you can use GitHub Actions 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 the article [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. 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 snippet shows how to specify the startup command in the workflow file:

```
yml
```

```
  startup-command: startup.txt
```

When using Django, you typically want to migrate the data models using `python manage.py migrate` command after deploying the app code. You can run the `migrate` command in a post-deployment script.

Disconnect GitHub Actions

Disconnecting GitHub Actions from your App Service allows you to reconfigure the app deployment. You can choose what happens to your workflow file after you disconnect, whether to save or delete the file.

Azure CLI

Disconnect GitHub Actions with Azure CLI [az webapp deployment github-actions remove](#) command.

Bash

```
az webapp deployment github-actions remove \
--repo "<github-user>/<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 tutorial, delete the resource group that contains the App Service 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 the resource group.

Bash

```
az group delete --name <resource-group-name>
```

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.

If you deleted the Azure resource group, consider also making the following modifications to the GitHub account and repo that was connected for continuous deployment:

- In the repository, remove the *.github/workflows/<workflow-name>.yml* file.
- In the 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.

Use Azure Pipelines to build and deploy a Python web app to Azure App Service

Article • 04/01/2024

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

- An Azure subscription. If you don't have one, create a [free account](#).
- A GitHub account. If you don't have one, [create one for free](#).
- An Azure DevOps Services organization. [Create one for free](#).

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
```

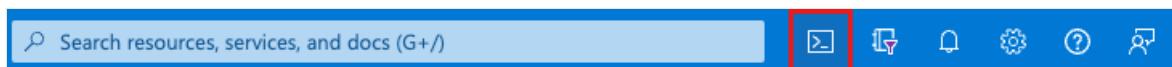
2. Build and run the app

```
Linux  
Bash  
  
python -m venv .env  
source .env/bin/activate  
pip install --upgrade pip  
pip install -r ./requirements.txt  
export set FLASK_APP=hello_app.webapp  
python3 -m flask run
```

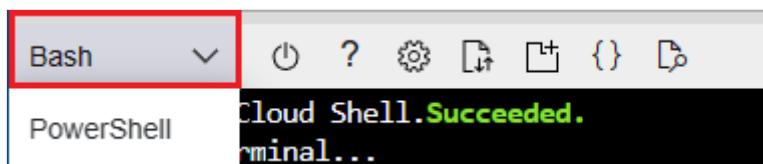
3. To view the app, open a browser window and go to <http://localhost:5000>. Verify that you see the title `Visual Studio Flask Tutorial`.
4. When you're finished, close the browser window and stop the Flask server with `Ctrl + C`.

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",  
  "status": "Running",  
  "type": "WebApp",  
  "id": "https://<your-web-app-name>.azurewebsites.net"
```

```
"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.

➊ 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

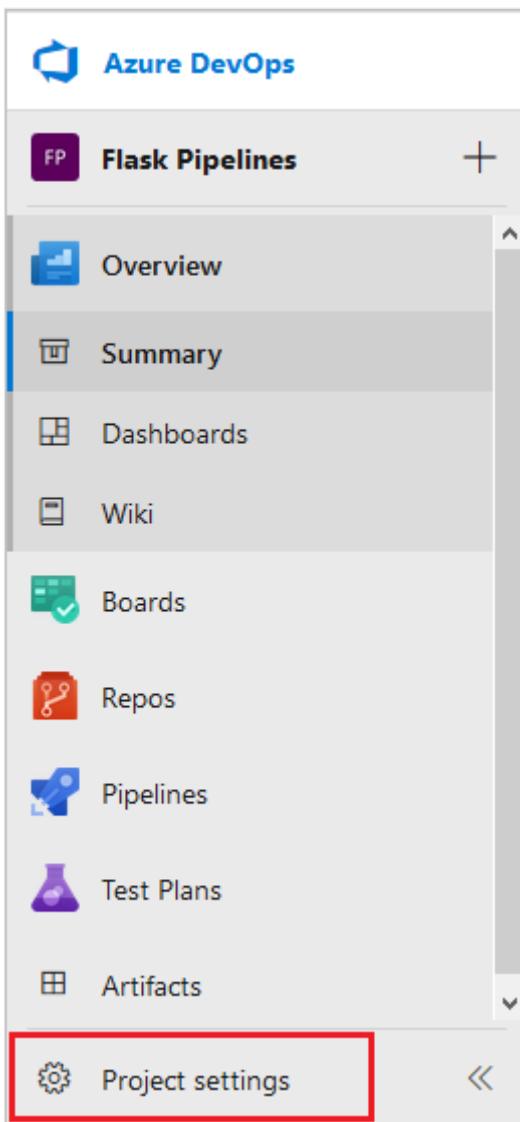
Create a new Azure DevOps project.

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](#)

Next

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.

Field	Description
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.

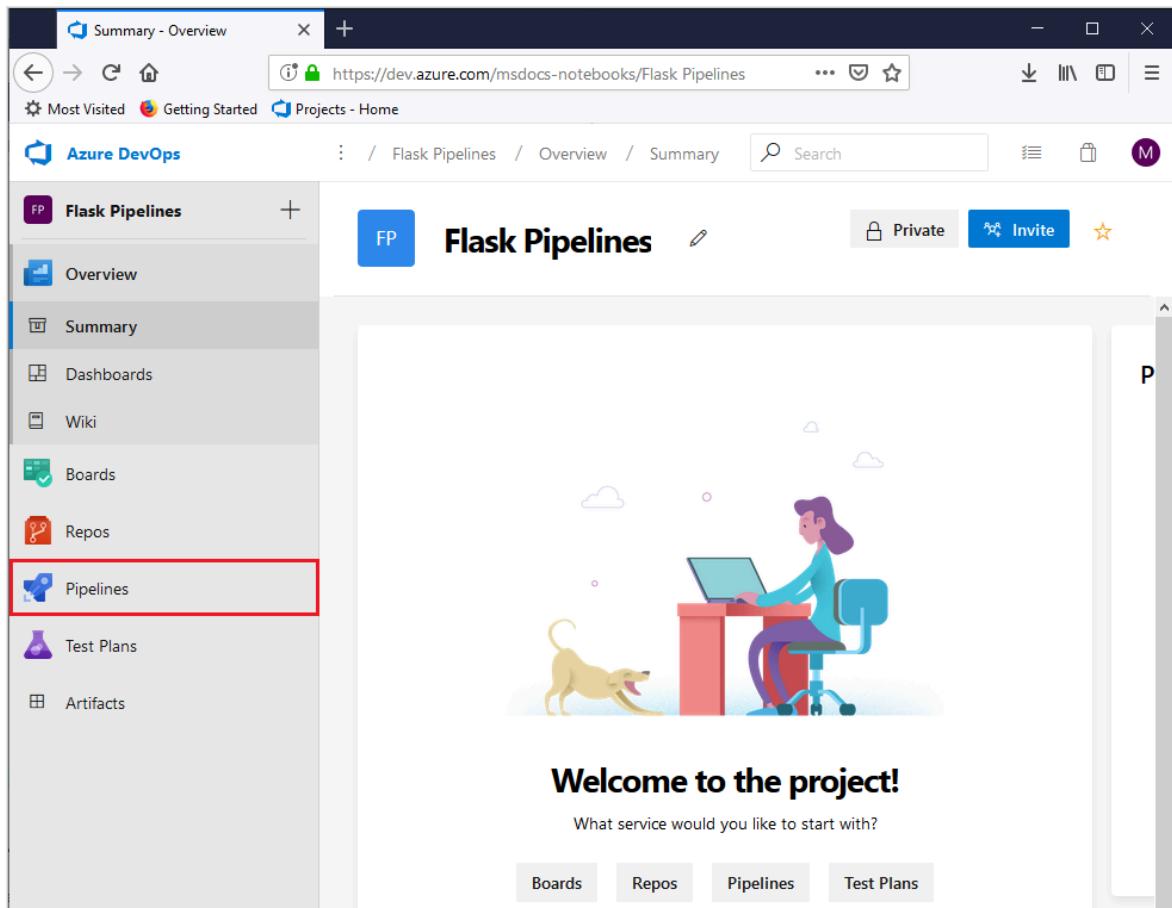
The new connection appears in the **Service connections** list, and is ready for use in your Azure Pipeline.

Create a pipeline

Create a pipeline to build and deploy your Python web app to Azure App Service. To understand pipeline concepts, watch:

<https://www.microsoft.com/en-us/videoplayer/embed/RWMIMo?postJsIMsg=true>

1. On the left navigation menu, select **Pipelines**.



2. Select Create Pipeline.



Create your first Pipeline

Automate your build and release processes using our wizard, and go from code to cloud-hosted within minutes.

[Create Pipeline](#)

:

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.

The screenshot shows the 'Select' step of the Azure Pipelines pipeline creation wizard. At the top, there are tabs: 'Connect', 'Select' (which is highlighted in blue), 'Configure', and 'Review'. Below the tabs, it says 'New pipeline' and 'Select a repository'. There is a search bar with 'Filter by keywords' and a dropdown menu 'My repositories' with a 'X' button. A list of repositories is shown:

Repository	Last updated
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 a GitHub repository page for 'Mycode/python-sample-vscode-flask-tutorial'. The top navigation bar includes 'Search or jump to...', 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the left, there's a sidebar with 'Personal settings' and links for 'Profile', 'Account', 'Emails', and 'Notifications'. The main content area features the 'Azure Pipelines' extension card:

Azure Pipelines
Installed 7 days ago | Developed by AzurePipelines | <https://azure.microsoft.com/services/devops/pipelines/>

Continuously build, test, and deploy to any platform and cloud
Azure Pipelines offers cloud-hosted pipelines for Linux, macOS, and Windows with 10 free parallel jobs and unlimited minutes for open source projects.

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' dialog box. It has two options:

- All repositories
This applies to all current *and* future repositories.
- Only select repositories
Select repositories ▾

At the bottom, there are 'Approve and install' and 'Cancel' buttons. The 'Approve and install' button is highlighted with a red border.

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](#).

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-web-app-name>'

# Agent VM image name
vmImageName: 'ubuntu-latest'

# Environment name
environmentName: '<your-web-app-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'
```

Variable	Description
<code>azureServiceConnectionId</code>	The ID or name of the Azure Resource Manager service connection.
<code>webAppName</code>	The name of the Azure App Service web app.
<code>vmImageName</code>	The name of the operating system to use for the build agent.
<code>environmentName</code>	The name of the environment used in the deployment stage. The environment is automatically created when the stage job is run.
<code>projectRoot</code>	The root folder containing the app code.
<code>pythonVersion</code>	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 `vmImageName` variable.

```
yml
- job: BuildJob
  pool:
    vmImage: $(vmImageName)
```

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` file. The `--target` parameter specifies the location to install the dependencies. 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 setup
    pip install --target="./.python_packages/lib/site-packages" -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.
<code>replaceExistingArchive</code>	Indicates whether to replace an existing archive if the file already exists. Set to <code>true</code> .

Parameter	Description
<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 <code>.zip</code> file to deploy.

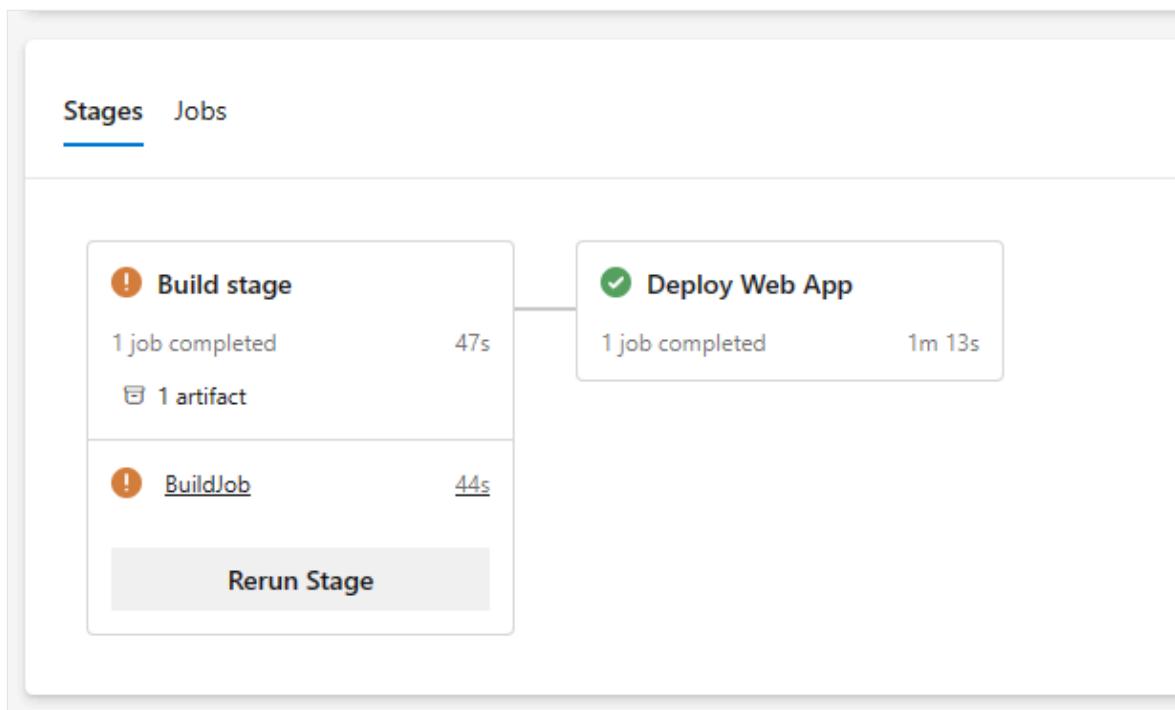
Also, because the `python-vscode-flask-tutorial` repository contains the same startup command in a file named `startup.txt`, you can specify that file by adding the parameter: `startUpCommand: 'startup.txt'`.

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

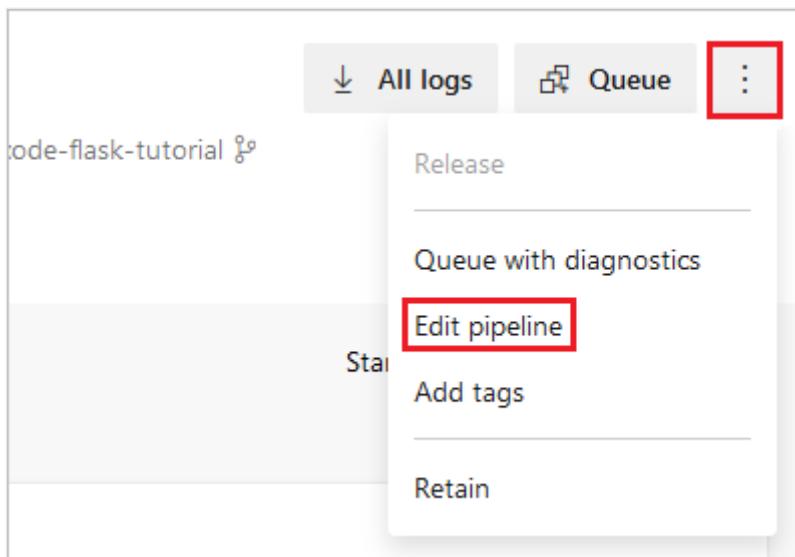
Build stage

▼	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

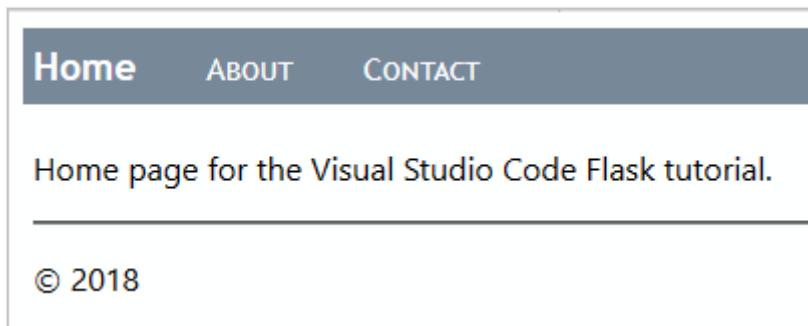
▼	DeploymentJob	59s
	Initialize job	4s
	Download Artifact	4s
	Use Python version	<1s
	Deploy Azure Web Ap...	36s
	Finalize Job	<1s

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](#)
-

Feedback

Was this page helpful?



[Provide product feedback ↗](#)

Quickstart: Sign in users and call Microsoft Graph from a Python Flask web app

Article • 04/19/2024

In this quickstart, you download and run a Python Flask web app sample that demonstrates how to authenticate users and call the Microsoft Graph API. Users in your Microsoft Entra organization can sign into the application.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- A Microsoft Entra tenant. For more information, see [how to get a Microsoft Entra tenant](#).
- [Python 3 +](#)

Step 1: Register your application

💡 Tip

Steps in this article might vary slightly based on the portal you start from.

Follow these steps to register your application in the Microsoft Entra admin center:

1. Sign in to the [Microsoft Entra admin center](#) as at least a [Cloud Application Administrator](#).
2. If you have access to multiple tenants, use the **Settings** icon  in the top menu to switch to the tenant in which you want to register the application from the [Directories + subscriptions](#) menu.
3. Browse to **Identity > Applications > App registrations** and select **New registration**.
4. Enter a **Name** for your application, for example *python-webapp*.
5. Under **Supported account types**, select **Accounts in this organizational directory only**.
6. Under **Redirect URIs**, select **Web** for the platform.
7. Enter a redirect URI of `http://localhost:5000/getAToken`. You can change this value later.

8. Select Register.

Step 2: Add a client secret

The sample app uses a client secret to prove its identity when it requests for tokens. Follow these steps to create a client secret for your Python web app:

1. On the app **Overview** page, note the **Application (client) ID** value for later use.
2. Under **Manage**, select the **Certificates & secrets** and from the **Client secrets** section, select **New client secret**.
3. Enter a description for the client secret, leave the default expiration, and select **Add**.
4. Save the **Value** of the **Client Secret** in a safe location. You need this value to configure the code, and you can't retrieve it later.

When creating credentials for a confidential client application, Microsoft recommends that you use a certificate instead of a client secret before moving the application to a production environment. For more information on how to use a certificate, see [these instructions](#).

Step 3: Add a scope

Since this app signs in users, you need to add delegated permissions:

1. Under **Manage**, select **API permissions > Add a permission**.
2. Ensure that the **Microsoft APIs** tab is selected.
3. From the **Commonly used Microsoft APIs** section, select **Microsoft Graph**.
4. From the **Delegated permissions** section, ensure that **User.Read** is selected. Use the search box if necessary.
5. Select **Add permissions**.

Step 4: Download the sample app

[Download the Python code sample](#) or clone the repository:

Console

```
git clone https://github.com/Azure-Samples/ms-identity-docs-code-python/
```

Step 5: Configure the sample app

1. Open the application you downloaded in an IDE and navigate to root folder of the sample app.

```
Console
```

```
cd flask-web-app
```

2. Create an `.env` file in the root folder of the project using `.env.sample` as a guide.

```
Python
```

```
# The following variables are required for the app to run.  
CLIENT_ID=<Enter_your_client_id>  
CLIENT_SECRET=<Enter_your_client_secret>  
AUTHORITY=<Enter_your_authority_url>
```

- Set the value of `CLIENT_ID` to the **Application (client) ID** for the registered application, available on the overview page.
- Set the value of `CLIENT_SECRET` to the client secret you created in the **Certificates & Secrets** for the registered application.
- Set the value of `AUTHORITY` to a
`https://login.microsoftonline.com/<TENANT_GUID>`. The **Directory (tenant) ID** is available on the app registration overview page.

The environment variables are referenced in `app_config.py`, and are kept in a separate `.env` file to keep them out of source control. The provided `.gitignore` file prevents the `.env` file from being checked in.

Step 6: Run the sample app

1. Create a virtual environment for the app:

```
Windows
```

```
Cmd
```

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

2. Install the requirements using `pip`:

```
Console
```

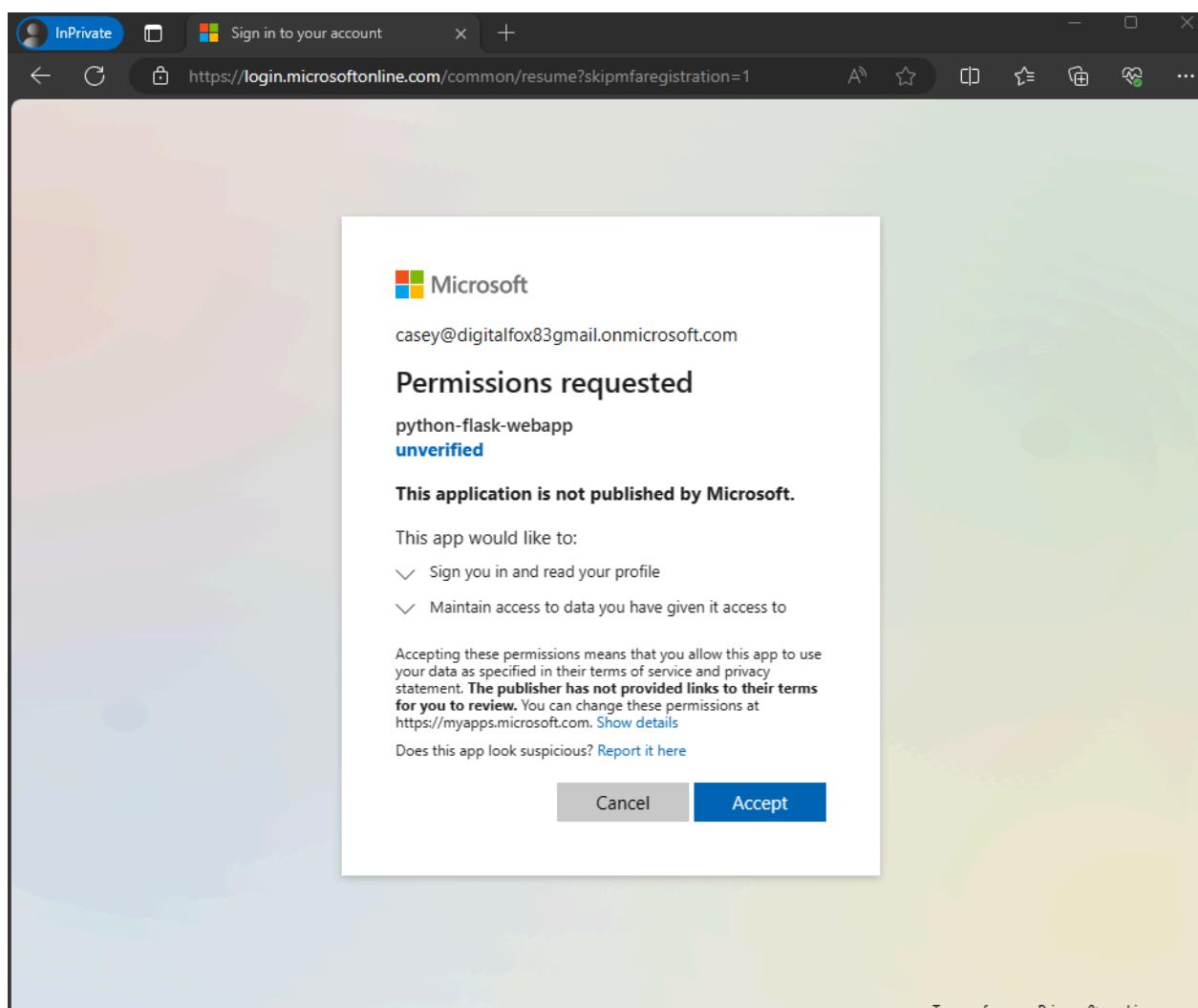
```
pip install -r requirements.txt
```

3. Run the app from the command line. Ensure your app is running on the same port as the redirect URI you configured earlier.

```
Console
```

```
flask run --debug --host=localhost --port=5000
```

4. Copy the https URL that appears in the terminal, for example, <https://localhost:5000>, and paste it into a browser. We recommend using a private or incognito browser session.
5. Follow the steps and enter the necessary details to sign in with your Microsoft account. You're requested to provide an email address and password to sign in.
6. The application requests permission to maintain access to data you've given it access to, and to sign you in and read your profile, as shown. Select Accept.

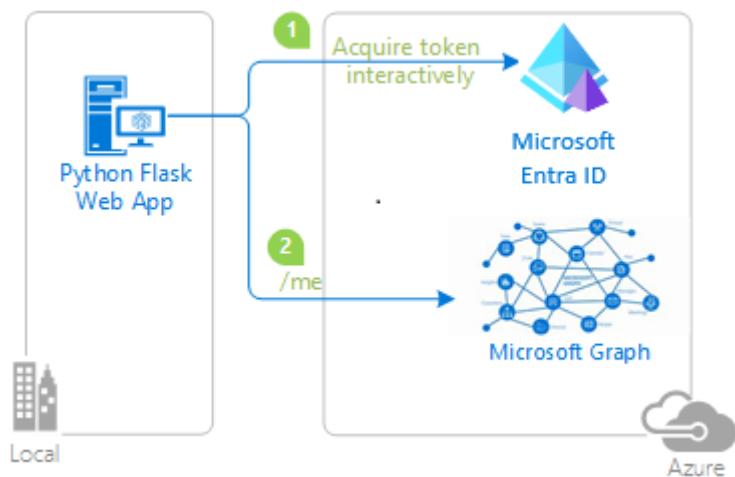


1. The following screenshot appears, indicating that you've successfully signed in to the application.

A screenshot of a web browser window titled "Microsoft Identity Python Web App". The address bar shows "localhost:5000". The page content includes the heading "Microsoft Identity Python Web App", a welcome message "Welcome Casey Jensen!", and two links: "Call a downstream API" and "Logout". At the bottom right, it says "Microsoft identity platform Web App Sample 0.8.0".

How it works

The following diagram demonstrates how the sample app works:



1. The application uses the [identity package](#) to obtain an access token from the Microsoft identity platform. This package is built on top of the Microsoft Authentication Library (MSAL) for Python to simplify authentication and authorization in web apps.
2. The access token you obtain in the previous step is used as a bearer token to authenticate the user when calling the Microsoft Graph API.

Next steps

Learn more by building a Python web app that signs in users and calls a protected web API in the following multi-part tutorial series:

Tutorial: Web app that signs in users

Quickstart: Azure Key Vault secret client library for Python

Article • 04/07/2024

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

Azure CLI

To grant your application permissions to your key vault through Role-Based Access Control (RBAC), assign a role using the Azure CLI command [az role assignment create](#).

Azure CLI

```
az role assignment create --role "Key Vault Secrets User" --assignee "<app-id>" --scope "/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>/providers/Microsoft.KeyVault/vaults/<your-unique-keyvault-name>"
```

Replace `<app-id>`, `<subscription-id>`, `<resource-group-name>` and `<your-unique-keyvault-name>` with your actual values. `<app-id>` is the Application (client) ID of your registered application in Azure AD.

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 a function in Azure with Python using Visual Studio Code

Article • 03/05/2024

ⓘ AI-assisted content. This article was partially created with the help of AI. An author reviewed and revised the content as needed. [Learn more](#)

In this article, you use Visual Studio Code to create a Python function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

This article uses the Python v2 programming model for Azure Functions, which provides a decorator-based approach for creating functions. To learn more about the Python v2 programming model, see the [Developer Reference Guide](#)

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [CLI-based version](#) of this article.

This video shows you how to create a Python function in Azure using Visual Studio Code.

[https://learn-video.azurefd.net/vod/player?id=a1e10f96-2940-489c-bc53-da2b915c8fc2&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Fcreate-first-function-vs-code-python ↗](https://learn-video.azurefd.net/vod/player?id=a1e10f96-2940-489c-bc53-da2b915c8fc2&locale=en-us&embedUrl=%2Fazure%2Fazure-functions%2Fcreate-first-function-vs-code-python)

The steps in the video are also described in the following sections.

Configure your environment

Before you begin, make sure that you have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free ↗](#).
- A Python version that is [supported by Azure Functions](#). For more information, see [How to install Python ↗](#).
- [Visual Studio Code ↗](#) on one of the [supported platforms ↗](#).
- The [Python extension ↗](#) for Visual Studio Code.
- The [Azure Functions extension ↗](#) for Visual Studio Code, version 1.8.1 or later.

- The [Azurite V3 extension](#) local storage emulator. While you can also use an actual Azure storage account, this article assumes you're using the Azurite emulator.

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.

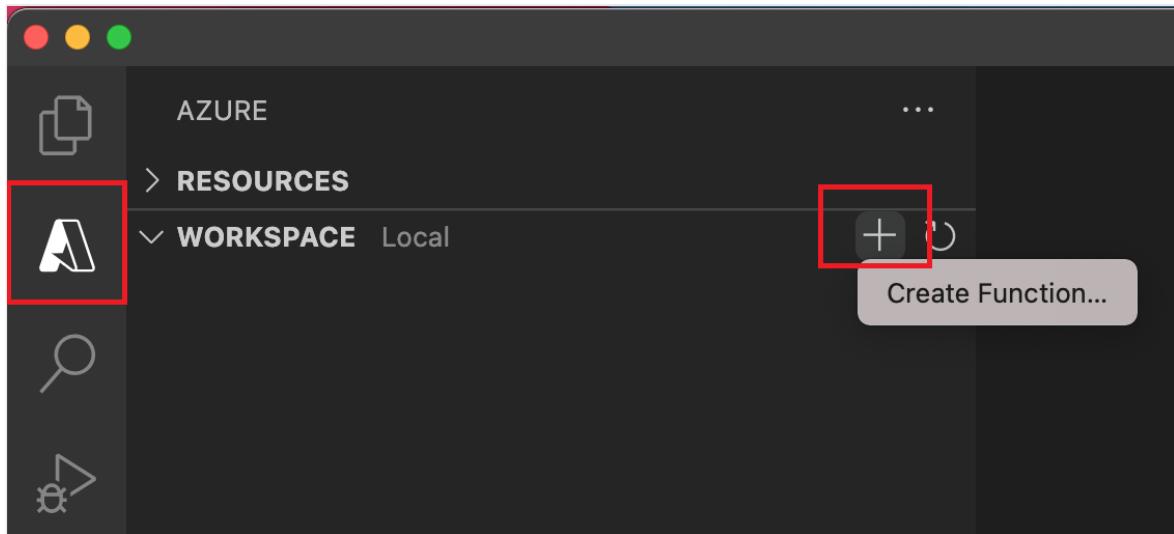
In Visual Studio Code, select F1 to open the command palette, and then search for and run the command **Azure Functions: Install or Update Core Tools**.

This command starts a package-based installation of the latest version of Core Tools.

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in Python. Later in this article, you publish your function code to Azure.

1. Choose the Azure icon in the Activity bar. Then in the **Workspace (local)** area, select the + button, choose **Create Function** in the dropdown. When prompted, choose **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:

Prompt	Selection
Select a language	Choose <code>Python (Programming Model V2)</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 about the authorization level, see Authorization keys .
Select how you would like to open your project	Choose <code>Open in current window</code> .

4. Visual Studio Code uses the provided information and generates an Azure Functions project with an HTTP trigger. You can view the local project files in the Explorer. The generated `function_app.py` project file contains your functions.
5. Open the `local.settings.json` project file and verify that the `AzureWebJobsFeatureFlags` setting has a value of `EnableWorkerIndexing`. This is required for Functions to interpret your project correctly as the Python v2 model when running locally.
6. 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 currently required by the Python v2 model. When you publish your project to Azure, you need to instead use the default storage account. If you're instead using an Azure Storage account, set your storage account connection string here.

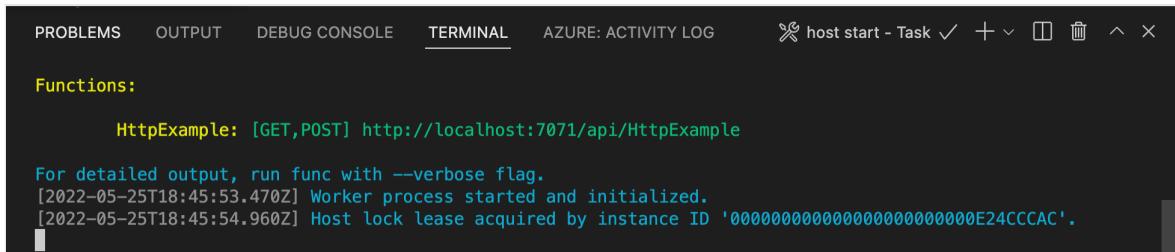
Start the emulator

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. :: zone-end

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:

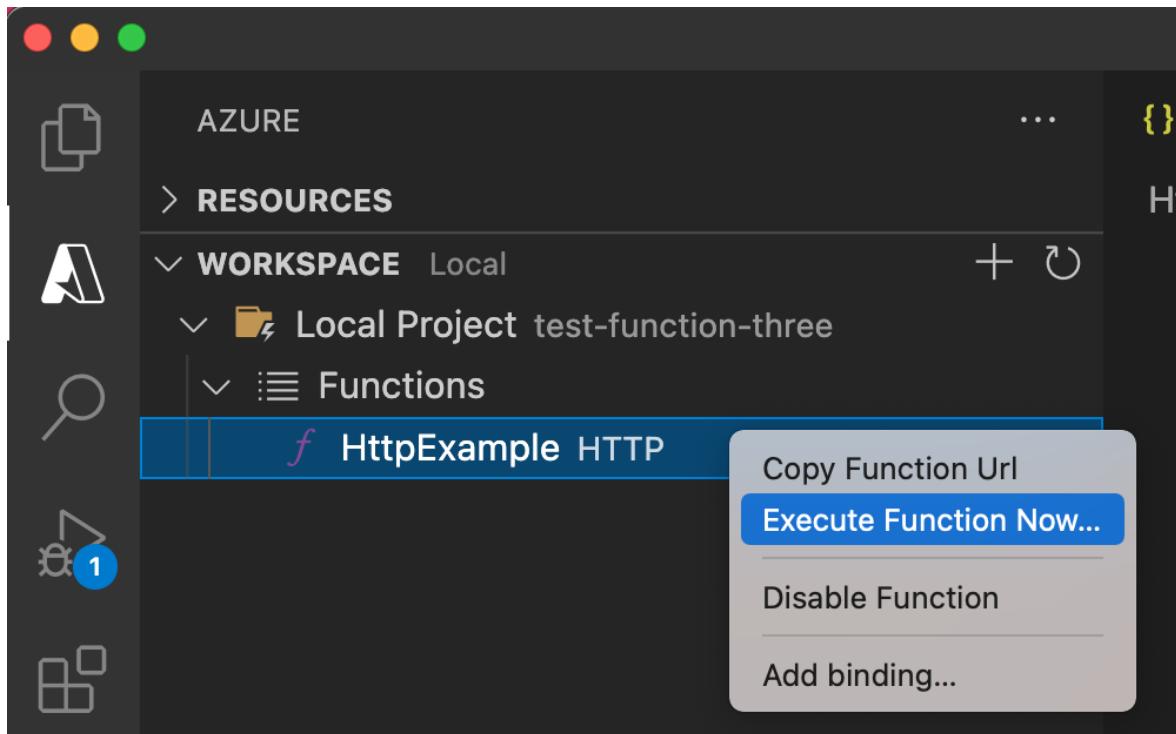
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AZURE: ACTIVITY LOG
✖ host start - Task ✓ + ×

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....**



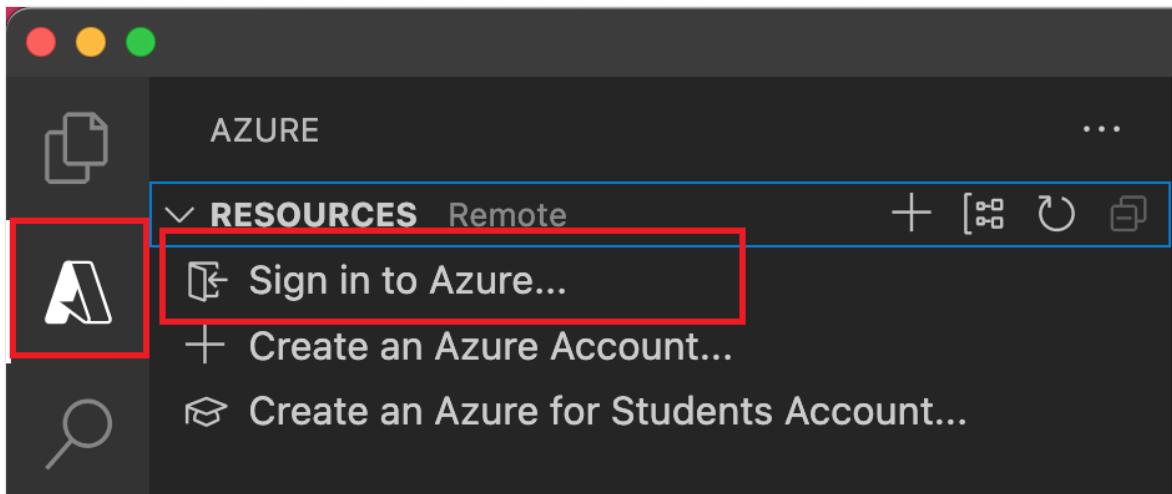
3. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
4. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution 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, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar. Then in the **Resources** area, choose **Sign in to Azure....**



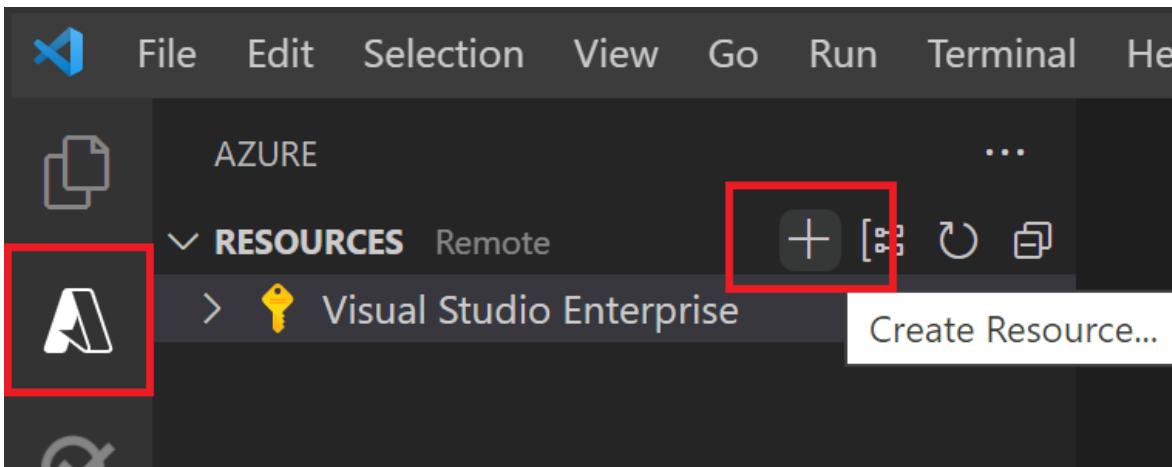
If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, choose [Create an Azure Account....](#) Students can choose [Create an Azure for Students Account....](#)

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the sidebar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription.

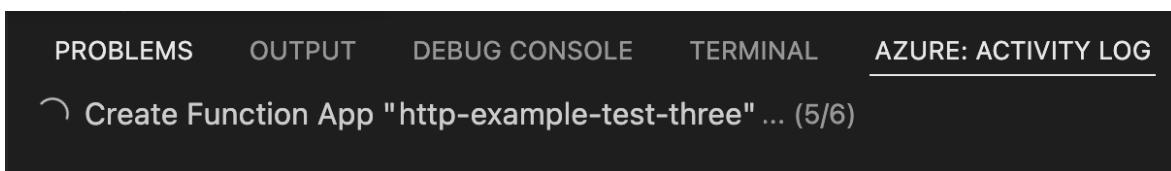
1. Choose the Azure icon in the Activity bar. Then in the Resources area, select the + icon and choose the [Create Function App in Azure](#) option.



2. Provide the following information at the prompts:

Prompt	Selection
Select subscription	Choose the subscription to use. You won't see this prompt when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Choose the language version on which you've been running locally.
Select a location for new resources	For better performance, choose a region ↗ near you.

The extension shows the status of individual resources as they're being created in Azure in the **Azure: Activity Log** panel.



- When the creation is complete, the following Azure resources are created in your subscription. The resources are named based on your function app name:

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- 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 App Service plan, which defines the underlying host for your function app.
- An Application Insights instance connected to the function app, which tracks usage of your functions in the app.

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 function app name you provide. By default, they're also created

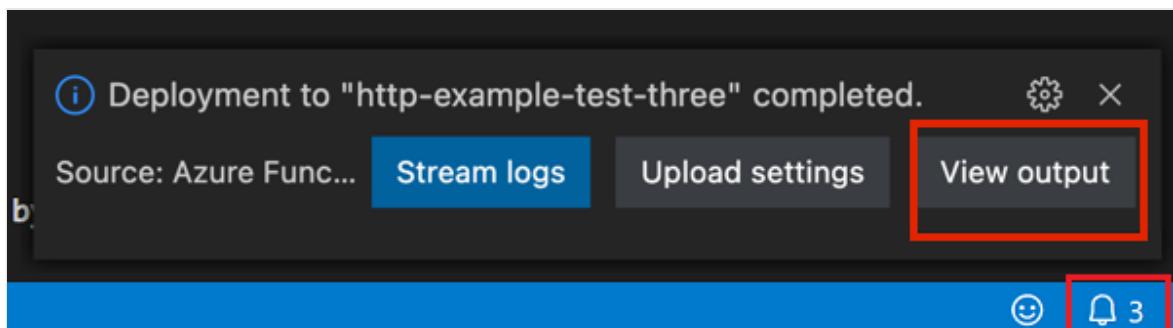
in the same new resource group with the function app. If you want to either customize the names of these resources or reuse existing resources, you need to [publish the project with advanced create options](#) instead.

Deploy the project to Azure

ⓘ Important

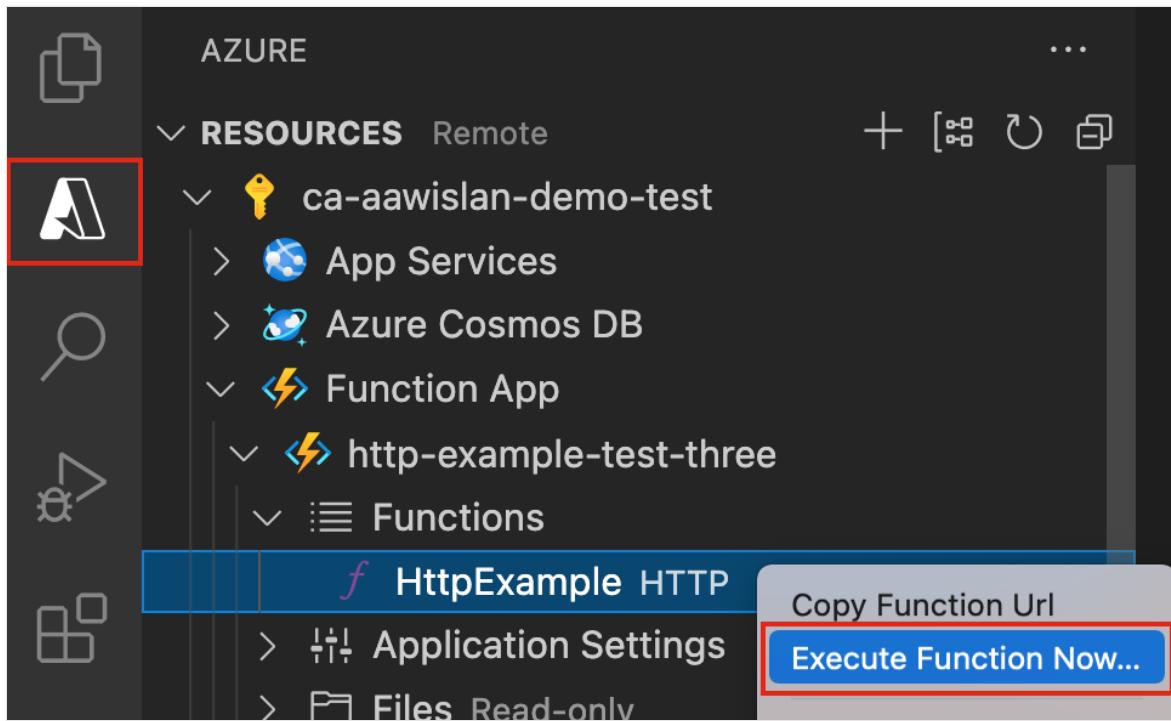
Deploying to an existing function app always overwrites the contents of that app in Azure.

1. In the **Resources** area of the Azure activity, locate the function app resource you just created, right-click the resource, and select **Deploy to function app....**
2. When prompted about overwriting previous deployments, select **Deploy** to deploy your function code to the new function app resource.
3. After deployment completes, 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. Back in the **Resources** area in the side bar, expand your subscription, your new function app, and **Functions**. Right-click (Windows) or **Ctrl - click** (macOS) the `HttpExample` function and choose **Execute Function Now....**



2. In **Enter request body** you see the request message body value of `{ "name": "Azure" }`. Press Enter to send this request message to your function.
3. When the function executes in Azure and returns a response, a notification is raised in Visual Studio Code.

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, 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 an App Service named 'myfunctionapp'. The left sidebar has a 'Functions' section with 'Overview' selected. The main content area shows the 'Resource group (change)' section with 'myResourceGroup' highlighted by a red box. Other details include Status: Running, Location: Central US, Subscription: Visual Studio Enterprise, Subscription ID: 11111111-1111-1111-1111-111111111111, and Tags: Click here to add tags. At the bottom, there are tabs for Metrics, Features (8), Notifications (0), and Quickstart.

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.

For more information about Functions costs, see [Estimating Consumption plan costs](#).

Next steps

You created and deployed a function app with a simple HTTP-triggered function. In the next articles, you expand that function by connecting to a storage service in Azure. To learn more about connecting to other Azure services, see [Add bindings to an existing function in Azure Functions](#).

[Connect to Azure Cosmos DB](#)

[Connect to an Azure Storage queue](#)

Having issues? Let us know. ↗

Quickstart: Create a Python function in Azure from the command line

Article • 03/05/2024

In this article, you use command-line tools to create a Python function that responds to HTTP requests. After testing the code locally, you deploy it to the serverless environment of Azure Functions.

This article uses the Python v2 programming model for Azure Functions, which provides a decorator-based approach for creating functions. To learn more about the Python v2 programming model, see the [Developer Reference Guide](#)

Completing this quickstart incurs a small cost of a few USD cents or less in your Azure account.

There's also a [Visual Studio Code-based version](#) of this article.

Configure your local environment

Before you begin, you must have the following requirements in place:

- An Azure account with an active subscription. [Create an account for free](#).
- One of the following tools for creating Azure resources:
 - [Azure CLI](#) version 2.4 or later.
 - The Azure [Az PowerShell module](#) version 5.9.0 or later.
- A [Python version supported by Azure Functions](#).
- The [Azurite storage emulator](#). While you can also use an actual Azure Storage account, the article assumes you're using this emulator.

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.

Use the `func --version` command to make sure your version of Core Tools is at least [4.0.5530](#).

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 that you're using a [version of Python 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 function

In Azure Functions, a function project is a container for one or more individual functions that each responds to a specific trigger. All functions in a project share the same local and hosting configurations.

In this section, you create a function project and add an HTTP triggered function.

1. Run the `func init` command as follows to create a Python v2 functions project in the virtual environment.

```
Console  
func init --python
```

The environment now contains various files for the project, including configuration 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.

2. Add a function to your project by using the following command, where the `--name` argument is the unique name of your function (`HttpExample`) and the `--template` argument specifies the function's trigger (HTTP).

```
Console  
func new --name HttpExample --template "HTTP trigger" --authlevel  
"anonymous"
```

If prompted, choose the **ANONYMOUS** option. `func new` adds an HTTP trigger endpoint named `HttpExample` to the `function_app.py` file, which is accessible without authentication.

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.

Create supporting Azure resources for your function

Before you can deploy your function code to Azure, you need to create three resources:

- A resource group, which is a logical container for related resources.
- A storage account, which maintains the state and other information about your projects.
- 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 following commands to create these items. Both Azure CLI and PowerShell are supported.

1. If needed, sign in to Azure.

```
Azure CLI
```

```
Azure CLI
```

```
az login
```

The `az login` command signs you into your Azure account.

2. Create a resource group named `AzureFunctionsQuickstart-rg` in your chosen region.

```
Azure CLI
```

```
Azure CLI
```

```
az group create --name AzureFunctionsQuickstart-rg --location  
<REGION>
```

The `az group create` command creates a resource group. In the above command, replace `<REGION>` with a region near you, using an available region code returned from the [az account list-locations](#) command.

ⓘ Note

You can't host Linux and Windows apps in the same resource group. If you have an existing resource group named `AzureFunctionsQuickstart-rg` with a Windows function app or web app, you must use a different resource group.

3. Create a general-purpose storage account in your resource group and region.

```
Azure CLI
```

```
Azure CLI
```

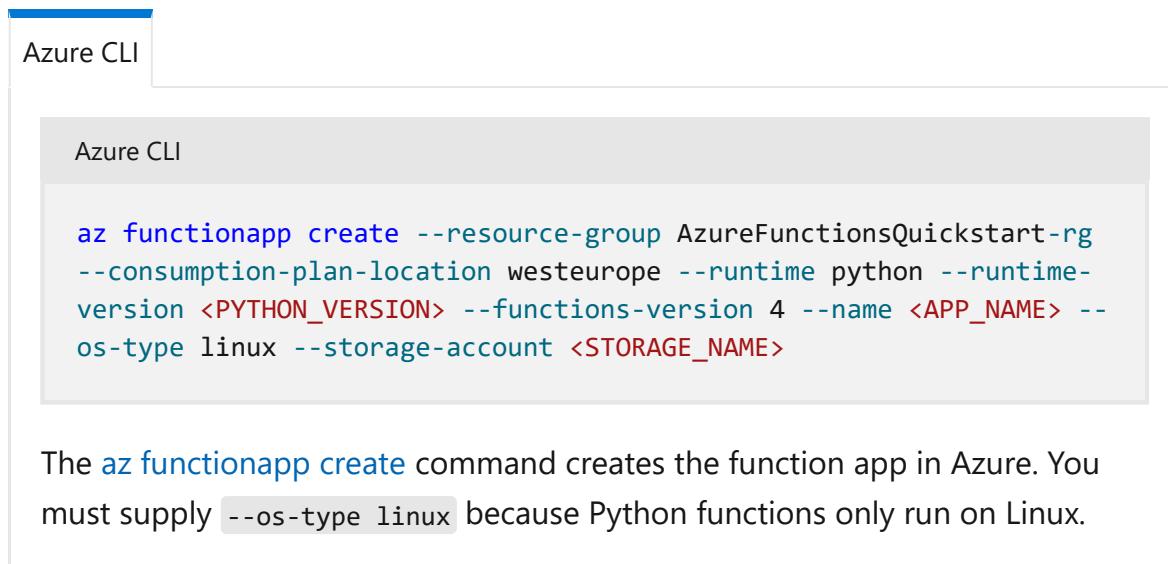
```
az storage account create --name <STORAGE_NAME> --location <REGION>  
--resource-group AzureFunctionsQuickstart-rg --sku Standard_LRS
```

The `az storage account create` command creates the storage account.

In the previous example, replace `<STORAGE_NAME>` with a name that's appropriate to you and unique in Azure Storage. Names must contain 3 to 24 characters numbers and lowercase letters only. `Standard_LRS` specifies a general-purpose account supported by Functions.

The storage account incurs only a few cents (USD) for this quickstart.

4. Create the function app in Azure.



Azure CLI

```
az functionapp create --resource-group AzureFunctionsQuickstart-rg --consumption-plan-location westeurope --runtime python --runtime-version <PYTHON_VERSION> --functions-version 4 --name <APP_NAME> --os-type linux --storage-account <STORAGE_NAME>
```

The `az functionapp create` command creates the function app in Azure. You must supply `--os-type linux` because Python functions only run on Linux.

In the previous example, replace `<APP_NAME>` with a globally unique name appropriate to you. The `<APP_NAME>` is also the default subdomain for the function app. Make sure that the value you set for `<PYTHON_VERSION>` is a version supported by Functions and is the same version you used during local development.

This command creates a function app running in your specified language runtime under the [Azure Functions 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 monitor your function app and view logs. For more information, see [Monitor Azure Functions](#). The instance incurs no costs until you activate it.

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 the following example, replace `<APP_NAME>` with the name of your app.

Console

```
func azure functionapp publish <APP_NAME>
```

The publish command 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, you invoke it by making an HTTP request to its URL in the browser or with a tool like curl.

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 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 Cosmos DB](#)

[Connect to an Azure Storage queue](#)

Having issues with this article?

- [Troubleshoot Python function apps in Azure Functions](#)
- [Let us know ↗](#)

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')

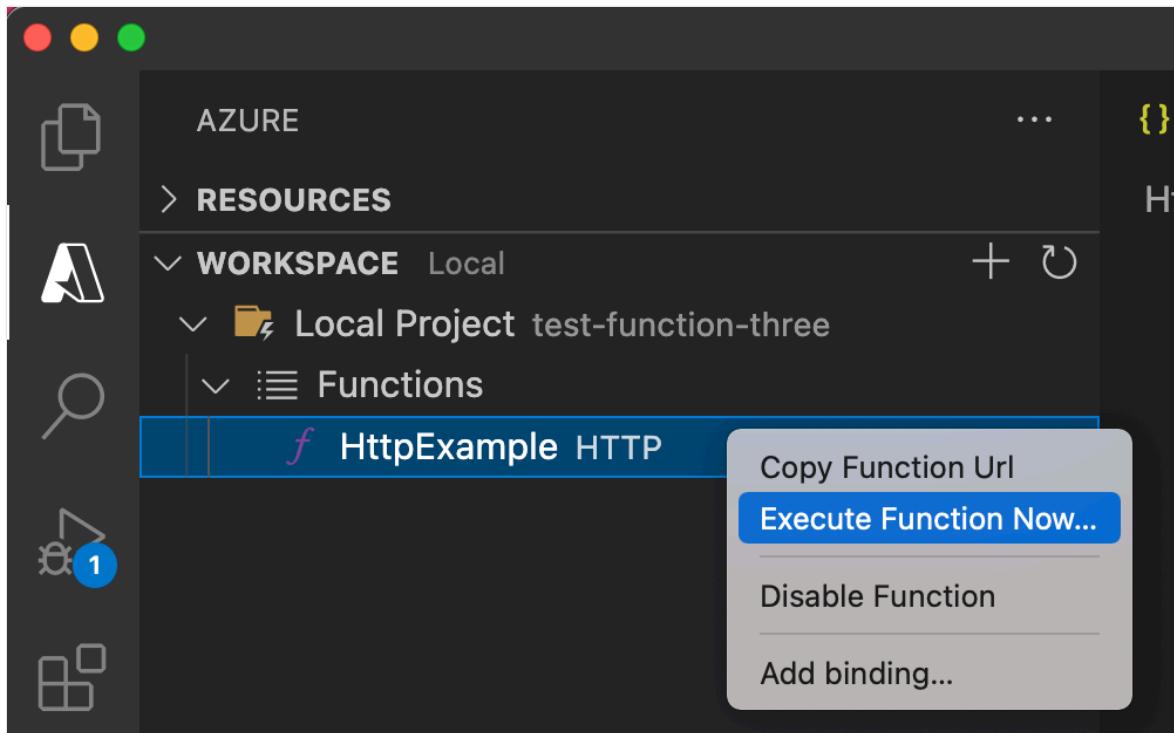
if name:
    msg.set(name)
    return func.HttpResponse(f"Hello, {name}. This HTTP triggered
function executed successfully.")
else:
    return func.HttpResponse(
        "This HTTP triggered function executed successfully. Pass a
name in the query string or in the request body for a personalized
response.",
        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.

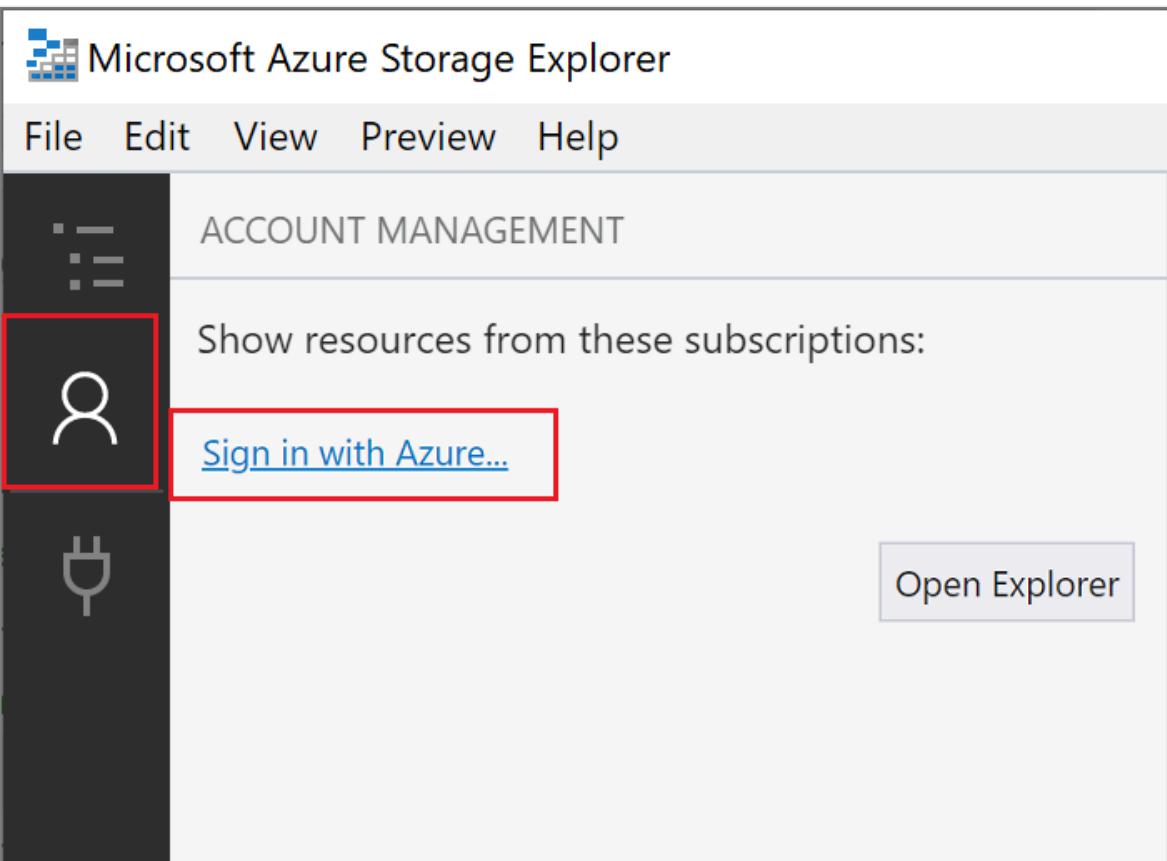
4. After a response is returned, press **Ctrl + C** to stop Core Tools.

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.

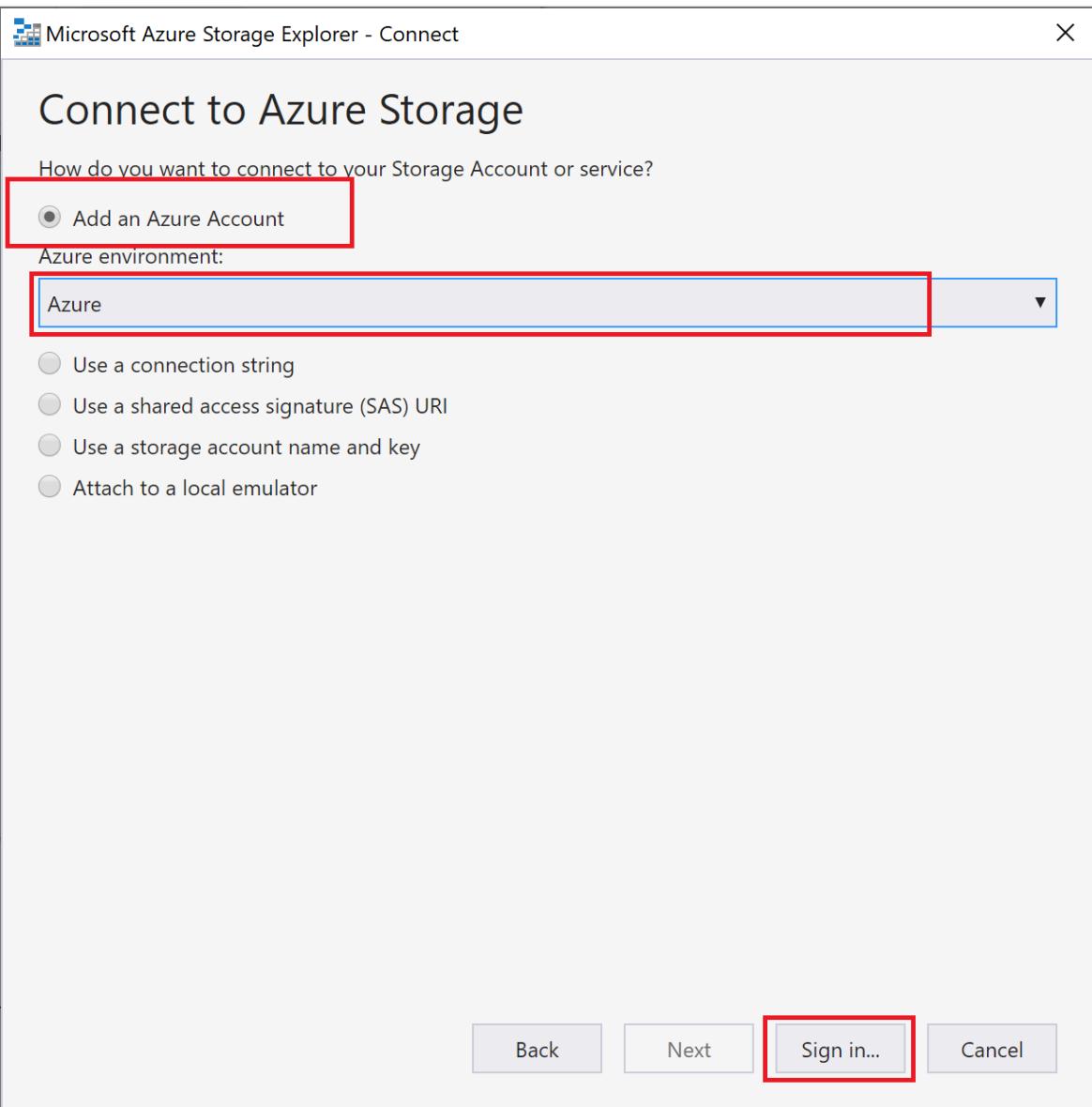
Connect Storage Explorer to your account

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....**

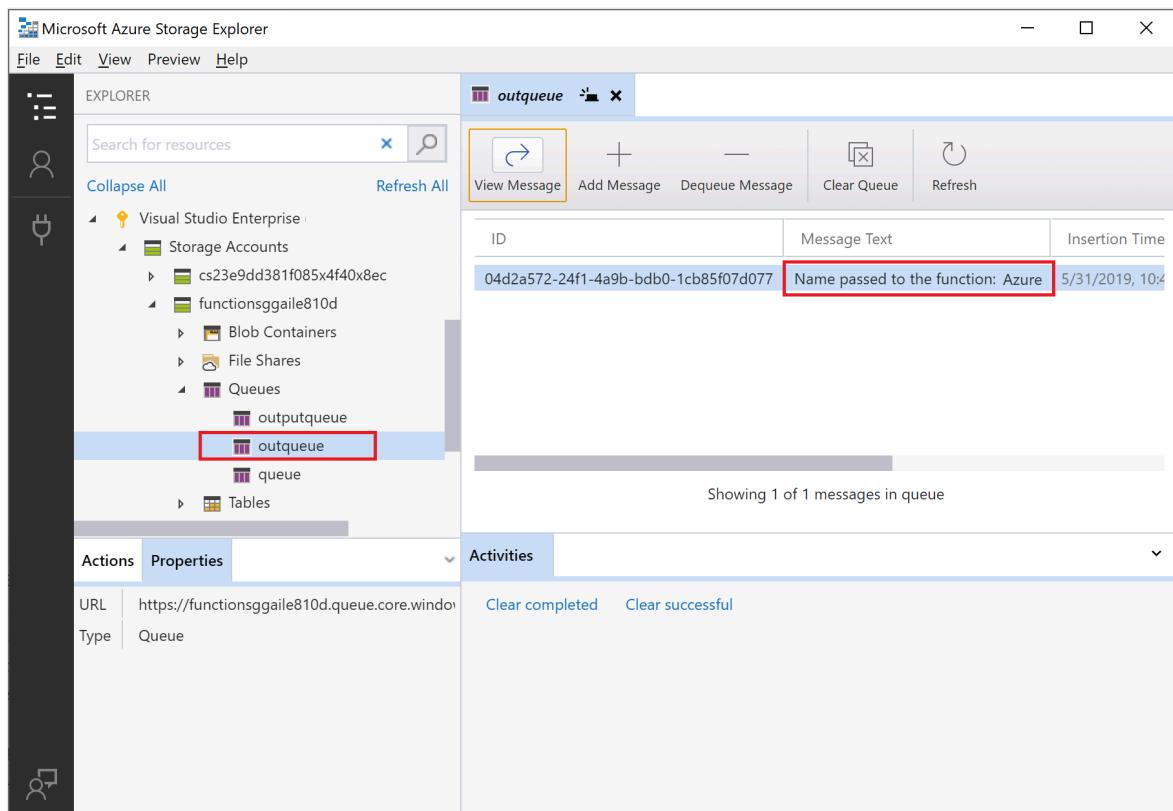


After you successfully sign in to your account, you see all of the Azure subscriptions associated with your account. Choose your subscription and select **Open Explorer**.

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

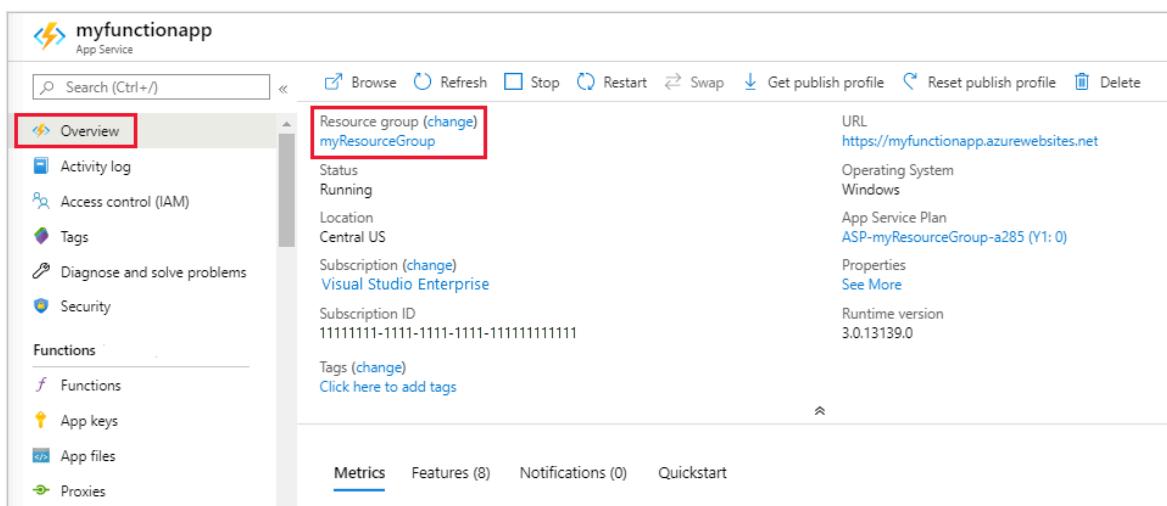
1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select **Azure Functions: Deploy to function app....**
2. Choose the function app that you created in the first article. Because you're redeploying your project to the same app, select **Deploy** 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
- Metrics, Features (8), Notifications (0), Quickstart buttons at the bottom

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 • 04/25/2024

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 additional 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

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, replace `<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

Now that you've verified 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've finished, 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](#)

Data solutions for Python apps on Azure

Article • 03/14/2024

Azure offers a choice of fully managed relational, NoSQL, and in-memory databases, spanning proprietary and open-source engines in addition to storage services for object, block, and file storage. The following articles help you get started with Python data solutions on Azure.

Databases

- **PostgreSQL:** Build scalable, secure, and fully managed enterprise-ready apps on open-source PostgreSQL, scale out single-node PostgreSQL with high performance, or migrate 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 apps that scale with managed and intelligent SQL 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 apps that scale with managed and intelligent SQL database 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 applications with guaranteed low latency and high availability anywhere, at any 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:** Massively scalable and secure object storage for cloud-native workloads, archives, data lakes, high-performance computing, and machine learning.
 - [Quickstart: Azure Blob Storage client library for Python](#)
 - [Azure Storage samples using v12 Python client libraries](#)
- **Azure Data Lake Storage Gen2:** Massively scalable and secure data lake for your high-performance analytics workloads.
 - [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:** Power fast, scalable applications with an open-source-compatible in-memory data store.
 - [Quickstart: Use Azure Cache for Redis in Python](#)

Big data and analytics

- **Azure Data Lake analytics:** A fully managed on-demand pay-per-job analytics service with 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 data integration service to orchestrate and automate data movement and transformation.
 - [Quickstart: Create a data factory and pipeline using Python](#)
 - [Transform data by running a Python activity in Azure Databricks](#)
- **Azure Event Hubs:** A hyper-scale telemetry ingestion service that collects, transforms, and stores millions of events.
 - [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 Hadoop and Spark service backed by 99.9% SLA for your enterprise
 - [Use Spark & Hive Tools for Visual Studio Code](#)

- **Azure Databricks:** A fully managed, fast, easy and collaborative Apache® Spark™ based analytics platform optimized for 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 limitless analytics service that brings together enterprise data warehousing and big data analytics.
 - [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

Article • 03/14/2024

The following articles help you get started with Azure Machine Learning. Azure Machine Learning v2 REST APIs, Azure CLI extension, and Python SDK accelerate the production machine learning lifecycle. The links in this article target v2, which is recommended if you're starting a new machine learning project.

Getting started

The workspace is the top-level resource for Azure Machine Learning, providing a centralized place to work with all the artifacts you create when you use Azure Machine Learning.

- [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 machine learning models for real-time inference.

- [Tutorial: Designer - deploy a machine learning model](#)
- [Deploy and score a machine learning model by using an online endpoint](#)

Automated machine learning

Automated machine learning, also referred to as automated ML or AutoML, is the process of automating the time-consuming, iterative tasks of machine learning model development.

- [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 bring data from a local machine or an existing cloud-based storage.

- [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 create a workflow that stitches together various ML phases.

- [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 • 01/12/2024

This article describes how to go from Python project code (for example, a web app) to a deployed Docker container in Azure. Discussed are the general process of containerization, deployment options for containers in Azure, and Python-specific configuration of containers in Azure.

The nature of Docker containers is that creating a Docker image from code and deploying that image to a container in Azure is similar across programming languages. The language-specific considerations - Python in this case - are in the configuration during the containerization process in Azure, in particular the Dockerfile structure and configuration supporting Python web frameworks such as [Django](#), [Flask](#), and [FastAPI](#).

Container workflow scenarios

For Python container development, some typical workflows for moving from code to container are:

[+] Expand table

Scenario	Description	Workflow
Dev	Build Python Docker images in your dev environment.	<p>Code: git clone code to dev environment (with Docker installed).</p> <p>Build: Use Docker CLI, VS Code (with extensions), PyCharm (with plugin). Described in section Working with Python Docker images and containers.</p> <p>Test: In dev environment in a Docker container.</p> <p>Push: To a registry like Azure Container Registry, Docker Hub, or private registry.</p> <p>Deploy: To Azure service from registry.</p>
Hybrid	From your dev environment, build Python Docker images in Azure.	Code: git clone code to dev environment (not necessary for Docker to be installed).

Scenario	Description	Workflow
		<p>Build: VS Code (with extensions), Azure CLI.</p> <p>Push: To Azure Container Registry</p> <p>Deploy: To Azure service from registry.</p>
Azure	All in the cloud; use Azure Cloud Shell to build Python Docker images code from GitHub repo.	<p>Code: git clone GitHub repo to Azure Cloud Shell.</p> <p>Build: In Azure Cloud Shell, use Azure CLI or Docker CLI.</p> <p>Push: To registry like Azure Container Registry, Docker Hub, or private registry.</p> <p>Deploy: To Azure service from registry.</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 that's hosted in the cloud), or [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>A fully managed hosting service for containerized web applications including websites and web APIs. Containerized web apps on Azure App Service can scale as needed and use streamlined CI/CD workflows with Docker Hub, Azure Container Registry, and GitHub. Ideal as an easy on-ramp for developers to take advantage of the fully managed Azure App Service platform, but who also want a single deployable artifact containing an app and all of its dependencies.</p> <p>Example: Deploy a Flask or FastAPI web app on Azure App Service.</p>
Azure Container Apps (ACA)	<p>A fully managed serverless container service powered by Kubernetes and open-source technologies like Dapr, KEDA, and envoy. Based on best practices and optimized for general purpose containers. Cluster infrastructure is managed by ACA and direct access to the Kubernetes API is not supported. Provides</p>

Service	Description
	<p>many application-specific concepts on top of containers, including certificates, revisions, scale, and environments. Ideal for teams that want to start building container microservices without having to manage the underlying complexity of Kubernetes.</p> <p>Example: Deploy a Flask or FastAPI web app on Azure Container Apps.</p>
Azure Container Instances (ACI) ↗	<p>A serverless offering that provides a single pod of Hyper-V isolated containers on demand. Billed on consumption rather than provisioned resources. Concepts like scale, load balancing, and certificates aren't provided with ACI containers. Users often interact with ACI through other services; for example, AKS for orchestration. Ideal if you need a less "opinionated" building block that doesn't align with the scenarios Azure Container Apps is optimizing for.</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>A fully managed Kubernetes option in Azure. Supports direct access to the Kubernetes API and runs any Kubernetes workload. The full cluster resides in your subscription, with the cluster configurations and operations within your control and responsibility. Ideal for teams looking for a fully managed version of Kubernetes in Azure.</p> <p>Example: Deploy an Azure Kubernetes Service cluster using the Azure CLI.</p>
Azure Functions ↗	<p>An event-driven, serverless functions-as-a-service (FaaS) solution. Shares many characteristics with Azure Container Apps around scale and integration with events, but is optimized for ephemeral functions deployed as either code or containers. Ideal for teams looking to trigger the execution of functions on events; for example, to bind to other data sources.</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

When you're running a Python project in a dev environment, using a virtual environment is a common way of managing dependencies and ensuring reproducibility of your project setup. A virtual environment has a Python interpreter, libraries, and scripts installed that are required by the project code running in that environment.

Dependencies for Python projects are managed through the *requirements.txt* file.

💡 Tip

With containers, virtual environments aren't needed unless you're using them for testing or other reasons. If you use virtual environments, don't copy them into the Docker image. Use the `.dockerignore` file to exclude them.

You can think of Docker containers as providing similar capabilities as virtual environments, but with further advantages in reproducibility and portability. Docker container can be run anywhere containers can be run, regardless of OS.

A Docker container contains your Python project code and everything that code needs to run. To get to that point, you need to build your Python project code into a Docker image, and then create container, a runnable instance of that image.

For containerizing Python projects, the key files are:

[+] Expand table

Project file	Description
<code>requirements.txt</code>	Used during the building of the Docker image to get the correct dependencies into the image.
<code>Dockerfile</code>	Used to specify how to build the Python Docker image. For more information, see the section Dockerfile instructions for Python .
<code>.dockerignore</code>	Files and directories in <code>.dockerignore</code> aren't copied to the Docker image with the <code>COPY</code> command in the <code>Dockerfile</code> . 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 have default ports on which they listen for web requests. When working with some Azure container solutions, you need to specify the port your container is listening on that will receive traffic.

[+] 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 difference 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 WEBSITES_PORT app setting in your App Service app. For more information, see Configure a custom container for Azure App Service .
Azure Container Apps	Azure Container Apps allows you to expose your container app to the public web, to your VNET, or to other container apps within your 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	Set port during creation of a container. You need to ensure your solution has a web framework, application server (for example, gunicorn, uvicorn), and web server (for example, nginx). For example, you can create two containers, one container with a web framework and application server, and another framework with a web server. The two containers communicate on one port, and the web server container exposes 80/443 for external requests.

Python Dockerfile

A Dockerfile is a text file that contains instructions for building a Docker image. The first line states the base image to begin with. This line is followed by instructions to install required programs, copy files, and other instructions to create a working environment. For example, some Python-specific examples for key Python Dockerfile instructions show in the table below.

[\[+\] 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 on the specified network ports 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'll 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

Working in an integrated development environment (IDE) for Python container development isn't necessary but can simplify many container-related tasks. 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

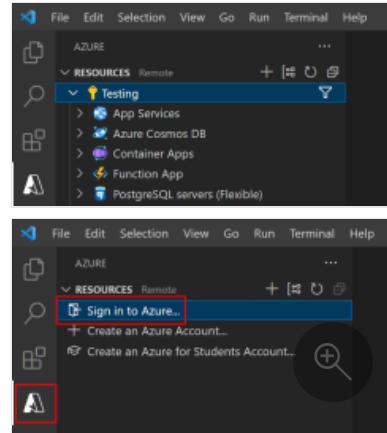
Screenshot

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.

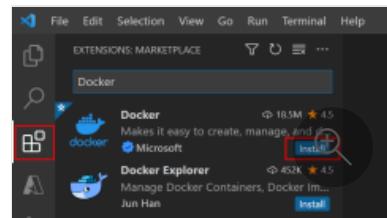
If you aren't signed in, select **Sign in to Azure** and follow the prompts.

If you have trouble accessing your Azure subscription, it may be because you are behind a proxy. To resolve connection issues, see [Network Connections in Visual Studio Code](#).

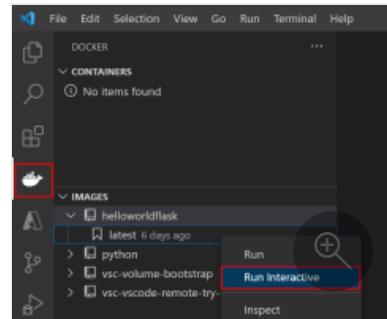


Step 2: Use **CTRL + SHIFT + X** to open **Extensions**, search for the [Docker extension](#), and install the extension.

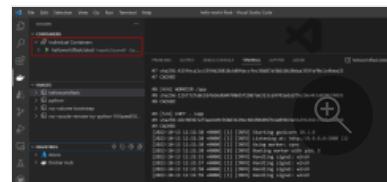
You can also select the **Extensions** icon on the VS Code extensions bar.



Step 3: Select the **Docker** icon in the extension bar, expand **IMAGES**, and right-click an 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.

Bash

```
az acr build --registry <registry-name> \
--resource-group <resource-group> \
--target pythoncontainerwebapp: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 often make use of environment variables to pass data to code. For example, you might specify database connection information in an environment variable so that it can be easily changed during testing. Or, when deploying the project to production, the database connection can be changed to refer to a production database instance.

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 above 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>
```

If you're using VS Code or PyCharm, the UI options for working with images and containers ultimately use Docker CLI commands like the one shown above.

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'll 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.

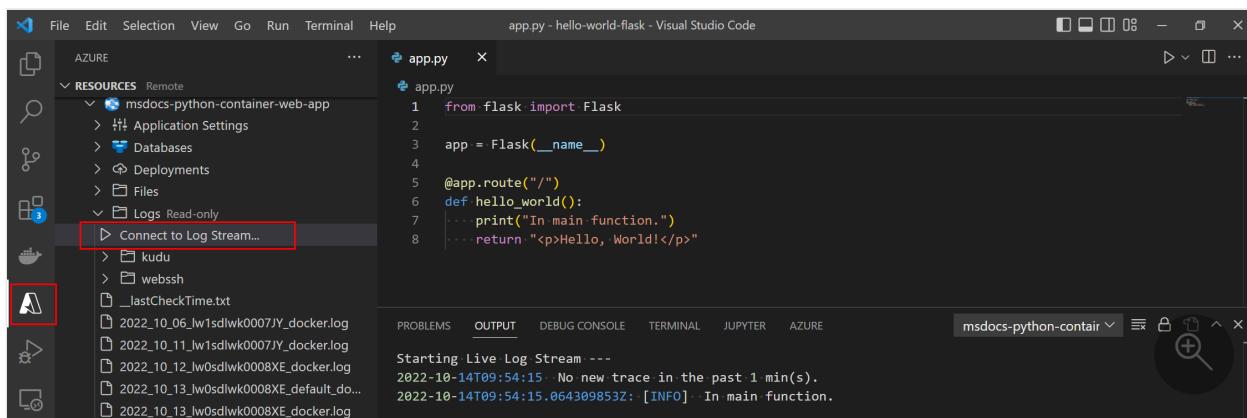
Azure service	How to access logs in Azure portal
Web App for Containers	Go to the Diagnose and solve problems resource to view logs. Diagnostics is an intelligent and interactive experience to help you troubleshoot your app with no configuration required. For a real-time view of logs, go to the Monitoring - 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'll 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 Container Instances	Go to the Containers resource and select Logs .

For the same services listed above, 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

Article • 04/12/2024

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. Web App for Containers provides an easy on-ramp for developers to take advantage of the fully managed Azure App Service platform, but who also want a single deployable artifact containing an app and all of its dependencies. 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 create a Docker image and test it locally. And, you use the [Azure CLI](#) to create a Docker image in Azure and deploy it to Azure App Service. 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 shows creating a Docker image that can then be run on App Service. This is not required to use App Service. You can deploy code directly from a 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](#).
- [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
```

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
```

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 .
```

① 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. Create a group with the `az group create` command.

```
Azure CLI
```

```
az group create --name web-app-simple-rg --location eastus
```

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*.

2. Create an Azure Container Registry with the [az acr create](#) command.

Azure CLI

```
az acr create --resource-group web-app-simple-rg \
--name webappacr123 --sku Basic --admin-enabled true
```

 **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. The second command saves the password to a variable with the [az credential show](#) command. The password is used to authenticate to the registry in a later step.

3. Set an environment variable to the value of the password for the registry.

Bash

```
ACR_PASSWORD=$(az acr credential show \
--resource-group web-app-simple-rg \
--name webappacr123 \
--query "passwords[?name == 'password'].value" \
--output tsv)
```

The command for creating the environment variable is shown for the Bash shell. Change the syntax and continuation character (`\`) as appropriate for other shells.

You can also get the password (`ACR_PASSWORD`) from the [Azure portal](#) by going to the registry, selecting **Access keys**, and copying the password.

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 web-app-simple-rg \
--registry webappacr123 \
--image webappsimple: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 web-app-simple-rg \
--sku B1 \
--is-linux
```

2. Create the web app with the `az webapp create` command.

```
Azure CLI

az webapp create \
--resource-group web-app-simple-rg \
--plan webplan --name webappsimple123 \
--docker-registry-server-password $ACR_PASSWORD \
--docker-registry-server-user webappacr123 \
--role acrpull \
--deployment-container-image-name
webappacr123.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 `--docker-registry-server-user` and `--deployment-container-image-name` parameters appropriately.
- 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 web-app-simple-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 a Python web app on Azure Container Apps
 - Quickstart: Deploy a Python (Django or Flask) web app to Azure App Service
-

Feedback

Was this page helpful?



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

Overview: Containerized Python web app on Azure with MongoDB

Article • 10/13/2023

This tutorial shows you how to containerize a Python web app and deploy it to Azure. The single container web app is hosted in [Azure App Service](#) and uses [MongoDB for Azure Cosmos DB](#) to store data. App Service [Web App for Containers](#) allows you to focus on composing your containers without worrying about managing and maintaining an underlying container orchestrator. When building web apps, Azure App Service is a good option for taking your first steps with containers. For more information about using containers in Azure, see [Comparing Azure container options](#).

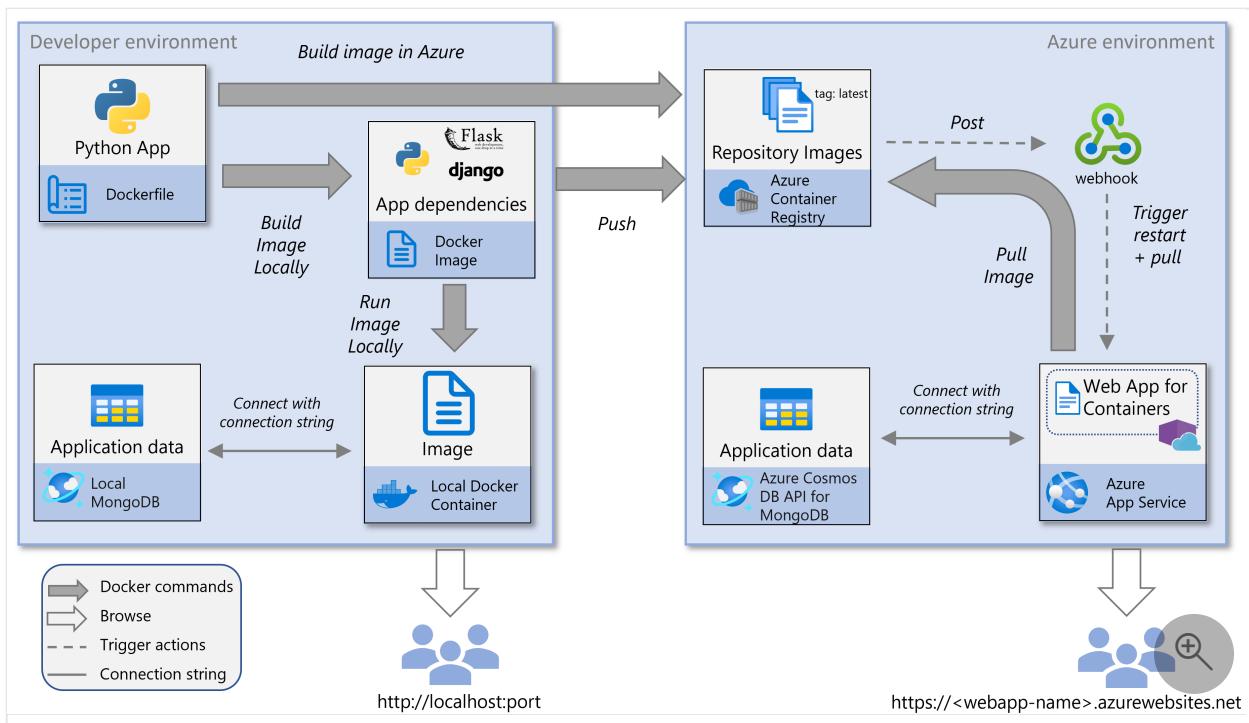
In this tutorial you will:

- Build and run a [Docker](#) container locally. *This step is optional.*
- Build a [Docker](#) container image directly in Azure.
- Configure an App Service to create a web app based on the Docker container image.

Following this tutorial, 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) and the different Azure services used in the tutorial.



The components supporting this tutorial and shown in the diagram above are:

- [Azure App Service](#)
 - The underlying App Service functionality that enables containerization is Web App for Containers. Azure App Service uses the [Docker](#) container technology to host both built-in images and custom images. In this tutorial, you'll build an image from Python code and deploy it to Web App for Containers.
 - Web App for Containers uses a webhook in the registry to get notified of new images. A push of a new image to the repository triggers App Service to pull the image and restart.
- [Azure Container Registry](#)
 - Azure Container Registry enables you to work with Docker images and its components in Azure. It provides a registry that's close to your deployments in Azure and that gives you control over access, making it possible to use your Microsoft Entra groups and permissions.
 - In this tutorial, the registry source is Azure Container Registry, 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 to store data.

- Access to Azure Cosmos DB resource is via a connection string, which is passed as an environment variable to the containerized app.

Authentication

In this tutorial, you'll build a Docker image (either locally or directly in Azure) and deploy it to Azure App Service. The App Service pulls the container image from an Azure Container Registry repository.

The App Service uses [managed identity](#) to pull images from Azure Container Registry. Managed identity allows you to grant permissions to the web app so that it can access other Azure resources without the need to specify credentials. Specifically, this tutorial uses a system assigned managed identity. 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'll 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, we recommend you follow the steps in [part 2 of this tutorial](#).
- [Visual Studio Code](#) or [Azure CLI](#), depending on what tool you'll use.
 - For Visual Studio Code, you'll need the [Docker extension](#) and [Azure App Service extension](#).
- Python packages:
 - [PyMongo](#) for connecting to MongoDB.
 - [Flask](#) or [Django](#) as a web framework.
- [Docker](#) installed locally if you want to run container locally.

Sample app

The Python sample app is a restaurant review app that saves restaurant and review data in MongoDB. For an example of a web app using PostgreSQL, see [Create and deploy a Flask web app to Azure with a managed identity](#).

At the end of the tutorial, you'll have a restaurant review app deployed and running in Azure that looks like the screenshot below.

Azure Restaurant Review

Azure Docs ▾

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 test locally](#)

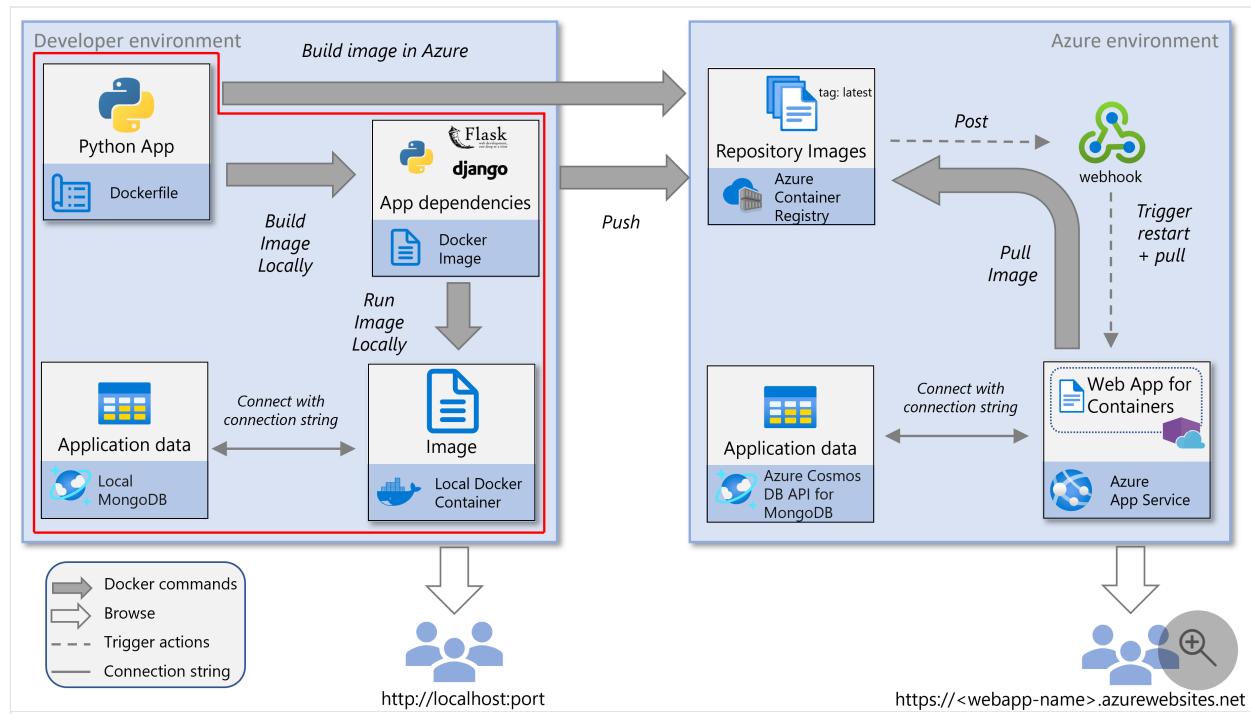
Build and run a containerized Python web app locally with MongoDB

Article • 12/29/2023

This article is part of a tutorial about how to containerize and deploy a containerized Python web app to Azure App Service. App Service enables you to run containerized web apps and deploy through continuous integration/continuous deployment (CI/CD) capabilities with Docker Hub, Azure Container Registry, and Visual Studio Team Services. In this part of the tutorial, you learn how to build and run the containerized Python web app locally. *This step is optional and isn't required to deploy the sample app to Azure.*

Running a Docker image locally in your development environment requires setup beyond deployment to Azure. Think of it as an investment that can make future development cycles easier, especially when you move beyond sample apps and you start to create your own web apps. To deploy the sample apps for [Django](#) and [Flask](#), you can skip this step and go to the next step in this tutorial. You can always return after deploying to Azure and work through these steps.

The following service diagram highlights the components covered in this article.



1. Clone or download the sample app

Git clone

Clone the repository:

```
terminal

# Django
git clone https://github.com/Azure-Samples/msdocs-python-django-
container-web-app.git

# Flask
git clone https://github.com/Azure-Samples/msdocs-python-flask-
container-web-app.git
```

Then navigate into that folder:

```
terminal

# Django
cd msdocs-python-django-container-web-app

# Flask
cd msdocs-python-flask-container-web-app
```

2. Build a Docker image

If you're using one of the framework sample apps available for [Django](#) and [Flask](#), you're set to go. If you're working with your own sample app, take a look to see how the sample apps are set up, in particular the *Dockerfile* in the root directory.

VS Code

These instructions require [Visual Studio Code](#) and the [Docker extension](#). Go to the sample folder you cloned or downloaded and open VS Code with the command `code .`

⚠ Note

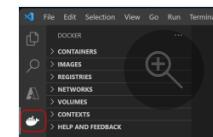
The steps in this section require the Docker daemon to be running. In some installations, for example on Windows, you need to open [Docker Desktop](#), which starts the daemon, before proceeding.

[+] [Expand table](#)

Instructions

Screenshot

Open the Docker extension.



If the Docker extension reports an error "Failed to connect", make sure [Docker](#) is installed and running. If this is your first time working with Docker, you probably won't have any containers, images, or connected registries.

Build the image.

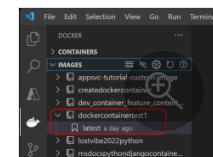
- In the project Explorer showing the project files, right-click the *Dockerfile* and select **Build Image....**
- Alternately, you can use the Command Palette (**F1** or **Ctrl+Shift+P**) and type "Docker Images: Build Images" to invoke the command.



For more information about Dockerfile syntax, see the [Dockerfile reference](#).

Confirm the image was built.

- Go to the **IMAGES** section of the Docker extension.
- Look for recently built image. The name of the container image is "msdocpythoncontainerwebapp", which is set in the *.vscode/tasks.json* file.



At this point, you have built an image locally. The image you created has the name "msdocpythoncontainerwebapp" and tag "latest". Tags are a way to define version information, intended use, stability, or other information. For more information, see [Recommendations for tagging and versioning container images](#).

Images that are built from VS Code or from using the Docker CLI directly can also be viewed with the [Docker Desktop](#) application.

3. Set up MongoDB

For this tutorial, you need a MongoDB database named *restaurants_reviews* and a collection named *restaurants_reviews*. The steps in this section show you how to use a local installation of MongoDB or [Azure Cosmos DB for MongoDB](#) to create and access the database and collection.

ⓘ Important

Don't use a MongoDB database you'll use in production. In this tutorial, you'll store the MongoDB connection string in an environment variable. This makes it

observable by anyone capable of inspecting your container (for example, using `docker inspect`).

Local MongoDB

Step 1: Install [MongoDB](#) if it isn't already.

You can check for the installation of MongoDB by using the [MongoDB Shell \(mongosh\)](#).

- The following command enters the shell and gives you the version of both mongosh and mongoDB server installed on your system:

terminal

```
mongosh
```

- The following command gives you just the version of MongoDB server installed on your system:

terminal

```
mongosh --quiet --exec 'db.version()'
```

If these commands don't work, you might need to explicitly [install mongosh](#) or [connect mongosh to your MongoDB server](#).

An alternative in some installations is to directly invoke the Mongo daemon.

terminal

```
mongod --version
```

Step 2: Edit the `mongod.cfg` file to add your computer's IP address.

The [mongod configuration file](#) has a `bindIp` key that defines hostnames and IP addresses that MongoDB listens for client connections. Add the current IP of your local development computer. The sample app running locally in a Docker container will communicate to the host machine 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>
```

Restart MongoDB to pick up changes to the configuration file.

Step 3: Create a database and collection in the local MongoDB database.

Set the database name to "restaurants_reviews" and the collection name to "restaurants_reviews". You can create a database and collection with the VS Code [MongoDB extension](#), the [MongoDB Shell \(mongosh\)](#), or any other MongoDB-aware tool.

For the MongoDB shell, here are example commands to create the database and collection:

```
mongosh  
  
> help  
> use restaurants_reviews  
> db.restaurants_reviews.insertOne({})  
> show dbs  
> exit
```

At this point, your local MongoDB connection string is "mongodb://127.0.0.1:27017/", the database name is "restaurants_reviews", and the collection name is "restaurants_reviews".

4. Run the image locally in a container

With information on how to connect to a MongoDB, you're ready to run the container locally. The sample app expects MongoDB connection information to be passed in 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 deployed to Azure, the web app will get connection info from environment values set as App Service configuration settings and none of the modifications for the local development environment scenario apply.

VS Code

[+] Expand table

Instructions

Screenshot



In the `.vscode` folder of the sample app, the `settings.json` file defines what happens when you use the Docker extension and select **Run** or **Run Interactive** from the context menu of a Tag. The `settings.json` file contains two templates each for the `(MongoDB local)` and `(MongoDB Azure)` scenarios.

If you're using a local MongoDB database:

- Replace both instances of `<YOUR_IP_ADDRESS>` with your IP address.
- Replace both instances of `<CONNECTION_STRING>` with the connection string for your MongoDB database.

If you're using an Azure Cosmos DB for MongoDB database:

- Replace both instances of `<CONNECTION_STRING>` with the Azure Cosmos DB for MongoDB connection string.

Set the `docker.dockerPath` configuration setting used by the templates. To set `docker.dockerPath`, open the VS Code **Command Palette** (**Ctrl+Shift+P**), enter "Preferences: Open Workspace Settings", then enter "docker.dockerPath" in the **Search settings** box. Enter "docker" (without the quotes) for the value of the setting.

! Note

Both the database name and collection name are assumed to be `restaurants_reviews`.

Run the image.

- In the **IMAGES** section of the Docker extension, find the built image.
- Expand the image to find the **latest** tag, right-click and select **Run Interactive**.
- You'll be prompted to select the task appropriate for your scenario, either "Interactive run configuration (MongoDB local)" or "Interactive run configuration (MongoDB Azure)".



Instructions

Screenshot

With interactive run, you'll see any print statements in the code, which can be useful for debugging. You can also select **Run** which is non-interactive and doesn't keep standard input open.

ⓘ Important

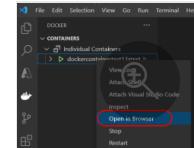
This step fails if the default terminal profile is set to (Windows) Command Prompt. To change the default profile, open the VS Code **Command Palette** (**Ctrl+Shift+P**), enter "Terminal: Select Default Profile", and then select a different profile from the dropdown menu; for example *Git Bash* or *PowerShell*.

Confirm that the container is running.

- In the **CONTAINERS** section of the Docker extension, find the container.
- Expand the **Individual Containers** node and confirm that "msdocspythoncontainerwebapp" is running. You should see a green triangle symbol next to the container name if it's running.



Test the web app by right-clicking the container name and selecting **Open in Browser**.



The browser will open into your default browser as "http://127.0.0.1:8000" for Django or "http://127.0.0.1:5000/" for Flask.

Stop the container.

- In the **CONTAINERS** section of the Docker extension, find the running container.
- Right click the container and select **Stop**.



💡 Tip

You can also run the container selecting a run or debug configuration. The Docker extension tasks in *tasks.json* are called when you run or debug. The task called depends on what launch configuration you select. For the task "Docker: Python (MongoDB local)", specify <YOUR-IP-ADDRESS>. For the task "Docker: Python (MongoDB Azure)", specify <CONNECTION-STRING>.

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 the cloud

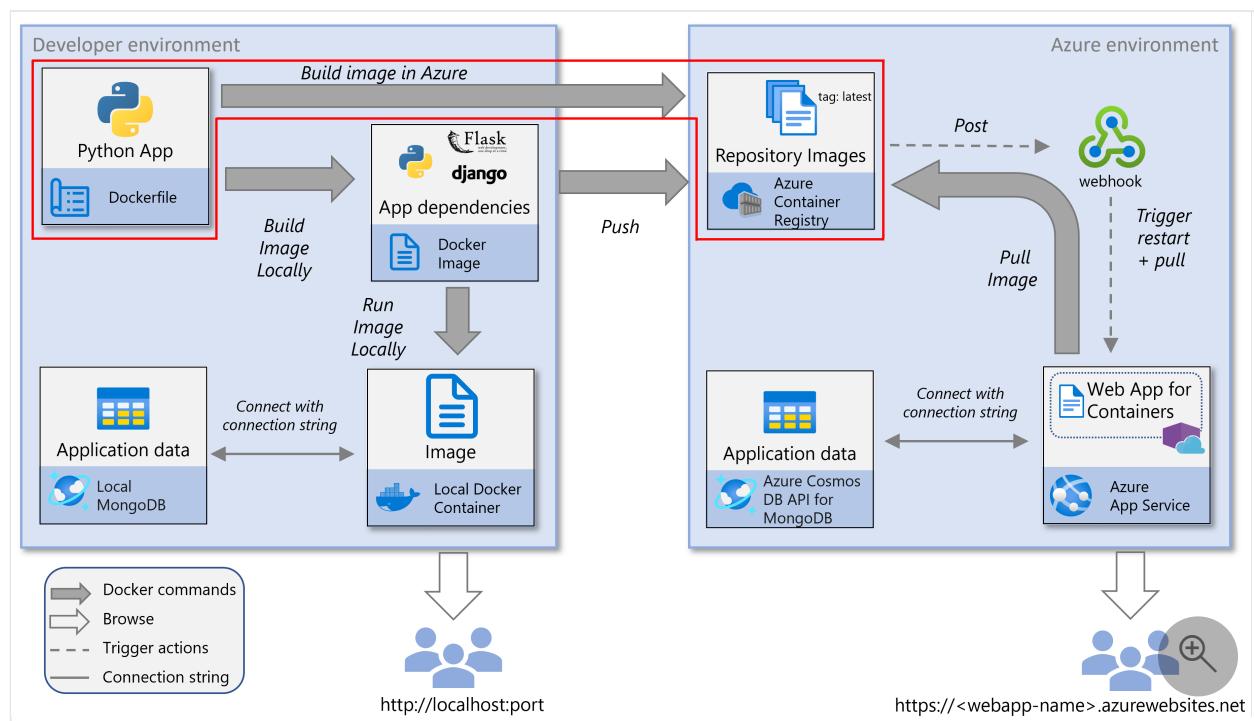
Article • 10/13/2023

This article is part of a tutorial about how to containerize and deploy a Python web app to Azure App Service. App Service enables you to run containerized web apps and deploy through continuous integration/continuous deployment (CI/CD) capabilities with Docker Hub, Azure Container Registry, and Visual Studio Team Services. In this part of the tutorial, you learn how to build the containerized Python web app in the cloud.

In the previous *optional* part of this tutorial, a container image was build and run locally. In contrast, in this part of the tutorial, you build (containerize) a Python web app into a Docker image directly in [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.

The service diagram shown below highlights the components covered in this article.



1. Create an Azure Container Registry

If you already have an Azure Container Registry you can use, go to the next step. If you don't, create one.

Azure CLI

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI installed](#). When running in Cloud Shell, skip **Step 3**.

Step 1. Create a resource group if needed with the `az group create` command. If you've already set up an Azure Cosmos DB for MongoDB account in part 2. **Build and test container locally** of this tutorial, set RESOURCE_GROUP_NAME to the name of the resource group you used for that account and move on to Step 2.

Azure CLI

```
RESOURCE_GROUP_NAME='msdocs-web-app-rg'  
LOCATION='eastus'  
  
az group create -n $RESOURCE_GROUP_NAME -l $LOCATION
```

LOCATION should be an Azure location value. Choose a location near you. You can list Azure location values with the following command: `az account list-locations -o table`.

Step 2. Create a container registry with the `az acr create` command.

Azure CLI

```
REGISTRY_NAME='<your Azure Container Registry name>'  
  
az acr create -g $RESOURCE_GROUP_NAME -n $REGISTRY_NAME --sku Basic
```

REGISTRY_NAME must be unique within Azure and contain 5-50 alphanumeric characters.

In the JSON output of the command look for the `loginServer` value, which is the fully qualified registry name (all lowercase) and which should include the registry name you specified.

Step 3. If you're running the Azure CLI locally, log in to the registry using the `az acr login` command.

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'll see the message "Login Succeeded".

ⓘ Note

The `az acr login` command isn't needed or supported in Cloud Shell.

2. Build an image in Azure Container Registry

You can build the container image directly in Azure in a few ways. First, you can use the Azure Cloud Shell, which builds the image without using your local environment at all. You can also build the container image in Azure from your local environment using VS Code or the Azure CLI. Building the image in the cloud doesn't require Docker to be running in your local environment. If you need to, you can follow the instructions in [Clone or download the sample app](#) in part 2 of this tutorial to get the sample Flask or Django web app.

Azure CLI

Azure CLI commands can be run on a workstation with the [Azure CLI installed](#) or in [Azure Cloud Shell](#). When running in Cloud Shell, skip **Step 1**.

Step 1. If you're running the Azure CLI locally, log into registry if you haven't done so already with the `az acr login` command.

Azure CLI

```
az acr login -n $REGISTRY_NAME
```

If you're accessing the registry from a subscription different from the one in which the registry was created, use the `--suffix` switch.

ⓘ Note

The `az acr login` command isn't needed or supported in Cloud Shell.

Step 2. Build the image with the `az acr build` command.

Azure CLI

```
az acr build -r $REGISTRY_NAME -g $RESOURCE_GROUP_NAME -t  
msdocspythoncontainerwebapp:latest .
```

In this command:

- 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 root directory, specify the path to the code.

Rather than a path to the code in your environment, you can, optionally, specify a path to the sample GitHub repo: <https://github.com/Azure-Samples/msdocs-python-django-container-web-app> or <https://github.com/Azure-Samples/msdocs-python-flask-container-web-app>.

- 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.

Step 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

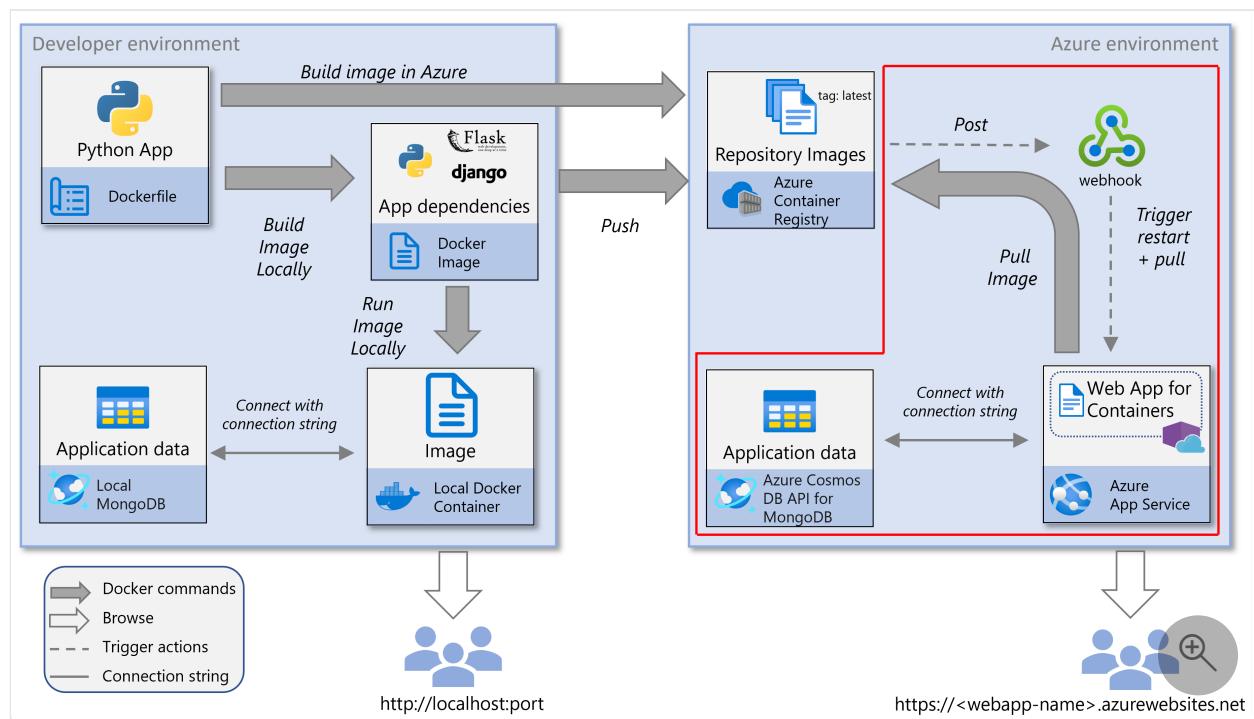
Article • 10/13/2023

This article is part of a tutorial about how to containerize and deploy a Python web app to Azure App Service. App Service enables you to run containerized web apps and deploy through continuous integration/continuous deployment (CI/CD) capabilities with Docker Hub, Azure Container Registry, and Visual Studio Team Services.

In this part of the tutorial, you learn how to deploy the containerized Python web app to App Service using the [App Service Web App for Containers](#). Web App for Containers allows you to focus on composing your containers without worrying about managing and maintaining an underlying container orchestrator.

Following the steps here, you'll end up with an App Service website using a Docker container image. The App Service pulls the initial image from Azure Container Registry using managed identity for authentication.

The service diagram shown below highlights the components covered in this article.



1. Create the web app

Azure CLI

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI installed](#).

Step 1. Get the resource ID of the group containing Azure Container Registry with the [az group show](#) command.

```
Azure CLI

# RESOURCE_GROUP_NAME='msdocs-web-app-rg'

RESOURCE_ID=$(az group show \
    --resource-group $RESOURCE_GROUP_NAME \
    --query id \
    --output tsv)
echo $RESOURCE_ID
```

In the command above, RESOURCE_GROUP_NAME should still be set in your environment to the resource group name you used in [part 3. Build container in Azure](#) of this tutorial. If it isn't, uncomment the first line and make sure it's set to the name you used.

Step 2. Create an App Service plan with the [az appservice plan create](#) command.

```
Azure CLI

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
```

Step 3. Create a web app with the [az webapp create](#) command.

The following command also enables the [system-assigned managed identity](#) for the web app and assigns it the [AcrPull role](#) on the specified resource--in this case, the resource group that contains the Azure Container Registry. This grants the system-assigned managed identity pull privileges on any Azure Container Registry in the resource group.

```
Azure CLI

APP_SERVICE_NAME='<website-name>'
# REGISTRY_NAME='<your Azure Container Registry name>'
CONTAINER_NAME=$REGISTRY_NAME.azurecr.io/msdocspythoncontainerwebapp:1a
```

```
test'

az webapp create \
--resource-group $RESOURCE_GROUP_NAME \
--plan $APP_SERVICE_PLAN_NAME \
--name $APP_SERVICE_NAME \
--assign-identity '[system]' \
--scope $RESOURCE_ID \
--role acrpull \
--deployment-container-image-name $CONTAINER_NAME
```

In the command above:

- APP_SERVICE_NAME must be globally unique as it becomes the website name in the URL `https://<website-name>.azurewebsites.net`.
- CONTAINER_NAME is of the form "`yourregistryname.azurecr.io/repo_name:tag`".
- REGISTRY_NAME should still be set in your environment to the registry name you used in part 3. **Build container in Azure** of this tutorial. If it isn't, uncomment the line where it's set above and make sure it's set to the name you used.

① Note

You may see an error similar to the following when running the command:

Output

```
No credential was provided to access Azure Container Registry.
Trying to look up...
Retrieving credentials failed with an exception:'No resource or
more than one were found with name ...'
```

This error occurs because the web app defaults to using the Azure Container Registry's admin credentials to authenticate with the registry and admin credentials haven't been enabled on the registry. You can safely ignore this error because you will set the web app to use the system-assigned managed identity for authentication in the next command.

2. Configure managed identity and webhook

Step 1. Configure the web app to use managed identities to pull from the Azure Container Registry with the [az webapp config set](#) command.

Azure CLI

```
az webapp config set \
--resource-group $RESOURCE_GROUP_NAME \
--name $APP_SERVICE_NAME \
--generic-configurations '{"acrUseManagedIdentityCreds": true}'
```

Because you enabled the system-assigned managed identity when you created the web app, it will be the managed identity used to pull from the Azure Container Registry.

Step 2. Get the application scope credential with the [az webapp deployment list-publishing-credentials](#) command.

Azure CLI

```
CREDENTIAL=$(az webapp deployment list-publishing-credentials \
--resource-group $RESOURCE_GROUP_NAME \
--name $APP_SERVICE_NAME \
--query publishingPassword \
--output tsv)
echo $CREDENTIAL
```

Step 3. Use the application scope credential to create a webhook with the [az acr webhook create](#) command.

Azure CLI

```
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 msdocspythoncontainerwebapp:* \
--uri $SERVICE_URI \
--actions push
```

By default, this command creates the webhook in the same resource group and location as the specified Azure Container registry. If desired, you can use the `--resource-group` and `--location` parameters to override this behavior.

3. Configure connection to MongoDB

In this step, you specify environment variables needed to connect to MongoDB.

If you need to create an Azure Cosmos DB for MongoDB, we recommend you follow the steps to [set up Cosmos DB for MongoDB](#) in part 2. **Build and test container locally** of this tutorial. When you're finished, you should 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>`.

You'll need the MongoDB connection string info to follow the steps below.

Azure CLI

To set environment variables in App Service, you create *app settings* with the following [az webapp config appsettings set](#) command.

Azure CLI

```
MONGO_CONNECTION_STRING='your Mongo DB connection string in single quotes'  
MONGO_DB_NAME=restaurants_reviews  
MONGO_COLLECTION_NAME=restaurants_reviews  
  
az webapp config appsettings set \  
    --resource-group $RESOURCE_GROUP_NAME \  
    --name $APP_SERVICE_NAME \  
    --settings CONNECTION_STRING=$MONGO_CONNECTION_STRING \  
        DB_NAME=$MONGO_DB_NAME \  
        COLLECTION_NAME=$MONGO_COLLECTION_NAME
```

- CONNECTION_STRING: A connection string that starts with "mongodb://".
- DB_NAME: Use "restaurants_reviews".
- COLLECTION_NAME: Use "restaurants_reviews".

4. Browse the site

To verify the site is running, go to `https://<website-name>.azurewebsites.net`; where website name is your app service name. If successful, you should see the restaurant review sample app. It can take a few moments for the site to start the first time. When the site appears, add a restaurant, and a review for that restaurant to confirm the sample app is functioning.

Azure CLI

If you're running the Azure CLI locally, you can use the [az webapp browse](#) command to browse to the web site. If you're using Cloud Shell, open a browser window and navigate to the website URL.

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.

5. 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 will 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 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".
- [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 • 10/09/2023

This article is part of a tutorial about how to containerize and deploy a Python web app to Azure App Service. In this article, you'll clean up resources used in Azure so you don't incur other charges and help keep your Azure subscription uncluttered. You can leave the Azure resources running if you want to use them for further development work.

1. Clean up resources

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.

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.

bash

```
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.

2. 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 \(azd\)](#).

3. 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 • 12/04/2023

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.



The screenshot shows a terminal window with a light gray background. On the left, there is a vertical toolbar with two items: "Flask" (which is highlighted with a blue border) and "Bash". The main area of the terminal is a light gray box containing the following text:

```
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 ENTRYPPOINT 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==2.0.2
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 `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](#).

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 .
```

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.)

Flask

Azure CLI

```
az containerapp up \
--resource-group web-flask-aca-rg --name web-aca-app \
--ingress external --target-port 50505 --source .
```

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-acra-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](#)

Overview: Deploy a Python web app on Azure Container Apps

Article • 01/31/2024

This tutorial shows you how to containerize a Python web app and deploy it to [Azure Container Apps](#). A sample web app will be containerized and the Docker image 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 an [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 covers Azure Container Apps. Container Apps are good for running general purpose containers, especially for applications that span many microservices deployed in containers. In this tutorial, you'll create one container. To deploy a Python web app as a container to Azure App Service, see [Containerized Python web app on App Service](#).

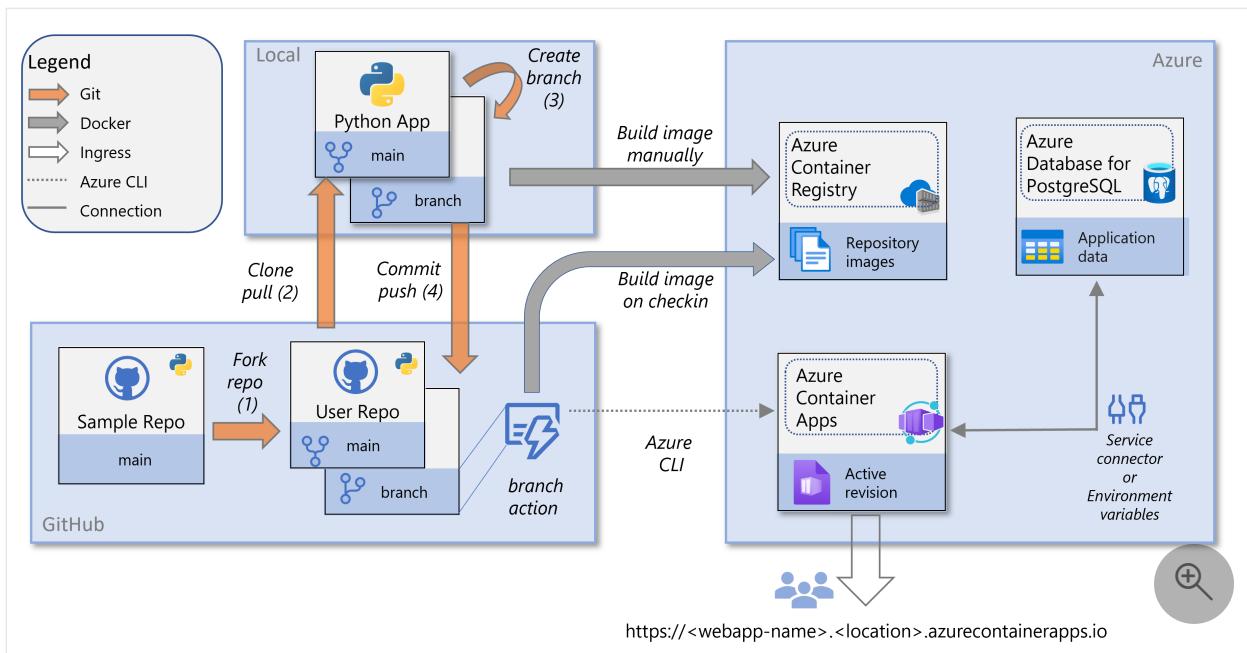
In this tutorial you'll:

- 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 a [GitHub Action](#) that updates the container with a new Docker image triggered by changes to your GitHub repository. *This last step is optional.*

Following this tutorial, you'll be set up for Continuous Integration (CI) and Continuous Deployment (CD) of a Python web app to Azure.

Service overview

The service diagram supporting this tutorial shows how your local environment, GitHub repositories, and Azure services are used in the tutorial.



The components supporting this tutorial and shown in the diagram above are:

- [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, making it possible to use your Microsoft Entra groups and permissions to control access to Docker images.
 - In this tutorial, 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 through environment variables configured explicitly or with [Azure Service Connector](#).
- [GitHub](#)
 - The sample code for this tutorial is in a GitHub repo that you'll fork and clone locally. To set up a CI/CD workflow with [GitHub Actions](#), you'll need a GitHub

account.

- You can still follow along with this tutorial without a GitHub account, working locally or in the [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 change. 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 shows you how to build a continuous integration and continuous delivery (CI/CD) pipeline with GitHub Actions. The pipeline automatically builds and deploys your code to the Container App whenever a new commit is pushed to your GitHub repository.

Authentication and security

In this tutorial, you'll build a Docker container image directly in Azure and deploy it to Azure Container Apps. Container Apps run in the context of an [environment](#), which is supported by an [Azure Virtual Networks \(VNet\)](#). VNets 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 continuous integration and continuous delivery (CI/CD), you'll 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 using credentials of a Microsoft Entra service principal (or existing one) with role-based access for Container Apps and username and password for Azure Container Registry. Credentials are stored securely in your GitHub repo.

Finally, the tutorial sample web app stores data in a PostgreSQL database. The sample code connects to PostgreSQL via a connection string. During the configuration of the Container App, the tutorial walks you through setting up environment variables containing connection information. You can also use an Azure Service Connector to accomplish the same thing.

Prerequisites

To complete this tutorial, you'll need:

- An Azure account where you can create:
 - Azure Container Registry
 - Azure Container Apps environment
 - Azure Database for PostgreSQL
- [Visual Studio Code](#) or [Azure CLI](#), depending on what tool you'll use
 - For Visual Studio Code, you'll need the [Container Apps extension](#).
 - You can also use Azure CLI through the [Azure Cloud Shell](#).
- Python packages:
 - [psycopg2-binary](#) for connecting to PostgreSQL.
 - [Flask](#) or [Django](#) 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, you'll have a restaurant review app deployed and running in Azure Container Apps that looks like the screenshot below.

The screenshot shows a web application interface for a restaurant review. At the top left is the logo 'Azure Restaurant Review'. At the top right is a link 'Azure Docs ▾'. The main title is 'Contoso Café'. Below the title, there are three sections: 'Street address:' with '1 Main Street', 'Description:' with 'Friendly coffee shop.', and 'Rating:' with a 5-star icon and '4.5 (2 reviews)'. Below these is a section titled 'Reviews' with a green button 'Add new review'. A table lists two reviews: one from 'Davide Sagese' on 26/07/2022 at 14:46:54 with rating 5 and review 'Great cappuccino.'; and another from 'Francesca Lombo' on 26/07/2022 at 14:47:58 with rating 4 and review 'Healthy breakfast choices.'. To the right of the reviews table is a circular icon with a magnifying glass and a plus sign.

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 to Azure Container Apps](#)

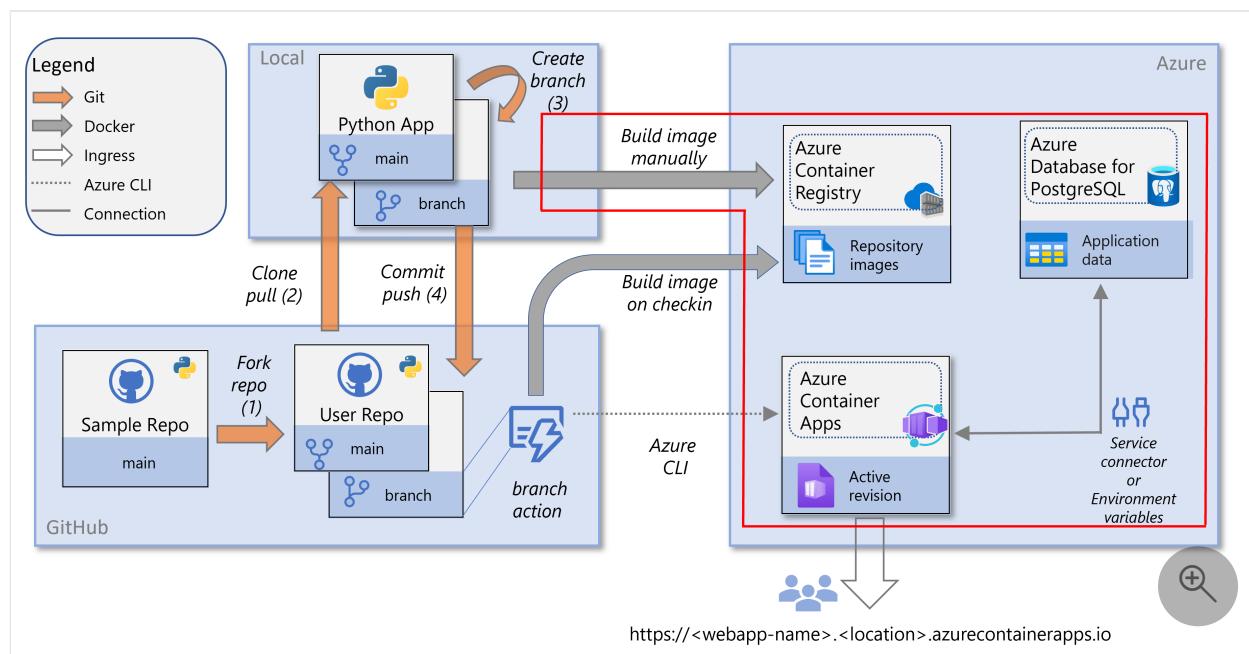
Build and deploy a Python web app with Azure Container Apps and PostgreSQL

Article • 01/31/2024

This article is part of a tutorial 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 part of the tutorial, you learn how to containerize and deploy a Python sample web app (Django or Flask). Specifically, you build the container image in the cloud and deploy it to Azure Container Apps. You 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.

This service diagram highlights the components covered in this article: building and deploying a container image.



Get the sample app

Fork and clone the sample code to your developer environment.

Step 1. Go to the GitHub repository of the sample app ([Django](#) or [Flask](#)) and select Fork.

Follow the steps to fork the directory to your GitHub account. You can also download the code repo directly to your local machine without forking or a GitHub account,

however, you won't be able to set up CI/CD discussed later in the tutorial.

Step 2. Use the [git clone](#) command to clone the forked repo into the *python-container* folder:

Console

```
# Django
git clone https://github.com/$USERNAME/msdocs-python-django-azure-container-
apps.git python-container

# Flask
# git clone https://github.com/$USERNAME/msdocs-python-flask-azure-
container-apps.git python-container
```

Step 3. Change directory.

Console

```
cd python-container
```

Build a container image from web app code

After following these steps, you'll have an Azure Container Registry that contains a Docker container image built from the sample code.

Azure CLI

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI](#) installed.

Step 1. Create a resource group with the [az group create](#) command.

Azure CLI

```
az group create \
--name pythoncontainer-rg \
--location <location>
```

<location> is one of the Azure location *Name* values from the output of the command `az account list-locations -o table`.

Step 2. Create a container registry with the [az acr create](#) command.

Azure CLI

```
az acr create \
--resource-group pythoncontainer-rg \
--name <registry-name> \
--sku Basic \
--admin-enabled
```

<registry-name> must be unique within Azure, and contain 5-50 alphanumeric characters.

You can view the credentials created for admin with:

Azure CLI

```
az acr credential show \
--name <registry-name> \
--resource-group pythoncontainer-rg
```

Step 3. Sign in to the registry 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 successful, you'll see the message "Login Succeeded". If you're accessing the registry from a subscription different from the one in which the registry was created, use the `--suffix` switch.

Step 4. Build the image with the [az acr build](#) command.

Azure CLI

```
az acr build \
--registry <registry-name> \
--resource-group pythoncontainer-rg \
--image pythoncontainer:latest .
```

Note that:

- 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 root directory, specify the path to the code.
- If you are running the command in Azure Cloud Shell, use `git clone` to first pull the repo into the Cloud Shell environment first and change directory into the root of the project so that 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.

Step 5. Confirm the container image was created with the [az acr repository list](#) command.

Azure CLI

```
az acr repository list --name <registry-name>
```

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

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI](#) installed.

Step 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 to complete.

Azure CLI

```
az postgres flexible-server create \
--resource-group pythoncontainer-rg \
--name <postgres-server-name> \
--location <location> \
--admin-user <admin-username> \
--admin-password <admin-password> \
--sku-name Standard_D2s_v3 \
--public-access 0.0.0.0
```

- "pythoncontainer-rg" → The resource group name used in this tutorial. If you used a different name, change this value.
- <postgres-server-name> → The PostgreSQL database server name. This name must be **unique across all Azure**. The server endpoint is "https://<postgres-

server-name>.postgres.database.azure.com". Allowed characters are "A"- "Z", "0"- "9", and "-".

- <location> → Use the same location 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> → Username for the administrator account. It can't be "azure_superuser", "admin", "administrator", "root", "guest", or "public". Use "demoadmin" for this tutorial.
- <admin-password> 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 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.

- <sku-name> → The name of the pricing tier and compute configuration, for example "Standard_D2s_v3". For more information, see [Azure Database for PostgreSQL pricing](#). To list available SKUs, use `az postgres flexible-server list-skus --location <location>`.
- <public-access> → Use "0.0.0.0", which allows public access to the server from any Azure service, such as Container Apps.

Note

If you plan on working the PostgreSQL server from your local workstation with tools other than Azure CLI, you'll need to add a firewall rule with the `az postgres flexible-server firewall-rule create` command.

Create a database on the server

At this point, you have a PostgreSQL server. In this section, you create a database on the server.

```
psql
```

You can use the PostgreSQL interactive terminal [psql](#) in your local environment, or in the [Azure Cloud Shell](#), which is also accessible in the [Azure portal](#). When working with psql, it's often easier to use the [Cloud Shell](#) because all the dependencies are included for you in the shell.

Step 1. Connect to the database with psql.

```
Bash
```

```
psql --host=<postgres-server-name>.postgres.database.azure.com \
      --port=5432 \
      --username=demoadmin@<postgres-server-name> \
      --dbname=postgres
```

Where *<postgres-server-name>* is the name of the PostgreSQL server. The command will prompt you for the admin password.

If you have trouble connecting, restart the database and try again. If you're connecting from your local environment, your IP address must be added to the firewall rule list for the database service.

Step 2. Create the database.

At the `postgres=>` prompt type:

```
SQL
```

```
CREATE DATABASE restaurants_reviews;
```

The semicolon (";") at the end of the command is necessary. To verify that the database was successfully created, use the command `\c restaurants_reviews`. Type `\?` to show help or `\q` to quit.

You can also connect to Azure PostgreSQL Flexible server and create a database using [Azure Data Studio](#) or any other IDE that supports PostgreSQL.

Deploy the web app to Container Apps

Container apps are deployed to Container Apps [environments](#), which act as a secure boundary. In the following steps, you create the environment, a container inside the

environment, and configure the container so that the website is visible externally.

Azure CLI

Step 1. Sign in to Azure and authenticate, if needed.

```
Azure CLI
```

```
az login
```

Step 2. Install or upgrade the extension for Azure Container Apps with the [az extension add](#) command.

```
Azure CLI
```

```
az extension add --name containerapp --upgrade
```

Step 3. Create a Container Apps environment with the [az containerapp env create](#) command.

```
Azure CLI
```

```
az containerapp env create \
--name python-container-env \
--resource-group pythoncontainer-rg \
--location <location>
```

<location> is one of the Azure location *Name* values from the output of the command `az account list-locations -o table`.

Step 4. Get the sign-in credentials for the Azure Container Registry.

```
Azure CLI
```

```
az acr credential show -n <registry-name>
```

Use the username and one of the passwords returned from the output of the command.

Step 5. Create a container app in the environment with the [az containerapp create](#) command.

```
Azure CLI
```

```
az containerapp create \
--name python-container-app \
--resource-group pythoncontainer-rg \
--image <registry-name>.azurecr.io/pythoncontainer:latest \
--environment python-container-env \
--ingress external \
--target-port 8000 \
--registry-server <registry-name>.azurecr.io \
--registry-username <registry-username> \
--registry-password <registry-password> \
--env-vars <env-variable-string>
--query properties.configuration.ingress.fqdn
```

<env-variable-string> is a string composed of space-separated values in the key="value" format with the following values.

- AZURE_POSTGRESQL_HOST=<postgres-server-name>.postgres.database.azure.com
- AZURE_POSTGRESQL_DATABASE=restaurants_reviews
- AZURE_POSTGRESQL_USERNAME=demoadmin
- AZURE_POSTGRESQL_PASSWORD=<db-password>
- RUNNING_IN_PRODUCTION=1
- AZURE_SECRET_KEY=<YOUR-SECRET-KEY>

Generate AZURE_SECRET_KEY value using output of `python -c 'import secrets; print(secrets.token_hex())'`.

Here's an example:

```
--env-vars AZURE_POSTGRESQL_HOST="my-postgres-
server.postgres.database.azure.com"
AZURE_POSTGRESQL_DATABASE="restaurants_reviews"
AZURE_POSTGRESQL_USERNAME="demoadmin" AZURE_POSTGRESQL_PASSWORD="somepassword"
RUNNING_IN_PRODUCTION="1" AZURE_SECRET_KEY=asdfasdfasdf.
```

Step 7. For Django only, migrate and create database schema. (In the Flask sample app, it's done automatically, and you can skip this step.)

Connect with the az containerapp exec command:

Azure CLI

```
az containerapp exec \
--name python-container-app \
--resource-group pythoncontainer-rg
```

Then, at the shell command prompt type `python manage.py migrate`.

You don't need to migrate for revisions of the container.

Step 8. Test the website.

The `az containerapp create` command you entered previously outputs an application URL you can use to browse to the app. The URL ends in "azurecontainerapps.io". Navigate to the URL in a browser. Alternatively, you can use the [az containerapp browse](#) command.

Here's an example of the sample website after adding a restaurant and two reviews.

A screenshot of a web application titled "Azure Restaurant Review". The main heading is "Restaurants". Below it is a table with three columns: "Name", "Rating", and "Details". The first row shows "Contoso Café" with a yellow star rating icon and "4.0 (2 reviews)". To the right of the rating is a blue "Details" button. At the bottom of the table is a green button labeled "Add new restaurant" with a magnifying glass and plus sign icon. The entire interface has a light gray background with some shadows and rounded corners.

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 the **Application Url**.
 - In VS Code, go to the Azure extension and select the **Container Apps** section. Expand the subscription, expand the container environment, and when you find the container app, right-click **python-container-app** and select **Browse**.
 - With 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, then 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 to see if your registry from a previous run exists and if so, use it.
- In the Azure portal during the creation of a Container App, you see an access error that contains "Cannot access ACR '<name>.azurecr.io'".

- This error occurs when admin credentials on the ACR are disabled. To check admin status in the portal, go to your Azure Container Registry, select the **Access keys** resource, and ensure that **Admin user** is enabled.
- Your container image doesn't appear in the Azure Container Registry.
 - 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 Azure CLI, run `az login`.
- Website returns "Bad Request (400)".
 - 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** on the **Overview** page for the container. If the Application Url contains the word "internal", then ingress isn't set correctly.
 - Check the ingress of the container. For example, in Azure portal, go to the **Ingress** resource of the container and make sure **HTTP Ingress** is enabled and **Accepting traffic from anywhere** is selected.
- Website doesn't start, you see "stream timeout", or nothing is returned.
 - Check the logs.
 - In the Azure portal, go to the Container App's Revision management resource and check the **Provision Status** of the container.
 - If "Provisioning", then wait until provisioning has completed.
 - If "Failed", then 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 first and look for Python `stderr` and `stdout` messages in the "Stream_s" column. Python 'print' output will be `stdout` messages.
 - With the Azure CLI, use the [az containerapp logs show](#) command.
 - If 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](#)

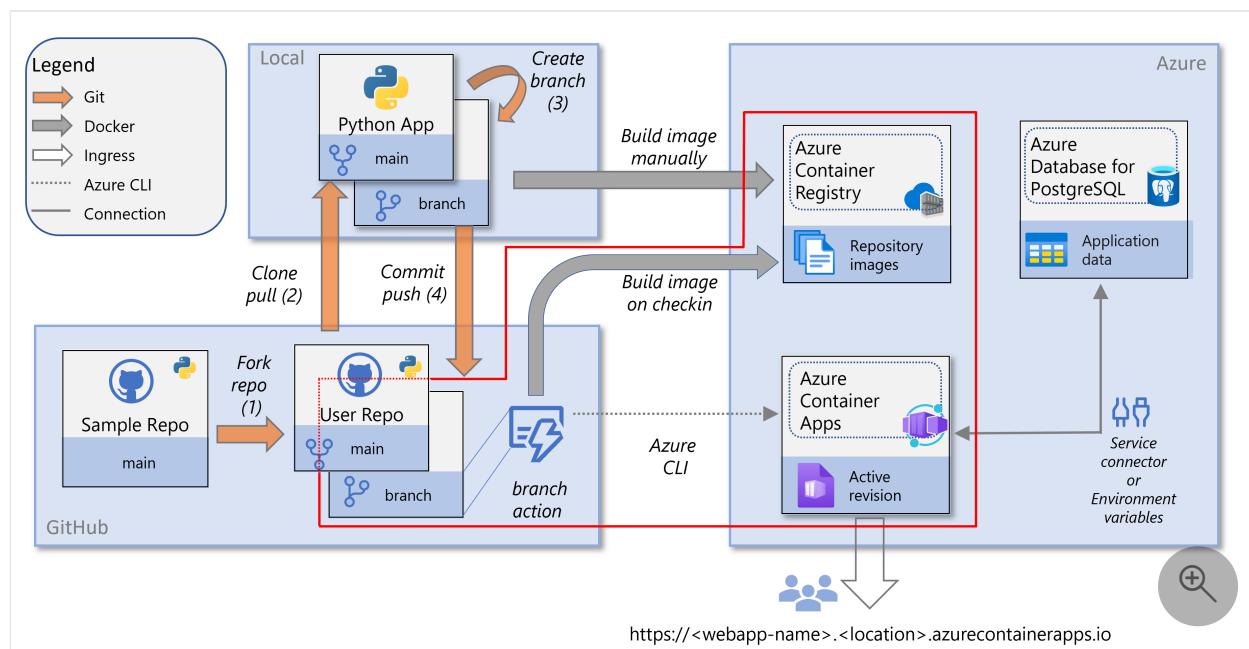
Configure continuous deployment for a Python web app in Azure Container Apps

Article • 01/31/2024

This article is part of a tutorial 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 part of the tutorial, you learn how to configure continuous deployment or delivery (CD) for the container app. CD is part of the DevOps practice of continuous integration / continuous delivery (CI/CD), which is automation of your app development workflow. Specifically, you use [GitHub Actions](#) for continuous deployment.

This service diagram highlights the components covered in this article: configuration of CI/CD.



Prerequisites

To set up continuous deployment, you need:

- The resources and their configuration created in the [previous article](#) of this tutorial series, which includes an [Azure Container Registry](#) and a container app in [Azure Container Apps](#).

- A GitHub account where you forked the sample code ([Django](#) or [Flask](#)) and you can connect to from Azure Container Apps. (If you downloaded the sample code instead of forking, make sure you 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 CD for a container

In a previous article of this 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. The container image is pulled from the registry when creating a container *revision*, such as when you first set up the container app.

In this section, you set up continuous deployment 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 (PR). When triggered, the workflow creates a new Docker image, pushes it to the Azure Container Registry, and updates the container app to a new revision using the new image.

Azure CLI

Azure CLI commands can be run in the [Azure Cloud Shell](#) or on a workstation with the [Azure CLI](#) 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
```

Step 1. Create a *service principal* with 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>"
```

Where:

- <*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 article in this tutorial, the resource group name is `pythoncontainer-rg`.

Save the output of this command for the next step, in particular, the client ID (`appId` property), client secret (`password` property), and tenant ID (`tenant` property).

Step 2. Configure GitHub Actions with [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
```

Where:

- <*resource-group-name*> is the name of the resource group. If you are following this tutorial, it is "pythoncontainer-rg".
- <<https://github.com/userid/repo>> is the URL of your GitHub repository. If you're following the steps in this tutorial, it will be either <https://github.com/userid/msdocs-python-django-azure-container-apps> or <https://github.com/userid/msdocs-python-flask-azure-container-apps>; where `userid` is your GitHub user ID.
- <*registry-name*> is the existing Container Registry you created for this 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 00000000-0000-0000-0000-00000000.
- <*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 similar to the client ID.
- <*client-secret*> is the value of the `password` property from the previous `az ad sp create-for-rbac` command.

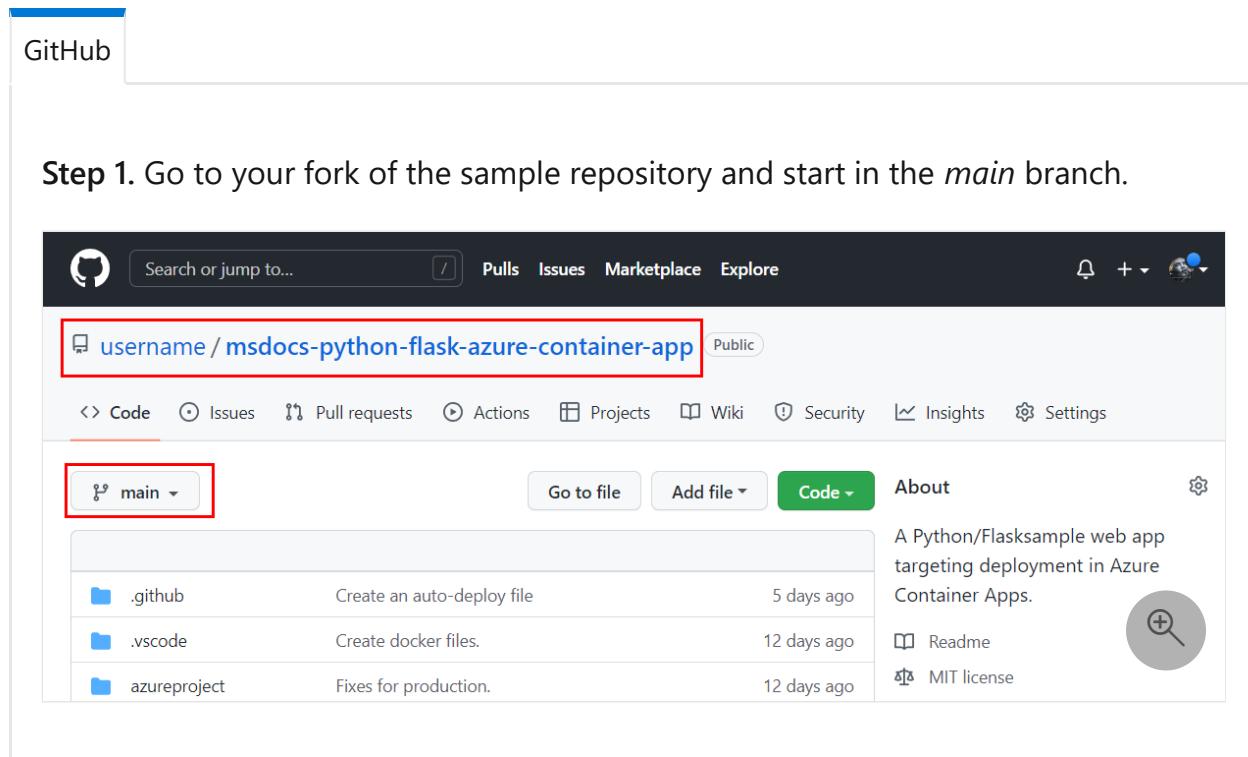
In the configuration of continuous deployment, a *service principal* is used to enable GitHub Actions to access and modify Azure resources. Access to resources is restricted by the roles assigned to the service principal. The service principal was assigned the built-in *Contributor* role on the resource group containing 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 first before configuring continuous deployment.

Redeploy web app with GitHub Actions

In this section, you make a change to your forked copy of the sample repository and confirm that the change is automatically deployed to the web site.

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](#).



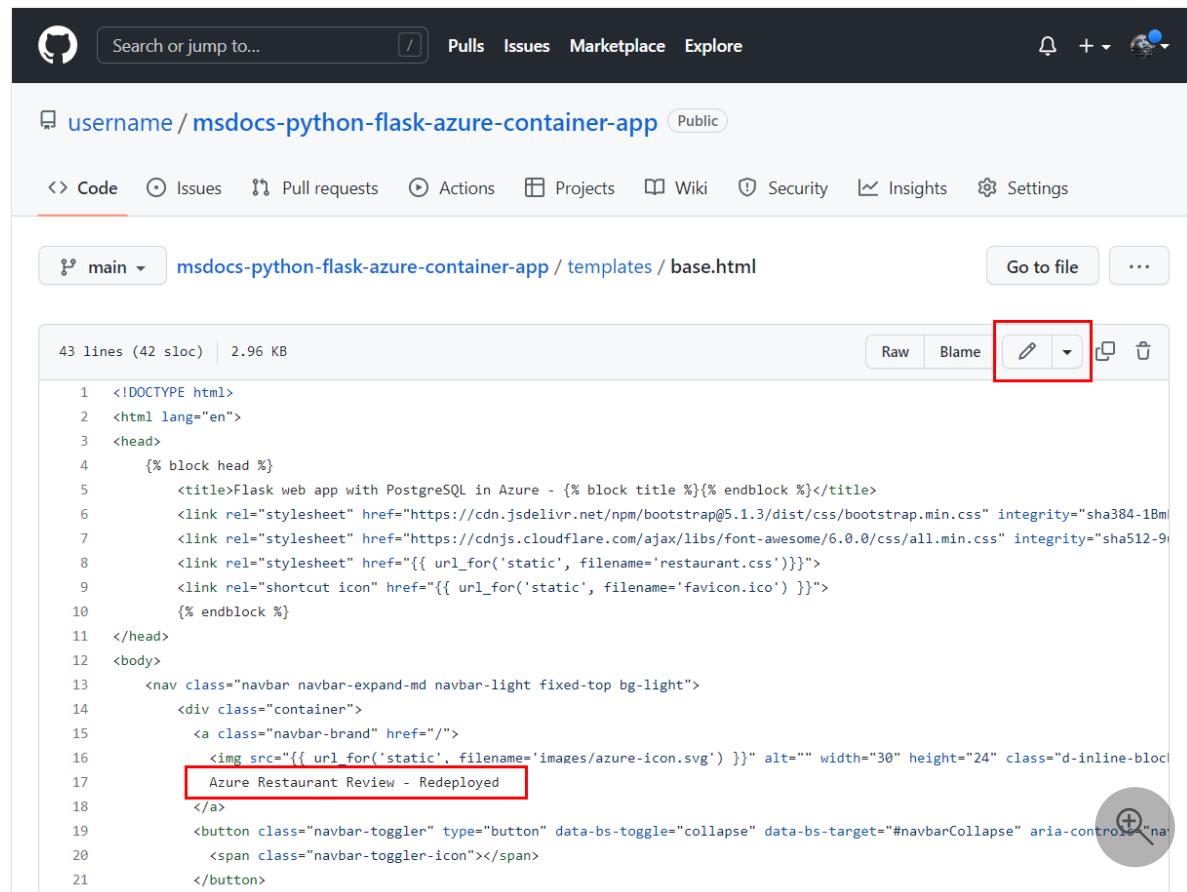
The screenshot shows a GitHub repository interface. At the top, there's a navigation bar with links for Pulls, Issues, Marketplace, and Explore. On the far right, there are icons for notifications, a plus sign, and a user profile. Below the navigation bar, the repository URL is displayed: `username / msdocs-python-flask-azure-container-app`. A red box highlights this URL. To the right of the URL, there's a 'Public' badge. Below the URL, there's a dropdown menu showing 'main' with a red box around it, indicating the active branch. The main content area shows a list of commits:

Commit	Description	Date
<code>.github</code>	Create an auto-deploy file	5 days ago
<code>.vscode</code>	Create docker files.	12 days ago
<code>azureproject</code>	Fixes for production.	12 days ago

To the right of the commits, there's an 'About' section with the following text: "A Python/Flask sample web app targeting deployment in Azure Container Apps." Below the 'About' section are links for 'Readme' and 'MIT license'. At the bottom right of the commit list, there's a circular icon with a plus sign and a magnifying glass.

Step 2. Make a change.

- Go to the `/templates/base.html` file. (For Django, the path is: `restaurant_review/templates/restaurant_review/base.html`.)
- Select **Edit** and change the phrase "Azure Restaurant Review" to "Azure Restaurant Review - Redeployed".

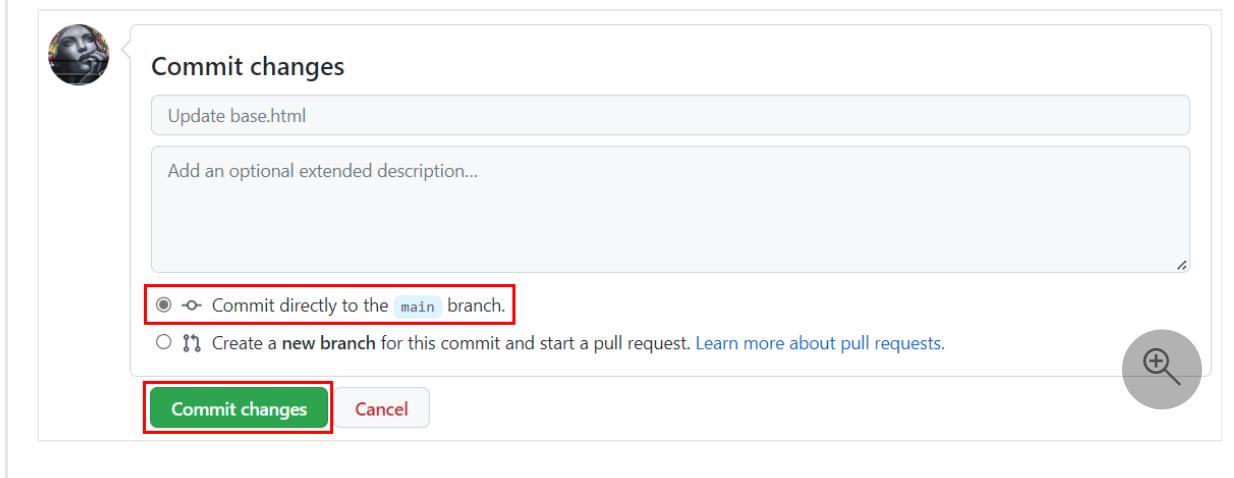


The screenshot shows a GitHub repository page for a Python Flask application. The user is viewing the `base.html` file in the `main` branch. The code editor interface includes a toolbar with buttons for Raw, Blame, and Edit (which is highlighted with a red box). The code itself contains the line `Azure Restaurant Review`, which is also highlighted with a red box. A magnifying glass icon is visible in the bottom right corner of the code area.

```
43 lines (42 sloc) | 2.96 KB
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="sha384-1BmE4ngvV2EJjI01IOAlPnZaF6r7H+HClL777uSGCvDd7jqVgqGhH&lt;br&gt;
7          <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css" integrity="sha512-9i&lt;br&gt;
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>
```

Step 3. Commit the change directly to the `main` branch.

- On the bottom of the page you editing, select the **Commit** button.
- The commit kicks off the GitHub Actions workflow.

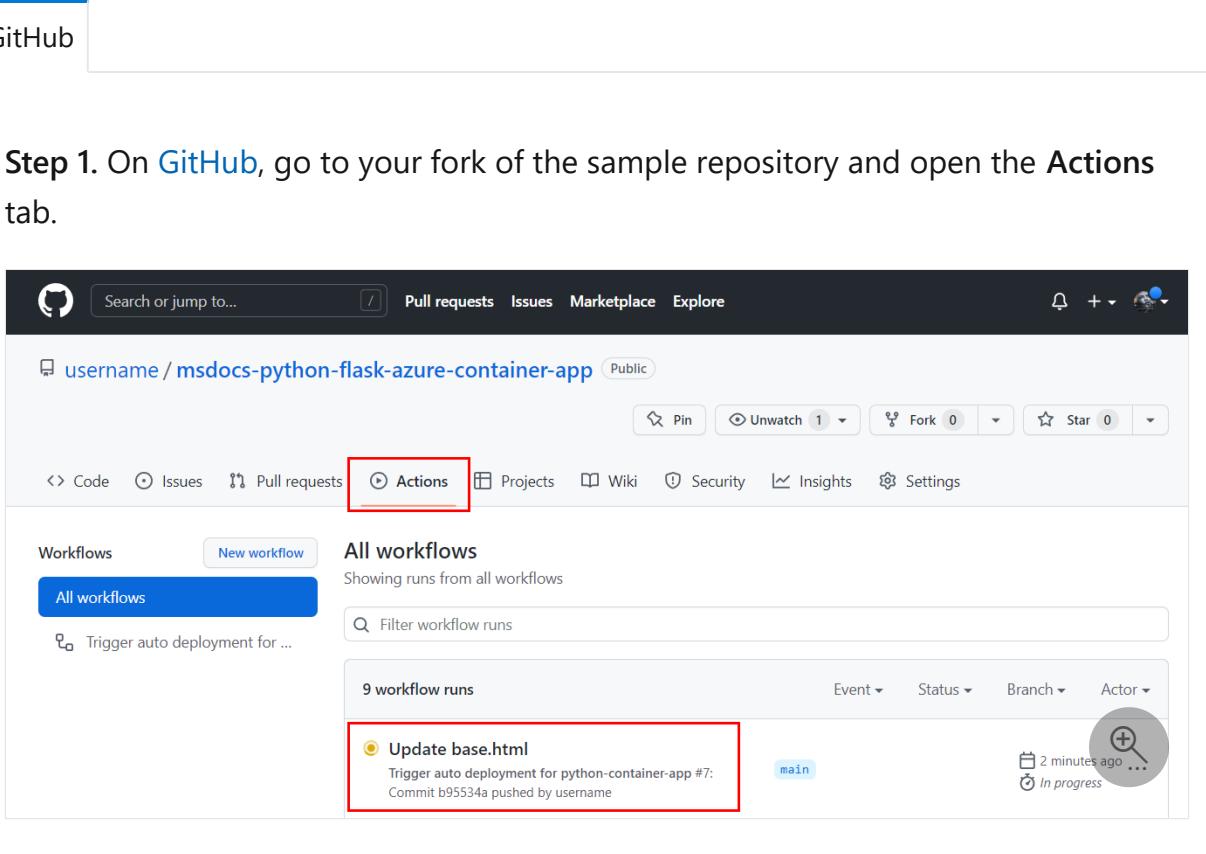


① Note

We showed making a change directly in the *main* branch. In typical software workflows, you'll make a change in a branch other than *main* and then create a pull request (PR) to merge those changes into *main*. PRs also kick off the workflow.

About GitHub Actions

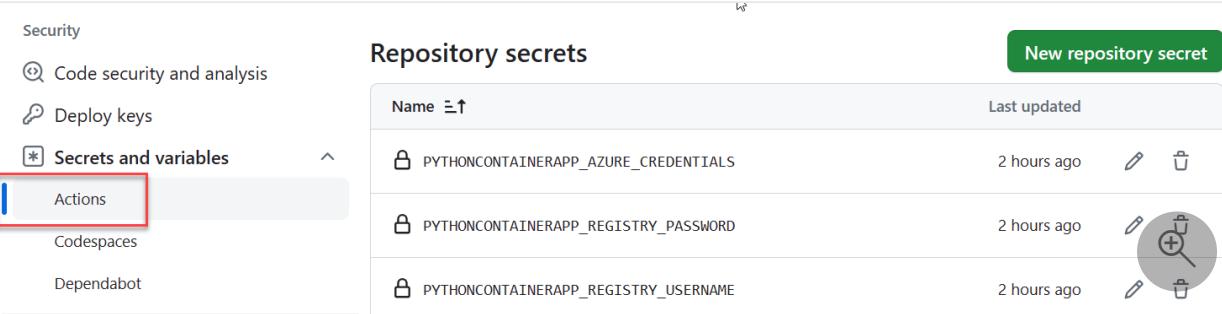
Viewing workflow history



The screenshot shows the GitHub Actions tab for a repository named "username / msdocs-python-flask-azure-container-app". The "Actions" tab is selected, highlighted with a red box. Below it, the "All workflows" section displays a list of workflow runs. One run titled "Update base.html" is highlighted with a red box. This run was triggered by "Trigger auto deployment for ...", occurred on the "main" branch, and is currently "In progress" (indicated by a circular icon). The run was updated 2 minutes ago.

Workflow secrets

In the `.github/workflows/<workflow-name>.yml` workflow file that was added to the repo, you'll see 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 **Settings** under **Security/Secrets and variables/Actions**.



The screenshot shows the "Repository secrets" page in GitHub settings. The left sidebar has a "Secrets and variables" section, which is expanded and highlighted with a red box. It includes options for "Actions", "Codespaces", and "Dependabot". The main area lists three secrets: "PYTHONCONTAINERAPP_AZURE_CREDENTIALS", "PYTHONCONTAINERAPP_REGISTRY_PASSWORD", and "PYTHONCONTAINERAPP_REGISTRY_USERNAME". Each secret entry includes a "Last updated" timestamp (2 hours ago), edit, and delete icons.

If credential information changes, you can update it here. For example, if the Azure Container Registry passwords are regenerated, you'll 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 authorize Azure Container Apps as an authorized OAuth App for your GitHub account. Container Apps uses the authorized access to create a GitHub Actions YML file in `.github/workflows/<workflow-name>.yml`. You can see your authorized apps and revoke permissions under [Integrations/Applications](#) of your account.

The screenshot shows the GitHub 'Applications' section. On the left, there's a sidebar with 'Public profile', 'Account', 'Appearance', 'Accessibility', and 'Notifications'. Below that is an 'Access' section with 'Billing and plans', 'Emails', 'Password and authentication', and 'SSH and GPG keys'. The main area is titled 'Applications' and has tabs for 'Installed GitHub Apps', 'Authorized GitHub Apps', and 'Authorized OAuth Apps'. A red box highlights the 'Authorized OAuth Apps' tab. Below it, a message says 'You have granted 9 applications access to your account.' There are 'Sort' and 'Revoke all' buttons. Two app entries are listed: 'Azure App Service' (last used within the last 2 months, owned by AzureAppService) and 'Azure App Service Container Apps' (last used within the last week, owned by AzureAppService). Both entries have a red box around them, and the 'Azure App Service Container Apps' entry also has a red box around its icon.

Troubleshooting tips

Errors setting up a service principal with the Azure CLI `az ad sp create-for-rba` command.

- You receive an error containing "InvalidSchema: No connection adapters were found".
 - Check the shell you're running in. If using Bash shell, set the MSYS_NO_PATHCONV variables as follows `export MSYS_NO_PATHCONV=1`. For more information, see the GitHub issue [Unable to create service principal with Azure CLI from git bash shell, no connection adapters were found.](#)
- You receive an error containing "More than one application have the same display name".
 - This error indicates the name is already taken for the service principal. Choose another name or leave off the `--name` argument and 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 its workflow file. There should be two jobs "build" and "deploy". For a failed job, look at the output of the job's tasks to look for problems.
- If you see an error message with "TLS handshake timeout", run the workflow manually by selecting **Trigger auto deployment** under the **Actions** tab of the repo 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 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 Azure portal: check the Azure Container Registry to see if a new Docker image was created with a timestamp after your change to the branch.
- In Azure portal: check the logs of the container app. If there's a programming error, you'll see it here.
 - Go to the Container App | Revision Management | <active container> | Revision details | Console logs
 - Choose the order of the columns to show "Time Generated", "Stream_s", and "Log_s". Sort the logs by most-recent first and look for Python `stderr` and `stdout` messages in the "Stream_s" column. Python 'print' output will be `stdout` messages.
- With the Azure CLI, use the [az containerapp logs show](#) command.

What happens when I disconnect continuous deployment?

- Stopping continuous deployment means disconnecting your container app from your repo. To disconnect:
 - In Azure portal, go the container app, select the **Continuous deployment** resource, select **Disconnect**.
 - With the Azure CLI, use the [az container app github-action remove](#) command.
- After disconnecting, in your GitHub repo:
 - The `.github/workflows/<workflow-name>.yml` file is removed from your repo.
 - Secret keys aren't removed.
 - Azure Container Apps remains as an authorized OAuth App for your GitHub account.
- After disconnecting, in Azure:

- The container is left with last deployed container. You can reconnect the container app with the Azure Container Registry, so that new container revisions pick up the latest image.
- Service principals created and used for continuous deployment aren't deleted.

Next steps

If you're done with the tutorial and don't want to incur extra costs, remove the resources 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](#).

If you plan on building on this tutorial, here are some next steps 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

Article • 04/09/2024



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 simulating a retail scenario.

ⓘ Note

To get started with quickly provisioning an AKS cluster, this article includes steps to deploy a cluster with default settings for evaluation purposes only. Before deploying 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 subscription](#), create an [Azure free account](#) before you begin.
- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in 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 [Sign in with the Azure CLI](#).

- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use 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`.
- This article requires version 2.0.64 or later of the Azure CLI. If you're using Azure Cloud Shell, the latest version is already installed there.
- Make sure that the identity you're using to create your cluster has the appropriate minimum permissions. For more details 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.

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="westeurope"
export MY_AKS_CLUSTER_NAME="myAKSCluster$RANDOM_ID"
export MY_DNS_LABEL="mydnslabel$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
```

Results:

JSON

```
{  
  "id": "/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/resourceGroups/myAKSResourceGroupxxxxxx",  
  "location": "eastus",  
  "managedBy": null,  
  "name": "testResourceGroup",  
  "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 --enable-managed-identity --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
```

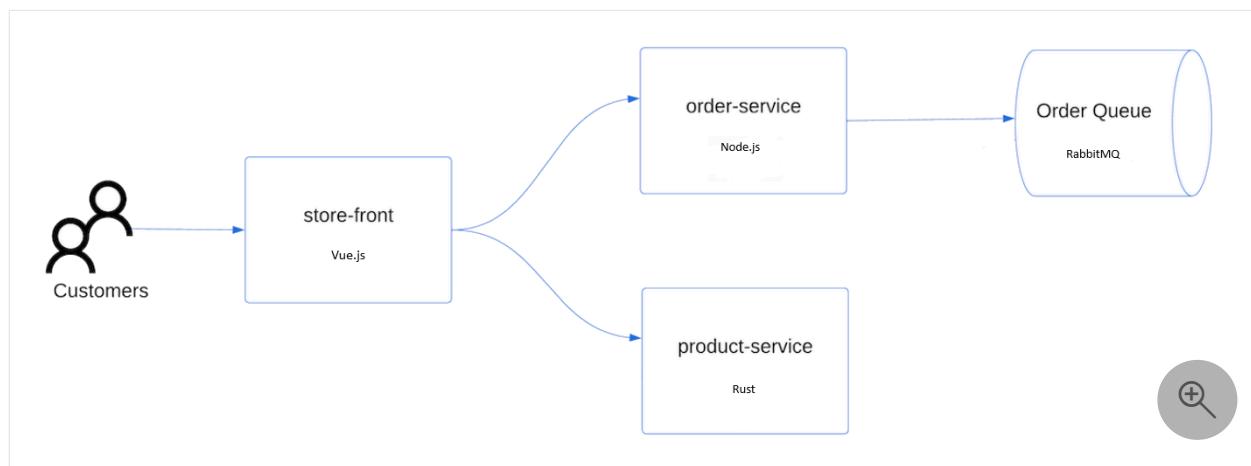
2. Verify the connection to your cluster using the [kubectl get](#) command. This command returns a list of the cluster nodes.

```
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.
- **Rabbit MQ:** Message queue for an order queue.

ⓘ Note

We don't recommend running stateful containers, such as Rabbit MQ, without persistent storage for production. These are used 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: Deployment
metadata:
  name: rabbitmq
spec:
  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
    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
resources:
  requests:
    cpu: 1m
    memory: 1Mi
  limits:
    cpu: 1m
    memory: 7Mi
---
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
---
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 CloudShell 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 minute"
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:

JSON

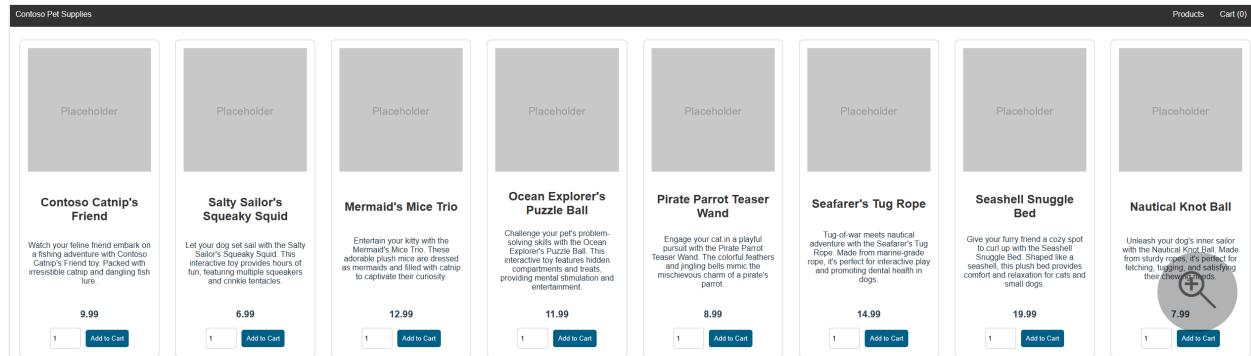
```

<!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>

```

JSON

```
echo "You can now visit your web server at $IP_ADDRESS"
```



Delete the cluster

If you don't plan on going through the [AKS tutorial](#), clean up unnecessary resources to avoid Azure charges. You can remove the resource group, container service, and all related resources using the [az group delete](#) command.

Note

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 on creating full solutions with AKS for production, see [AKS solution guidance](#).

To learn more about AKS and walk through a complete code-to-deployment example, continue to the Kubernetes cluster tutorial.

[AKS tutorial](#)

Use the Azure libraries (SDK) for Python

Article • 10/19/2023

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 >>>](#)

Azure libraries for Python usage patterns

Article • 10/18/2023

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](#). 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](#).

Library installation

pip

To install a specific library package, use `pip install`:

Windows Command Prompt

```
# Install the management library for Azure Storage  
pip install azure-mgmt-storage
```

Windows Command Prompt

```
# Install the client library for Azure Blob Storage  
pip install azure-storage-blob
```

`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.

The following code shows 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 `WebSiteManagementClient.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, which is due to version differences. 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

```
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](#) method. 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](#)

Feedback

Was this page helpful?

 Yes

 No

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

Authenticate Python apps to Azure services by using the Azure SDK for Python

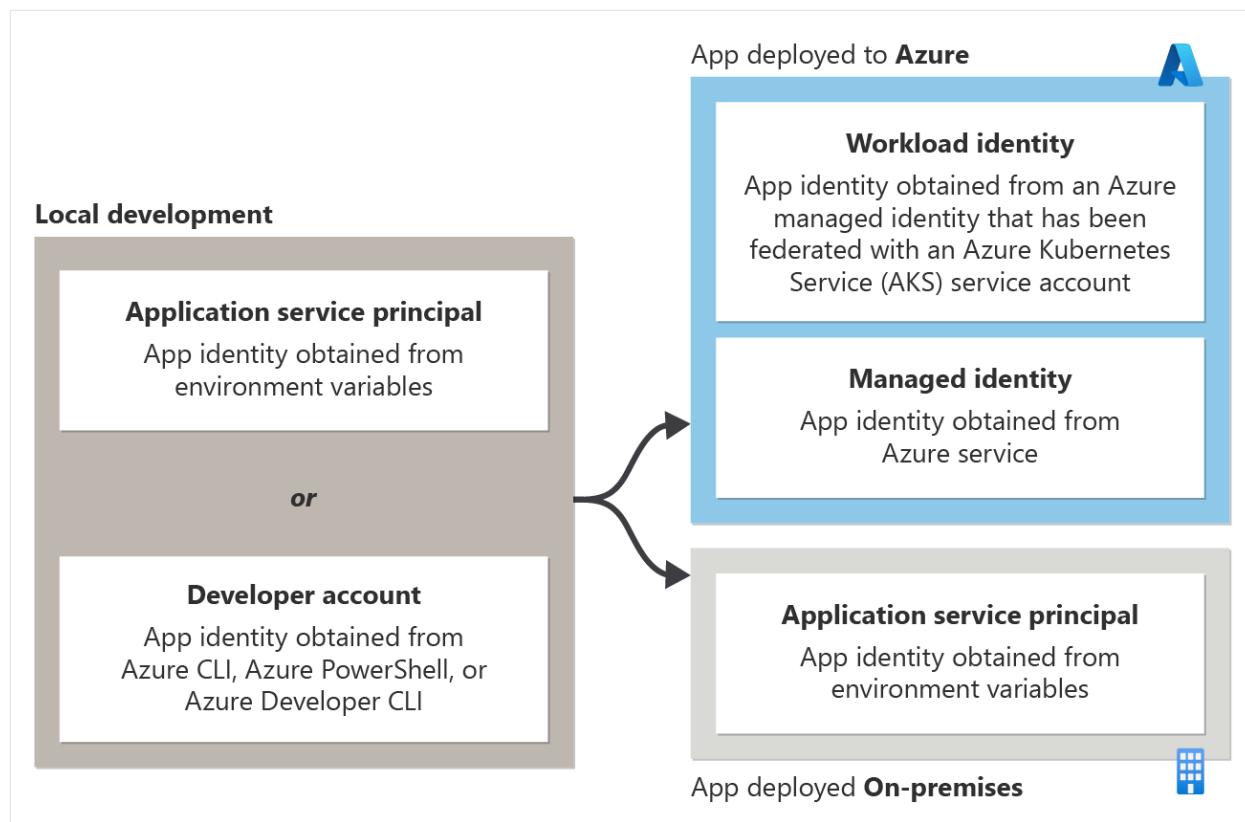
Article • 02/26/2024

When an application needs to access an Azure resource like Azure Storage, Azure Key Vault, or Azure AI services, the application must be authenticated to Azure. This requirement is true for all applications, 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 SDK for Python provides classes that support token-based authentication. Apps can 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 SDK 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 in the Azure SDK manages tokens for you behind the scenes. Managed tokens make 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 SDK are always preferred when they're authenticating to Azure resources.

Authentication in server environments

When you're hosting in a server environment, each application is assigned a unique *application identity* per environment where the application runs. In Azure, an app 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>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>Learn about auth from apps hosted outside of Azure</p>

Authentication during local development

When an application 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	<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</p>

Authentication method	Description
used during local development.	<p>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>
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>

[Learn about auth using developer accounts](#)

Use `DefaultAzureCredential` in an application

To use `DefaultAzureCredential` in a Python app, add the [azure.identity](#) package to your application.

terminal

```
pip install azure-identity
```

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)

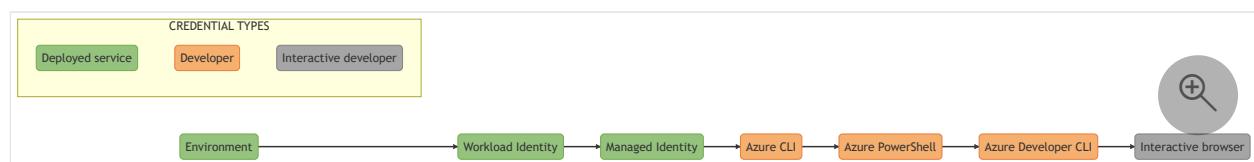
```

The `DefaultAzureCredential` object automatically detects the authentication mechanism configured for the app and obtains the necessary tokens to authenticate the app to Azure. If an application makes use of more than one SDK client, you can use the same credential object with each SDK client object.

Sequence of authentication methods when you use `DefaultAzureCredential`

Internally, `DefaultAzureCredential` implements a chain of credential providers for authenticating applications to Azure resources. Each credential provider can detect if credentials of that type are configured for the app. The `DefaultAzureCredential` object sequentially checks each provider in order and uses the credentials from the first provider that has credentials configured.

The order in which `DefaultAzureCredential` looks for credentials is shown in the following diagram and table:



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

Credential type	Description
Environment	<p>The <code>DefaultAzureCredential</code> object reads a set of environment variables to determine if an application service principal (application user) was set for the app. If so, <code>DefaultAzureCredential</code> uses these values to authenticate the app to Azure.</p> <p>This method is most often used in server environments, but you can also use it when you develop locally.</p>

Credential type	Description
Workload identity	If the application is deployed to Azure Kubernetes Service (AKS) with managed identity enabled, <code>DefaultAzureCredential</code> authenticates the app to Azure by using that managed identity. Workload identity represents a trust relationship between a user-assigned managed identity and an AKS service account. Authentication by using a workload identity is discussed in the AKS article Use Microsoft Entra Workload ID with Azure Kubernetes Service .
Managed identity	If the application is deployed to an Azure host with managed identity enabled, <code>DefaultAzureCredential</code> authenticates the app to Azure by using that managed identity. Authentication by using a managed identity is discussed in the section Authentication in server environments . This method is only available when an application is hosted in Azure by using a service like Azure App Service, Azure Functions, or Azure Virtual Machines.
Azure CLI	If you've authenticated to Azure by using the <code>az login</code> command in the Azure CLI, <code>DefaultAzureCredential</code> authenticates the app to Azure by using that same account.
Azure PowerShell	If you've authenticated to Azure by using the <code>Connect-AzAccount</code> cmdlet from Azure PowerShell, <code>DefaultAzureCredential</code> authenticates the app to Azure by using that same account.
Azure Developer CLI	If you've authenticated to Azure by using the <code>azd auth login</code> command in the Azure Developer CLI, <code>DefaultAzureCredential</code> authenticates the app to Azure by using that same account.
Interactive	If enabled, <code>DefaultAzureCredential</code> interactively authenticates you via the current system's default browser. By default, this option is disabled.

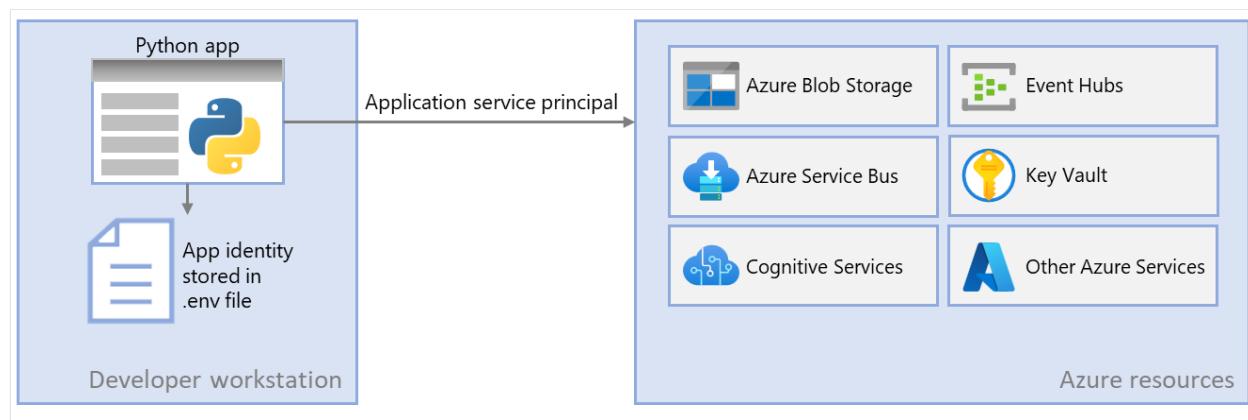
ⓘ Note

Due to a [known issue](#), `VisualStudioCodeCredential` has been removed from the `DefaultAzureCredential` token chain. When the issue is resolved in a future release, this change will be reverted. For more information, see [Azure Identity client library for Python](#).

Authenticate Python apps to Azure services during local development using service principals

Article • 12/01/2023

When creating cloud applications, developers need to debug and test applications on their local workstation. When an application is run on a developer's workstation during local development, it still must authenticate to any Azure services used by the app. This article covers how to set up dedicated application service principal objects to be used during local development.



Dedicated application service principals for local development allow you to follow the principle of least privilege during app development. Since permissions are scoped to exactly what is needed for the app during development, app code is prevented from accidentally accessing an Azure resource intended for use by a different app. This also prevents bugs from occurring when the app is moved to production because the app was overprivileged in the dev environment.

An application service principal is set up for the app when the app is registered in Azure. When registering apps for local development, it's recommended to:

- Create separate app registrations for each developer working on the app. This will create separate application service principals for each developer to use during local development and avoid the need for developers to share credentials for a single application service principal.
- Create separate app registrations per app. This scopes the app's permissions to only what is needed by the app.

During local development, environment variables are set with the application service principal's identity. The Azure SDK for Python reads these environment variables and uses this information to authenticate the app to the Azure resources it needs.

1 - Register the application in Azure

Application service principal objects are created with an app registration in Azure. This can be done using either the Azure portal or Azure CLI.

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

```
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 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": "00000000-0000-0000-0000-000000000000",
  "displayName": "{service-principal-name}",
  "password": "abcdefghijklmnopqrstuvwxyz",
  "tenant": "33333333-3333-3333-3333-333333333333"
}
```

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 `--main nickname` 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

```
az ad group create \
--display-name MyDisplay \
--mail-nickname MyDisplay \
--description "<group-description>"
```

Copy the value of the `id` property in the output of the command. This is the object ID for the group. You need it in later steps. You can also use the [az ad group show](#) command to retrieve this property.

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

```
az ad sp list \
--filter "startswith(displayName, 'msdocs')" \
--query "[].{objectId:id, displayName:displayName}" \
--output table
```

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-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](#).

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

```
az role assignment create --assignee {appId or objectId} \
--scope
/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName} \
--role "{roleName}"
```

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 `00000000-0000-0000-0000-000000000000` 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 `11111111-1111-1111-1111-111111111111`, 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 00000000-0000-0000-0000-000000000000 \
    --scope /subscriptions/11111111-1111-1111-1111-111111111111/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=00000000-0000-0000-0000-000000000000
AZURE_TENANT_ID=11111111-1111-1111-1111-111111111111
AZURE_CLIENT_SECRET=abcdefghijklmnopqrstuvwxyz
```

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

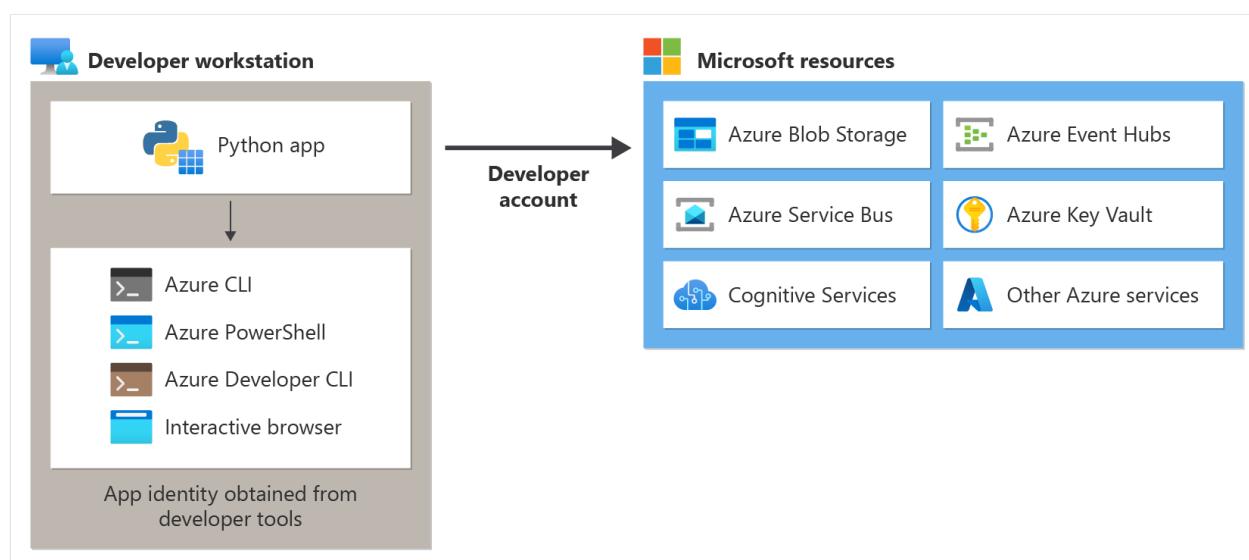
# Acquire a credential object
token_credential = DefaultAzureCredential()

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

Article • 02/24/2024

When developers create cloud applications, they typically debug and test applications on their local workstation. When an application is run on a developer's workstation during local development, it still must authenticate to any Azure services used by the app. This article covers how to use a developer's Azure credentials to authenticate the app to Azure during local development.



For an app to authenticate to Azure during local development using the developer's Azure credentials, a developer must be signed-in to Azure from the Azure CLI, Azure PowerShell, or Azure Developer CLI. The Azure SDK for Python is able to detect that the developer is signed-in from one of these tools and then obtain the necessary credentials from the credentials cache to authenticate the app to Azure as the signed-in user.

This approach is easiest to set up for a development team since it takes advantage of the developers' existing Azure accounts. However, a developer's account will likely have more permissions than required by the application, therefore exceeding the permissions the app will run with in production. As an alternative, you can [create application service principals to use during local development](#), which can be scoped to have only the access needed by the app.

1 - Create Microsoft Entra security group for local development

Since there are almost always multiple developers who work on an application, it's recommended to first create a Microsoft Entra security group to encapsulate the roles (permissions) the app needs in local development. This approach 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, they simply must be added to the correct Microsoft Entra group to get the correct permissions to work on the app.

If you have an existing Microsoft Entra security group for your development team, you can use that group. Otherwise, complete the following steps to create a Microsoft Entra security group.

Azure CLI

The [az ad group create](#) command is used to create groups in Microsoft Entra ID. The `--display-name` and `--main nickname` 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

```
az ad group create \
    --display-name MyDisplay \
    --mail-nickname MyDisplay \
    --description "<group-description>"
```

Copy the value of the `id` property in the output of the command. This is the object ID for the group. You need it in later steps. You can also use the [az ad group show](#) command to retrieve this property.

To add members to the group, you need the object ID of Azure user. Use the [az ad user list](#) to list the available service principals. The `--filter` parameter command accepts OData style filters and can be used to filter the list on the display name of the user as shown. The `--query` parameter limits the output to columns of interest.

Azure CLI

```
az ad user list \
    --filter "startswith(displayName, 'Bob')" \
```

```
--query "[].{objectId:id, displayName:displayName}" \
--output table
```

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-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

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 will be assigned to the Microsoft Entra group created in step 1. 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.

Azure CLI

```
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.

Azure CLI

```
az role definition list --query "sort_by([],{roleName:roleName, description:description}, &roleName)" --output table
```

For example, to allow the members of a group with an object ID of `00000000-0000-0000-0000-000000000000` 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 `11111111-1111-1111-1111-111111111111`, you would assign the *Storage Blob Data Contributor* role to the group using the following command.

Azure CLI

```
az role assignment create --assignee 00000000-0000-0000-0000-000000000000 \
--scope /subscriptions/11111111-1111-1111-1111-111111111111/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

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 to Azure, your application should use the [DefaultAzureCredential](#) class from the `azure.identity` package. In this scenario, `DefaultAzureCredential` will sequentially check to see if the developer has signed-in to Azure using the Azure CLI, Azure PowerShell, or Azure developer CLI. If the developer is signed-in to Azure using any of these tools, then the credentials used to sign into the tool will be used by the app to authenticate to Azure.

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 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 • 01/05/2024

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.

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": "99999999-9999-9999-9999-999999999999",  
  "tenantId": "33333333-3333-3333-3333-333333333333",  
  "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 `99999999-9999-9999-9999-999999999999` 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 `11111111-1111-1111-1111-111111111111`, 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 99999999-9999-9999-9999-  
999999999999 \  
    --scope /subscriptions/11111111-1111-1111-1111-  
111111111111/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.

Authenticate to Azure resources from Python apps hosted on-premises

Article • 11/29/2023

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

```
az ad sp create-for-rbac --name <app-name>
```

The output of the command will be 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": "00000000-0000-0000-0000-000000000000",
  "displayName": "msdocs-python-sdk-auth-prod",
  "password": "abcdefghijklmnopqrstuvwxyz",
  "tenant": "33333333-3333-3333-3333-333333333333"
}
```

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

```
az role assignment create --assignee {appId} \
    --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 service principal with the appId of `00000000-0000-0000-0000-000000000000` 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 `11111111-1111-1111-1111-111111111111`, 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 00000000-0000-0000-0000-
000000000000 \
    --scope /subscriptions/11111111-1111-1111-1111-111111111111-
```

```
111111111111/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 `unicorn.server` file as shown below.

```
unicorn.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 • 12/01/2023

This article lists additional methods apps can use to authenticate to Azure resources. The methods on this page are less commonly used and, when possible, it's encouraged to use one of the methods outlined in the [authenticating Python apps to Azure using the Azure SDK overview](#) article.

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.
 - a. 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 URLs, you can supply specific arguments to `InteractiveBrowserCredential` such as `redirect_uri`.

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.

Authentication with a username and password

This method authenticates an application using previous-collected credentials and the [UsernamePasswordCredential](#) object.

This method of authentication is discouraged because it's less secure than other flows. Also, this method isn't interactive and is therefore **not compatible with any form of multi-factor authentication or consent prompting**. The application must already have consent from the user or a directory administrator.

Furthermore, this method authenticates only work and school accounts; Microsoft accounts aren't supported. For more information, see [Sign up your organization to use Microsoft Entra ID](#).

Python

```
# Show Azure subscription information

import os
from azure.mgmt.resource import SubscriptionClient
from azure.identity import UsernamePasswordCredential

# Retrieve the information necessary for the credentials, which are assumed
# to
# be in environment variables for the purpose of this example.
client_id = os.environ["AZURE_CLIENT_ID"]
tenant_id = os.environ["AZURE_TENANT_ID"]
username = os.environ["AZURE_USERNAME"]
password = os.environ["AZURE_PASSWORD"]

credential = UsernamePasswordCredential(client_id=client_id, tenant_id =
tenant_id,
    username = username, password = password)

subscription_client = SubscriptionClient(credential)

subscription = next(subscription_client.subscriptions.list())
print(subscription.subscription_id)
```

Walkthrough: Integrated authentication for Python apps with Azure services

Article • 02/20/2024

Microsoft Entra ID along with Azure Key Vault provide a comprehensive and convenient means for applications to authenticate with Azure services and third-party services where access keys are involved.

After providing some background, this walkthrough explains these authentication features in the context of the sample, [github.com/Azure-Samples/python-integrated-authentication ↗](https://github.com/Azure-Samples/python-integrated-authentication).

Part 1: Background

Although many Azure services rely solely on role-based access control for authorization, certain services control access to their respective resources by using secrets or keys. Such services include Azure Storage, databases, Azure AI services, Key Vault, and Event Hubs.

When creating a cloud app that accesses these services, you can use the Azure portal, the Azure CLI, or Azure PowerShell to create and configure keys for your app. The keys you create are tied to specific access policies and prevent access to those app-specific resources by any other unauthorized code.

Within this general design, cloud apps must typically manage those keys and authenticate with each service individually, a process that can be both tedious and error-prone. Managing keys directly in app code also risks exposing those keys in source control and keys might be stored on unsecured developer workstations.

Fortunately, Azure provides two specific services to simplify the process and provide greater security:

- Azure Key Vault provides secure cloud-based storage for access keys (along with cryptographic keys and certificates, which aren't covered in this article). By using Key Vault, the app accesses such keys only at run time so that they never appear directly in source code.
- With [Microsoft Entra managed identities](#), an app needs to authenticate only once with Microsoft Entra ID. The app is then automatically authenticated with other Azure services, including Key Vault. As a result, your code never needs to concern itself with keys or other credentials for those Azure services. Better still, you can

run the same code both locally and in the cloud with minimal configuration requirements.

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.

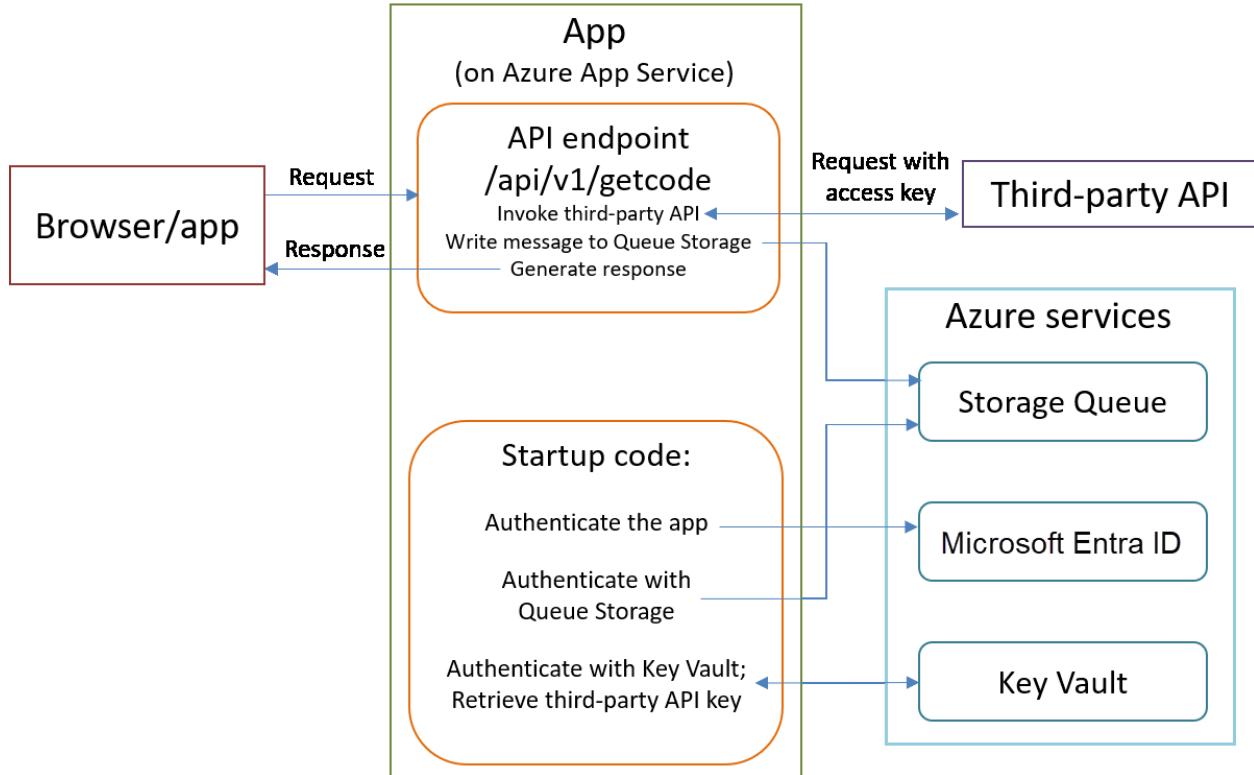
 **Important**

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 should not 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 main app exposes a public (non-authenticated) API endpoint that responds to HTTP requests with JSON data. The example endpoint as shown in this article is implemented as a simple Flask app deployed to Azure App Service.
- To generate its response, the API invokes a third-party API that requires an access key. The app retrieves that access key from Azure Key Vault at run time.
- Before the API returns a response, it writes a message to an Azure Storage Queue for later processing. (The specific processing of these messages isn't relevant to the main scenario.)



ⓘ Note

Although a public API endpoint is usually protected by its own access key, for the purposes of this article we assume the endpoint is open and unauthenticated. This assumption avoids any confusion between the app's authentication needs with those of an *external* caller of this endpoint. This scenario doesn't demonstrate such an external caller.

[Part 2 - Authentication requirements >>>](#)

Part 2: Authentication needs in the scenario

Article • 02/20/2024

[Previous part: Introduction and background](#)

Within this example scenario, the main app has the following authentication requirements:

- Authenticate with Azure Key Vault to access the stored third-party API key.
- Authenticate with the third-party API using the API key.
- Authenticate with Azure Queue Storage using the necessary credentials for the storage account.

With these three distinct requirements, the application has to manage three sets of credentials: two for Azure resources (Key Vault and Queue Storage) and one for an external resource (the third-party API).

As noted earlier, you can securely manage all the credentials in Key Vault except for those credentials needed for Key Vault itself. Once the application is authenticated with Key Vault, it can then retrieve any other keys at run time to authenticate with services like Queue Storage.

This approach, however, still requires the app to separately manage credentials for Key Vault. How then can you manage that credential securely and have it work both for local development and in your production deployment in the cloud?

A partial solution is to store the key in a server-side environment variable, which at least keeps the key out of source control. For example, you can set an environment variable through an application setting with Azure App Service and Azure Functions. The downside of this approach is that code on a developer workstation you must replicate that environment variable locally, which risks exposure of the credentials and/or accidental inclusion in source control. You could work around the problem to some extent by implementing special procedures in the development version of your code, but doing so adds complexity to your development process.

Fortunately, integrated authentication with Microsoft Entra ID allows an app to avoid handling any Azure credentials at all.

Integrated authentication with managed identity

Many Azure services, like Storage and Key Vault, are integrated with Microsoft Entra such that when an application authenticates with Microsoft Entra ID using a [managed identity](#), it's automatically authenticated with other connected resources. Authorization for the identity is handled through [role-based access control \(RBAC\)](#) and occasionally through other access policies.

This integration means that you never need to handle any Azure-related credentials in your app code and those credentials never appear on developer workstations or in source control. Furthermore, any handling of keys for third-party APIs and services is done entirely at run time, thus keeping those keys secure.

Managed identity only works with apps that are deployed to Azure. For local development, you create a separate service principal to serve as the app identity when running locally. You make this service principal available to the Azure libraries using environment variables as described in [Authenticate Python apps to Azure services during local development using service principals](#). You also assign role permissions to this service principal alongside the managed identity used in the cloud.

Once you configure and assign roles for the local service principal, the same code works both locally and in the cloud to authenticate the app with Azure resources. These details are discussed in [How to authenticate and authorize apps](#), but the short version is as follows:

1. In your code, create a `DefaultAzureCredential` object that automatically uses your managed identity when running on Azure and your separate service principal when running locally.
2. Use this credential when you create the appropriate client object for whatever resource you want to access (Key Vault, Queue Storage, etc.).
3. Authentication then takes place when you call an operation method through the client object, which generates a REST API call to the resource.
4. If the app identity is valid, then Azure also checks whether that identity is authorized for the specific operation.

The remainder of this tutorial demonstrates all the details of the process in the context of the example scenario and the accompanying sample code.

In the sample's provisioning script, all of the resources are created under a resource group named `auth-scenario-rg`. This group is created using the Azure CLI [az group create](#) command.

[Part 3 - Third-party API implementation >>>](#)

Part 3: Example third-party API implementation

Article • 02/26/2024

[Previous part: Authentication requirements](#)

In our example scenario, the main app's public endpoint uses a third-party API that's secured by an access key. This section shows an implementation of the third-party API using Azure Functions, but the API could be implemented in other ways and deployed to a different cloud server or web host. The only important aspect is that client requests to the protected endpoint must include the access key. Any app that invokes this API must securely manage that key.

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've deployed 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](#).
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

Article • 02/20/2024

[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

Article • 02/26/2024

[Previous part: Main app implementation](#)

This part examines the Python libraries brought into the main app and the environment variables required by the code. When deployed to Azure, you use application settings in Azure App Service to provide environment variables.

Dependencies and import statements

The app code requires the following libraries: Flask, the standard HTTP requests library, and the Azure libraries for Microsoft Entra ID token authentication ([azure.identity](#)), Key Vault ([azure.keyvault.secrets](#)), and Queue Storage ([azure.storage.queue](#)). These libraries are included in the app's *requirements.txt* file:

```
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 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've 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's been configured 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. (If you deploy the app to a virtual machine, you would create similar server-side variables.) You can use a library like [python-dotenv](#), which reads key-value pairs from a `.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. In this case, you create [application settings](#) with the same names, which then appear to the app 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 do not need to set these values as the app's system-assigned managed identity will be 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

Article • 02/26/2024

[Previous part: Dependencies and environment variables](#)

The app's startup code, which follows the `import` statements, initializes different variables used in the functions that handle HTTP requests.

First, it creates the Flask app object and retrieves the third-party API endpoint URL from the environment variable:

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 connect to the key vault, we must create a suitable client object. Because we want to retrieve a secret, we use the [SecretClient](#), which requires the key vault URL and the credential object that represents the identity under which the app is running.

Python

```
keyvault_client = SecretClient(vault_url=key_vault_url,  
                                credential=credential)
```

Creating the [SecretClient](#) object doesn't authenticate the credential in any way. The [SecretClient](#) is simply a client-side construct that internally manages the resource URL and the credential. 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 the app identity is authorized to access the key vault, it must still be authorized to access secrets. Otherwise, the [get_secret](#) call fails. For this reason, 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 Key Vault, we use a specific client object from the Azure libraries, [QueueClient](#), and its [from_queue_url](#) method to connect to the resource located at the URL in question. Once again, attempting to create this client object validates that the app identity represented by the credential is authorized to access the queue. As noted earlier, this authorization was granted by assigning the "Storage Queue Data Contributor" role to the main app.

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

Article • 02/26/2024

[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":"objE-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):

```
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 will receive 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've 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](#)

How to install Azure library packages for Python

Article • 11/27/2023

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](#).

Example: Use the Azure libraries to create a resource group

Article • 03/06/2024

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](#). 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 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:

```
Windows Command Prompt  
  
pip install -r requirements.txt
```

3: Write code to create a resource group

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"]

# 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(
    "PythonAzureExample-rg", {"location": "centralus"}
)

# 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(
    "PythonAzureExample-rg",
    {
        "location": "centralus",
        "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 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 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):

cmd

Windows Command Prompt

```
set AZURE_SUBSCRIPTION_ID=00000000-0000-0000-0000-000000000000
```

3. Run the script:

Windows Command Prompt

```
python provision_rg.py
```

5: Verify the resource group

You can verify that the 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 you've already had the portal open, use the **Refresh** command to update the list.
- Azure CLI: use the [az group show](#) command:

Azure CLI

```
az group show -n PythonAzureExample-rg
```

6: 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.

Azure CLI

```
az group delete -n PythonAzureExample-rg --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 • 03/06/2024

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](#). 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 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}}"
        f"{str(resource.changed_time):<{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):

```
cmd
```

```
Windows Command Prompt
```

```
set 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

Article • 01/18/2024

In this article, you learn how to use the Azure management libraries in a Python script to create a resource group that contains an Azure Storage account and a Blob storage container.

After creating the resources, see [Example: Use Azure Storage](#) to use the Azure client libraries in Python application code to upload a file to the Blob storage container.

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. 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](#). 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 the needed Azure library packages

1. Create a *requirements.txt* file that lists the management libraries used in this example:

txt

```
azure-mgmt-resource  
azure-mgmt-storage  
azure-identity
```

2. In your terminal with the virtual environment activated, install the requirements:

Console

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

3: Write code to create storage resources

Create a Python file named *provision_blob.py* with the following code. The comments explain the details. The script reads your subscription ID from an environment variable, `AZURE_SUBSCRIPTION_ID`. You set this variable in a later step. The resource group name, location, storage account name, and container name are all defined as constants in the code.

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

# 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)

# 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-Storage-rg"
LOCATION = "centralus"

# Step 1: 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}")
```

```
# 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 = f"pythonazurorestorage{random.randint(1,100000):05}"

# You can replace the storage account here with any unique name. A random
# number is used
# by default, but note that the name changes every time you run this script.
# The name must be 3-24 lower case letters and numbers only.

# 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 = "blob-container-01"  
container = storage_client.blob_containers.create(RESOURCE_GROUP_NAME,  
STORAGE_ACCOUNT_NAME, CONTAINER_NAME, {})  
  
# The fourth argument is a required BlobContainer object, but because we  
don't need any  
# special values there, so we just pass empty JSON.  
  
print(f"Provisioned blob container {container.name}")
```

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 resource groups and storage resources 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 resource groups and storage resources 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 \(azure.mgmt.resource\)](#)
- [StorageManagementClient \(azure.mgmt.storage\)](#)

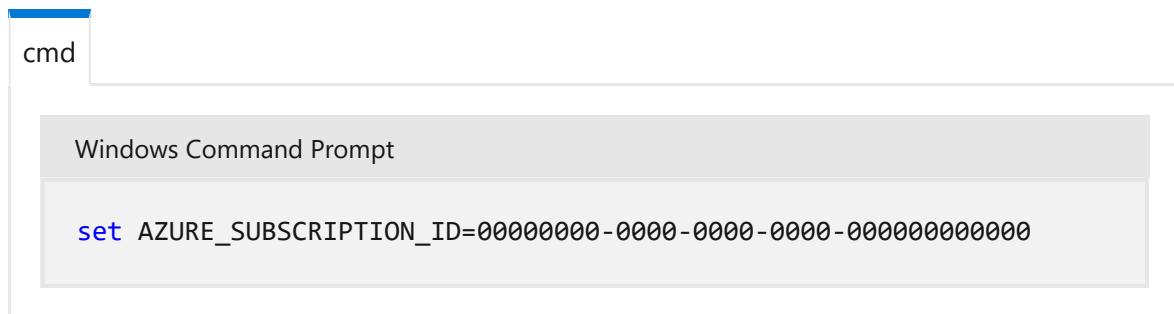
4. Run the script

1. If you haven't already, sign in to Azure using the Azure CLI:

```
Azure CLI
```

```
az login
```

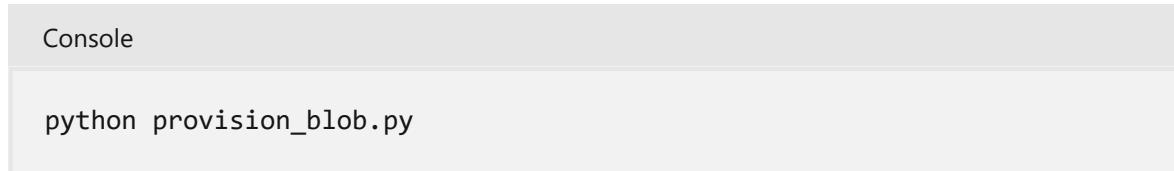
- Set the `AZURE_SUBSCRIPTION_ID` environment variable to your subscription ID. (You can run the `az account show` command and get your subscription ID from the `id` property in the output):



```
Windows Command Prompt

set AZURE_SUBSCRIPTION_ID=00000000-0000-0000-0000-000000000000
```

- Run the script:



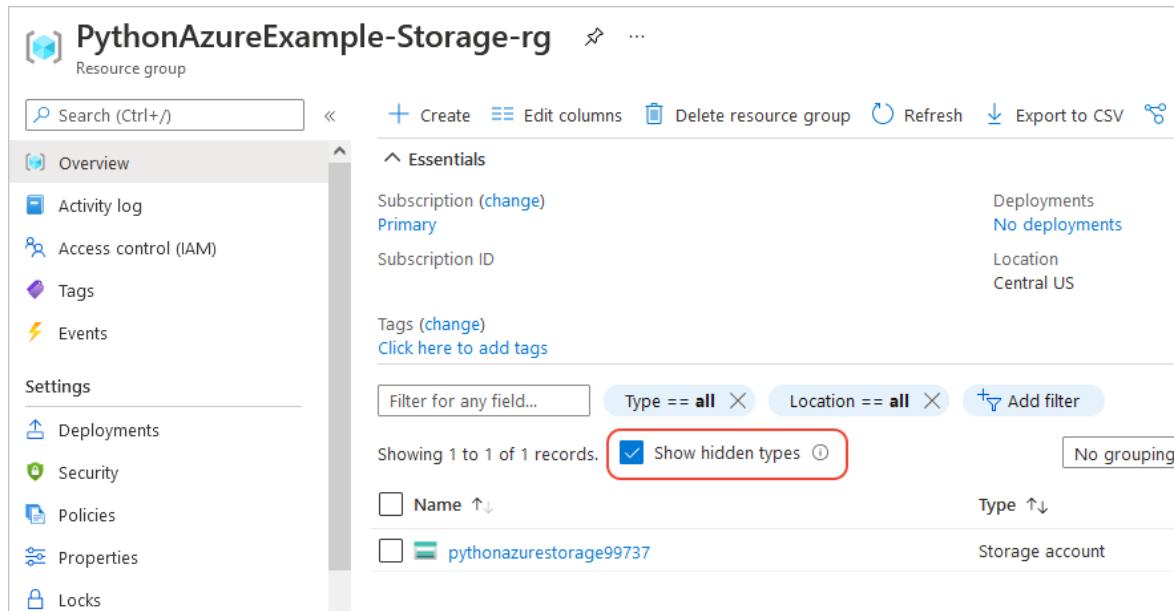
```
Console

python provision_blob.py
```

The script will take a minute or two to complete.

5: Verify the resources

- 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.



Subscription (change)	Deployments
Primary	No deployments
Subscription ID	Location
Tags (change) Click here to add tags	Central US
Filter for any field... Type == all X Location == all X Add filter	
Showing 1 to 1 of 1 records. <input checked="" type="checkbox"/> Show hidden types ⓘ No grouping	
Name ↑	Type ↑
pythonazurestorage99737	Storage account

- 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 Explorer interface. On the left, there's a sidebar with options: Data storage (highlighted with a red box), Containers (highlighted with a red box), File shares, Queues, and Tables. The main area shows a search bar at the top with the placeholder 'Search containers by prefix'. Below it is a table with a single row. The row has a checkbox column, a 'Name' column containing 'blob-container-01', and a small preview thumbnail.

3. If you want to try using these resources from application code, continue with [Example: Use Azure Storage](#).

For an additional example of using the Azure Storage management library, see the [Manage Python Storage sample](#).

For reference: equivalent Azure CLI commands

The following Azure CLI commands complete the same creation steps as the Python script:

```
cmd

Azure CLI

rem Provision the resource group

az group create ^
-n PythonAzureExample-Storage-rg ^
-l centralus

rem Provision the storage account

set account=pythonazurestorage%random%
echo Storage account name is %account%

az storage account create ^
-g PythonAzureExample-Storage-rg ^
-l centralus ^
-n %account% ^
--kind StorageV2 ^
--sku Standard_LRS

rem Retrieve the connection string
```

```
FOR /F %i IN ('az storage account show-connection-string -g PythonAzureExample-Storage-rg -n %account% --query connectionString') do  
(SET connstr=%i)  
  
rem Provision the blob container  
  
az storage container create ^  
--name blob-container-01 ^  
--account-name %account% ^  
--connection-string %connstr%
```

6: 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.

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.

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 • 01/18/2024

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](#). 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'll use 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

The following two sections demonstrate two ways to access the blob container created through [Example: Create Azure Storage](#).

The [first method \(with authentication\)](#) authenticates the app with `DefaultAzureCredential` as described in [Authenticate Python apps to Azure services during local development using service principals](#). With this method, you must first assign the appropriate permissions to the app identity, which is the recommended practice.

The [second method \(with connection string\)](#) uses a connection string to access the storage account directly. Although this method seems simpler, it has two significant drawbacks:

- A connection string inherently 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.
- 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.

For these reasons, we recommend using the authentication method in production code.

4a: Use blob storage with authentication

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,
)

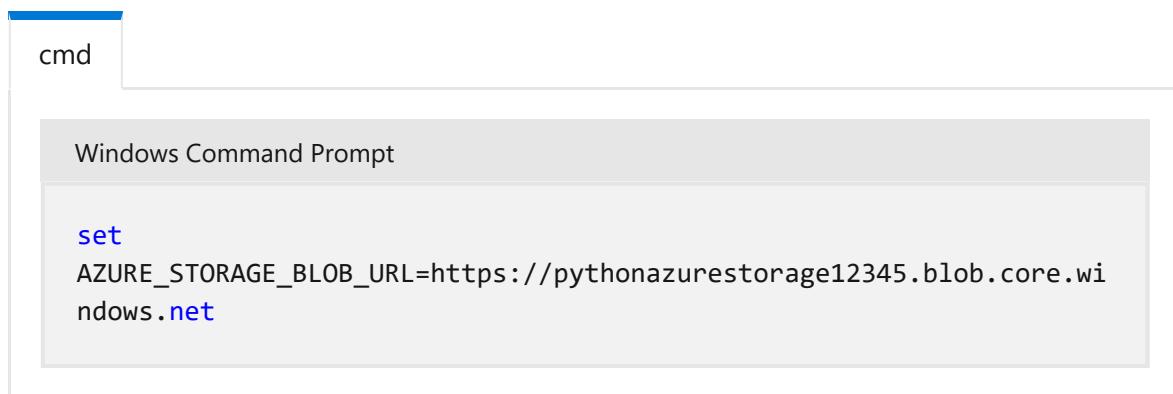
# Open a local file and upload its contents to Blob Storage
with open("./sample-source.txt", "rb") as data:
    blob_client.upload_blob(data)
    print(f"Uploaded sample-source.txt to {blob_client.url}")

```

Reference links:

- [DefaultAzureCredential \(azure.identity\)](#)
- [BlobClient \(azure.storage.blob\)](#)

2. Create an environment variable named `AZURE_STORAGE_BLOB_URL`:



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": "00000000-0000-0000-0000-000000000000",  
  "displayName": "{service-principal-name}",  
  "password": "abcdefghijklmnopqrstuvwxyz",  
  "tenant": "11111111-1111-1111-1111-111111111111"  
}
```

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=00000000-0000-0000-0000-000000000000  
set AZURE_TENANT_ID=11111111-1111-1111-1111-111111111111  
set AZURE_CLIENT_SECRET=abcdefghijklmnopqrstuvwxyz
```

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 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, "Can not 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](#).

4b: Use blob storage with a connection string

1. Create a Python file named `use_blob_conn_string.py` with the following code. The comments explain the steps.

Python

```
import os
import uuid

# Import the client object from the SDK library
from azure.storage.blob import BlobClient

# Retrieve the connection string from an environment variable. Note
# that a
# connection string grants all permissions to the caller, making it
# less
# secure than obtaining a BlobClient object using credentials.
conn_string = os.environ["AZURE_STORAGE_CONNECTION_STRING"]

# Create the client object for the resource identified by the
# connection
# string, indicating also the blob container and the name of the
# specific
# blob we want.
blob_client = BlobClient.from_connection_string(
    conn_string,
    container_name="blob-container-01",
    blob_name=f"sample-blob-{str(uuid.uuid4())[0:5]}.txt",
)

# 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}")
```

2. Create an environment variable named `AZURE_STORAGE_CONNECTION_STRING`, the value of which is the full connection string for the storage account. (This environment variable is also used by various Azure CLI commands.) You can get the connection string for your storage account by running the [az storage account show-connection-string](#) command.

```
Azure CLI
```

```
az storage account show-connection-string --resource-group  
PythonAzureExample-Storage-rg --name pythonazurestorage12345
```

Replace `PythonAzureExample-Storage-rg` and `pythonazurestorage12345` with the resource group that contains your storage account and the exact name of your storage account.

When you set the environment variable, use the entire value of the `connectionString` property in the output including the quotes.

3. Run the code:

```
Console
```

```
python use_blob_conn_string.py
```

Again, although this method is simple, a connection string authorizes all operations in a storage account. With production code, it's better to use specific permissions as described in the previous section.

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 links for 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 a single row for 'sample-blob.txt'. A red box highlights the file name 'sample-blob.txt'.

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:

Azure CLI

```
az storage blob list --container-name blob-container-01
```

If you followed the instructions to use blob storage with authentication, you can add the `--connection-string` parameter to the preceding command with the connection string for your storage account. To learn how to get the connection string, see the instructions in [4b: Use blob storage with a connection string](#). Use the whole connection string including the quotes.

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 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 blob storage with 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.

Console

```
az ad app delete --id <AZURE_CLIENT_ID>
```

See also

- [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 • 12/29/2023

This example demonstrates how to use the Azure SDK *management* libraries in a Python script to create and deploy a web app to Azure App Service. The app code is deployed from a GitHub repository.

With the management libraries (namespaces beginning with `azure-mgmt`, for example, `azure-mgmt-web`), you can write configuration and deployment programs to perform the same tasks that you can through the Azure portal, Azure CLI, or other resource management tools. For examples, see [Quickstart: Deploy a Python \(Django or Flask\) web app to Azure App Service](#). (Equivalent Azure CLI commands are given at later in this article.)

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](#). 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 the required Azure library packages

Create a file named *requirements.txt* with the following contents:

txt

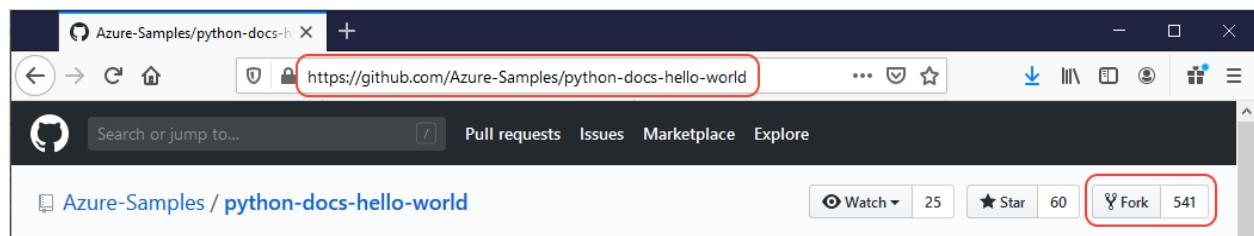
```
azure-mgmt-resource
azure-mgmt-web
azure-identity
```

In a terminal or command prompt with the virtual environment activated, install the requirements:

```
Windows Command Prompt  
pip install -r requirements.txt
```

3: Fork the sample repository

Visit <https://github.com/Azure-Samples/python-docs-hello-world> and fork the repository into your own GitHub account. You'll use a fork to ensure that you have permissions to deploy the repository to Azure.



Then create an environment variable named `REPO_URL` with the URL of your fork. The example code in the next section depends on this environment variable:

```
cmd  
Windows Command Prompt  
set REPO_URL=<url_of_your_fork>  
set AZURE_SUBSCRIPTION_ID=<subscription_id>
```

4: Write code to create and deploy a web app

Create a Python file named `provision_deploy_web_app.py` with the following code. The comments explain the details of the code. Be sure to define the `REPO_URL` and `AZURE_SUBSCRIPTION_ID` environment variables before running the script.

```
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.
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_N
AME,
    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_G
ROUP_NAME,
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("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 may 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) or 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

Windows Command Prompt

```
python provision_deploy_web_app.py
```

6: Verify the web app deployment

Visit the deployed web site by running the following command:

Azure CLI

```
az webapp browse --name PythonAzureExample-WebApp-12345 --resource-group PythonAzureExample-WebApp-rg
```

Replace the web app name (`--name` option) and resource group name (`--resource-group` option) with the values you used in the script. You should see "Hello, World!" in the browser.

If you don't see the expected output, wait a few minutes and try again.

If you still don't see the expected output, then:

1. Go to the [Azure portal](#).
2. Select **Resource groups**, and find the resource group you created.
3. Select the resource group name to view the resources it contains. Specifically, verify that there's an App Service Plan and the App Service.
4. Select the App Service, and then select **Deployment Center**.
5. Select the **Logs** tab to view deployment logs.

7: Redeploy the web app code (optional)

The script sets up the resources needed to host your web app and sets the deployment source to your fork using manual integration. With manual integration, you must trigger the web app to pull from the configured repository and branch.

The script calls the `WebSiteManagementClient.web_apps.sync_repository` method to trigger a pull from the web app. If you push subsequent code changes to your repository, you can redeploy your code by invoking this API or by using other Azure tooling like the Azure CLI or Azure portal.

You can deploy your code with 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
```

Replace the web app name (`--name` option) and resource group name (`--resource-group` option) with the values you used in the script.

To deploy your code from Azure portal:

1. Go to the [Azure portal](#).

2. Select **Resource groups**, and find the resource group you created.
3. Select the resource group name to view the resources it contains. Specifically, verify that there's an App Service Plan and the App Service.
4. Select the App Service, and then select **Deployment Center**.
5. On the top menu, select **Sync** to deploy your code.

8: Clean up resources

Azure CLI

```
az group delete --name PythonAzureExample-WebApp-rg --no-wait
```

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 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.

You can also use the `ResourceManagementClient.resource_groups.begin_delete` method to delete a resource group from code.

For reference: equivalent Azure CLI commands

The following Azure CLI commands complete the same provisioning steps as the Python script:

cmd

Azure CLI

```
rem Replace <your_github_user_name> with the account name of the fork.

set repoUrl=https://github.com/<your_github_user_name>/python-docs-hello-world
set appName=PythonAzureExample-WebApp-%random%

az group create -l centralus -n PythonAzureExample-WebApp-rg

az appservice plan create -n PythonAzureExample-WebApp-plan -g PythonAzureExample-WebApp-rg ^
    --is-linux --sku F1

echo Creating app: %appName%
```

```
az webapp create -g PythonAzureExample-WebApp-rg -n %appName% ^
--plan PythonAzureExample-WebApp-plan --runtime "python|3.8"

rem You can use --deployment-source-url with the first create command.
It is shown here
rem to match the sequence of the Python code.

az webapp create -n %appName% -g PythonAzureExample-WebApp-rg ^
--plan PythonAzureExample-WebApp-plan --runtime "python|3.8" ^
--deployment-source-url %repoUrl%

rem The previous command sets up External Git deployment from the
specified repository. This
rem command triggers a pull from the repository.

az webapp deployment source sync --name %appName% --resource-group
PythonAzureExample-WebApp-rg
```

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

Article • 02/12/2024

This example demonstrates how to use the Azure SDK management libraries in a Python script to create an Azure Database for MySQL flexible server instance and database. It also provides a simple script to query the database using the mysql-connector library (not part of the Azure SDK). You can use similar code to create an Azure Database for PostgreSQL flexible server instance and database.

[Equivalent Azure CLI commands](#) are at later in this article. If you prefer to use the Azure portal, see [Create a MySQL server](#) or [Create a PostgreSQL server](#).

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 the code. Here are some options:

- [Configure a Python virtual environment](#). 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 the needed Azure library packages

Create a file named *requirements.txt* with the following contents:

```
txt  
  
azure-mgmt-resource  
azure-mgmt-rdbms  
azure-identity  
mysql-connector-python
```

In a terminal 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: Write code to create the database

Create a Python file named *provision_db.py* with the following code. The comments explain the details. In particular, specify environment variables for

`AZURE_SUBSCRIPTION_ID` and `PUBLIC_IP_ADDRESS`. The latter variable is your workstation's IP address for this sample to run. You can use [WhatIsMyIP](#) to find your IP address.

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"]

# 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-DB-rg'
LOCATION = "southcentralus"

# 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

# We use a random number to create a reasonably unique database server name.
# If you've already provisioned a database and need to re-run the script,
set
# the DB_SERVER_NAME environment variable to that name instead.
#
# Also set DB_USER_NAME and DB_USER_PASSWORD variables to avoid using the
defaults.

db_server_name = os.environ.get("DB_SERVER_NAME", f"python-azure-example-
mysql-{random.randint(1,100000):05}")
db_admin_name = os.environ.get("DB_ADMIN_NAME", "azureuser")
db_admin_password = os.environ.get("DB_ADMIN_PASSWORD", "ChangePa$$w0rd24")

# Obtain the management client object
mysql_client = MySQLManagementClient(credential, subscription_id)

# Provision the server and wait for the result
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.FIVE7
    )
)

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"]

# For the above code, create an environment variable named PUBLIC_IP_ADDRESS
# that
# contains your workstation's public IP address as reported by a site like
# https://whatismyipaddress.com/.

# 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 with the Azure CLI to run the sample code. If your account has permissions to create resource groups and storage resources 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 resource groups and storage resources 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

- [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 \(azure.mgmt.rdbms.postgresql_flexibleServers\)](#)

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` and `PUBLIC_IP_ADDRESS` environment variables. You can run the `az account show` command to get your subscription ID from the `id` property in the output. You can use [WhatIsMyIP](#) to find your IP address.

cmd

Windows Command Prompt

```
set AZURE_SUBSCRIPTION_ID=00000000-0000-0000-0000-000000000000
set PUBLIC_IP_ADDRESS=<Your public IP address>
```

3. Optionally, set the `DB_SERVER_NAME`, `DB_ADMIN_NAME`, and `DB_ADMIN_PASSWORD` environment variables; otherwise, code defaults are used.

4. Run the script:

Console

```
python provision_db.py
```

5: Insert a record and query the database

Create a file named `use_db.py` with the following code. Note the dependencies on the `DB_SERVER_NAME`, `DB_ADMIN_NAME`, and `DB_ADMIN_PASSWORD` environment variables. You get these values from the output of running the previous code `provision_db.py` or in the code itself.

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", "azureuser")
db_admin_password = os.getenv("DB_ADMIN_PASSWORD", "ChangePa$$w0rd24")

db_name = os.getenv("DB_NAME", "example-db1")
```

```

db_port = os.getenv("DB_PORT", 3306)

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; \
                  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).

Next, download the certificate needed to communicate over TSL/SSL with your Azure Database for MySQL server from

<https://www.digicert.com/CACerts/BaltimoreCyberTrustRoot.crt.pem> and save the certificate file to the same folder as the Python file. For more information, see [Obtain an SSL Certificate](#) in the Azure Database for MySQL documentation.

Finally, run the code:

```
Windows Command Prompt
```

```
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.

6: 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.

```
Azure CLI
```

```
az group delete -n PythonAzureExample-DB-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.

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.

```
cmd
```

Azure CLI

```
az group create --location southcentralus --name PythonAzureExample-DB-rg

az mysql flexible-server create --location southcentralus --resource-group PythonAzureExample-DB-rg ^
    --name python-azure-example-mysql-12345 --admin-user azureuser --
    admin-password ChangePa$$w0rd24 ^
    --sku-name Standard_B1ms --version 5.7 --yes

# Change the IP address to the public IP address of your workstation,
# that is, the address shown
# by a site like https://whatismyipaddress.com/.

az mysql flexible-server firewall-rule create --resource-group PythonAzureExample-DB-rg --name python-azure-example-mysql-12345 ^
    --rule-name allow_ip --start-ip-address 10.11.12.13 --end-ip-address 10.11.12.13

az mysql flexible-server db create --resource-group PythonAzureExample-DB-rg --server-name python-azure-example-mysql-12345 ^
    --database-name example-db1
```

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

Article • 03/14/2024

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.

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. 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:

- [Configure a Python virtual environment](#). 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 the needed Azure library packages

Create a *requirements.txt* file that lists the management libraries used in this example:

```
txt
```

```
azure-mgmt-resource  
azure-mgmt-compute  
azure-mgmt-network  
azure-identity
```

Then, in your terminal or command prompt with the virtual environment activated, install the management libraries listed 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 with the Azure CLI to run the sample code. If your account has permissions to create resource groups and network and compute resources 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 resource groups and network and compute resources 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 \(azure.mgmt.resource\)](#)
- [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):

```
cmd
```

```
Windows Command Prompt
```

```
set AZURE_SUBSCRIPTION_ID=00000000-0000-0000-0000-000000000000
```

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' view for 'PythonAzureExample-VM-rg'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Events, Deployments, Security, Policies, Properties, Locks, Cost Management, Cost analysis, and Cost alerts (preview). The main area displays the 'Essentials' blade with sections for Subscription (Primary), Tags, and Deployments. Below these are filters for Type and Location. A table lists five resources: ExampleVM, ExampleVM_disk1_f1eda1ebf9f84a11983a1b550f37b865, python-example-ip, python-example-nic, and python-example-vnet. The last three items 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:

```
Azure CLI
az vm list --resource-group PythonAzureExample-VM-rg
```

Equivalent Azure CLI commands

```
cmd
Azure CLI
rem Provision the resource group
az group create -n PythonAzureExample-VM-rg -l westus2
rem 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

rem Provision a public IP address

az network public-ip create -g PythonAzureExample-VM-rg -n python-
example-ip ^
    --allocation-method Dynamic --version IPv4

rem 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

rem 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:

- [Create and manage Windows VMs in Azure using Python](#). You can use this example to create Linux VMs by changing the `storage_profile` parameter.
- [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

Article • 03/14/2024

Azure Managed Disks are high-performance, durable block storage designed to be used with Azure Virtual Machines and Azure VMware Solution. Azure Managed Disks provide simplified disk management, enhanced scalability, improved security, and better scaling without having to work directly with storage accounts. For more information, see [Azure Managed Disks](#).

You use the `azure-mgmt-compute` library to administer Managed Disks for an existing virtual machine.

For an example of how to create a virtual machine with the `azure-mgmt-compute` library, see [Example - Create a virtual machine](#).

The code examples in this article demonstrate how to perform some common tasks with managed disks using the `azure-mgmt-compute` library. They aren't runnable as-is, but are designed for you to incorporate into your own code. You can consult [Example - Create a virtual machine](#) to learn how to create an instance of `azure.mgmt.compute.ComputeManagementClient` in your code to run the examples.

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

You can create standalone Managed Disks in many ways as illustrated in the following sections.

Create an empty Managed Disk

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

The managed disk is created from a virtual hard disk (VHD) stored as a blob.

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

The managed disk image is created from a virtual hard disk (VHD) stored as a blob.

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',
                'blob_uri': '/subscriptions/<subscription-
id>/resourceGroups/<resource-group-
name>/providers/Microsoft.Storage/storageAccounts/<storage-account-name>/vms/<vm-name>/osDisk'
            }
        }
    }
)
image_resource = poller.result()
```

```

        '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

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 implicit Managed Disk for a specific disk image, which relieves you from specifying all the details.

A Managed Disk is created implicitly when creating a VM from an OS image in Azure. In the `storage_profile` parameter, the `os_disk` is optional and you don't have to create a storage account as required precondition to create a Virtual Machine.

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 on how to create a virtual machine using the Azure management libraries, for Python, see [Example - Create a virtual machine](#). In the create example, you 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
    '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 Managed Disks, you needed to create a storage account manually for all the VMs you wanted inside your Scale Set, and then use the list parameter `vhd_containers` to provide all the storage account name to the Scale Set RestAPI.

Because you don't have to manage storage accounts with Azure Managed Disks, your `storage_profile` for [Virtual Machine Scale Sets](#) can now be exactly the same as the one used in 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"  
            }  
        },  
        'os_profile': {  
            'computer_name_prefix': naming_infix,  
            'admin_username': 'Foo12',  
            'admin_password': 'P@ssw0rd!',  
            'linux_configuration': {  
                'ssh': {  
                    'public_keys': [  
                        {  
                            'key_data': '-----'  
                        }  
                    ]  
                }  
            }  
        }  
    }  
}
```

```

        '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

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

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

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

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 • 01/24/2024

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 (as DEBUG level), 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 (DEBUG level)

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 (DEBUG level)

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 (DEBUG level)

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': 'SUytm0872jZh+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 with authentication](#).

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

Article • 02/08/2024

If your organization requires the use of a proxy server to access internet resources, you'll 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.

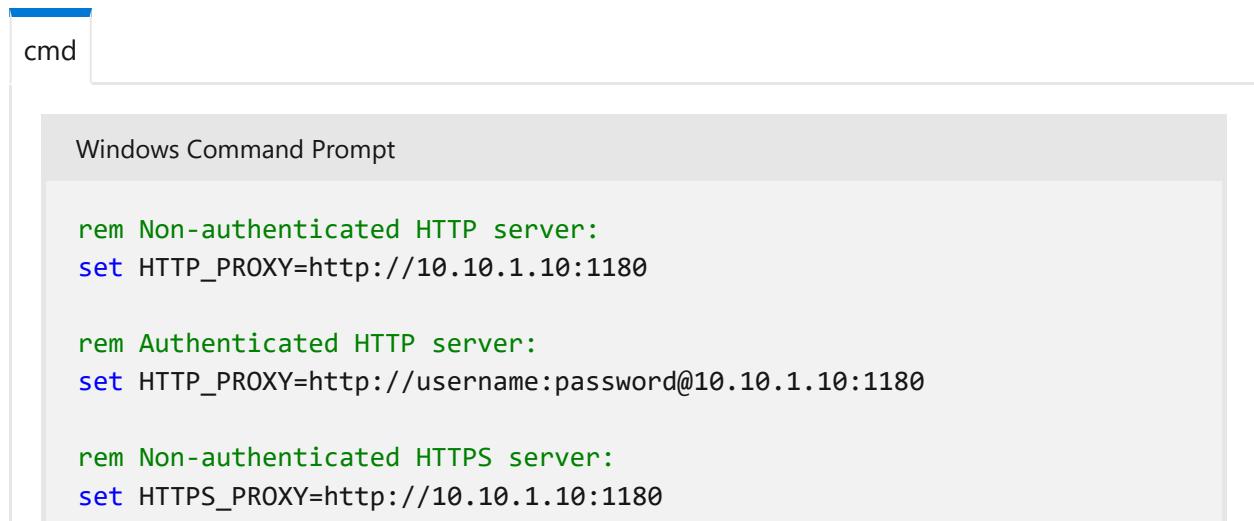
You can then configure a proxy globally by using environment variables, or you can specify a proxy by passing an argument named `proxies` to an individual client constructor or operation method.

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



```
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

Article • 12/11/2023

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

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.azuresynthesize.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.azuresynthesize.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](#).

Azure libraries package index

Article • 01/16/2024

Azure Python SDK packages are published to [PyPI](#), including beta releases designated with "b" following the version. For more information, see [Azure SDK Releases: Python](#).

If you're looking for information on how to use a specific package:

- Find the package in the table and select the GitHub link under the **Source** column. This link takes you to the source code of the package. Each package repo has an *README.md* file with some code samples to get you started.
- Find the package in the table and select the docs link under the **Docs** column. This link takes you to the API documentation. The API doc page gives an overview of the package and is useful when you're looking for an overview of all the classes of a package and an overview of all the methods of a class.
- Go to the [Azure for Python developers](#) documentation. In these docs, you can find more examples of using packages, and quickstarts and tutorials. For example, to learn about installing packages, see [How to install Azure library packages for Python](#).

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*.

 **Note**

For Conda libraries, see the [Microsoft channel on anaconda.org](#).

Libraries using `azure.core`

 Expand table

Name	Package	Docs	Source
AI Generative	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
AI Resources	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Anomaly Detector	PyPI 3.0.0b6	docs	GitHub 3.0.0b6

Name	Package	Docs	Source
App Configuration	PyPI 1.5.0	docs	GitHub 1.5.0
App Configuration Provider	PyPI 1.0.0	docs	GitHub 1.0.0
	PyPI 1.1.0b3		GitHub 1.1.0b3
Attestation	PyPI 1.0.0	docs	GitHub 1.0.0
Azure AI Search	PyPI 11.4.0	docs	GitHub 11.4.0
Azure AI Vision SDK	PyPI 0.15.1b1		GitHub 0.15.1b1
Azure Blob Storage Checkpoint Store	PyPI 1.1.4	docs	GitHub 1.1.4
Azure Blob Storage Checkpoint Store AIO	PyPI 1.1.4	docs	GitHub 1.1.4
Azure Monitor OpenTelemetry	PyPI 1.1.1	docs	GitHub 1.1.1
Azure Remote Rendering	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Communication Call Automation	PyPI 1.1.0	docs	GitHub 1.1.0
Communication Chat	PyPI 1.2.0	docs	GitHub 1.2.0
Communication Email	PyPI 1.0.0	docs	GitHub 1.0.0
Communication Identity	PyPI 1.4.0	docs	GitHub 1.4.0
Communication JobRouter	PyPI 1.0.0	docs	GitHub 1.0.0
Communication Network Traversal	PyPI 1.1.0b1	docs	GitHub 1.1.0b1
Communication Phone Numbers	PyPI 1.1.0	docs	GitHub 1.1.0
	PyPI 1.2.0b1		GitHub 1.2.0b1
Communication Rooms	PyPI 1.0.0	docs	GitHub 1.0.0
	PyPI 1.1.0b1		GitHub 1.1.0b1
Communication Sms	PyPI 1.0.1	docs	GitHub 1.0.1
Confidential Ledger	PyPI 1.1.1	docs	GitHub 1.1.1
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 1.29.6	docs	GitHub 1.29.6
Core - Client - Experimental	PyPI 1.0.0b4	docs	GitHub 1.0.0b4

Name	Package	Docs	Source
Core - Client - Tracing Opentelemetry	PyPI 1.0.0b11 ↗	docs	GitHub 1.0.0b11 ↗
Cosmos DB	PyPI 4.5.1 ↗ PyPI 4.5.2b3 ↗	docs	GitHub 4.5.1 ↗ GitHub 4.5.2b3 ↗
Defender EASM	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Dev Center	PyPI 1.0.0b3 ↗	docs	GitHub 1.0.0b3 ↗
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 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Document Translation	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Event Grid	PyPI 4.16.0 ↗ PyPI 4.17.0b1 ↗	docs	GitHub 4.16.0 ↗ GitHub 4.17.0b1 ↗
Event Hubs	PyPI 5.11.5 ↗	docs	GitHub 5.11.5 ↗
FarmBeats	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Form Recognizer	PyPI 3.3.2 ↗	docs	GitHub 3.3.2 ↗
Health Insights Cancer Profiling	PyPI 1.0.0b1.post1 ↗	docs	GitHub 1.0.0b1.post1 ↗
Health Insights Clinical Matching	PyPI 1.0.0b1.post1 ↗	docs	GitHub 1.0.0b1.post1 ↗
Identity	PyPI 1.15.0 ↗	docs	GitHub 1.15.0 ↗
Key Vault - Administration	PyPI 4.3.0 ↗ PyPI 4.4.0b2 ↗	docs	GitHub 4.3.0 ↗ GitHub 4.4.0b2 ↗
Key Vault - Certificates	PyPI 4.7.0 ↗ PyPI 4.8.0b3 ↗	docs	GitHub 4.7.0 ↗ GitHub 4.8.0b3 ↗
Key Vault - Keys	PyPI 4.8.0 ↗ PyPI 4.9.0b3 ↗	docs	GitHub 4.8.0 ↗ GitHub 4.9.0b3 ↗
Key Vault - Secrets	PyPI 4.7.0 ↗ PyPI 4.8.0b2 ↗	docs	GitHub 4.7.0 ↗ GitHub 4.8.0b2 ↗
Load Testing	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Machine Learning	PyPI 1.12.1 ↗	docs	GitHub 1.12.1 ↗
Machine Learning - Feature Store	PyPI 1.0.1 ↗		GitHub 1.0.1 ↗

Name	Package	Docs	Source
Managed Private Endpoints	PyPI 0.4.0	docs	GitHub 0.4.0
Maps Geolocation	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Maps Render	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Maps Route	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Maps Search	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Media Analytics Edge	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Metrics Advisor	PyPI 1.0.0	docs	GitHub 1.0.0
Mixed Reality Authentication	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Monitor Ingestion	PyPI 1.0.3	docs	GitHub 1.0.3
Monitor Query	PyPI 1.2.0 PyPI 1.3.0b2	docs	GitHub 1.2.0 GitHub 1.3.0b2
OpenTelemetry Exporter	PyPI 1.0.0b21	docs	GitHub 1.0.0b21
Personalizer	PyPI 1.0.0b1		GitHub 1.0.0b1
Purview Account	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Purview Catalog	PyPI 1.0.0b4	docs	GitHub 1.0.0b4
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 1.2.0 PyPI 1.3.0b3	docs	GitHub 1.2.0 GitHub 1.3.0b3
Schema Registry - Avro	PyPI 1.0.0	docs	GitHub 1.0.0
Schema Registry - Avro	PyPI 1.0.0b4.post1	docs	GitHub 1.0.0b4.post1
Service Bus	PyPI 7.11.4	docs	GitHub 7.11.4
Spark	PyPI 0.7.0	docs	GitHub 0.7.0
Storage - Blobs	PyPI 12.19.0	docs	GitHub 12.19.0

Name	Package	Docs	Source
Storage - Blobs Changefeed	PyPI 12.0.0b4	docs	GitHub 12.0.0b4
Storage - Files Data Lake	PyPI 12.14.0	docs	GitHub 12.14.0
Storage - Files Share	PyPI 12.15.0	docs	GitHub 12.15.0
Storage - Queues	PyPI 12.9.0	docs	GitHub 12.9.0
Synapse - AccessControl	PyPI 0.7.0	docs	GitHub 0.7.0
Synapse - Artifacts	PyPI 0.18.0	docs	GitHub 0.18.0
Synapse - Monitoring	PyPI 0.2.0	docs	GitHub 0.2.0
Tables	PyPI 12.5.0	docs	GitHub 12.5.0
Text Analytics	PyPI 5.3.0	docs	GitHub 5.3.0
Text Translation	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Video Analyzer Edge	PyPI 1.0.0b4	docs	GitHub 1.0.0b4
Web PubSub	PyPI 1.0.1 PyPI 1.1.0b1	docs	GitHub 1.0.1 GitHub 1.1.0b1
Core - Management - Core	PyPI 1.4.0	docs	GitHub 1.4.0
Resource Management - Dev Center	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
Resource Management - Elastic SAN	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
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 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - API Management	PyPI 4.0.0	docs	GitHub 4.0.0
Resource Management - App Compliance	PyPI 1.0.0b1	docs	GitHub 1.0.0b1

Name	Package	Docs	Source
Automation			
Resource Management - App Configuration	PyPI 3.0.0 ↗	docs	GitHub 3.0.0 ↗
Resource Management - App Platform	PyPI 8.0.0 ↗	docs	GitHub 8.0.0 ↗
Resource Management - App Service	PyPI 7.2.0 ↗	docs	GitHub 7.2.0 ↗
Resource Management - Application Insights	PyPI 4.0.0 ↗	docs	GitHub 4.0.0 ↗
Resource Management - Arc Data	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
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 - Automange	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 1.1.0b3 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b3 ↗
Resource Management - Azure AD B2C	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Azure AI Search	PyPI 9.1.0 ↗	docs	GitHub 9.1.0 ↗
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 8.0.0b3 ↗	docs	GitHub 7.0.0 ↗ GitHub 8.0.0b3 ↗
Resource Management - Azure VMware Solution	PyPI 8.0.0 ↗	docs	GitHub 8.0.0 ↗
Resource Management - Backup	PyPI 8.0.0 ↗	docs	GitHub 8.0.0 ↗
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 17.2.0 ↗	docs	GitHub 17.2.0 ↗
Resource Management - Billing	PyPI 6.0.0 ↗ PyPI 6.1.0b1 ↗	docs	GitHub 6.0.0 ↗ GitHub 6.1.0b1 ↗
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 ↗

Name	Package	Docs	Source
Resource Management - Change Analysis	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Chaos	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Cognitive Services	PyPI 13.5.0 ↗	docs	GitHub 13.5.0 ↗
Resource Management - Commerce	PyPI 6.0.0 ↗ PyPI 6.1.0b1 ↗	docs	GitHub 6.0.0 ↗ GitHub 6.1.0b1 ↗
Resource Management - Communication	PyPI 2.0.0 ↗ PyPI 2.1.0b2 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b2 ↗
Resource Management - Compute	PyPI 30.4.0 ↗	docs	GitHub 30.4.0 ↗
Resource Management - Confidential Ledger	PyPI 1.0.0 ↗ PyPI 2.0.0b3 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b3 ↗
Resource Management - Confluent	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
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 3.0.0 ↗	docs	GitHub 3.0.0 ↗
Resource Management - Container Instances	PyPI 10.1.0 ↗	docs	GitHub 10.1.0 ↗
Resource Management - Container Registry	PyPI 10.3.0 ↗	docs	GitHub 10.3.0 ↗
Resource Management - Container Service	PyPI 28.0.0 ↗	docs	GitHub 28.0.0 ↗
Resource Management - Container Service Fleet	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Content Delivery Network	PyPI 13.0.0 ↗	docs	GitHub 13.0.0 ↗
Resource Management - Cosmos DB	PyPI 9.4.0 ↗ PyPI 10.0.0b1 ↗	docs	GitHub 9.4.0 ↗ GitHub 10.0.0b1 ↗
Resource Management - Cosmos DB for PostgreSQL	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
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 2.0.0 ↗	docs	GitHub 2.0.0 ↗

Name	Package	Docs	Source
Resource Management - Data Box Edge	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Data Factory	PyPI 4.0.0 ↗	docs	GitHub 4.0.0 ↗
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 1.3.0 ↗	docs	GitHub 1.3.0 ↗
Resource Management - Data Share	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
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 - Deployment Manager	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Desktop Virtualization	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
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 Update	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - DevTest Labs	PyPI 9.0.0 ↗ PyPI 10.0.0b1 ↗	docs	GitHub 9.0.0 ↗ GitHub 10.0.0b1 ↗
Resource Management - Digital Twins	PyPI 6.4.0 ↗	docs	GitHub 6.4.0 ↗
Resource Management - DNS	PyPI 8.1.0 ↗	docs	GitHub 8.1.0 ↗
Resource Management - DNS Resolver	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Dynatrace	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Edge Order	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗

Name	Package	Docs	Source
Resource Management - Education	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Elastic	PyPI 1.0.0 PyPI 1.1.0b3	docs	GitHub 1.0.0 GitHub 1.1.0b3
Resource Management - Event Grid	PyPI 10.2.0 PyPI 10.3.0b3	docs	GitHub 10.2.0 GitHub 10.3.0b3
Resource Management - Event Hubs	PyPI 11.0.0	docs	GitHub 11.0.0
Resource Management - Extended Location	PyPI 1.1.0 PyPI 1.2.0b1	docs	GitHub 1.1.0 GitHub 1.2.0b1
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 1.1.0	docs	GitHub 1.1.0
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 - HDInsight	PyPI 9.0.0	docs	GitHub 9.0.0
Resource Management - HDInsight Containers	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Health Bot	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Healthcare APIs	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Hybrid Compute	PyPI 8.0.0 PyPI 9.0.0b1	docs	GitHub 8.0.0 GitHub 9.0.0b1
Resource Management - Hybrid Connectivity	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Hybrid Container Service	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Hybrid Kubernetes	PyPI 1.1.0 PyPI 1.2.0b1	docs	GitHub 1.1.0 GitHub 1.2.0b1
Resource Management - Hybrid Network	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Image Builder	PyPI 1.3.0	docs	GitHub 1.3.0
Resource Management - IoT Central	PyPI 9.0.0 PyPI 10.0.0b2	docs	GitHub 9.0.0 GitHub 10.0.0b2

Name	Package	Docs	Source
Resource Management - IoT Firmware Defense	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - IoT Hub	PyPI 3.0.0	docs	GitHub 3.0.0
Resource Management - Key Vault	PyPI 10.3.0	docs	GitHub 10.3.0
Resource Management - Kubernetes Configuration	PyPI 3.1.0	docs	GitHub 3.1.0
Resource Management - Kusto	PyPI 3.3.0	docs	GitHub 3.3.0
Resource Management - Lab Services	PyPI 2.0.0 PyPI 2.1.0b1	docs	GitHub 2.0.0 GitHub 2.1.0b1
Resource Management - Load Testing	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Log Analytics	PyPI 12.0.0 PyPI 13.0.0b6	docs	GitHub 12.0.0 GitHub 13.0.0b6
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 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
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 2.2.0b1	docs	GitHub 2.1.0 GitHub 2.2.0b1
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 7.0.0b1	docs	GitHub 6.0.0 GitHub 7.0.0b1
Resource Management - Management Groups	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1

Name	Package	Docs	Source
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 1.2.0b1 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b1 ↗
Resource Management - Media Services	PyPI 10.2.0 ↗	docs	GitHub 10.2.0 ↗
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 3.1.0 ↗	docs	GitHub 3.1.0 ↗
Resource Management - Monitor	PyPI 6.0.2 ↗	docs	GitHub 6.0.2 ↗
Resource Management - NetApp Files	PyPI 11.0.0 ↗ PyPI 12.0.0b1 ↗	docs	GitHub 11.0.0 ↗ GitHub 12.0.0b1 ↗
Resource Management - Network	PyPI 25.2.0 ↗	docs	GitHub 25.2.0 ↗
Resource Management - Network Function	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Networkanalytics	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Networkcloud	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - New Relic Observability	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Nginx	PyPI 3.0.0 ↗	docs	GitHub 3.0.0 ↗
Resource Management - Notification Hubs	PyPI 8.0.0 ↗ PyPI 8.1.0b1 ↗	docs	GitHub 8.0.0 ↗ GitHub 8.1.0b1 ↗
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 - 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 - Playwright Testing	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗

Name	Package	Docs	Source
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 - PostgreSQL	PyPI 10.1.0 ↗ PyPI 10.2.0b13 ↗	docs	GitHub 10.1.0 ↗ GitHub 10.2.0b13 ↗
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 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Purview	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Quantum	PyPI 1.0.0b4 ↗	docs	GitHub 1.0.0b4 ↗
Resource Management - Qumulo	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Quota	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Recovery Services	PyPI 2.5.0 ↗	docs	GitHub 2.5.0 ↗
Resource Management - Recoveryservicesdatareplication	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Red Hat OpenShift	PyPI 1.4.0 ↗	docs	GitHub 1.4.0 ↗
Resource Management - Redis	PyPI 14.3.0 ↗	docs	GitHub 14.3.0 ↗
Resource Management - Redis Enterprise	PyPI 2.0.0 ↗ PyPI 2.1.0b2 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b2 ↗
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 ↗
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 1.0.0b5 ↗	docs	GitHub 1.0.0b5 ↗
Resource Management - Resource Mover	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗

Name	Package	Docs	Source
Resource Management - Resources	PyPI 23.0.1 ↗ PyPI 23.1.0b2 ↗	docs	GitHub 23.0.1 ↗ GitHub 23.1.0b2 ↗
Resource Management - Resources	PyPI 23.0.1 ↗ PyPI 23.1.0b2 ↗	docs	GitHub 23.0.1 ↗ GitHub 23.1.0b2 ↗
Resource Management - Scheduler	PyPI 2.0.0 ↗ PyPI 7.0.0b1 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Scvmm	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Security	PyPI 5.0.0 ↗	docs	GitHub 5.0.0 ↗
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 2.0.0b2 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b2 ↗
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 8.2.0 ↗	docs	GitHub 8.2.0 ↗
Resource Management - Service Fabric	PyPI 2.1.0 ↗	docs	GitHub 2.1.0 ↗
Resource Management - Service Fabric Managed Clusters	PyPI 1.0.0 ↗ PyPI 2.0.0b5 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b5 ↗
Resource Management - Service Linker	PyPI 1.1.0 ↗ PyPI 1.2.0b1 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b1 ↗
Resource Management - Service Networking	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - SignalR	PyPI 1.2.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.2.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Site Recovery	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Sphere	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - SQL	PyPI 3.0.1 ↗ PyPI 4.0.0b15 ↗	docs	GitHub 3.0.1 ↗ GitHub 4.0.0b15 ↗
Resource Management - SQL Virtual Machine	PyPI 1.0.0b6 ↗	docs	GitHub 1.0.0b6 ↗
Resource Management - Storage	PyPI 21.1.0 ↗	docs	GitHub 21.1.0 ↗
Resource Management - Storage Cache	PyPI 1.5.0 ↗	docs	GitHub 1.5.0 ↗
Resource Management - Storage Import/Export	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗

Name	Package	Docs	Source
Resource Management - Storage Mover	PyPI 2.0.0	docs	GitHub 2.0.0
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 2.0.0b1	docs	GitHub 1.0.0 GitHub 2.0.0b1
Resource Management - Subscriptions	PyPI 3.1.1 PyPI 3.2.0b1	docs	GitHub 3.1.1 GitHub 3.2.0b1
Resource Management - Support	PyPI 6.0.0 PyPI 6.1.0b2	docs	GitHub 6.0.0 GitHub 6.1.0b2
Resource Management - Synapse	PyPI 2.0.0 PyPI 2.1.0b7	docs	GitHub 2.0.0 GitHub 2.1.0b7
Resource Management - Test Base	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
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 - 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 1.1.0 PyPI 2.0.0b2	docs	GitHub 1.1.0 GitHub 2.0.0b2
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

All libraries

[Expand table](#)

Name	Package	Docs	Source
AI Generative	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
AI Resources	PyPI 1.0.0b2	docs	GitHub 1.0.0b2

Name	Package	Docs	Source
Anomaly Detector	PyPI 3.0.0b6 ↗	docs	GitHub 3.0.0b6 ↗
App Configuration	PyPI 1.5.0 ↗	docs	GitHub 1.5.0 ↗
App Configuration Provider	PyPI 1.0.0 ↗ PyPI 1.1.0b3 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b3 ↗
Attestation	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Azure AI Search	PyPI 11.4.0 ↗	docs	GitHub 11.4.0 ↗
Azure AI Vision SDK	PyPI 0.15.1b1 ↗		GitHub 0.15.1b1 ↗
Azure Blob Storage Checkpoint Store	PyPI 1.1.4 ↗	docs	GitHub 1.1.4 ↗
Azure Blob Storage Checkpoint Store AIO	PyPI 1.1.4 ↗	docs	GitHub 1.1.4 ↗
Azure Monitor OpenTelemetry	PyPI 1.1.1 ↗	docs	GitHub 1.1.1 ↗
Azure Remote Rendering	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Communication Call Automation	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Communication Chat	PyPI 1.2.0 ↗	docs	GitHub 1.2.0 ↗
Communication Email	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Communication Identity	PyPI 1.4.0 ↗	docs	GitHub 1.4.0 ↗
Communication JobRouter	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Communication Network Traversal	PyPI 1.1.0b1 ↗	docs	GitHub 1.1.0b1 ↗
Communication Phone Numbers	PyPI 1.1.0 ↗ PyPI 1.2.0b1 ↗	docs	GitHub 1.1.0 ↗ GitHub 1.2.0b1 ↗
Communication Rooms	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Communication Sms	PyPI 1.0.1 ↗	docs	GitHub 1.0.1 ↗
Confidential Ledger	PyPI 1.1.1 ↗	docs	GitHub 1.1.1 ↗
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 1.29.6 ↗	docs	GitHub 1.29.6 ↗

Name	Package	Docs	Source
Core - Client - Experimental	PyPI 1.0.0b4	docs	GitHub 1.0.0b4
Core - Client - Tracing Opentelemetry	PyPI 1.0.0b11	docs	GitHub 1.0.0b11
Cosmos DB	PyPI 4.5.1 PyPI 4.5.2b3	docs	GitHub 4.5.1 GitHub 4.5.2b3
Defender EASM	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Dev Center	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
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 1.0.0b1	docs	GitHub 1.0.0b1
Document Translation	PyPI 1.0.0	docs	GitHub 1.0.0
Event Grid	PyPI 4.16.0 PyPI 4.17.0b1	docs	GitHub 4.16.0 GitHub 4.17.0b1
Event Hubs	PyPI 5.11.5	docs	GitHub 5.11.5
FarmBeats	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Form Recognizer	PyPI 3.3.2	docs	GitHub 3.3.2
Health Insights Cancer Profiling	PyPI 1.0.0b1.post1	docs	GitHub 1.0.0b1.post1
Health Insights Clinical Matching	PyPI 1.0.0b1.post1	docs	GitHub 1.0.0b1.post1
Identity	PyPI 1.15.0	docs	GitHub 1.15.0
Key Vault - Administration	PyPI 4.3.0 PyPI 4.4.0b2	docs	GitHub 4.3.0 GitHub 4.4.0b2
Key Vault - Certificates	PyPI 4.7.0 PyPI 4.8.0b3	docs	GitHub 4.7.0 GitHub 4.8.0b3
Key Vault - Keys	PyPI 4.8.0 PyPI 4.9.0b3	docs	GitHub 4.8.0 GitHub 4.9.0b3
Key Vault - Secrets	PyPI 4.7.0 PyPI 4.8.0b2	docs	GitHub 4.7.0 GitHub 4.8.0b2
Load Testing	PyPI 1.0.0	docs	GitHub 1.0.0
Machine Learning	PyPI 1.12.1	docs	GitHub 1.12.1

Name	Package	Docs	Source
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 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Maps Render	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Maps Route	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Maps Search	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Media Analytics Edge	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Metrics Advisor	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Mixed Reality Authentication	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Monitor Ingestion	PyPI 1.0.3 ↗	docs	GitHub 1.0.3 ↗
Monitor Query	PyPI 1.2.0 ↗ PyPI 1.3.0b2 ↗	docs	GitHub 1.2.0 ↗ GitHub 1.3.0b2 ↗
OpenTelemetry Exporter	PyPI 1.0.0b21 ↗	docs	GitHub 1.0.0b21 ↗
Personalizer	PyPI 1.0.0b1 ↗		GitHub 1.0.0b1 ↗
Purview Account	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Purview Catalog	PyPI 1.0.0b4 ↗	docs	GitHub 1.0.0b4 ↗
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 1.2.0 ↗ PyPI 1.3.0b3 ↗	docs	GitHub 1.2.0 ↗ GitHub 1.3.0b3 ↗
Schema Registry - Avro	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Service Bus	PyPI 7.11.4 ↗	docs	GitHub 7.11.4 ↗
Spark	PyPI 0.7.0 ↗	docs	GitHub 0.7.0 ↗
Storage - Blobs	PyPI 12.19.0 ↗	docs	GitHub 12.19.0 ↗
Storage - Blobs Changefeed	PyPI 12.0.0b4 ↗	docs	GitHub 12.0.0b4 ↗

Name	Package	Docs	Source
Storage - Files Data Lake	PyPI 12.14.0	docs	GitHub 12.14.0
Storage - Files Share	PyPI 12.15.0	docs	GitHub 12.15.0
Storage - Queues	PyPI 12.9.0	docs	GitHub 12.9.0
Synapse - AccessControl	PyPI 0.7.0	docs	GitHub 0.7.0
Synapse - Artifacts	PyPI 0.18.0	docs	GitHub 0.18.0
Synapse - Monitoring	PyPI 0.2.0	docs	GitHub 0.2.0
Tables	PyPI 12.5.0	docs	GitHub 12.5.0
Text Analytics	PyPI 5.3.0	docs	GitHub 5.3.0
Text Translation	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Video Analyzer Edge	PyPI 1.0.0b4	docs	GitHub 1.0.0b4
Web PubSub	PyPI 1.0.1 PyPI 1.1.0b1	docs	GitHub 1.0.1 GitHub 1.1.0b1
Core - Management - Core	PyPI 1.4.0	docs	GitHub 1.4.0
Resource Management - Dev Center	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
Resource Management - Elastic SAN	PyPI 1.0.0b3	docs	GitHub 1.0.0b3
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 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - API Management	PyPI 4.0.0	docs	GitHub 4.0.0
Resource Management - App Compliance Automation	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - App Configuration	PyPI 3.0.0	docs	GitHub 3.0.0

Name	Package	Docs	Source
Resource Management - App Platform	PyPI 8.0.0	docs	GitHub 8.0.0
Resource Management - App Service	PyPI 7.2.0	docs	GitHub 7.2.0
Resource Management - Application Insights	PyPI 4.0.0	docs	GitHub 4.0.0
Resource Management - Arc Data	PyPI 1.0.0 PyPI 2.0.0b1	docs	GitHub 1.0.0 GitHub 2.0.0b1
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 - Automange	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 1.1.0b3	docs	GitHub 1.0.0 GitHub 1.1.0b3
Resource Management - Azure AI Search	PyPI 9.1.0	docs	GitHub 9.1.0
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 8.0.0b3	docs	GitHub 7.0.0 GitHub 8.0.0b3
Resource Management - Azure VMware Solution	PyPI 8.0.0	docs	GitHub 8.0.0
Resource Management - Backup	PyPI 8.0.0	docs	GitHub 8.0.0
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 17.2.0	docs	GitHub 17.2.0
Resource Management - Billing	PyPI 6.0.0 PyPI 6.1.0b1	docs	GitHub 6.0.0 GitHub 6.1.0b1
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 - Change Analysis	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Chaos	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Cognitive Services	PyPI 13.5.0	docs	GitHub 13.5.0

Name	Package	Docs	Source
Resource Management - Commerce	PyPI 6.0.0 ↗ PyPI 6.1.0b1 ↗	docs	GitHub 6.0.0 ↗ GitHub 6.1.0b1 ↗
Resource Management - Communication	PyPI 2.0.0 ↗ PyPI 2.1.0b2 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b2 ↗
Resource Management - Compute	PyPI 30.4.0 ↗	docs	GitHub 30.4.0 ↗
Resource Management - Confidential Ledger	PyPI 1.0.0 ↗ PyPI 2.0.0b3 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b3 ↗
Resource Management - Confluent	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
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 3.0.0 ↗	docs	GitHub 3.0.0 ↗
Resource Management - Container Instances	PyPI 10.1.0 ↗	docs	GitHub 10.1.0 ↗
Resource Management - Container Registry	PyPI 10.3.0 ↗	docs	GitHub 10.3.0 ↗
Resource Management - Container Service	PyPI 28.0.0 ↗	docs	GitHub 28.0.0 ↗
Resource Management - Container Service Fleet	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Content Delivery Network	PyPI 13.0.0 ↗	docs	GitHub 13.0.0 ↗
Resource Management - Cosmos DB	PyPI 9.4.0 ↗ PyPI 10.0.0b1 ↗	docs	GitHub 9.4.0 ↗ GitHub 10.0.0b1 ↗
Resource Management - Cosmos DB for PostgreSQL	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
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 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Data Box Edge	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Data Factory	PyPI 4.0.0 ↗	docs	GitHub 4.0.0 ↗
Resource Management - Data Lake Analytics	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗

Name	Package	Docs	Source
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 1.3.0 ↗	docs	GitHub 1.3.0 ↗
Resource Management - Data Share	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
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 - Deployment Manager	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Desktop Virtualization	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
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 Update	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - DevTest Labs	PyPI 9.0.0 ↗ PyPI 10.0.0b1 ↗	docs	GitHub 9.0.0 ↗ GitHub 10.0.0b1 ↗
Resource Management - Digital Twins	PyPI 6.4.0 ↗	docs	GitHub 6.4.0 ↗
Resource Management - DNS	PyPI 8.1.0 ↗	docs	GitHub 8.1.0 ↗
Resource Management - DNS Resolver	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Dynatrace	PyPI 2.0.0 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Edge Order	PyPI 1.0.0 ↗ PyPI 2.0.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 2.0.0b1 ↗
Resource Management - Education	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
Resource Management - Elastic	PyPI 1.0.0 ↗ PyPI 1.1.0b3 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b3 ↗
Resource Management - Event Grid	PyPI 10.2.0 ↗ PyPI 10.3.0b3 ↗	docs	GitHub 10.2.0 ↗ GitHub 10.3.0b3 ↗

Name	Package	Docs	Source
Resource Management - Event Hubs	PyPI 11.0.0	docs	GitHub 11.0.0
Resource Management - Extended Location	PyPI 1.1.0 PyPI 1.2.0b1	docs	GitHub 1.1.0 GitHub 1.2.0b1
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 1.1.0	docs	GitHub 1.1.0
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 - HDInsight	PyPI 9.0.0	docs	GitHub 9.0.0
Resource Management - HDInsight Containers	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - Health Bot	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Healthcare APIs	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Hybrid Compute	PyPI 8.0.0 PyPI 9.0.0b1	docs	GitHub 8.0.0 GitHub 9.0.0b1
Resource Management - Hybrid Connectivity	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Hybrid Container Service	PyPI 1.0.0b2	docs	GitHub 1.0.0b2
Resource Management - Hybrid Kubernetes	PyPI 1.1.0 PyPI 1.2.0b1	docs	GitHub 1.1.0 GitHub 1.2.0b1
Resource Management - Hybrid Network	PyPI 2.0.0	docs	GitHub 2.0.0
Resource Management - Image Builder	PyPI 1.3.0	docs	GitHub 1.3.0
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 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - IoT Hub	PyPI 3.0.0	docs	GitHub 3.0.0
Resource Management - Key Vault	PyPI 10.3.0	docs	GitHub 10.3.0
Resource Management - Kubernetes Configuration	PyPI 3.1.0	docs	GitHub 3.1.0

Name	Package	Docs	Source
Resource Management - Kusto	PyPI 3.3.0	docs	GitHub 3.3.0
Resource Management - Lab Services	PyPI 2.0.0 PyPI 2.1.0b1	docs	GitHub 2.0.0 GitHub 2.1.0b1
Resource Management - Load Testing	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - Log Analytics	PyPI 12.0.0 PyPI 13.0.0b6	docs	GitHub 12.0.0 GitHub 13.0.0b6
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 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
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 2.2.0b1	docs	GitHub 2.1.0 GitHub 2.2.0b1
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 7.0.0b1	docs	GitHub 6.0.0 GitHub 7.0.0b1
Resource Management - Management Groups	PyPI 1.0.0 PyPI 1.1.0b1	docs	GitHub 1.0.0 GitHub 1.1.0b1
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 1.2.0b1	docs	GitHub 1.1.0 GitHub 1.2.0b1
Resource Management - Media Services	PyPI 10.2.0	docs	GitHub 10.2.0

Name	Package	Docs	Source
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 3.1.0 ↗	docs	GitHub 3.1.0 ↗
Resource Management - Monitor	PyPI 6.0.2 ↗	docs	GitHub 6.0.2 ↗
Resource Management - NetApp Files	PyPI 11.0.0 ↗ PyPI 12.0.0b1 ↗	docs	GitHub 11.0.0 ↗ GitHub 12.0.0b1 ↗
Resource Management - Network	PyPI 25.2.0 ↗	docs	GitHub 25.2.0 ↗
Resource Management - Network Function	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - NetworkAnalytics	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Networkcloud	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - New Relic Observability	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Nginx	PyPI 3.0.0 ↗	docs	GitHub 3.0.0 ↗
Resource Management - Notification Hubs	PyPI 8.0.0 ↗ PyPI 8.1.0b1 ↗	docs	GitHub 8.0.0 ↗ GitHub 8.1.0b1 ↗
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 - 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 - Playwright Testing	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
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 - PostgreSQL	PyPI 10.1.0 ↗ PyPI 10.2.0b13 ↗	docs	GitHub 10.1.0 ↗ GitHub 10.2.0b13 ↗

Name	Package	Docs	Source
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 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Purview	PyPI 1.0.0 ↗ PyPI 1.1.0b1 ↗	docs	GitHub 1.0.0 ↗ GitHub 1.1.0b1 ↗
Resource Management - Quantum	PyPI 1.0.0b4 ↗	docs	GitHub 1.0.0b4 ↗
Resource Management - Qumulo	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
Resource Management - Quota	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Recovery Services	PyPI 2.5.0 ↗	docs	GitHub 2.5.0 ↗
Resource Management - Recoveryservicesdatareplication	PyPI 1.0.0b1 ↗	docs	GitHub 1.0.0b1 ↗
Resource Management - Red Hat OpenShift	PyPI 1.4.0 ↗	docs	GitHub 1.4.0 ↗
Resource Management - Redis	PyPI 14.3.0 ↗	docs	GitHub 14.3.0 ↗
Resource Management - Redis Enterprise	PyPI 2.0.0 ↗ PyPI 2.1.0b2 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b2 ↗
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 1.0.0b5 ↗	docs	GitHub 1.0.0b5 ↗
Resource Management - Resource Mover	PyPI 1.1.0 ↗	docs	GitHub 1.1.0 ↗
Resource Management - Resources	PyPI 23.0.1 ↗ PyPI 23.1.0b2 ↗	docs	GitHub 23.0.1 ↗ GitHub 23.1.0b2 ↗
Resource Management - Resources	PyPI 23.0.1 ↗ PyPI 23.1.0b2 ↗	docs	GitHub 23.0.1 ↗ GitHub 23.1.0b2 ↗
Resource Management - Scheduler	PyPI 2.0.0 ↗ PyPI 7.0.0b1 ↗	docs	GitHub 2.0.0 ↗
Resource Management - Scvmm	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗

Name	Package	Docs	Source
Resource Management - Security	PyPI 5.0.0	docs	GitHub 5.0.0
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 2.0.0b2	docs	GitHub 1.0.0 GitHub 2.0.0b2
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 8.2.0	docs	GitHub 8.2.0
Resource Management - Service Fabric	PyPI 2.1.0	docs	GitHub 2.1.0
Resource Management - Service Fabric Managed Clusters	PyPI 1.0.0 PyPI 2.0.0b5	docs	GitHub 1.0.0 GitHub 2.0.0b5
Resource Management - Service Linker	PyPI 1.1.0 PyPI 1.2.0b1	docs	GitHub 1.1.0 GitHub 1.2.0b1
Resource Management - Service Networking	PyPI 1.0.0	docs	GitHub 1.0.0
Resource Management - SignalR	PyPI 1.2.0 PyPI 2.0.0b2	docs	GitHub 1.2.0 GitHub 2.0.0b2
Resource Management - Site Recovery	PyPI 1.1.0	docs	GitHub 1.1.0
Resource Management - Sphere	PyPI 1.0.0b1	docs	GitHub 1.0.0b1
Resource Management - SQL	PyPI 3.0.1 PyPI 4.0.0b15	docs	GitHub 3.0.1 GitHub 4.0.0b15
Resource Management - SQL Virtual Machine	PyPI 1.0.0b6	docs	GitHub 1.0.0b6
Resource Management - Storage	PyPI 21.1.0	docs	GitHub 21.1.0
Resource Management - Storage Cache	PyPI 1.5.0	docs	GitHub 1.5.0
Resource Management - Storage Mover	PyPI 2.0.0	docs	GitHub 2.0.0
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 2.0.0b1	docs	GitHub 1.0.0 GitHub 2.0.0b1

Name	Package	Docs	Source
Resource Management - Subscriptions	PyPI 3.1.1 ↗ PyPI 3.2.0b1 ↗	docs	GitHub 3.1.1 ↗ GitHub 3.2.0b1 ↗
Resource Management - Support	PyPI 6.0.0 ↗ PyPI 6.1.0b2 ↗	docs	GitHub 6.0.0 ↗ GitHub 6.1.0b2 ↗
Resource Management - Synapse	PyPI 2.0.0 ↗ PyPI 2.1.0b7 ↗	docs	GitHub 2.0.0 ↗ GitHub 2.1.0b7 ↗
Resource Management - Test Base	PyPI 1.0.0b2 ↗	docs	GitHub 1.0.0b2 ↗
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 - 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 1.1.0 ↗ PyPI 2.0.0b2 ↗	docs	GitHub 1.1.0 ↗ GitHub 2.0.0b2 ↗
Resource Management - Workloads	PyPI 1.0.0 ↗	docs	GitHub 1.0.0 ↗
azure-communication-administration	PyPI 1.0.0b4 ↗		
Unknown Display Name	PyPI 1.0.0 ↗		
Auto Suggest	PyPI 0.2.0 ↗		GitHub 0.2.0 ↗
azureml-fsspec	PyPI 1.0.0 ↗		
Batch	PyPI 14.1.0 ↗	docs	GitHub 14.1.0 ↗
Computer Vision	PyPI 0.9.0 ↗	docs	GitHub 0.9.0 ↗
Content Safety	PyPI 1.0.0 ↗		GitHub 1.0.0 ↗
Conversational Language Understanding	PyPI 0.7.0 ↗	docs	GitHub 0.7.0 ↗
Core - Client - Tracing Opencensus	PyPI 1.0.0b9 ↗	docs	GitHub 1.0.0b9 ↗
Custom Image Search	PyPI 0.2.0 ↗		GitHub 0.2.0 ↗
Custom Search	PyPI 0.3.0 ↗		GitHub 0.3.0 ↗
Custom Vision	PyPI 3.1.0 ↗	docs	GitHub 3.1.0 ↗
Device Provisioning Services	PyPI 1.0.0b1 ↗		

Name	Package	Docs	Source
Device Provisioning Services	PyPI 1.2.0		
Entity Search	PyPI 2.0.0		GitHub 2.0.0
Face	PyPI 0.6.0	docs	GitHub 0.6.0
Image Search	PyPI 2.0.0		GitHub 2.0.0
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		
IoT Models Repository	PyPI 1.0.0b1		GitHub 1.0.0b1
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		
News Search	PyPI 2.0.0		GitHub 2.0.0
Personalizer	PyPI 0.1.0		GitHub 0.1.0
Purview Administration	PyPI 1.0.0b1		GitHub 1.0.0b1
Question Answering	PyPI 0.3.0	docs	GitHub 0.3.0
Service Fabric	PyPI 8.2.0.0	docs	GitHub 8.2.0.0
Speech	PyPI 1.14.0		
Spell Check	PyPI 2.0.0		GitHub 2.0.0
Storage	PyPI 0.37.0		GitHub 0.37.0
Storage - Files Data Lake	PyPI 0.0.51		
Synapse	PyPI 0.1.1	docs	GitHub 0.1.1
Text Analytics	PyPI 1.0.2		

Name	Package	Docs	Source
Uamqp	PyPI 1.6.6 ↗		GitHub 1.6.6 ↗
Video Search	PyPI 2.0.0 ↗		GitHub 2.0.0 ↗
Visual Search	PyPI 0.2.0 ↗		GitHub 0.2.0 ↗
Web PubSub - Client	PyPI 1.0.0b1 ↗		
Web Search	PyPI 2.0.0 ↗		GitHub 2.0.0 ↗
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 ↗		
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 ↗

Name	Package	Docs	Source
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 ↗		
Service Management - Legacy	PyPI 0.20.7 ↗		GitHub 0.20.7 ↗
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 ↗
Unknown Display Name	PyPI 0.3.7 ↗		
Unknown Display Name	PyPI 0.0.8 ↗		

Reference

Reference

Services

Advisor	Container Service Fleet	Grafana
Alerts Management	Content Delivery Network	Graph Services
API Center	Cosmos DB	HANA on Azure
App Configuration	Cosmos DB for PostgreSQL	HDInsight
App Platform	Cost Management	Healthcare APIs
App Service	Custom Providers	Hybrid Compute
Application Insights	Data Box	Hybrid Connectivity
Arc Data	Data Box Edge	Hybrid Container Service
Attestation	Data Explorer	Hybrid Kubernetes
Authorization	Data Factory	Hybrid Network
Automanage	Data Protection	Identity
Automation	Data Share	Image Builder
Azure Stack	Database Migration Service	Image Search
Azure Stack HCI	Databricks	IoT
Azure VMware Solution	Datadog	Key Vault
BareMetal Infrastructure	Deployment Manager	Kubernetes Configuration
Batch	Desktop Virtualization	Lab Services
Billing	Dev Center	Load Testing
Bot Service	DevTest Labs	Log Analytics
Change Analysis	DNS	Logic Apps
Chaos	DNS Resolver	Logz
Cognitive Services	Dynatrace	Machine Learning
Commerce	Edge Order	Maintenance
Communication	Elastic	Managed Network Fabric
Compute	Elastic SAN	Managed Service Identity
Confidential Ledger	Entity Search	Managed Services
Confluent	Event Grid	Management Groups
Connected VMware	Event Hubs	Management Partner
Consumption	Extended Location	Maps
Container Apps	Fluid Relay	Marketplace Ordering
Container Instances	Front Door	Media Services
Container Registry	Functions	Metrics Advisor
Container Service	Functions	Mixed Reality

Mobile Network	Private DNS	Service Fabric
Monitor	Purview	Service Linker
NetApp Files	Qumulo	Service Networking
Network	Recovery Services	Sphere
Network Analytics	Red Hat OpenShift (ARO)	SQL
New Relic Observability	Redis	Storage
News Search	Relay	Stream Analytics
Nginx	Resource Connector	Subscriptions
Notification Hubs	Resource Graph	Support
Operations Management	Resource Mover	Synapse
Operator Nexus - Network	Resources	Tables
Cloud	Scheduler	Traffic Manager
Orbital	Schema Registry	unknown
Palo Alto Networks	Search	Video Search
Peering	Security	Voice Services
Policy Insights	Security Insights	Web PubSub
Portal	Self Help	Web Search
PostgreSQL	Serial Console	Workloads
Power BI Dedicated	Service Bus	Other

Hosting Python apps on Azure

Article • 01/23/2024

Azure provides various different ways to host your app depending on your needs. The article [Hosting applications on Azure](#) provides an overview of the different options.

Generally speaking, choosing an Azure hosting option is a matter of choosing on the continuum of control versus responsibility. The more control you need, the more responsibility you take on for management of the resource(s). In this continuum, we recommend starting with Azure App Service, with the least administrative responsibility on your part. Then, consider other options in the continuum moving toward taking more administrative responsibility of your Azure resources. At the other end of the continuum from App Service is Azure Virtual Machines, where you have the most control and more administrative responsibility for maintaining your resources.

The sections in this article are arranged approximately from more managed options (less management overhead for you) to less managed options (more control for you).

- **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](#)
 - [Azure Virtual Machines Management Samples - Python](#)

Data solutions for Python apps on Azure

Article • 03/14/2024

Azure offers a choice of fully managed relational, NoSQL, and in-memory databases, spanning proprietary and open-source engines in addition to storage services for object, block, and file storage. The following articles help you get started with Python data solutions on Azure.

Databases

- **PostgreSQL:** Build scalable, secure, and fully managed enterprise-ready apps on open-source PostgreSQL, scale out single-node PostgreSQL with high performance, or migrate 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 apps that scale with managed and intelligent SQL 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 apps that scale with managed and intelligent SQL database 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 applications with guaranteed low latency and high availability anywhere, at any 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:** Massively scalable and secure object storage for cloud-native workloads, archives, data lakes, high-performance computing, and machine learning.
 - [Quickstart: Azure Blob Storage client library for Python](#)
 - [Azure Storage samples using v12 Python client libraries](#)
- **Azure Data Lake Storage Gen2:** Massively scalable and secure data lake for your high-performance analytics workloads.
 - [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:** Power fast, scalable applications with an open-source-compatible in-memory data store.
 - [Quickstart: Use Azure Cache for Redis in Python](#)

Big data and analytics

- **Azure Data Lake analytics:** A fully managed on-demand pay-per-job analytics service with 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 data integration service to orchestrate and automate data movement and transformation.
 - [Quickstart: Create a data factory and pipeline using Python](#)
 - [Transform data by running a Python activity in Azure Databricks](#)
- **Azure Event Hubs:** A hyper-scale telemetry ingestion service that collects, transforms, and stores millions of events.
 - [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 Hadoop and Spark service backed by 99.9% SLA for your enterprise
 - [Use Spark & Hive Tools for Visual Studio Code](#)

- **Azure Databricks:** A fully managed, fast, easy and collaborative Apache® Spark™ based analytics platform optimized for 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 limitless analytics service that brings together enterprise data warehousing and big data analytics.
 - [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

Article • 03/11/2024

Identity and access management for Python apps on Azure are fundamentally about the *authentication* of the identity of a user, group, application, or service and *authorization* of that identity to perform requested actions on Azure resources. There are different identity and access management options you can choose from depending on your application and security needs. This article provides links to resources to help you get started.

For an overview of authentication and authorization in Azure, see [Recommendations for identity and access management](#).

Passwordless connections

Whenever possible, we recommend you use managed identities to simplify overall management and improve security. Specifically, use *passwordless connections* to avoid using embedding sensitive data such as passwords in code or environment variables.

- [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](#)

The resources listed show how to use Azure Python SDK and passwordless connections with the [DefaultAzureCredential](#). The `DefaultAzureCredential` is appropriate for most applications that will run in Azure because it combines common production credentials with development credentials.

Service Connector

Many Azure resources you're likely to use with your Python apps enable the [Service Connector](#) service. Service Connector helps you configure network settings and connection information between Azure services such as App Service and Container Apps and other services such as storage or databases.

- [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 like [Azure Key Vault](#) gives you more control but with an increase in 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 build Python applications that enable your users and customers to sign in using their Microsoft identities or social accounts. Your app authorizes access to your own APIs or Microsoft APIs like Microsoft Graph.

- [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

Article • 03/14/2024

The following articles help you get started with Azure Machine Learning. Azure Machine Learning v2 REST APIs, Azure CLI extension, and Python SDK accelerate the production machine learning lifecycle. The links in this article target v2, which is recommended if you're starting a new machine learning project.

Getting started

The workspace is the top-level resource for Azure Machine Learning, providing a centralized place to work with all the artifacts you create when you use Azure Machine Learning.

- [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 machine learning models for real-time inference.

- [Tutorial: Designer - deploy a machine learning model](#)
- [Deploy and score a machine learning model by using an online endpoint](#)

Automated machine learning

Automated machine learning, also referred to as automated ML or AutoML, is the process of automating the time-consuming, iterative tasks of machine learning model development.

- [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 bring data from a local machine or an existing cloud-based storage.

- [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 create a workflow that stitches together various ML phases.

- [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

Article • 03/08/2024

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 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

Article • 03/11/2024

The following articles help you get started with messaging, event ingestion and processing, and Internet of Things (IoT) services in Azure.

Messaging

Messaging services on Azure provide the interconnectivity between components and applications that are written in different languages and hosted in the same cloud, 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

Event Hubs is a big data streaming platform and event ingestion service. Event Grid is a scalable, serverless event broker that you can use to integrate applications using events.

- **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 or IoT refers to a collection of managed and platform services across edge and cloud that connect, monitor, and control IoT assets. IoT also includes security and operating systems for devices and data and analytics that help you build, deploy, and manage IoT applications.

- **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

Article • 03/14/2024

The services referenced in this article for Python are specialized or focused on solving a targeted problem. By other services we mean services other than the core services compute, networking, storage, and database. The references in this article include management and governance, media, genomics, and Internet of Things services. For a complete list of services, see [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:**
 - [Sample: Create and manage Windows VMs in Azure using Python](#)
 - [Example: Use the Azure libraries to create a virtual machine](#)