

Azure Container Apps documentation

Azure Container Apps allows you to run containerized applications without worrying about orchestration or infrastructure.

About Azure Container Apps

OVERVIEW

[What is Azure Container Apps?](#)

[Compare container options in Azure](#)

Get started

QUICKSTART

[Deploy your first container app - Azure CLI](#)

[Deploy your first container app - Azure portal](#)

CONCEPT

[Environments](#)

[Containers](#)

[Revisions](#)

[Application lifecycle management](#)

[Microservices](#)

[Observability](#)

[Jobs](#)

Common tasks

HOW-TO GUIDE

[Set scaling rules](#)

[Manage secrets](#)

[Manage revisions](#)

[Set up ingress](#)

[Connect multiple apps](#)

[Use a custom VNET](#)

[Quotas](#)

 CONCEPT

[Dapr integration](#)

Apps and microservices

 TUTORIAL

[Microservices with Dapr](#)

[Background processing](#)

Azure Container Apps overview

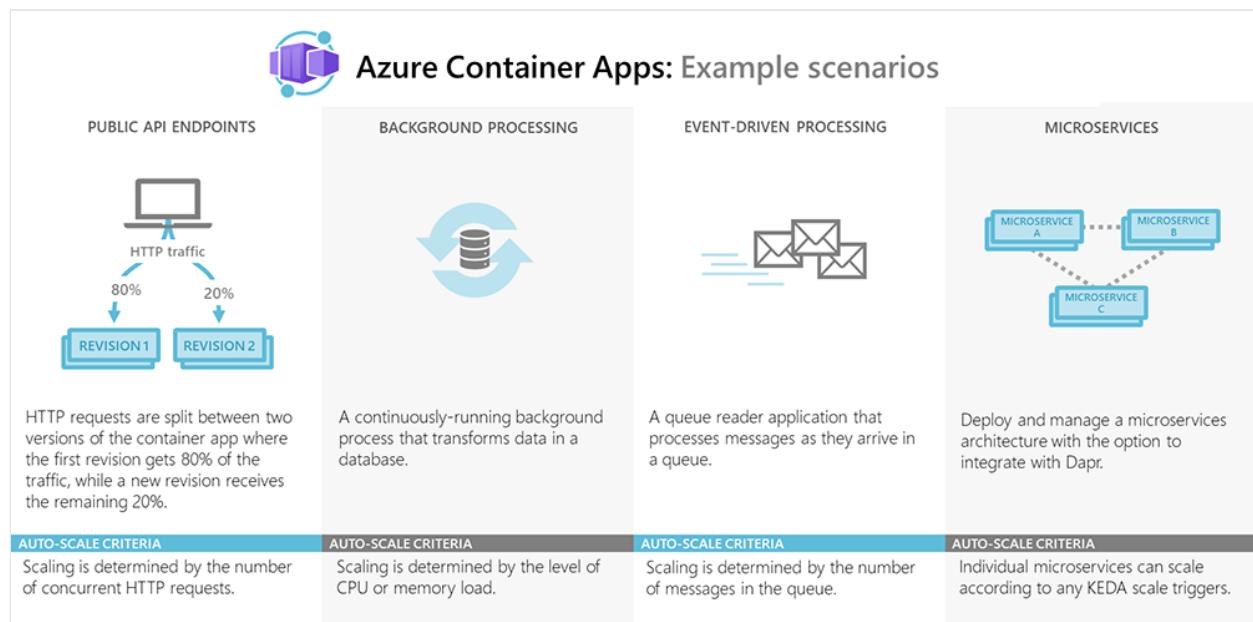
Article • 05/23/2023

Azure Container Apps is a fully managed environment that enables you to run microservices and containerized applications on a serverless platform. Common uses of Azure Container Apps include:

- Deploying API endpoints
- Hosting background processing applications
- Handling event-driven processing
- Running microservices

Applications built on Azure Container Apps can dynamically scale based on the following characteristics:

- HTTP traffic
- Event-driven processing
- CPU or memory load
- Any [KEDA-supported scaler](#)



Azure Container Apps enables executing application code packaged in any container and is unopinionated about runtime or programming model. With Container Apps, you enjoy the benefits of running containers while leaving behind the concerns of managing cloud infrastructure and complex container orchestrators.

Features

With Azure Container Apps, you can:

- **Use the Azure CLI extension, Azure portal or ARM templates** to manage your applications.
- **Enable HTTPS or TCP ingress** without having to manage other Azure infrastructure.
- **Build microservices with Dapr** and access its rich set of APIs.
- Add [Azure Functions](#) and [Azure Spring Apps](#) to your Azure Container Apps environment.
- **Use specialized hardware** for access to increased compute resources.
- **Run multiple container revisions** and manage the container app's application lifecycle.
- **Autoscale** your apps based on any KEDA-supported scale trigger. Most applications can scale to zero¹.
- **Split traffic** across multiple versions of an application for Blue/Green deployments and A/B testing scenarios.
- **Use internal ingress and service discovery** for secure internal-only endpoints with built-in DNS-based service discovery.
- **Run containers from any registry**, public or private, including Docker Hub and Azure Container Registry (ACR).
- **Provide an existing virtual network** when creating an environment for your container apps.
- **Securely manage secrets** directly in your application.
- **Monitor logs** using Azure Log Analytics.
- **Generous quotas** which can be overridden to increase limits on a per-account basis.

¹ Applications that [scale on CPU or memory load](#) can't scale to zero.

Introductory video

<https://www.youtube-nocookie.com/embed/b3dopSTnSRg>

Next steps

[Deploy your first container app](#)

Comparing Container Apps with other Azure container options

Article • 11/23/2022

There are many options for teams to build and deploy cloud native and containerized applications on Azure. This article will help you understand which scenarios and use cases are best suited for Azure Container Apps and how it compares to other container options on Azure including:

- [Azure Container Apps](#)
- [Azure App Service](#)
- [Azure Container Instances](#)
- [Azure Kubernetes Service](#)
- [Azure Functions](#)
- [Azure Spring Apps](#)
- [Azure Red Hat OpenShift](#)

There's no perfect solution for every use case and every team. The following explanation provides general guidance and recommendations as a starting point to help find the best fit for your team and your requirements.

Container option comparisons

Azure Container Apps

Azure Container Apps enables you to build serverless microservices based on containers. Distinctive features of Container Apps include:

- Optimized for running general purpose containers, especially for applications that span many microservices deployed in containers.
- Powered by Kubernetes and open-source technologies like [Dapr](#), [KEDA](#), and [envoy](#).
- Supports Kubernetes-style apps and microservices with features like [service discovery](#) and [traffic splitting](#).
- Enables event-driven application architectures by supporting scale based on traffic and pulling from [event sources like queues](#), including [scale to zero](#).
- Support of long running processes and can run [background tasks](#).

Azure Container Apps doesn't provide direct access to the underlying Kubernetes APIs. If you require access to the Kubernetes APIs and control plane, you should use [Azure Kubernetes Service](#). However, if you would like to build Kubernetes-style applications and don't require direct access to all the native Kubernetes APIs and cluster management, Container Apps provides a fully managed experience based on best-practices. For these reasons, many teams may prefer to start building container microservices with Azure Container Apps.

You can get started building your first container app [using the quickstarts](#).

Azure App Service

[Azure App Service](#) provides fully managed hosting for web applications including websites and web APIs. These web applications may be deployed using code or containers. Azure App Service is optimized for web applications. Azure App Service is integrated with other Azure services including Azure Container Apps or Azure Functions. When building web apps, Azure App Service is an ideal option.

Azure Container Instances

[Azure Container Instances \(ACI\)](#) provides a single pod of Hyper-V isolated containers on demand. It can be thought of as a lower-level "building block" option compared to Container Apps. Concepts like scale, load balancing, and certificates are not provided with ACI containers. For example, to scale to five container instances, you create five distinct container instances. Azure Container Apps provide many application-specific concepts on top of containers, including certificates, revisions, scale, and environments. Users often interact with Azure Container Instances through other services. For example, Azure Kubernetes Service can layer orchestration and scale on top of ACI through [virtual nodes](#). If you need a less "opinionated" building block that doesn't align with the scenarios Azure Container Apps is optimizing for, Azure Container Instances is an ideal option.

Azure Kubernetes Service

[Azure Kubernetes Service \(AKS\)](#) provides a fully managed Kubernetes option in Azure. It 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. Teams looking for a fully managed version of Kubernetes in Azure, Azure Kubernetes Service is an ideal option.

Azure Functions

[Azure Functions](#) is a serverless Functions-as-a-Service (FaaS) solution. It's optimized for running event-driven applications using the functions programming model. It shares many characteristics with Azure Container Apps around scale and integration with events, but optimized for ephemeral functions deployed as either code or containers. The Azure Functions programming model provides productivity benefits for teams looking to trigger the execution of your functions on events and bind to other data sources. When building FaaS-style functions, Azure Functions is the ideal option. The Azure Functions programming model is available as a base container image, making it portable to other container based compute platforms allowing teams to reuse code as environment requirements change.

Azure Spring Apps

[Azure Spring Apps](#) is a fully managed service for Spring developers. If you want to run Spring Boot, Spring Cloud or any other Spring applications on Azure, Azure Spring Apps is an ideal option. The service manages the infrastructure of Spring applications so developers can focus on their code. Azure Spring Apps provides lifecycle management using comprehensive monitoring and diagnostics, configuration management, service discovery, CI/CD integration, blue-green deployments, and more.

Azure Red Hat OpenShift

[Azure Red Hat OpenShift](#) is jointly engineered, operated, and supported by Red Hat and Microsoft to provide an integrated product and support experience for running Kubernetes-powered OpenShift. With Azure Red Hat OpenShift, teams can choose their own registry, networking, storage, and CI/CD solutions, or use the built-in solutions for automated source code management, container and application builds, deployments, scaling, health management, and more from OpenShift. If your team or organization is using OpenShift, Azure Red Hat OpenShift is an ideal option.

Next steps

[Deploy your first container app](#)

Quickstart: Deploy your first container app using the Azure portal

Article • 04/14/2023

Azure Container Apps enables you to run microservices and containerized applications on a serverless platform. With Container Apps, you enjoy the benefits of running containers while leaving behind the concerns of manually configuring cloud infrastructure and complex container orchestrators.

In this quickstart, you create a secure Container Apps environment and deploy your first container app using the Azure portal.

Prerequisites

An Azure account with an active subscription is required. If you don't already have one, you can [create an account for free](#). Also, please make sure to have the Resource Provider "Microsoft.App" registered.

Setup

Begin by signing in to the [Azure portal](#).

Create a container app

To create your container app, start at the Azure portal home page.

1. Search for **Container Apps** in the top search bar.
2. Select **Container Apps** in the search results.
3. Select the **Create** button.

Basics tab

In the *Basics* tab, do the following actions.

1. Enter the following values in the *Project details* section.

Setting	Action
Subscription	Select your Azure subscription.

Setting	Action
Resource group	Select Create new and enter my-container-apps .
Container app name	Enter my-container-app .

Create an environment

Next, create an environment for your container app.

1. Select the appropriate region.

Setting	Value
Region	Select Central US .

2. In the *Create Container Apps environment* field, select the **Create new** link.

3. In the *Create Container Apps Environment* page on the *Basics* tab, enter the following values:

Setting	Value
Environment name	Enter my-environment .
Zone redundancy	Select Disabled

4. Select the **Monitoring** tab to create a Log Analytics workspace.

5. Select the **Create new** link in the *Log Analytics workspace* field and enter the following values.

Setting	Value
Name	Enter my-container-apps-logs .

The *Location* field is pre-filled with *Central US* for you.

6. Select **OK**.

7. Select the **Create** button at the bottom of the *Create Container App Environment* page.

Deploy the container app

1. Select the **Review and create** button at the bottom of the page.

Next, the settings in the Container App are verified. If no errors are found, the **Create** button is enabled.

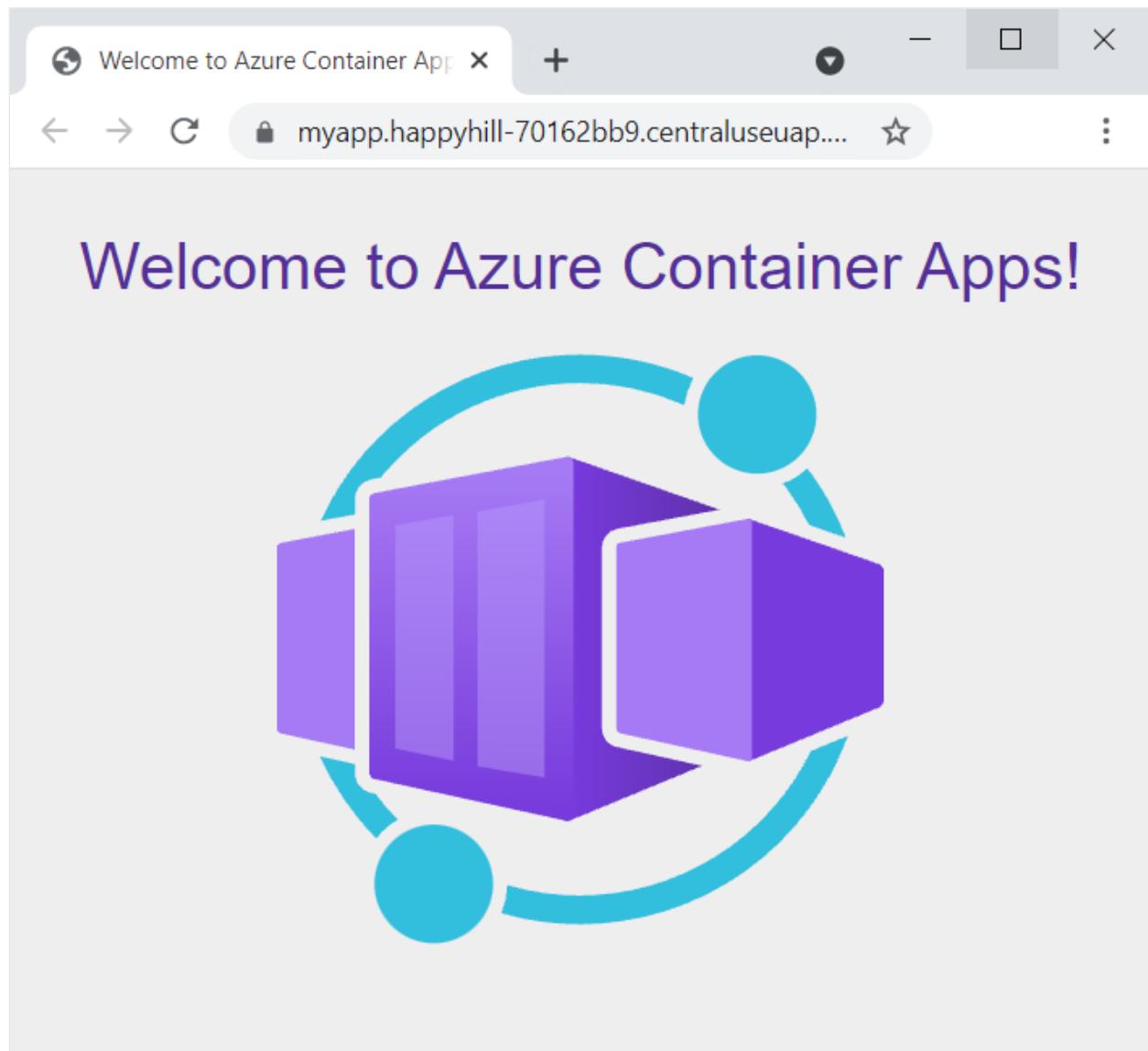
If there are errors, any tab containing errors is marked with a red dot. Navigate to the appropriate tab. Fields containing an error will be highlighted in red. Once all errors are fixed, select **Review and create** again.

2. Select **Create**.

A page with the message *Deployment is in progress* is displayed. Once the deployment is successfully completed, you'll see the message: *Your deployment is complete*.

Verify deployment

Select **Go to resource** to view your new container app. Select the link next to *Application URL* to view your application. You'll see the following message in your browser.



Clean up resources

If you're not going to continue to use this application, you can delete the Azure Container Apps instance and all the associated services by removing the resource group.

1. Select the **my-container-apps** resource group from the *Overview* section.
2. Select the **Delete resource group** button at the top of the resource group *Overview*.
3. Enter the resource group name **my-container-apps** in the *Are you sure you want to delete "my-container-apps"* confirmation dialog.
4. Select **Delete**.

The process to delete the resource group may take a few minutes to complete.

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#) ↗.

Next steps

[Communication between microservices](#)

Quickstart: Deploy your first container app with containerapp up

Article • 03/31/2023

The Azure Container Apps service enables you to run microservices and containerized applications on a serverless platform. With Container Apps, you enjoy the benefits of running containers while you leave behind the concerns of manually configuring cloud infrastructure and complex container orchestrators.

In this quickstart, you create and deploy your first container app using the `az containerapp up` command.

Prerequisites

- An Azure account with an active subscription.
 - If you don't have one, you [can create one for free](#).
- Install the [Azure CLI](#).

Setup

To sign in to Azure from the CLI, run the following command and follow the prompts to complete the authentication process.

```
Bash
Azure CLI
az login
```

Ensure you're running the latest version of the CLI via the upgrade command.

```
Bash
Azure CLI
az upgrade
```

Next, install or update the Azure Container Apps extension for the CLI.

Bash

Azure CLI

```
az extension add --name containerapp --upgrade
```

Register the `Microsoft.App` and `Microsoft.OperationalInsights` namespaces if you haven't already registered them in your Azure subscription.

Bash

Azure CLI

```
az provider register --namespace Microsoft.App
```

Azure CLI

```
az provider register --namespace Microsoft.OperationalInsights
```

Now that your Azure CLI setup is complete, you can define the environment variables that are used throughout this article.

Create and deploy the container app

Create and deploy your first container app with the `containerapp up` command. This command will:

- Create the resource group
- Create the Container Apps environment
- Create the Log Analytics workspace
- Create and deploy the container app using a public container image

Note that if any of these resources already exist, the command will use them instead of creating new ones.

Bash

Azure CLI

```
az containerapp up \
--name my-container-app \
--resource-group my-container-apps \
--location centralus \
--environment 'my-container-apps' \
--image mcr.microsoft.com/azuredocs/containerapps-helloworld:latest \
--target-port 80 \
--ingress external \
--query properties.configuration.ingress.fqdn
```

ⓘ Note

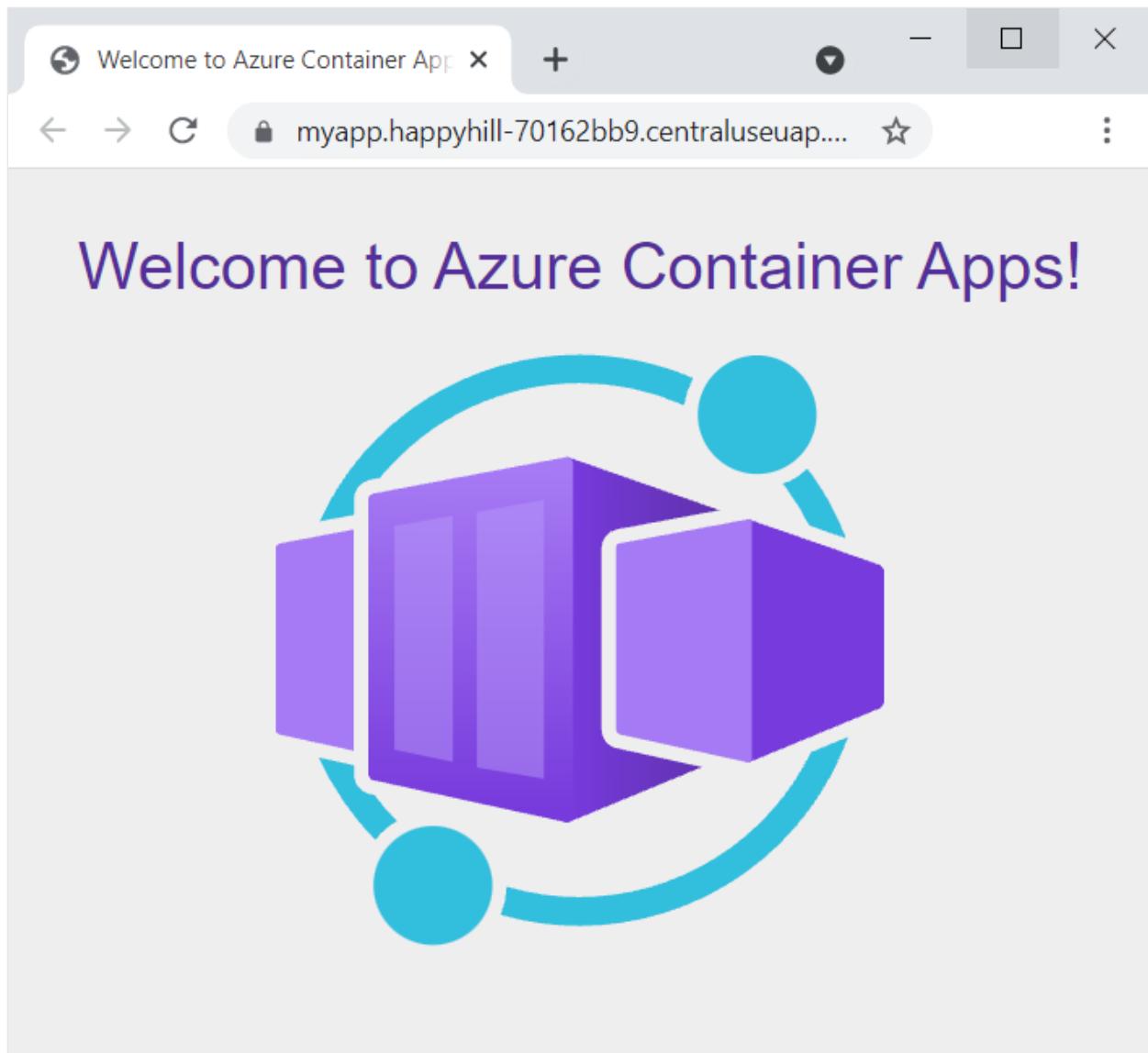
Make sure the value for the `--image` parameter is in lower case.

By setting `--ingress` to `external`, you make the container app available to public requests.

Verify deployment

The `up` command returns the fully qualified domain name for the container app. Copy this location to a web browser.

The following message is displayed when the container app is deployed:



Clean up resources

If you're not going to continue to use this application, run the following command to delete the resource group along with all the resources created in this quickstart.

⊗ Caution

The following command deletes the specified resource group and all resources contained within it. If resources outside the scope of this quickstart exist in the specified resource group, they will also be deleted.

Azure CLI

```
az group delete --name my-container-apps
```

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

[Communication between microservices](#)

Quickstart: Build and deploy your container app from a repository in Azure Container Apps

Article • 03/31/2023

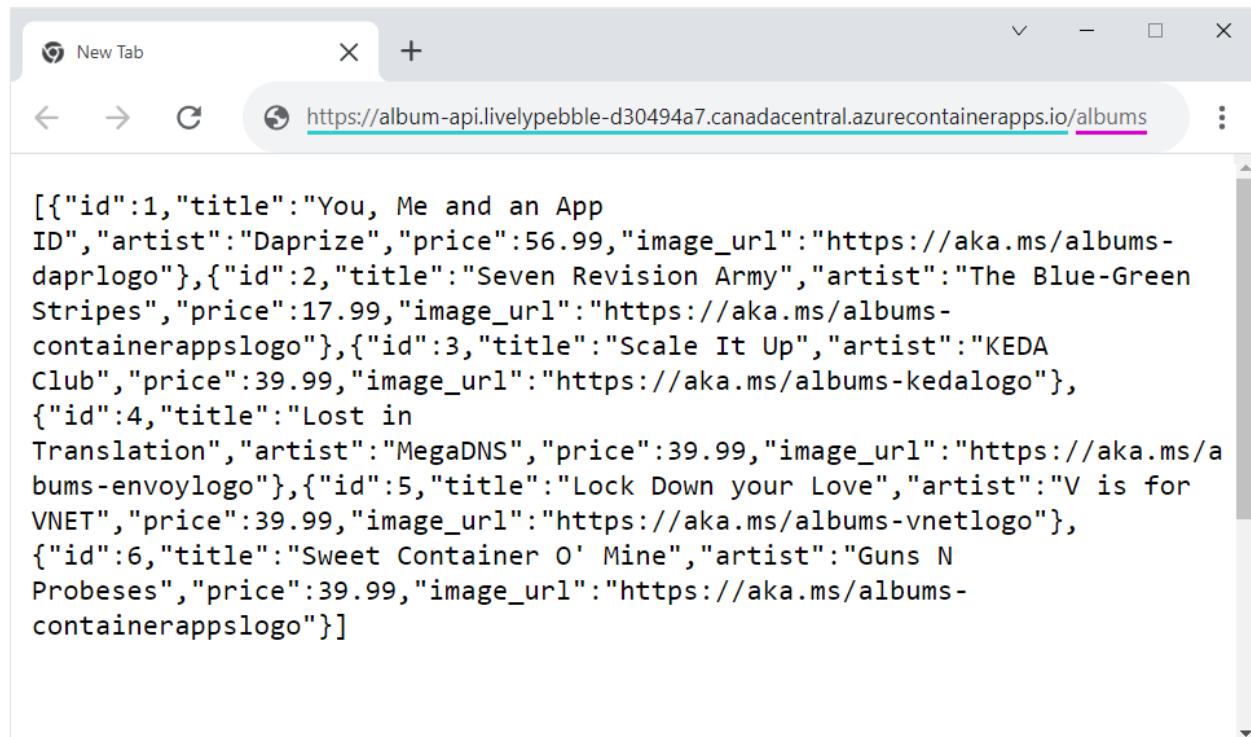
This article demonstrates how to build and deploy a microservice to Azure Container Apps from a source repository using the programming language of your choice.

In this quickstart, you create a backend web API service that returns a static collection of music albums. After completing this quickstart, you can continue to [Tutorial: Communication between microservices in Azure Container Apps](#) to learn how to deploy a front end application that calls the API.

ⓘ Note

You can also build and deploy this sample application using the `az containerapp up` command. For more information, see [Tutorial: Build and deploy your app to Azure Container Apps](#).

The following screenshot shows the output from the album API service you deploy.



Prerequisites

To complete this project, you need the following items:

Requirement	Instructions
Azure account	If you don't have one, create an account for free ↗ . You need the <i>Contributor</i> or <i>Owner</i> permission on the Azure subscription to proceed. Refer to Assign Azure roles using the Azure portal for details.
GitHub Account	Get one for free ↗ .
git	Install git ↗
Azure CLI	Install the Azure CLI .

Setup

To sign in to Azure from the CLI, run the following command and follow the prompts to complete the authentication process.

Bash

Azure CLI

```
az login
```

Ensure you're running the latest version of the CLI via the upgrade command.

Bash

Azure CLI

```
az upgrade
```

Next, install or update the Azure Container Apps extension for the CLI.

Bash

Azure CLI

```
az extension add --name containerapp --upgrade
```

Register the `Microsoft.App` and `Microsoft.OperationalInsights` namespaces if you haven't already registered them in your Azure subscription.

Bash

Azure CLI

```
az provider register --namespace Microsoft.App
```

Azure CLI

```
az provider register --namespace Microsoft.OperationalInsights
```

Now that your Azure CLI setup is complete, you can define the environment variables that are used throughout this article.

Bash

Define the following variables in your bash shell.

Azure CLI

```
RESOURCE_GROUP="album-containerapps"
LOCATION="canadacentral"
ENVIRONMENT="env-album-containerapps"
API_NAME="album-api"
```

Prepare the GitHub repository

In a browser window, go to the GitHub repository for your preferred language and fork the repository.

C#

Select the **Fork** button at the top of the [album API repo](#) to fork the repo to your account.

Build and deploy the container app

Build and deploy your first container app from your forked GitHub repository with the `containerapp up` command. This command will:

- Create the resource group
- Create an Azure Container Registry
- Build the container image and push it to the registry
- Create the Container Apps environment with a Log Analytics workspace
- Create and deploy the container app using a public container image
- Create a GitHub Action workflow to build and deploy the container app

The `up` command uses the Docker file in the root of the repository to build the container image. The target port is defined by the EXPOSE instruction in the Docker file. A Docker file isn't required to build a container app.

Replace the `<YOUR_GITHUB_REPOSITORY_NAME>` with your GitHub repository name in the form of `https://github.com/<owner>/<repository-name>` or `<owner>/<repository-name>`.

Bash

Azure CLI

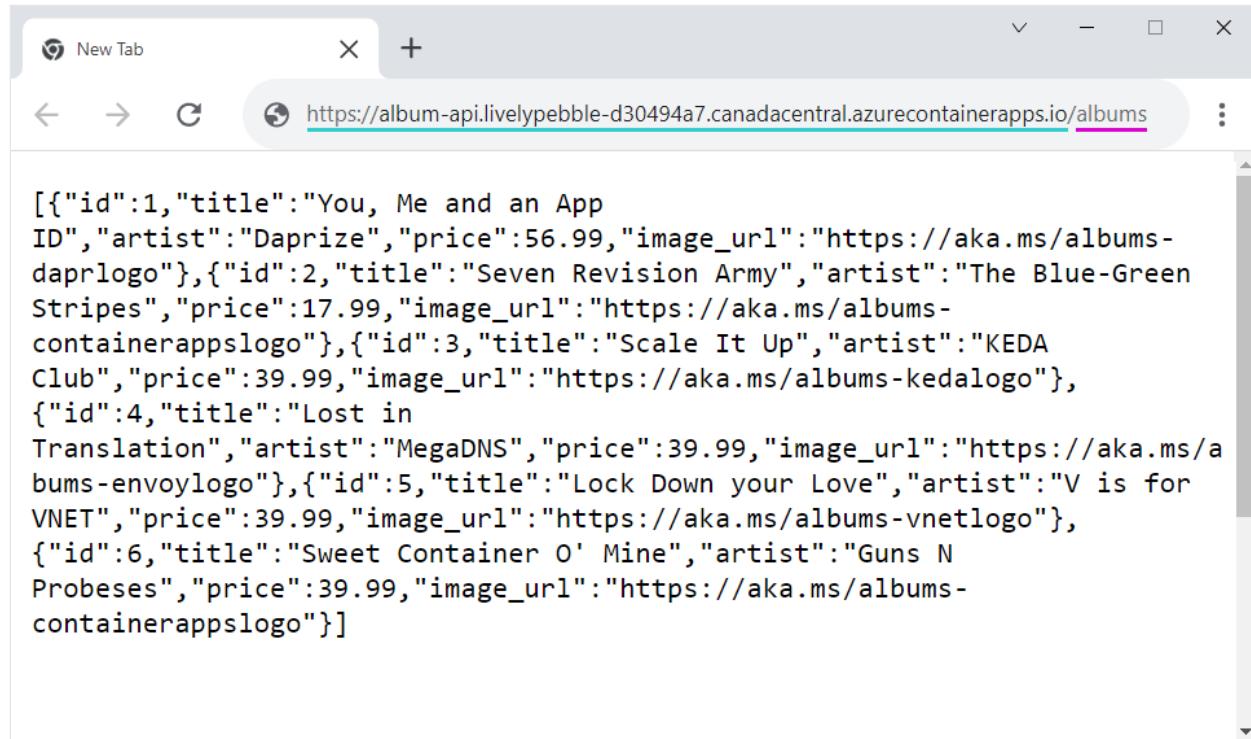
```
az containerapp up \
  --name $API_NAME \
  --resource-group $RESOURCE_GROUP \
  --location $LOCATION \
  --environment $ENVIRONMENT \
  --context-path ./src \
  --repo <YOUR_GITHUB_REPOSITORY_NAME>
```

Using the URL and the user code displayed in the terminal, go to the GitHub device activation page in a browser and enter the user code to the page. Follow the prompts to authorize the Azure CLI to access your GitHub repository.

The `up` command creates a GitHub Action workflow in your repository `.github/workflows` folder. The workflow is triggered to build and deploy your container app when you push changes to the repository.

Verify deployment

Copy the FQDN to a web browser. From your web browser, go to the `/albums` endpoint of the FQDN.



A screenshot of a Microsoft Edge browser window. The address bar shows the URL: `https://album-api.livelypebble-d30494a7.canadacentral.azurecontainerapps.io/albums`. The main content area displays a JSON array of album objects:

```
[{"id":1,"title":"You, Me and an App ID","artist":"Daprize","price":56.99,"image_url":"https://aka.ms/albums-daprllogo"}, {"id":2,"title":"Seven Revision Army","artist":"The Blue-Green Stripes","price":17.99,"image_url":"https://aka.ms/albums-containerappslogo"}, {"id":3,"title":"Scale It Up","artist":"KEDA Club","price":39.99,"image_url":"https://aka.ms/albums-kedalogo"}, {"id":4,"title":"Lost in Translation","artist":"MegaDNS","price":39.99,"image_url":"https://aka.ms/albums-envoylogo"}, {"id":5,"title":"Lock Down your Love","artist":"V is for VNET","price":39.99,"image_url":"https://aka.ms/albums-vnetlogo"}, {"id":6,"title":"Sweet Container O' Mine","artist":"Guns N Probeses","price":39.99,"image_url":"https://aka.ms/albums-containerappslogo"}]
```

Clean up resources

If you're not going to continue on to the [Deploy a frontend](#) tutorial, you can remove the Azure resources created during this quickstart with the following command.

⊗ Caution

The following command deletes the specified resource group and all resources contained within it. If the group contains resources outside the scope of this quickstart, they are also deleted.

Bash

Azure CLI

```
az group delete --name $RESOURCE_GROUP
```

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

[Tutorial: Communication between microservices](#)

Tutorial: Deploy to Azure Container Apps using Visual Studio

Article • 02/09/2023

Azure Container Apps enables you to run microservices and containerized applications on a serverless platform. With Container Apps, you enjoy the benefits of running containers while leaving behind the concerns of manually configuring cloud infrastructure and complex container orchestrators.

In this tutorial, you'll deploy a containerized ASP.NET Core 6.0 application to Azure Container Apps using Visual Studio. The steps below also apply to earlier versions of ASP.NET Core.

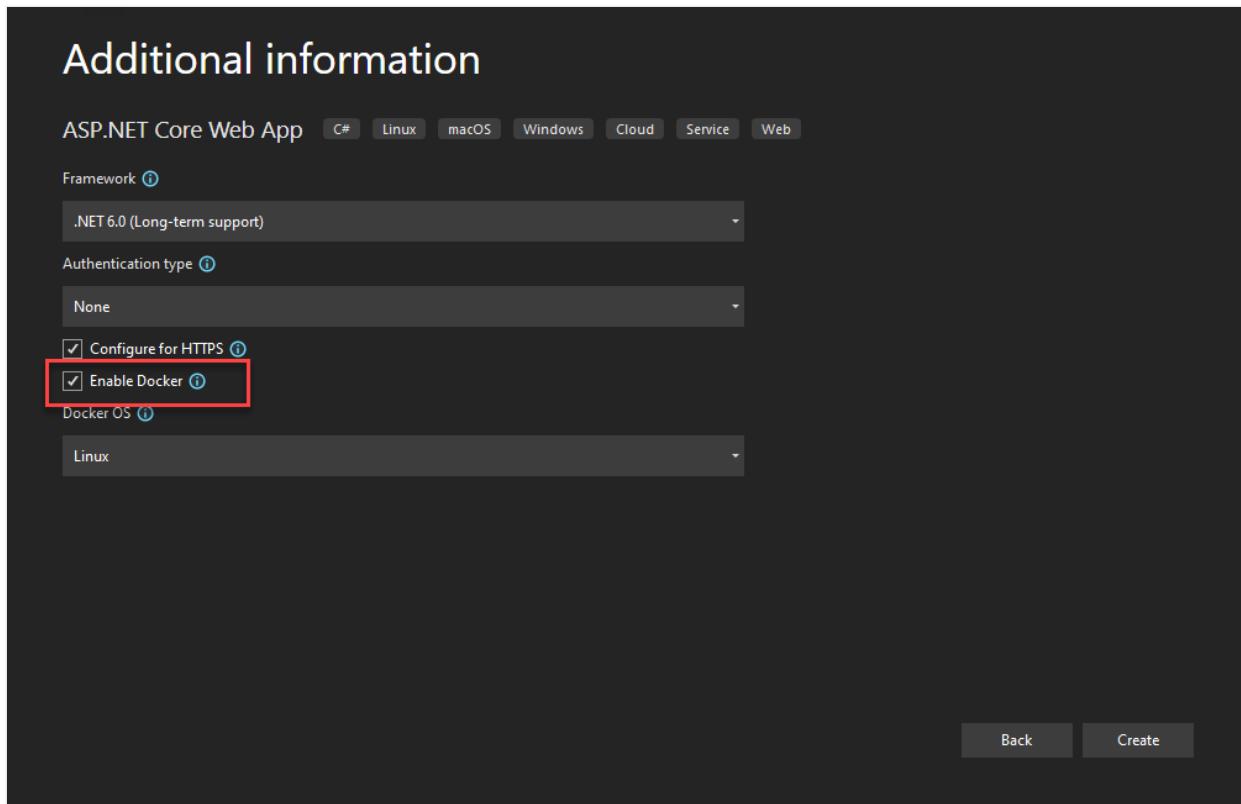
Prerequisites

- An Azure account with an active subscription is required. If you don't already have one, you can [create an account for free](#).
- Visual Studio 2022 version 17.2 or higher, available as a [free download](#).

Create the project

Begin by creating the containerized ASP.NET Core application to deploy to Azure.

1. Inside Visual Studio, select **File** and then choose **New => Project**.
2. In the dialog window, search for **ASP.NET**, and then choose **ASP.NET Core Web App** and select **Next**.
3. In the **Project Name** field, name the application *MyContainerApp* and then select **Next**.
4. On the **Additional Information** screen, make sure to select **Enable Docker**, and then make sure **Linux** is selected for the **Docker OS** setting. Azure Container Apps currently does not support Windows containers. This selection ensures the project template supports containerization by default. While enabled, the project uses a container as it is running or building.
5. Click **Create** and Visual Studio creates and loads the project.



Deploy to Azure Container Apps

The application includes a Dockerfile because the Enable Docker setting was selected in the project template. Visual Studio uses the Dockerfile to build the container image that is run by Azure Container Apps.

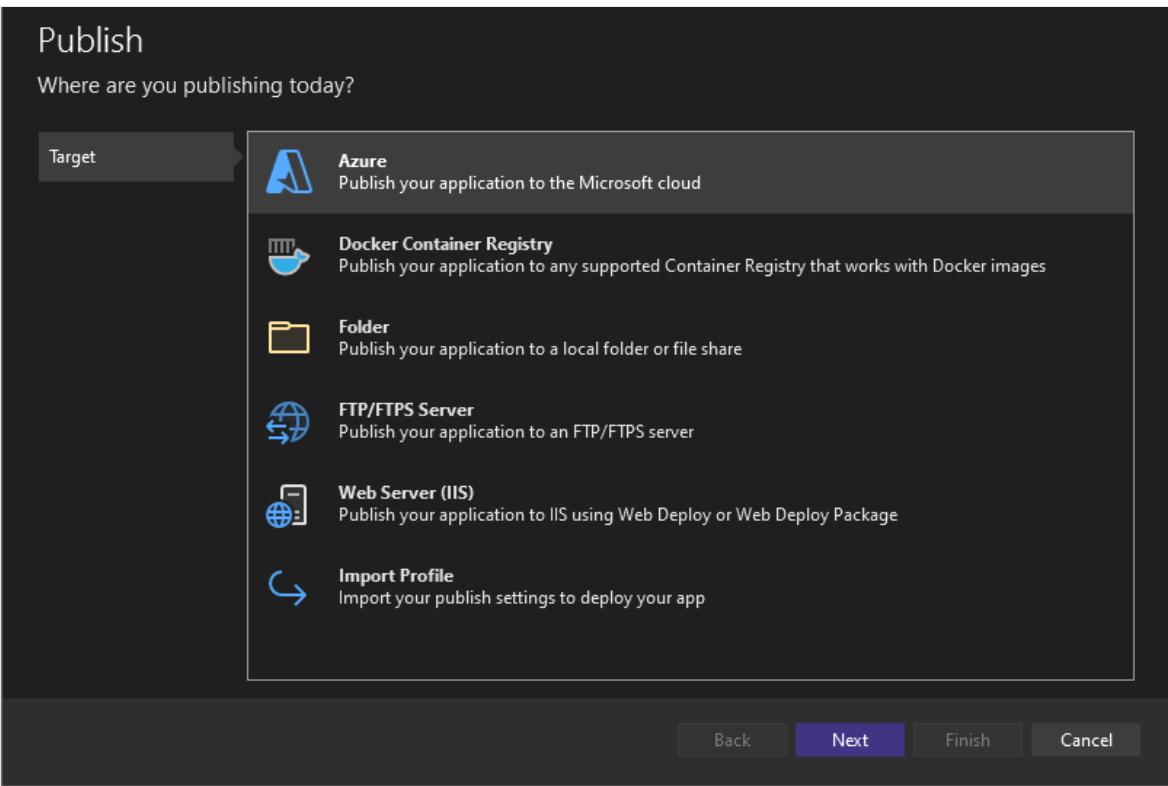
Refer to [How Visual Studio builds containerized apps](#) if you'd like to learn more about the specifics of this process.

You are now ready to deploy to the application to Azure Containers Apps.

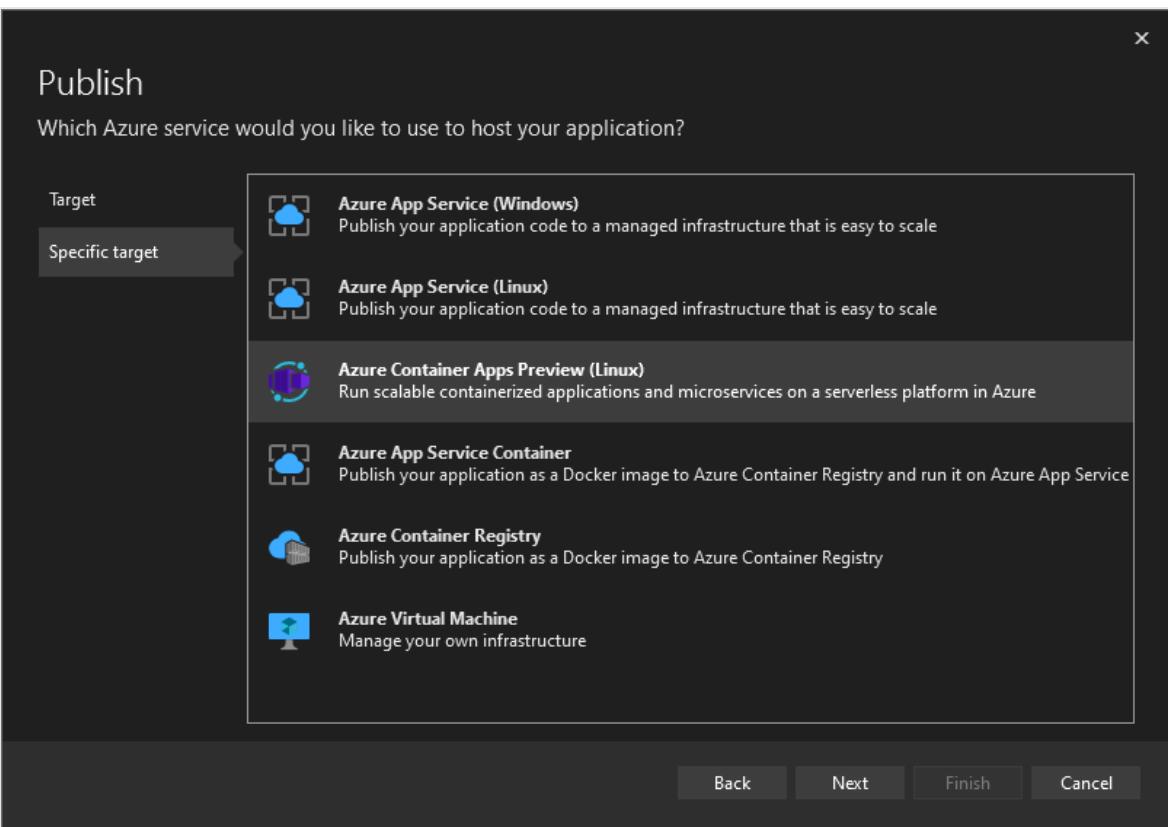
Create the resources

The Visual Studio publish dialogs will help you choose existing Azure resources, or create new ones to be used to deploy your applications to. It will also build the container image using the Dockerfile in the project, push this image to ACR, and finally deploy the new image to the container app selected.

1. Right-click the **MyContainerApp** project node and select **Publish**.
2. In the dialog, choose **Azure** from the list of publishing options, and then select **Next**.



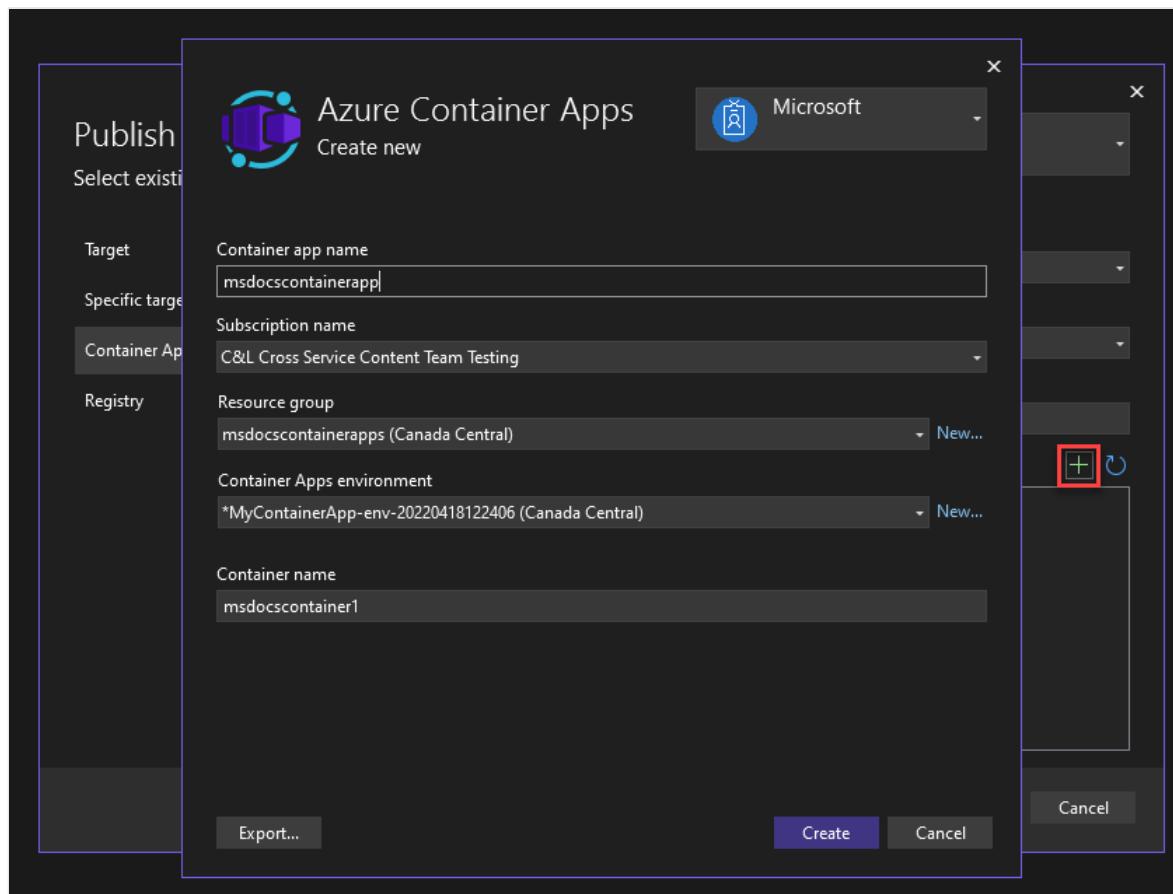
3. On the **Specific target** screen, choose **Azure Container Apps (Linux)**, and then select **Next** again.



4. Next, create an Azure Container App to host the project. Select the green plus icon on the right to open the create dialog. In the *Create new* dialog, enter the following values:

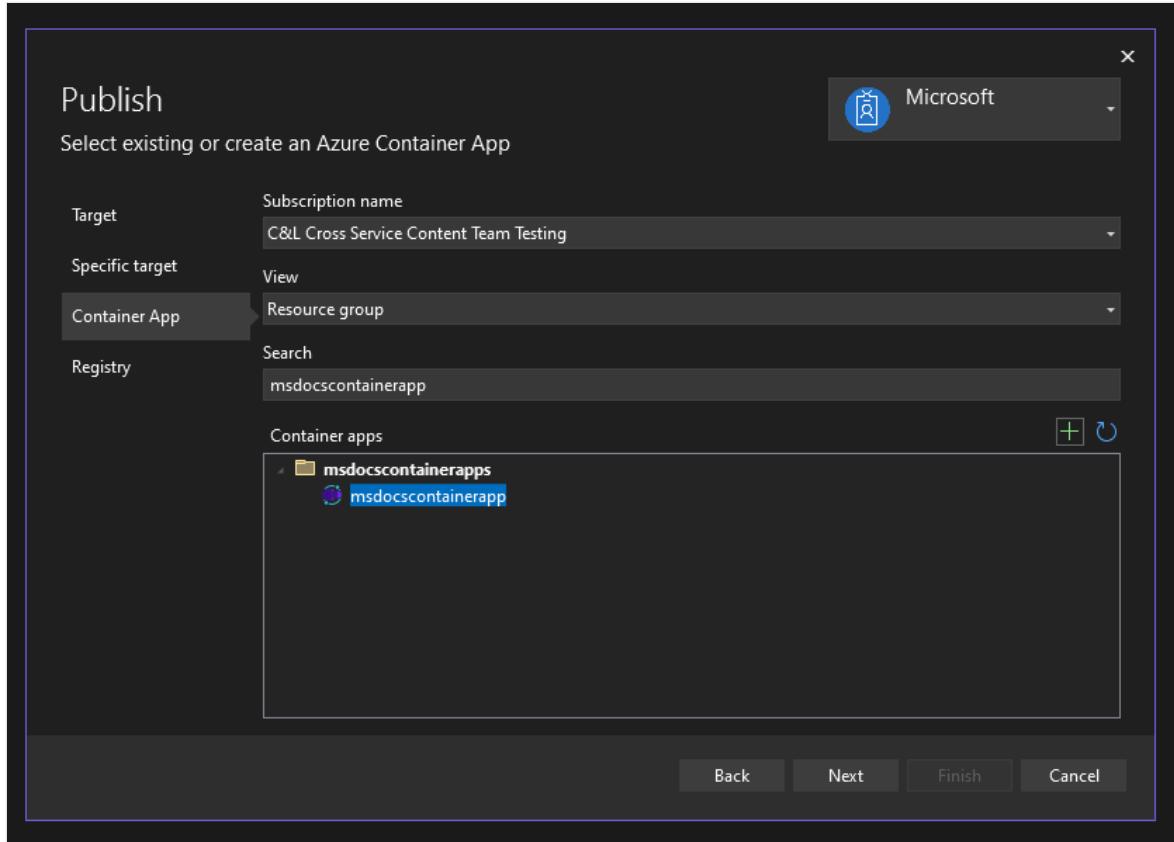
- **Container App name:** Enter a name of `msdocscontainerapp`.

- **Subscription name:** Choose the subscription where you would like to host your app.
- **Resource group:** A resource group acts as a logical container to organize related resources in Azure. You can either select an existing resource group, or select **New** to create one with a name of your choosing, such as `msdocscontainerapps`.
- **Container Apps Environment:** Container Apps Environment: Every container app must be part of a container app environment. An environment provides an isolated network for one or more container apps, making it possible for them to easily invoke each other. Click **New** to open the Create new dialog for your container app environment. Leave the default values and select **OK** to close the environment dialog.
- **Container Name:** This is the friendly name of the container that will run for this container app. Use the name `msdocscontainer1` for this quickstart. A container app typically runs a single container, but there are times when having more than one container is needed. One such example is when a sidecar container is required to perform an activity such as specialized logging or communications.



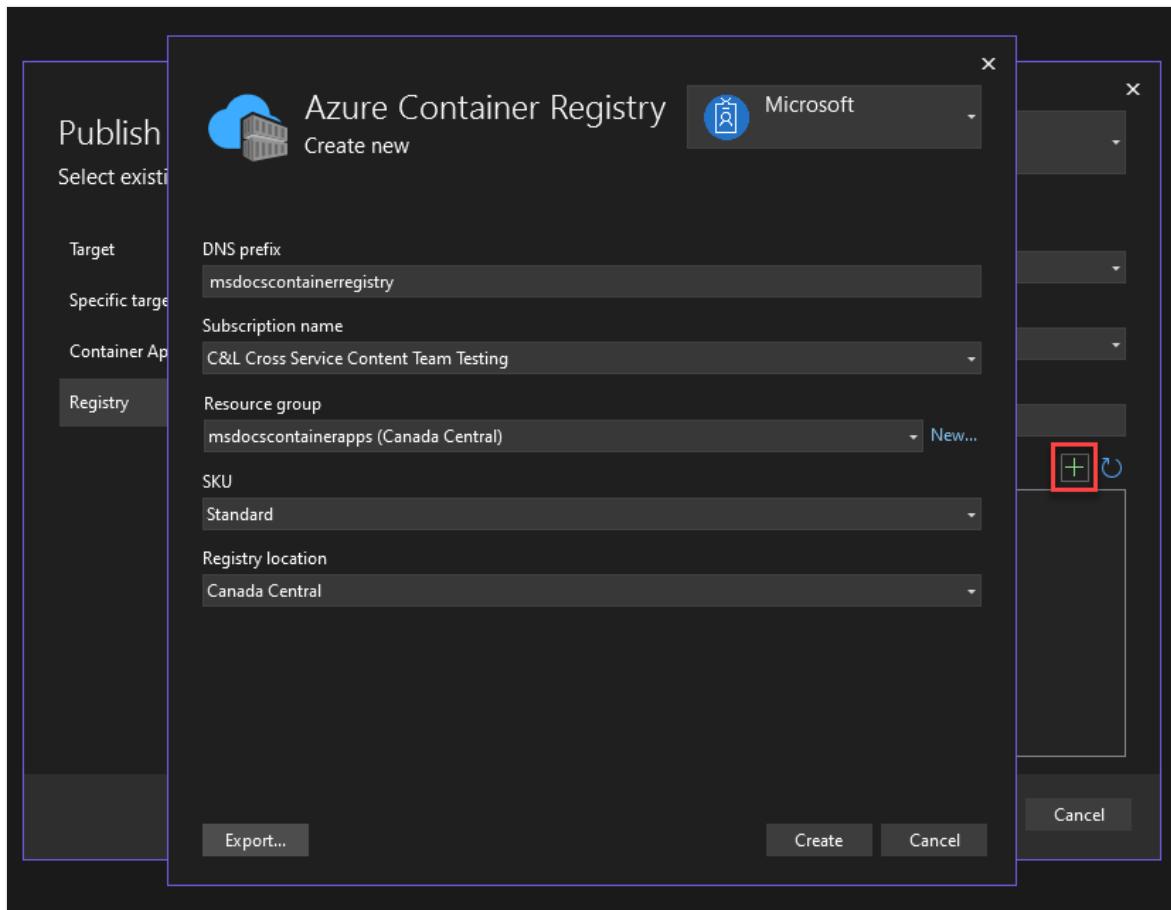
5. Select **Create** to finalize the creation of your container app. Visual Studio and Azure create the needed resources on your behalf. This process may take a couple minutes, so allow it to run to completion before moving on.

6. Once the resources are created, choose **Next**.

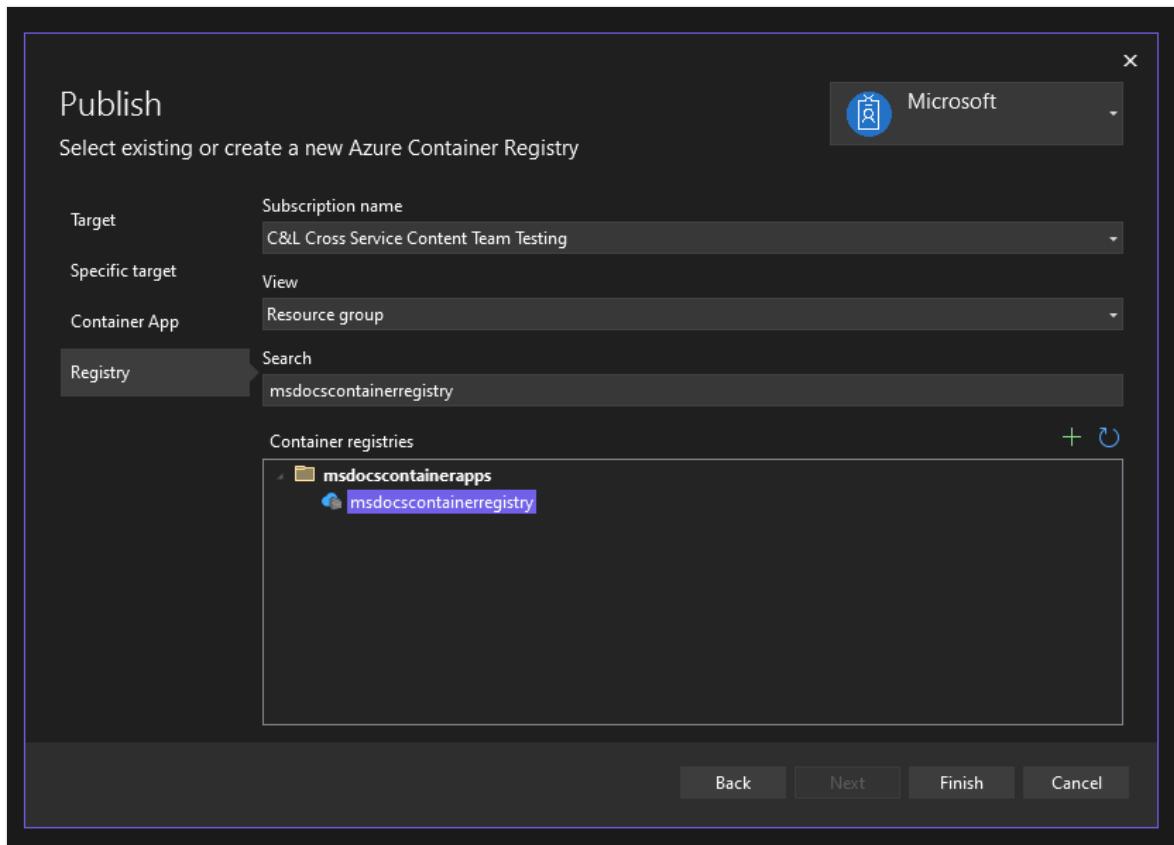


7. On the **Registry** screen, you can either select an existing Registry if you have one, or create a new one. To create a new one, click the green + icon on the right. On the **Create new registry** screen, fill in the following values:

- **DNS prefix:** Enter a value of `msdocscontainerregistry` or a name of your choosing.
- **Subscription Name:** Select the subscription you want to use - you may only have one to choose from.
- **Resource Group:** Choose the msdocs resource group you created previously.
- **Sku:** Select **Standard**.
- **Registry Location:** Select a region that is geographically close to you.



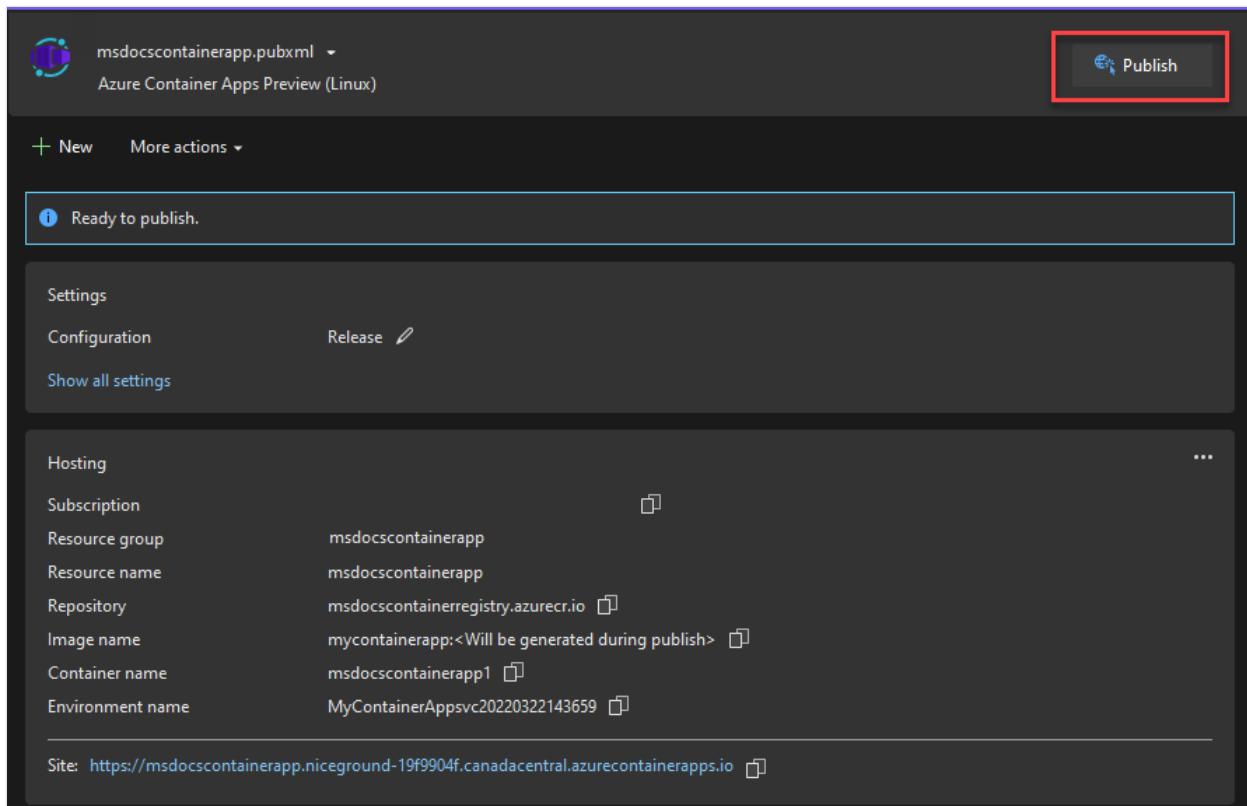
8. After you have populated these values, select **Create**. Visual Studio and Azure will take a moment to create the registry.
9. Once the container registry is created, make sure it is selected, and then choose **Finish**. Visual Studio will take a moment to create the publish profile. This publish profile is where VS stores the publish options and resources you chose so you can quickly publish again whenever you want. You can close the dialog once it finishes.



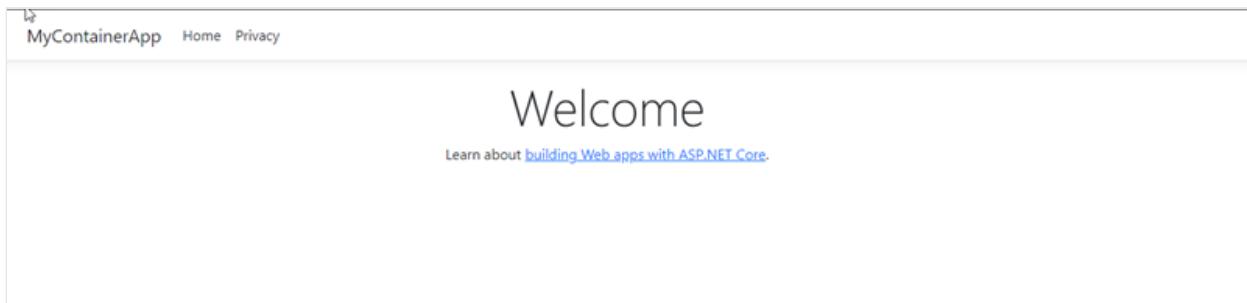
Publish the app using Visual Studio

While the resources and publishing profile are created, you still need to publish and deploy the app to Azure.

Choose **Publish** in the upper right of the publishing profile screen to deploy to the container app you created in Azure. This process may take a moment, so wait for it to complete.



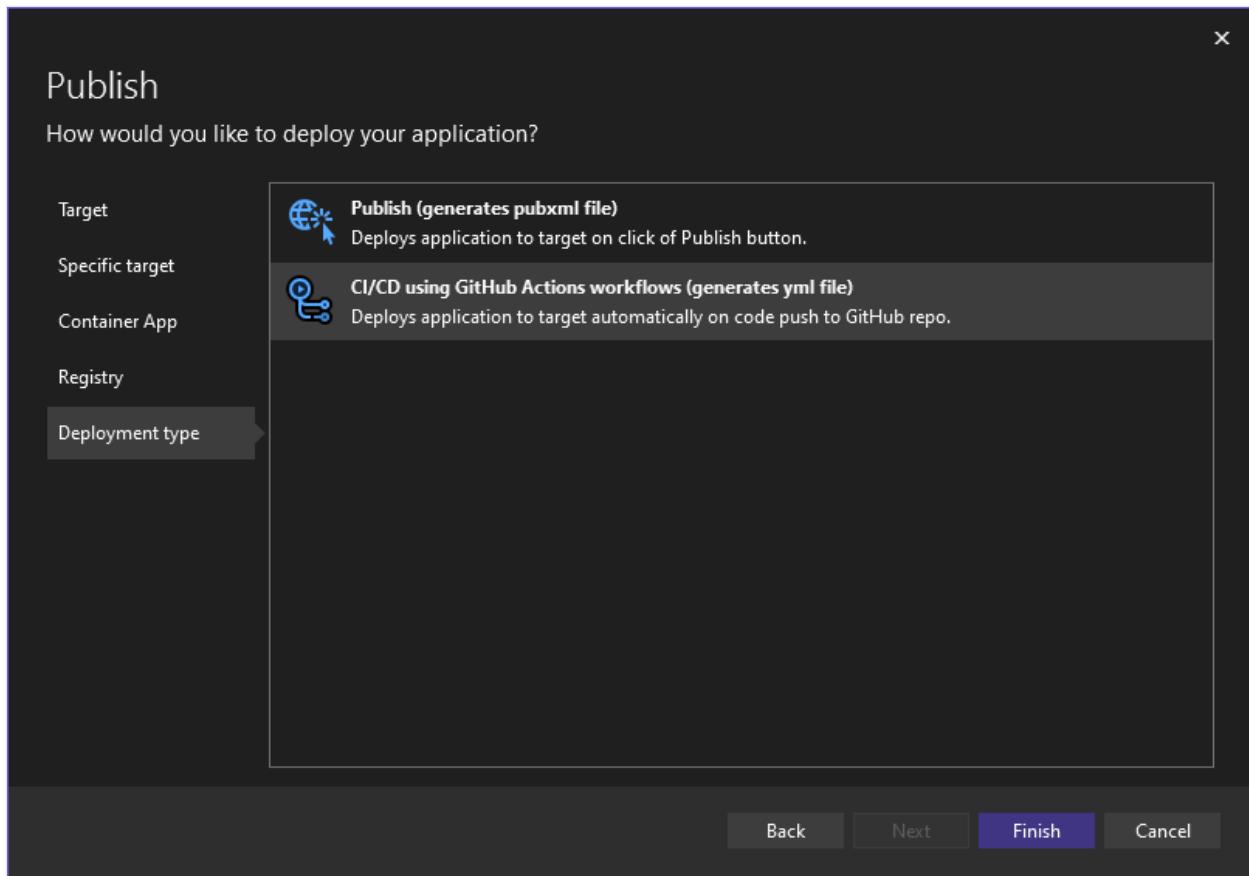
When the app finishes deploying, Visual Studio opens a browser to the URL of your deployed site. This page may initially display an error if all of the proper resources have not finished provisioning. You can continue to refresh the browser periodically to check if the deployment has fully completed.



Publish the app using GitHub Actions

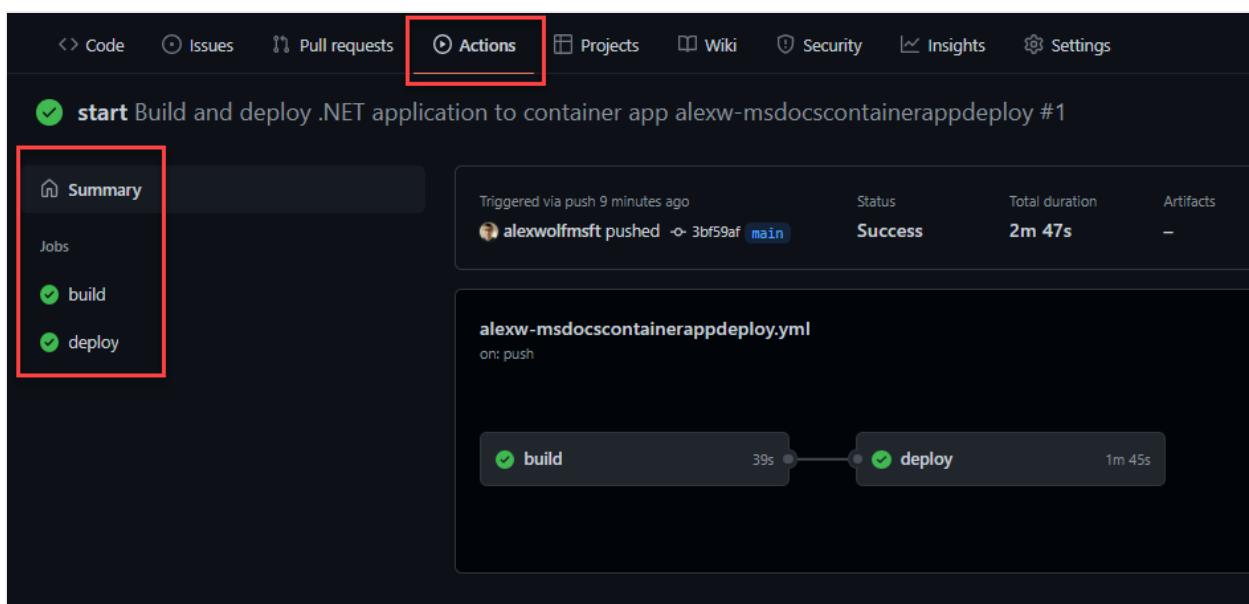
Container Apps can also be deployed using CI/CD through [GitHub actions](#), which are a powerful tool for automating, customizing, and executing development workflows directly through the GitHub repository of your project.

If Visual Studio detects the project you are publishing is hosted in GitHub, the publish flow presents an additional **Deployment type** step. This stage allows developers to choose whether to publish directly through Visual Studio using the steps shown earlier in the quickstart, or through a GitHub Actions workflow.



If you select the GitHub Actions workflow, Visual Studio will add a `.github` folder to the root directory of the project, along with a generated YAML file inside of it. The YAML file contains GitHub Actions configurations to build and deploy your app to Azure every time you push your code.

After you make a change and push your code, you can see the progress of the build and deploy process in GitHub under the **Actions** tab. This page provides detailed logs and indicators regarding the progress and health of the workflow.



Once you see a green checkmark next to the build and deploy jobs the workflow is complete. When you browse to your Container Apps site you should see the latest

changes applied. You can always find the URL for your container app using the Azure portal page.

Clean up resources

If you're not going to continue to use this application, you can delete the Azure Container Apps instance and all the associated services by removing the resource group.

Follow these steps in the Azure portal to remove the resources you created:

1. Select the **msdocscontainerapps** resource group from the *Overview* section.
2. Select the **Delete resource group** button at the top of the resource group *Overview*.
3. Enter the resource group name **msdocscontainerapps** in the *Are you sure you want to delete "my-container-apps"* confirmation dialog.
4. Select **Delete**.

The process to delete the resource group may take a few minutes to complete.

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

[Environments in Azure Container Apps](#)

Quickstart: Deploy to Azure Container Apps using Visual Studio Code

Article • 05/03/2023

Azure Container Apps enables you to run microservices and containerized applications on a serverless platform. With Container Apps, you enjoy the benefits of running containers while leaving behind the concerns of manually configuring cloud infrastructure and complex container orchestrators.

In this tutorial, you'll deploy a containerized application to Azure Container Apps using Visual Studio Code.

Prerequisites

- An Azure account with an active subscription is required. If you don't already have one, you can [create an account for free](#).
- Visual Studio Code, available as a [free download](#).
- The following Visual Studio Code extensions installed:
 - The [Azure Account extension](#)
 - The [Azure Container Apps extension](#)
 - The [Docker extension](#)

Clone the project

1. Begin by cloning the [sample repository](#) to your machine using the following command.

```
git
```

```
git clone https://github.com/Azure-Samples/containerapps-albumapi-javascript.git
```

 **Note**

This tutorial uses a JavaScript project, but the steps are language agnostic.

2. Open Visual Studio Code.
3. Select **F1** to open the command palette.

4. Select **File > Open Folder...** and select the folder where you cloned the sample project.

Sign in to Azure

1. Select **F1** to open the command palette.
2. Select **Azure: Sign In** and follow the prompts to authenticate.
3. Once signed in, return to Visual Studio Code.

Create the container registry and Docker image

Docker images contain the source code and dependencies necessary to run an application. This sample project includes a Dockerfile used to build the application's container. Since you can build and publish the image for your app directly in Azure, a local Docker installation isn't required.

Container images are stored inside container registries. You can create a container registry and upload an image of your app in a single workflow using Visual Studio Code.

1. In the *Explorer* window, expand the *src* folder to reveal the Dockerfile.
2. Right select on the Dockerfile, and select **Build Image in Azure**.

This action opens the command palette and prompts you to define a container tag.

3. Enter a tag for the container. Accept the default, which is the project name with a run ID suffix.
4. Select the Azure subscription that you want to use.

5. Select **+ Create new registry**, or if you already have a registry you'd like to use, select that item and skip to creating and deploying to the container app.

6. Enter a unique name for the new registry such as `msdocscapps123`, where `123` are unique numbers of your own choosing, and then select enter.

Container registry names must be globally unique across all over Azure.

7. Select **Basic** as the SKU.
8. Choose **+ Create new resource group**, or select an existing resource group you'd like to use.

For a new resource group, enter a name such as `msdocscontainerapps`, and press enter.

9. Select the location that is nearest to you. Select **Enter** to finalize the workflow, and Azure begins creating the container registry and building the image.

This process may take a few moments to complete.

10. Select **Linux** as the image base operating system (OS).

Once the registry is created and the image is built successfully, you're ready to create the container app to host the published image.

Create and deploy to the container app

The Azure Container Apps extension for Visual Studio Code enables you to choose existing Container Apps resources, or create new ones to deploy your applications to. In this scenario, you create a new Container App environment and container app to host your application. After installing the Container Apps extension, you can access its features under the Azure control panel in Visual Studio Code.

Create the Container Apps environment

Every container app must be part of a Container Apps environment. An environment provides an isolated network for one or more container apps, making it possible for them to easily invoke each other. You'll need to create an environment before you can create the container app itself.

1. Select `F1` to open the command palette.
2. Enter **Azure Container Apps: Create Container Apps Environment...** and enter the following values as prompted by the extension.

Prompt	Value
Name	Enter <code>my-aca-environment</code>
Region	Select the region closest to you

Once you issue this command, Azure begins to create the environment for you. This process may take a few moments to complete. Creating a container app environment also creates a log analytics workspace for you in Azure.

Create the container app and deploy the Docker image

Now that you have a container app environment in Azure you can create a container app inside of it. You can also publish the Docker image you created earlier as part of this workflow.

1. Select **F1** to open the command palette.
2. Enter **Azure Container Apps: Create Container App...** and enter the following values as prompted by the extension.

Prompt	Value	Remarks
Environment	Select my-aca-environment	
Name	Enter my-container-app	
Container registry	Select Azure Container Registries , then select the registry you created as you published the container image.	
Repository	Select the container registry repository where you published the container image.	
Tag	Select latest	
Environment variables	Select Skip for now	
Ingress	Select Enable	
HTTP traffic type	Select External	
Port	Enter 3500	You set this value to the port number that your container uses.

During this process, Visual Studio Code and Azure create the container app for you. The published Docker image you created earlier is also be deployed to the app. Once this process finishes, Visual Studio Code displays a notification with a link to browse to the site. Select this link, and to view your app in the browser.

```
[{"id":1,"title":"Sgt Peppers Lonely Hearts Club Band","artist":"The Beatles","price":10.99,"image_URL":"https://www.listchallenges.com/f/items/f3b05a20-31ae-4fdf-aebd-d1515af54eea.jpg"}, {"id":2,"title":"Pet Sounds","artist":"The Beach Boys","price":13.99,"image_URL":"https://www.listchallenges.com/f/items/fdef1440-e979-455a-90a7-14e77fac79af.jpg"}, {"id":3,"title":"The Beatles: Revolver","artist":"The Beatles","price":13.99,"image_URL":"https://www.listchallenges.com/f/items/19ff786d-d7a4-4fdc-bee2-59db8b33370d.jpg"}, {"id":4,"title":"Highway 61 Revisited","artist":"Bob Dylan","price":12.99,"image_URL":"https://www.listchallenges.com/f/items/428cf087-6c24-45ad-bf8c-bd3c57ddf444.jpg"}, {"id":5,"title":"Rubber Soul","artist":"The Beatles","price":12.99,"image_URL":"https://www.listchallenges.com/f/items/ebc794ef-1491-4672-a007-0081edc1a8ae.jpg"}, {"id":6,"title":"What's Going On","artist":"Marvin Gaye","price":14.99,"image_URL":"https://www.listchallenges.com/f/items/e5250d6c-1c15-4617-a943-b27e87e21704.jpg"}]
```

You can also append the `/albums` path at the end of the app URL to view data from a sample API request.

Congratulations! You successfully created and deployed your first container app using Visual Studio code.

Clean up resources

If you're not going to continue to use this application, you can delete the Azure Container Apps instance and all the associated services at once by removing the resource group.

Follow these steps in the Azure portal to remove the resources you created:

1. Select the **msdocscontainerapps** resource group from the *Overview* section.
2. Select the **Delete resource group** button at the top of the resource group *Overview*.
3. Enter the resource group name **msdocscontainerapps** in the *Are you sure you want to delete "my-container-apps"* confirmation dialog.
4. Select **Delete**.

The process to delete the resource group may take a few minutes to complete.



Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

[Environments in Azure Container Apps](#)

Create a job with Azure Container Apps (preview)

Article • 05/23/2023

Azure Container Apps [jobs](#) allow you to run containerized tasks that execute for a finite duration and exit. You can trigger a job manually, schedule their execution, or trigger their execution based on events.

Jobs are best suited to for tasks such as data processing, machine learning, or any scenario that requires on-demand processing.

In this quickstart, you create a manual or scheduled job. To learn how to create an event-driven job, see [Deploy an event-driven job with Azure Container Apps](#).

Prerequisites

- An Azure account with an active subscription.
 - If you don't have one, you [can create one for free](#).
- Install the [Azure CLI](#).
- See [Jobs preview limitations](#) for a list of limitations.

Setup

1. To sign in to Azure from the CLI, run the following command and follow the prompts to complete the authentication process.

```
Azure CLI
```

```
az login
```

2. Ensure you're running the latest version of the CLI via the upgrade command.

```
Azure CLI
```

```
az upgrade
```

3. Install the latest version of the Azure Container Apps CLI extension.

```
Azure CLI
```

```
az extension add --name containerapp --upgrade
```

4. Register the `Microsoft.App` and `Microsoft.OperationalInsights` namespaces if you haven't already registered them in your Azure subscription.

Azure CLI

```
az provider register --namespace Microsoft.App  
az provider register --namespace Microsoft.OperationalInsights
```

5. Now that your Azure CLI setup is complete, you can define the environment variables that are used throughout this article.

Azure CLI

```
RESOURCE_GROUP="jobs-quickstart"  
LOCATION="northcentralus"  
ENVIRONMENT="env-jobs-quickstart"  
JOB_NAME="my-job"
```

Create a Container Apps environment

The Azure Container Apps environment acts as a secure boundary around container apps and jobs so they can share the same network and communicate with each other.

1. Create a resource group using the following command.

Azure CLI

```
az group create \  
  --name "$RESOURCE_GROUP" \  
  --location "$LOCATION"
```

2. Create the Container Apps environment using the following command.

Azure CLI

```
az containerapp env create \  
  --name "$ENVIRONMENT" \  
  --resource-group "$RESOURCE_GROUP" \  
  --location "$LOCATION"
```

Create and run a manual job

To use manual jobs, you first create a job with trigger type `Manual` and then start an execution. You can start multiple executions of the same job and multiple job executions can run concurrently.

1. Create a job in the Container Apps environment using the following command.

Azure CLI

```
az containerapp job create \
  --name "$JOB_NAME" --resource-group "$RESOURCE_GROUP" --
  environment "$ENVIRONMENT" \
  --trigger-type "Manual" \
  --replica-timeout 60 --replica-retry-limit 1 --replica-completion-
  count 1 --parallelism 1 \
  --image "mcr.microsoft.com/k8se/quickstart-jobs:latest" \
  --cpu "0.25" --memory "0.5Gi"
```

Manual jobs don't execute automatically. You must start an execution of the job.

2. Start an execution of the job using the following command.

Azure CLI

```
az containerapp job start \
  --name "$JOB_NAME" \
  --resource-group "$RESOURCE_GROUP"
```

The command returns details of the job execution, including its name.

List recent job execution history

Container Apps jobs maintain a history of recent executions. You can list the executions of a job.

Azure CLI

```
az containerapp job execution list \
  --name "$JOB_NAME" \
  --resource-group "$RESOURCE_GROUP" \
  --output table \
  --query '[].{Status: properties.status, Name: name, StartTime:
  properties.startTime}'
```

Executions of scheduled jobs appear in the list as they run.

Console

Status	Name	StartTime
Succeeded	my-job-jvsgub6	2023-05-08T21:21:45+00:00

Query job execution logs

Job executions output logs to the logging provider that you configured for the Container Apps environment. By default, logs are stored in Azure Log Analytics.

1. Save the Log Analytics workspace ID for the Container Apps environment to a variable.

Azure CLI

```
LOG_ANALYTICS_WORKSPACE_ID=`az containerapp env show \
--name "$ENVIRONMENT" \
--resource-group "$RESOURCE_GROUP" \
--query
"properties.appLogsConfiguration.logAnalyticsConfiguration.customerId"
\
--output tsv`
```

2. Save the name of the most recent job execution to a variable.

Azure CLI

```
JOB_EXECUTION_NAME=`az containerapp job execution list \
--name "$JOB_NAME" \
--resource-group "$RESOURCE_GROUP" \
--query "[0].name" \
--output tsv`
```

3. Run a query against Log Analytics for the job execution using the following command.

Azure CLI

```
az monitor log-analytics query \
--workspace "$LOG_ANALYTICS_WORKSPACE_ID" \
--analytics-query "ContainerAppConsoleLogs_CL | where
ContainerGroupName_s startswith '$JOB_EXECUTION_NAME' | order by
_timestamp_d asc" \
--query "[ ].Log_s"
```

ⓘ Note

Until the `ContainerAppConsoleLogs_CL` table is ready, the command returns no results or with an error: `BadArgumentError: The request had some invalid properties.` Wait a few minutes and run the command again.

The following output is an example of the logs printed by the job execution.

JSON

```
[  
    "2023/04/24 18:38:28 This is a sample application that demonstrates  
    how to use Azure Container Apps jobs",  
    "2023/04/24 18:38:28 Starting processing...",  
    "2023/04/24 18:38:33 Finished processing. Shutting down!"  
]
```

Clean up resources

If you're not going to continue to use this application, run the following command to delete the resource group along with all the resources created in this quickstart.

⊗ Caution

The following command deletes the specified resource group and all resources contained within it. If resources outside the scope of this quickstart exist in the specified resource group, they will also be deleted.

Azure CLI

```
az group delete --name "$RESOURCE_GROUP"
```

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

Container Apps jobs

Azure Container Apps plan types

Article • 04/02/2023

Azure Container Apps features two different plan types.

Plan type	Description	In Preview
Consumption	Serverless environment with support for scale-to-zero and pay only for resources your apps use.	No
Consumption + Dedicated plan structures (preview)	Fully managed environment with support for scale-to-zero and pay only for resources your apps use. Optionally, run apps with customized hardware and increased cost predictability using Dedicated workload profiles.	Yes

Consumption plan

The Consumption plan features a serverless architecture that allows your applications to scale in and out on demand. Applications can scale to zero, and you only pay for running apps.

Use the Consumption plan when you don't have specific hardware requirements for your container app.

Consumption + Dedicated plan structure (preview)

The Consumption + Dedicated plan structure consists of a serverless plan that allows your applications to scale in and out on demand. Applications can scale to zero, and you only pay for running apps. It also consists of a fully managed plan you can optionally use that provides dedicated, customized hardware to run your apps on.

You can select from general purpose and memory optimized [workflow profiles](#) that provide larger amounts of CPU and memory. You pay per node, versus per app, and workload profile can scale in and out as demand changes.

Use the Consumption + Dedicated plan structure when you need any of the following in a single environment:

- **Consumption usage:** Use of the Consumption plan to run apps that need to scale to zero that don't have specific hardware requirements.

- **Secure outbound traffic:** You can create environments with no public inbound access, and customize the outbound network path from environments to use firewalls or other network appliances.

Use the Dedicated plan within the Consumption + Dedicated plan structure when you need any of the following features:

- **Environment isolation:** Use of the Dedicated workload profiles provides apps with dedicated hardware with a single tenant guarantee.
- **Customized compute:** Select from many types and sizes of Dedicated workload profiles based on your apps requirements. You can deploy many apps to each workload profile. Each workload profile can scale independently as more apps are added or removed or as apps scale their replicas up or down.
- **Cost control:** Traditional serverless compute options optimize for scale in response to events and may not provide cost control options. With Dedicated workload profiles, you can set minimum and maximum scaling to help you better control costs.

The Consumption + Dedicated plan structure can be more cost effective when you're running higher scale deployments with steady throughput.

Note

When configuring your cluster with a user defined route for egress, you must explicitly send egress traffic to a network virtual appliance such as Azure Firewall.

Next steps

Deploy an app with:

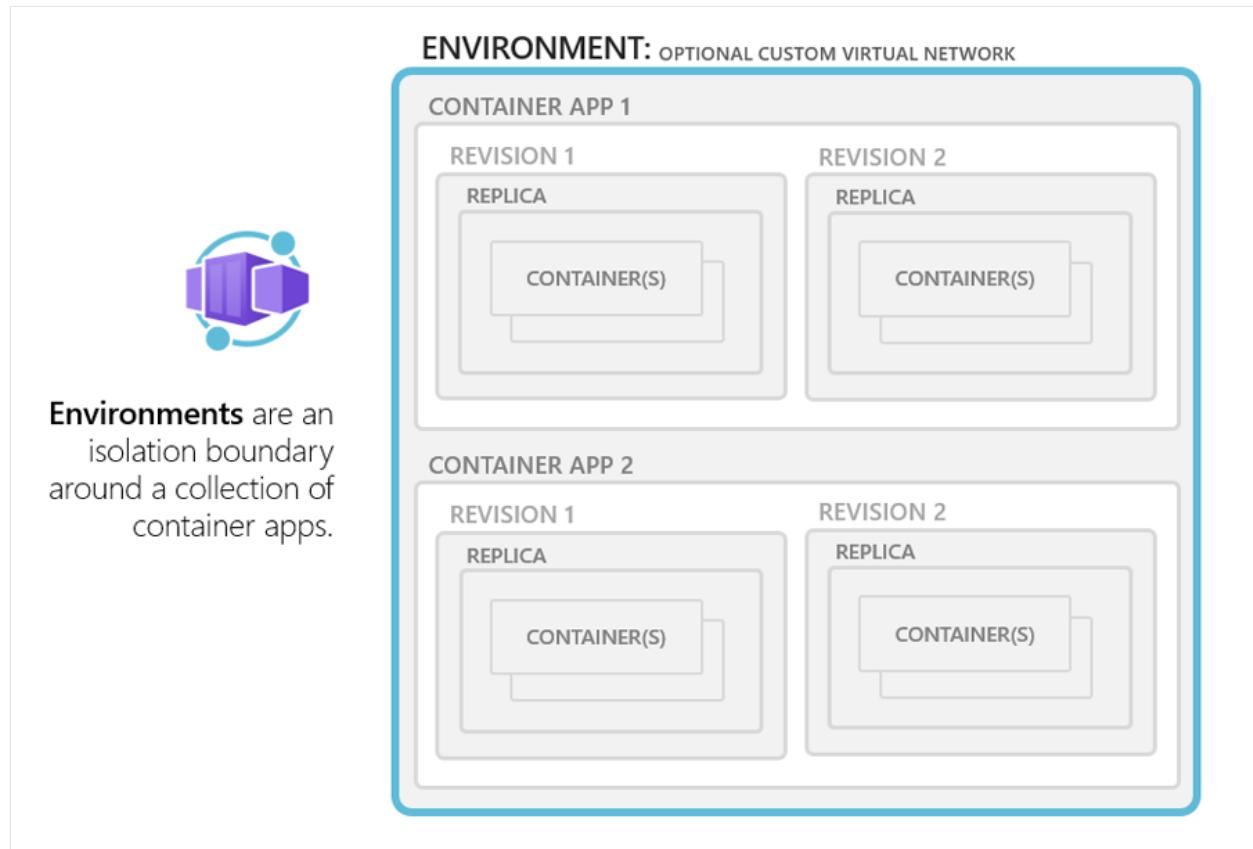
- [Consumption plan](#)
- [Consumption + Dedicated plan structure](#)

Azure Container Apps environments

Article • 05/23/2023

A Container Apps environment is a secure boundary around groups of container apps that share the same virtual network and write logs to the same logging destination.

Container Apps environments are fully managed where Azure handles OS upgrades, scale operations, failover procedures, and resource balancing.



Reasons to deploy container apps to the same environment include situations when you need to:

- Manage related services
- Deploy different applications to the same virtual network
- Instrument Dapr applications that communicate via the Dapr service invocation API
- Have applications to share the same Dapr configuration
- Have applications share the same log analytics workspace

Also, you may provide an [existing virtual network](#) when you create an environment.

Reasons to deploy container apps to different environments include situations when you want to ensure:

- Two applications never share the same compute resources

- Two Dapr applications can't communicate via the Dapr service invocation API

You can add [Azure Functions](#) and [Azure Spring Apps](#) to your Azure Container Apps environment.

Logs

Settings relevant to the Azure Container Apps environment API resource.

Property	Description
<code>properties.appLogsConfiguration</code>	Used for configuring the Log Analytics workspace where logs for all apps in the environment are published.
<code>properties.containerAppsConfiguration.daprAIInstrumentationKey</code>	App Insights instrumentation key provided to Dapr for tracing

Billing

Azure Container Apps has two different pricing structures.

- If you're using the Consumption only plan, or only the Consumption workload profile in the Consumption + Dedicated plan structure then billing is relevant only to individual container apps and their resource usage. There's no cost associated with the Container Apps environment.
- If you're using any Dedicated workload profiles in the Consumption + Dedicated plan structure, there's a fixed cost for the Dedicated plan management. This cost is for the entire environment regardless of how many Dedicated workload profiles you're using.

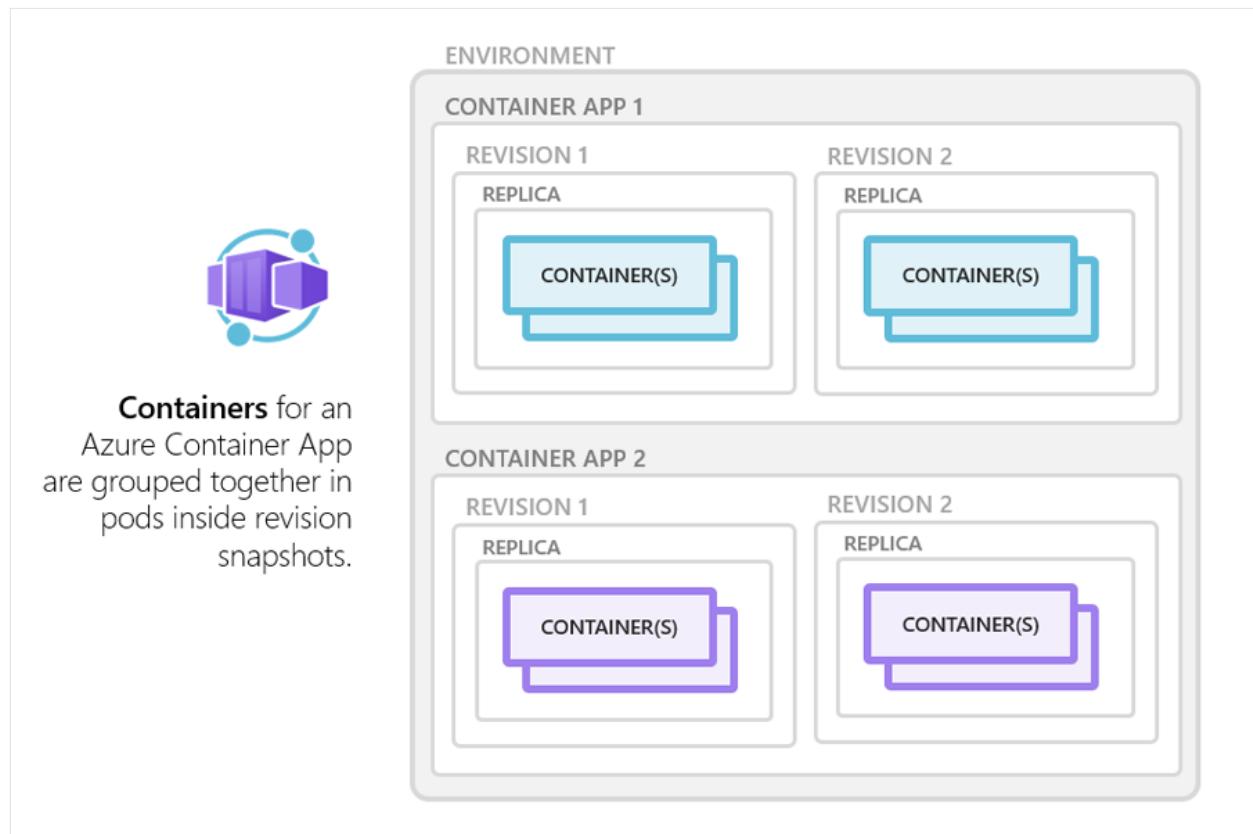
Next steps

[Containers](#)

Containers in Azure Container Apps

Article • 05/05/2023

Azure Container Apps manages the details of Kubernetes and container orchestration for you. Containers in Azure Container Apps can use any runtime, programming language, or development stack of your choice.



Azure Container Apps supports:

- Any Linux-based x86-64 (`linux/amd64`) container image
- Containers from any public or private container registry

Features include:

- There's no required base container image.
- Changes to the `template` configuration section trigger a new [container app revision](#).
- If a container crashes, it automatically restarts.

Configuration

The following code is an example of the `containers` array in the `properties.template` section of a container app resource template. The excerpt shows the available

configuration options when setting up a container.

JSON

```
{  
  "properties": {  
    "template": {  
      "containers": [  
        {  
          "name": "main",  
          "image": "[parameters('container_image')]",  
          "env": [  
            {  
              "name": "HTTP_PORT",  
              "value": "80"  
            },  
            {  
              "name": "SECRET_VAL",  
              "secretRef": "mysecret"  
            }  
          ],  
          "resources": {  
            "cpu": 0.5,  
            "memory": "1Gi"  
          },  
          "volumeMounts": [  
            {  
              "mountPath": "/appsettings",  
              "volumeName": "appsettings-volume"  
            }  
          ],  
          "probes": [  
            {  
              "type": "liveness",  
              "httpGet": {  
                "path": "/health",  
                "port": 8080,  
                "httpHeaders": [  
                  {  
                    "name": "Custom-Header",  
                    "value": "liveness probe"  
                  }  
                ]  
              },  
              "initialDelaySeconds": 7,  
              "periodSeconds": 3  
            },  
            {  
              "type": "readiness",  
              "tcpSocket": {  
                "port": 8081  
              },  
              "initialDelaySeconds": 10,  
              "periodSeconds": 3  
            }  
          ]  
        }  
      ]  
    }  
  }  
}
```

```

        },
        {
          "type": "startup",
          "httpGet": {
            "path": "/startup",
            "port": 8080,
            "httpHeaders": [
              {
                "name": "Custom-Header",
                "value": "startup probe"
              }
            ]
          },
          "initialDelaySeconds": 3,
          "periodSeconds": 3
        }
      ]
    }
  ],
  "initContainers": [
    {
      "name": "init",
      "image": "[parameters('init_container_image')]",
      "resources": {
        "cpu": 0.25,
        "memory": "0.5Gi"
      },
      "volumeMounts": [
        {
          "mountPath": "/appsettings",
          "volumeName": "appsettings-volume"
        }
      ]
    }
  ]
...
}
...
}

```

Setting	Description	Remarks
<code>image</code>	The container image name for your container app.	This value takes the form of <code>repository/image-name:tag</code> .
<code>name</code>	Friendly name of the container.	Used for reporting and identification.

Setting	Description	Remarks
<code>command</code>	The container's startup command.	Equivalent to Docker's entrypoint field.
<code>args</code>	Start up command arguments.	Entries in the array are joined together to create a parameter list to pass to the startup command.
<code>env</code>	An array of key/value pairs that define environment variables.	Use <code>secretRef</code> instead of the <code>value</code> field to refer to a secret.
<code>resources.cpu</code>	The number of CPUs allocated to the container.	<p>With the Consumption plan, values must adhere to the following rules:</p> <ul style="list-style-type: none"> • greater than zero • less than or equal to 2 • can be any decimal number (with a max of two decimal places) <p>For example, <code>1.25</code> is valid, but <code>1.555</code> is invalid. The default is 0.25 CPU per container.</p> <p>When you use the Consumption workload profile in the Consumption + Dedicated plan structure, the same rules apply, except CPU must be less than or equal to 4.</p> <p>When you use a Dedicated workload profile in the Consumption + Dedicated plan structure, the maximum CPU must be less than or equal to the number of cores available in the profile.</p>

Setting	Description	Remarks
<code>resources.memory</code>	The amount of RAM allocated to the container.	<p>With the Consumption plan, values must adhere to the following rules:</p> <ul style="list-style-type: none"> • greater than zero • less than or equal to <code>4Gi</code> • can be any decimal number (with a max of two decimal places) <p>For example, <code>1.25Gi</code> is valid, but <code>1.555Gi</code> is invalid. The default is <code>0.5Gi</code> per container.</p> <p>When you use the Consumption workload profile in the Consumption + Dedicated plan structure, the same rules apply except memory must be less than or equal to <code>8Gi</code>.</p> <p>When you use a dedicated workload profile in the Consumption + Dedicated plan structure, the maximum memory must be less than or equal to the amount of memory available in the profile.</p>
<code>volumeMounts</code>	An array of volume mount definitions.	You can define a temporary volume or multiple permanent storage volumes for your container. For more information about storage volumes, see Use storage mounts in Azure Container Apps .
<code>probes</code>	An array of health probes enabled in the container.	This feature is based on Kubernetes health probes. For more information about probes settings, see Health probes in Azure Container Apps .

In the Consumption plan and the Consumption workload profile in the [Consumption + Dedicated plan structure](#), the total CPU and memory allocations requested for all the containers in a container app must add up to one of the following combinations.

vCPUs (cores)	Memory	Consumption plan	Consumption workload profile
<code>0.25</code>	<code>0.5Gi</code>	✓	✓
<code>0.5</code>	<code>1.0Gi</code>	✓	✓
<code>0.75</code>	<code>1.5Gi</code>	✓	✓
<code>1.0</code>	<code>2.0Gi</code>	✓	✓
<code>1.25</code>	<code>2.5Gi</code>	✓	✓
<code>1.5</code>	<code>3.0Gi</code>	✓	✓

vCPUs (cores)	Memory	Consumption plan	Consumption workload profile
1.75	3.5Gi	✓	✓
2.0	4.0Gi	✓	✓
2.25	4.5Gi		✓
2.5	5.0Gi		✓
2.75	5.5Gi		✓
3.0	6.0Gi		✓
3.25	6.5Gi		✓
3.5	7.0Gi		✓
3.75	7.5Gi		✓
4.0	8.0Gi		✓

- The total of the CPU requests in all of your containers must match one of the values in the *vCPUs* column.
- The total of the memory requests in all your containers must match the memory value in the memory column in the same row of the CPU column.

When you use a Dedicated workload profile in the Consumption + Dedicated plan structure, the total CPU and memory allocations requested for all the containers in a container app must be less than or equal to the cores and memory available in the profile.

Multiple containers

In advanced scenarios, you can run multiple containers in a single container app. The containers share hard disk and network resources and experience the same [application lifecycle](#). There are two ways to run multiple containers in a container app: [sidecar containers](#) and [init containers](#).

Sidecar containers

You can define multiple containers in a single container app to implement the [sidecar pattern](#). Examples of sidecar containers include:

- An agent that reads logs from the primary app container on a [shared volume](#) and forwards them to a logging service.
- A background process that refreshes a cache used by the primary app container in a shared volume.

ⓘ Note

Running multiple containers in a single container app is an advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled. In most situations where you want to run multiple containers, such as when implementing a microservice architecture, deploy each service as a separate container app.

To run multiple containers in a container app, add more than one container in the `containers` array of the container app template.

Init containers (preview)

You can define one or more [init containers](#) in a container app. Init containers run before the primary app container and can be used to perform initialization tasks such as downloading data or preparing the environment.

Init containers are defined in the `initContainers` array of the container app template. The containers run in the order they are defined in the array and must complete successfully before the primary app container starts.

Container registries

You can deploy images hosted on private registries by providing credentials in the Container Apps configuration.

To use a container registry, you define the required fields in `registries` array in the [properties.configuration](#) section of the container app resource template. The `passwordSecretRef` field identifies the name of the secret in the `secrets` array name where you defined the password.

JSON

```
{  
  ...  
  "registries": [  
    {"server": "docker.io",
```

```
        "username": "my-registry-user-name",
        "passwordSecretRef": "my-password-secret-name"
    }]
}
```

With the registry information added, the saved credentials can be used to pull a container image from the private registry when your app is deployed.

The following example shows how to configure Azure Container Registry credentials in a container app.

JSON

```
{
...
"configuration": {
    "secrets": [
        {
            "name": "acr-password",
            "value": "my-acr-password"
        }
    ],
...
"registries": [
    {
        "server": "myacr.azurecr.io",
        "username": "someuser",
        "passwordSecretRef": "acr-password"
    }
]
}
```

ⓘ Note

Docker Hub [limits](#) the number of Docker image downloads. When the limit is reached, containers in your app will fail to start. You're recommended to use a registry with sufficient limits, such as [Azure Container Registry](#).

Managed identity with Azure Container Registry

You can use an Azure managed identity to authenticate with Azure Container Registry instead of using a username and password. For more information, see [Managed identities in Azure Container Apps](#).

When assigning a managed identity to a registry, use the managed identity resource ID for a user-assigned identity, or "system" for the system-assigned identity.

JSON

```
{  
    "identity": {  
        "type": "SystemAssigned,UserAssigned",  
        "userAssignedIdentities": {  
            "<IDENTITY1_RESOURCE_ID>": {}  
        }  
    }  
}  
"properties": {  
    "configuration": {  
        "registries": [  
            {  
                "server": "myacr1.azurecr.io",  
                "identity": "<IDENTITY1_RESOURCE_ID>"  
            },  
            {  
                "server": "myacr2.azurecr.io",  
                "identity": "system"  
            }]  
    }  
    ...  
}
```

For more information about configuring user-assigned identities, see [Add a user-assigned identity](#).

Limitations

Azure Container Apps has the following limitations:

- **Privileged containers:** Azure Container Apps can't run privileged containers. If your program attempts to run a process that requires root access, the application inside the container experiences a runtime error.
- **Operating system:** Linux-based (`linux/amd64`) container images are required.

Next steps

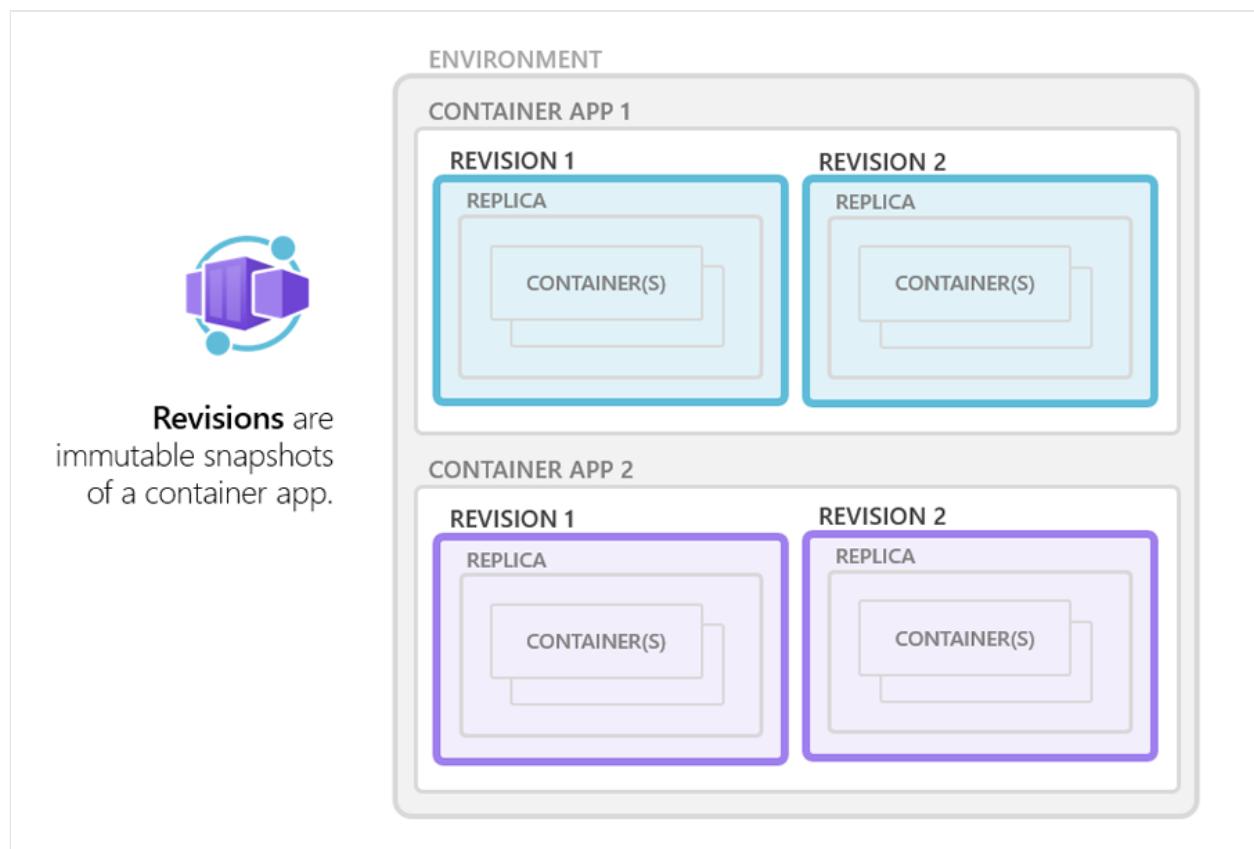
Revisions

Revisions in Azure Container Apps

Article • 06/01/2023

Azure Container Apps implements container app versioning by creating revisions. A revision is an immutable snapshot of a container app version.

- The first revision is automatically provisioned when you deploy your container app.
- New revisions are automatically provisioned when you make a *revision-scope* change to your container app.
- While revisions are immutable, they're affected by *application-scope* changes, which apply to all revisions.
- You can create new revisions by updating a previous revision.
- You can retain up to 100 revisions, giving you a historical record of your container app updates.
- You can run multiple revisions concurrently.
- You can split external HTTP traffic between active revisions.



Use cases

Container Apps revisions help you manage the release of updates to your container app by creating a new revision each time you make a *revision-scope* change to your app. You

can control which revisions are active, and the external traffic that is routed to each active revision.

You can use revisions to:

- Release a new version of your app.
- Quickly revert to an earlier version of your app.
- Split traffic between revisions for [A/B testing](#).
- Gradually phase in a new revision in blue-green deployments. For more information about blue-green deployment, see [BlueGreenDeployment](#).

Revision lifecycle

Revisions go through a series of states, based on status and availability.

Provisioning status

When a new revision is first created, it has to pass startup and readiness checks.

Provisioning status is set to *provisioning* during verification. Use *provisioning status* to follow progress.

Once the revision is verified, *running status* is set to *running*. The revision is available and ready for work.

Provisioning status values include:

- Provisioning
- Provisioned
- Provisioning failed

Running status

Revisions are fully functional after provisioning is complete. Use *running status* to monitor the status of a revision.

Running status values include:

Status	Description
Running	The revision is running. There are no issues to report.

Status	Description
Unhealthy	<p>The revision isn't operating properly. Use the revision state details for details.</p> <p>Common issues include:</p> <ul style="list-style-type: none"> • Container crashes • Resource quota exceeded • Image access issues, including ImagePullBackOff errors
Failed	<p>The revision isn't operating properly. Use the <i>revision state details</i> for more information. Common issues include:</p> <ul style="list-style-type: none"> • Container crashes • Resource quota exceeded • Image access issues, including ImagePullBackOff errors
Failed	<p>Critical errors caused revisions to fail. The <i>running state</i> provides details. Common causes include:</p> <ul style="list-style-type: none"> • Termination • Exit code <code>137</code>

Use running state details to learn more about the current status.

Inactive status

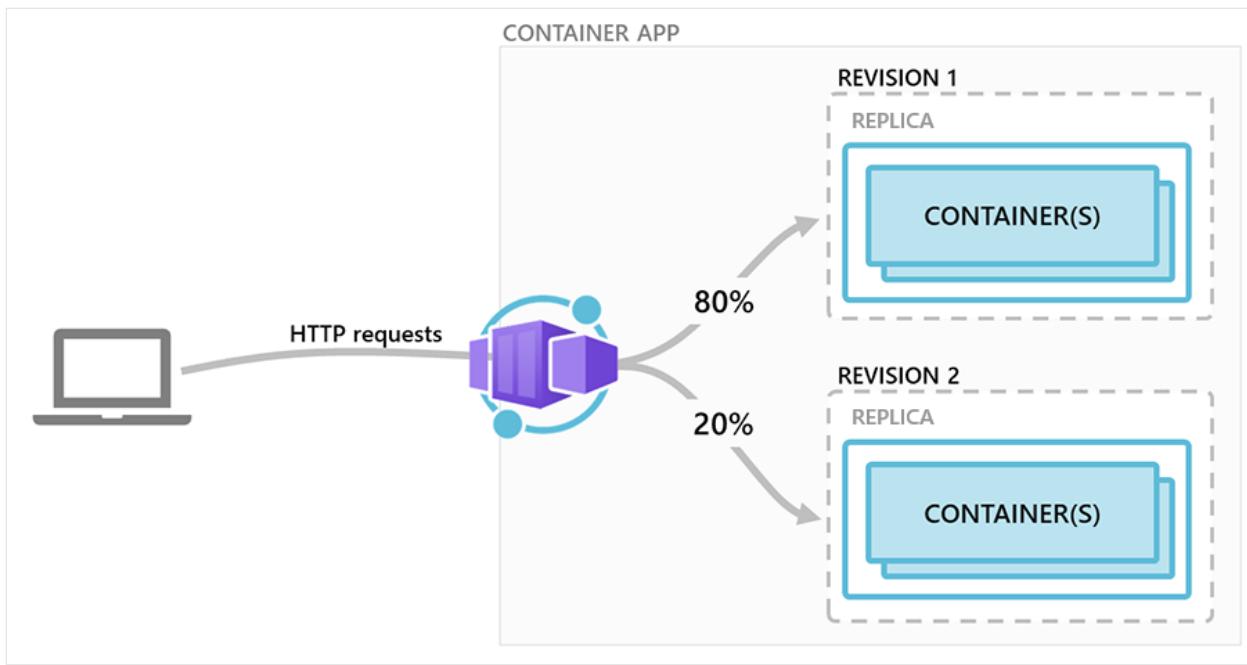
A revision can be set to active or inactive.

Inactive revisions don't have provisioning or running states.

Inactive revisions remain in a list of up to 100 inactive revisions.

Multiple revisions

The following diagram shows a container app with two revisions.



This scenario presumes the container app is in the following state:

- [Ingress](#) is enabled, making the container app available via HTTP or TCP.
- The first revision was deployed as *Revision 1*.
- After the container was updated, a new revision was activated as *Revision 2*.
- [Traffic splitting](#) rules are configured so that *Revision 1* receives 80% of the requests, and *Revision 2* receives the remaining 20%.

Revision name suffix

Revision names are used to identify a revision, and in the revision's URL. You can customize the revision name by setting the revision suffix.

The format of a revision name is:

```
text
<CONTAINER_APP_NAME>-<REVISION_SUFFIX>
```

By default, Container Apps creates a unique revision name with a suffix consisting of a semi-random string of alphanumeric characters. You can customize the name by setting a unique custom revision suffix.

For example, for a container app named *album-api*, setting the revision suffix name to *first-revision* would create a revision with the name *album-api--first-revision*.

A revision suffix name must:

- consist of lower case alphanumeric characters or dashes ('-')

- start with an alphabetic character
- end with an alphanumeric character
- not have two consecutive dashes (--)
- not be more than 64 characters

You can set the revision suffix in the [ARM template](#), through the Azure CLI `az containerapp create` and `az containerapp update` commands, or when creating a revision via the Azure portal.

Change types

Changes to a container app fall under two categories: *revision-scope* or *application-scope* changes. *Revision-scope* changes trigger a new revision when you deploy your app, while *application-scope* changes don't.

Revision-scope changes

A new revision is created when a container app is updated with *revision-scope* changes. The changes are limited to the revision in which they're deployed, and don't affect other revisions.

A *revision-scope* change is any change to the parameters in the [properties.template](#) section of the container app resource template.

These parameters include:

- [Revision suffix](#)
- Container configuration and images
- Scale rules for the container application

Application-scope changes

When you deploy a container app with *application-scope* changes:

- The changes are globally applied to all revisions.
- A new revision isn't created.

Application-scope changes are defined as any change to the parameters in the [properties.configuration](#) section of the container app resource template.

These parameters include:

- [Secret values](#) (revisions must be restarted before a container recognizes new secret values)
- [Revision mode](#)
- Ingress configuration including:
 - Turning [ingress](#) on or off
 - [Traffic splitting rules](#)
 - Labels
- Credentials for private container registries
- Dapr settings

Revision modes

The revision mode controls whether only a single revision or multiple revisions of your container app can be simultaneously active. You can set your app's revision mode from your container app's **Revision management** page in the Azure portal, using Azure CLI commands, or in the ARM template.

Single revision mode

By default, a container app is in *single revision mode*. In this mode, when a new revision is created, the latest revision replaces the active revision. For more information, see [Zero downtime deployment](#).

Multiple revision mode

Set the revision mode to *multiple revision mode*, to run multiple revisions of your app simultaneously. While in this mode, new revisions are activated alongside current active revisions.

For an app implementing external HTTP ingress, you can control the percentage of traffic going to each active revision from your container app's **Revision management** page in the Azure portal, using Azure CLI commands, or in an ARM template. For more information, see [Traffic splitting](#).

Revision Labels

For container apps with external HTTP traffic, labels are a portable means to direct traffic to specific revisions. A label provides a unique URL that you can use to route traffic to the revision that the label is assigned. To switch traffic between revisions, you can move the label from one revision to another.

- Labels keep the same URL when moved from one revision to another.
- A label can be applied to only one revision at a time.
- Allocation for traffic splitting isn't required for revisions with labels.
- Labels are most useful when the app is in *multiple revision mode*.
- You can enable labels, traffic splitting or both.

Labels are useful for testing new revisions. For example, when you want to give access to a set of test users, you can give them the label's URL. Then when you want to move your users to a different revision, you can move the label to that revision.

Labels work independently of traffic splitting. Traffic splitting distributes traffic going to the container app's application URL to revisions based on the percentage of traffic. When traffic is directed to a label's URL, the traffic is routed to one specific revision.

A label name must:

- consist of lower case alphanumeric characters or dashes ('-')
- start with an alphabetic character
- end with an alphanumeric character
- not have two consecutive dashes (--)
- not be more than 64 characters

You can manage labels from your container app's **Revision management** page in the Azure portal.

Name	Date created	Provision Status	Label	Traffic ↓	Active
album-api--v3	5/3/2022, 12:29...	Provisioned	label-1	0 %	<input checked="" type="checkbox"/>
album-api--v2	5/2/2022, 12:44...	Provisioned		100 %	<input checked="" type="checkbox"/>

You can find the label URL in the revision details pane.

Revision details

X

Revision name	album-api--v3
Revision URL	album-api--v3.grayplant-a453c292.canadacentral.azurecontainerapps.io
Label URL	album-api---label-1.grayplant-a453c292.canadacentral.azurecontainerapps.io
Status	Active
Time created	5/3/2022, 12:29:11 PM
Traffic	0%
Containers	View details
Scale	View details
Logs	View details

Activation state

In *multiple revision modes*, revisions remain active until you deactivate them. You can activate and deactivate revisions from your container app's **Revision management** page in the Azure portal or from the Azure CLI.

You aren't charged for the inactive revisions. You can have a maximum of 100 revisions, after which the oldest revision is purged.

Next steps

[Application lifecycle management](#)

Application lifecycle management in Azure Container Apps

Article • 03/15/2023

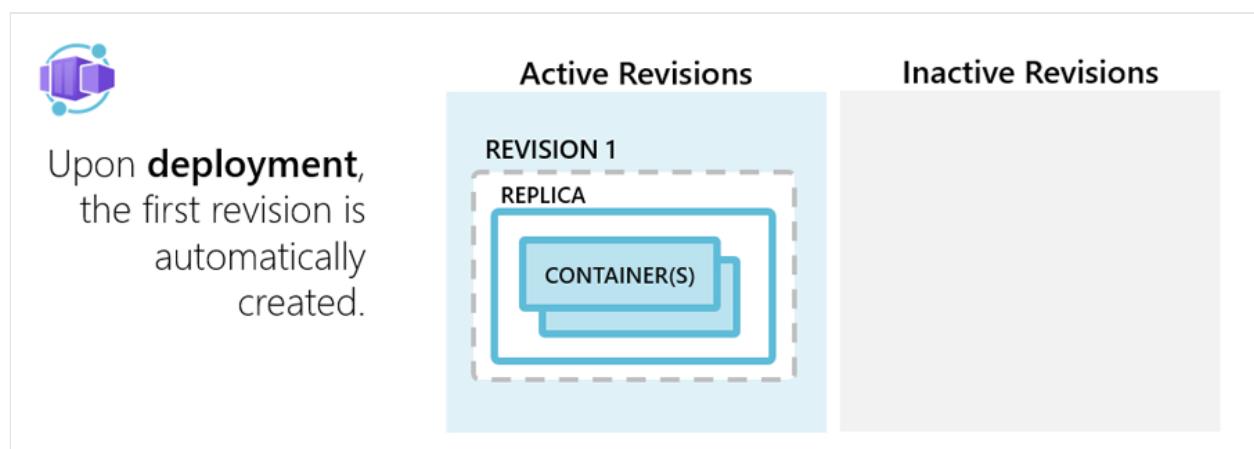
The Azure Container Apps application lifecycle revolves around [revisions](#).

When you deploy a container app, the first revision is automatically created. [More revisions are created](#) as [containers](#) change, or any adjustments are made to the [template](#) section of the configuration.

A container app flows through four phases: deployment, update, deactivation, and shut down.

Deployment

As a container app is deployed, the first revision is automatically created.

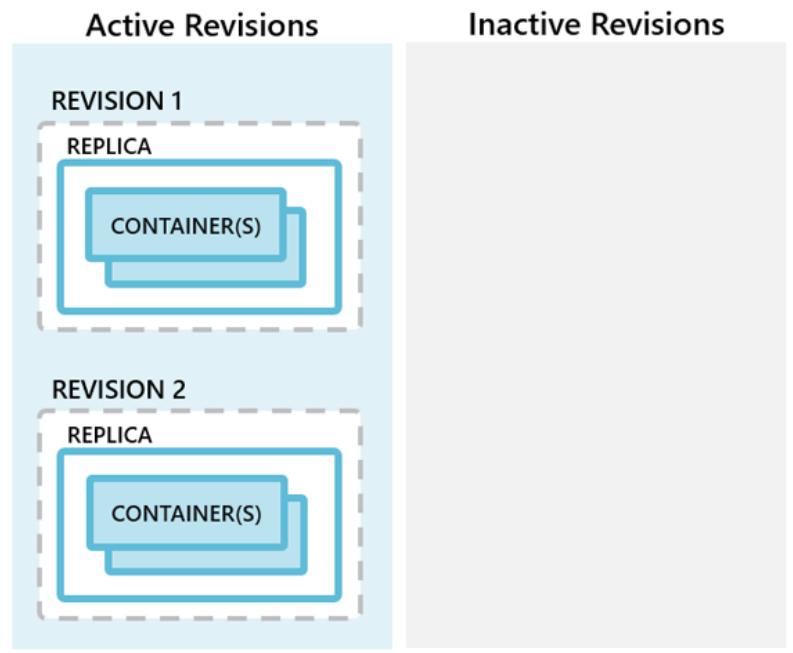


Update

As a container app is updated with a [revision scope-change](#), a new revision is created. You can [choose](#) whether to automatically deactivate old revisions (single revision mode), or allow them to remain available (multiple revision mode).



As the container app is **updated**, a new revision is created.



Zero downtime deployment

In single revision mode, Container Apps automatically ensures your app doesn't experience downtime when creating a new revision. The existing active revision isn't deactivated until the new revision is ready. If ingress is enabled, the existing revision continues to receive 100% of the traffic until the new revision is ready.

ⓘ Note

A new revision is considered ready when one of its replicas starts and becomes ready. A replica is ready when all of its containers start and pass their **startup** and **readiness probes**.

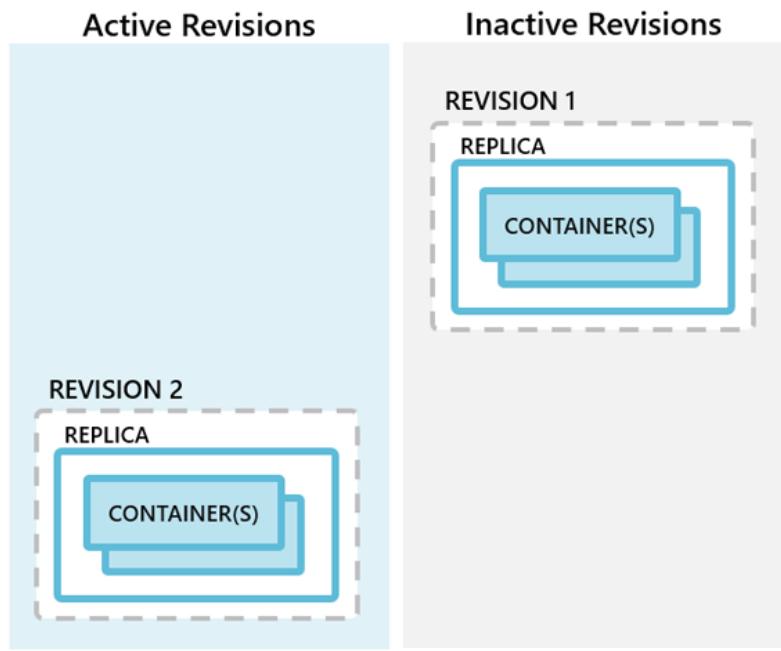
In multiple revision mode, you control when revisions are activated or deactivated and which revisions receive ingress traffic. If a [traffic splitting rule](#) is configured with `latestRevision` set to `true`, traffic doesn't switch to the latest revision until it's ready.

Deactivate

Once a revision is no longer needed, you can deactivate a revision with the option to reactivate later. During deactivation, containers in the revision are [shut down](#).



Once a revision is no longer needed, you can **deactivate** individual revisions, or choose to automatically deactivate old revisions.



Shutdown

The containers are shut down in the following situations:

- As a container app scales in
- As a container app is being deleted
- As a revision is being deactivated

When a shutdown is initiated, the container host sends a [SIGTERM message](#) to your container. The code implemented in the container can respond to this operating system-level message to handle termination.

If your application doesn't respond within 30 seconds to the `SIGTERM` message, then [SIGKILL](#) terminates your container.

Additionally, make sure your application can gracefully handle shutdowns. Containers restart regularly, so don't expect state to persist inside a container. Instead, use external caches for expensive in-memory cache requirements.

Next steps

[Microservices](#)

Jobs in Azure Container Apps (preview)

Article • 05/23/2023

Azure Container Apps jobs enable you to run containerized tasks that execute for a finite duration and exit. You can use jobs to perform tasks such as data processing, machine learning, or any scenario where on-demand processing is required.

Container apps and jobs run in the same [environment](#), allowing them to share capabilities such as networking and logging.

Compare container apps and jobs

There are two types of compute resources in Azure Container Apps: apps and jobs.

Apps are services that run continuously. If a container in an app fails, it's restarted automatically. Examples of apps include HTTP APIs, web apps, and background services that continuously process input.

Jobs are tasks that start, run for a finite duration, and exit when finished. Each execution of a job typically performs a single unit of work. Job executions start manually, on a schedule, or in response to events. Examples of jobs include batch processes that run on demand and scheduled tasks.

Job trigger types

A job's trigger type determines how the job is started. The following trigger types are available:

- **Manual:** Manual jobs are triggered on-demand.
- **Schedule:** Scheduled jobs are triggered at specific times and can run repeatedly.
- **Event:** Event-driven jobs are triggered by events such as a message arriving in a queue.

Manual jobs

Manual jobs are triggered on-demand using the Azure CLI or a request to the Azure Resource Manager API.

Examples of manual jobs include:

- One time processing tasks such as migrating data from one system to another.

- An e-commerce site running as container app starts a job execution to process inventory when an order is placed.

To create a manual job, use the job type `Manual`.

Azure CLI

To create a manual job using the Azure CLI, use the `az containerapp job create` command. The following example creates a manual job named `my-job` in a resource group named `my-resource-group` and a Container Apps environment named `my-environment`:

Azure CLI

```
az containerapp job create \
    --name "my-job" --resource-group "my-resource-group" --environment
"my-environment" \
    --trigger-type "Manual" \
    --replica-timeout 60 --replica-retry-limit 1 --replica-completion-
count 1 --parallelism 1 \
    --image "mcr.microsoft.com/k8se/quickstart-jobs:latest" \
    --cpu "0.25" --memory "0.5Gi"
```

The `mcr.microsoft.com/k8se/quickstart-jobs:latest` image is a sample container image that runs a job that waits a few seconds, prints a message to the console, and then exits.

The above command only creates the job. To start a job execution, see [Start a job execution on demand](#).

Scheduled jobs

To create a scheduled job, use the job type `Schedule`.

Container Apps jobs use cron expressions to define schedules. It supports the standard [cron](#) expression format with five fields for minute, hour, day of month, month, and day of week. The following are examples of cron expressions:

Expression	Description
<code>0 */2 * * *</code>	Runs every two hours.
<code>0 0 * * *</code>	Runs every day at midnight.
<code>0 0 * * 0</code>	Runs every Sunday at midnight.

Expression	Description
<code>0 0 1 * *</code>	Runs on the first day of every month at midnight.

Cron expressions in scheduled jobs are evaluated in Universal Time Coordinated (UTC).

Azure CLI

To create a scheduled job using the Azure CLI, use the `az containerapp job create` command. The following example creates a scheduled job named `my-job` in a resource group named `my-resource-group` and a Container Apps environment named `my-environment`:

Azure CLI

```
az containerapp job create \
    --name "my-job" --resource-group "my-resource-group" --environment
"my-environment" \
    --trigger-type "Schedule" \
    --replica-timeout 60 --replica-retry-limit 1 --replica-completion-
count 1 --parallelism 1 \
    --image "mcr.microsoft.com/k8se/quickstart-jobs:latest" \
    --cpu "0.25" --memory "0.5Gi" \
    --cron-expression "0 0 * * *"
```

The `mcr.microsoft.com/k8se/quickstart-jobs:latest` image is a sample container image that runs a job that waits a few seconds, prints a message to the console, and then exits.

The cron expression `0 0 * * *` runs the job every day at midnight UTC.

Event-driven jobs

Event-driven jobs are triggered by events from supported [custom scalers](#). Examples of event-driven jobs include:

- A job that runs when a new message is added to a queue such as Azure Service Bus, Kafka, or RabbitMQ.
- A self-hosted GitHub Actions runner or Azure DevOps agent that runs when a new job is queued in a workflow or pipeline.

Container apps and event-driven jobs use [KEDA](#) scalers. They both evaluate scaling rules on a polling interval to measure the volume of events for an event source, but the way they use the results is different.

In an app, each replica continuously processes events and a scaling rule determines the number of replicas to run to meet demand. In event-driven jobs, each job typically processes a single event, and a scaling rule determines the number of jobs to run.

Use jobs when each event requires a new instance of the container with dedicated resources or needs to run for a long time. Event-driven jobs are conceptually similar to [KEDA scaling jobs](#).

To create an event-driven job, use the job type `Event`.

Azure CLI

To create an event-driven job using the Azure CLI, use the `az containerapp job create` command. The following example creates an event-driven job named `my-job` in a resource group named `my-resource-group` and a Container Apps environment named `my-environment`:

Azure CLI

```
az containerapp job create \
    --name "my-job" --resource-group "my-resource-group" --environment
"my-environment" \
    --trigger-type "Event" \
    --replica-timeout 60 --replica-retry-limit 1 --replica-completion-
count 1 --parallelism 1 \
    --image "docker.io/myuser/my-event-driven-job:latest" \
    --cpu "0.25" --memory "0.5Gi" \
    --min-executions "0" \
    --max-executions "10" \
    --scale-rule-name "queue" \
    --scale-rule-type "azure-queue" \
    --scale-rule-metadata "accountName=mystorage" "queueName=myqueue"
"queueLength=1" \
    --scale-rule-auth "connection=connection-string-secret" \
    --secrets "connection-string-secret=<QUEUE_CONNECTION_STRING>"
```

The example configures an Azure Storage queue scale rule. For a complete tutorial, see [Deploy an event-driven job](#).

Start a job execution on demand

For any job type, you can start a job execution on demand.

Azure CLI

To start a job execution using the Azure CLI, use the `az containerapp job start` command. The following example starts an execution of a job named `my-job` in a resource group named `my-resource-group`:

Azure CLI

```
az containerapp job start --name "my-job" --resource-group "my-resource-group"
```

When you start a job execution, you can choose to override the job's configuration. For example, you can override an environment variable or the startup command to pass specific data to the job.

Azure CLI

Azure CLI doesn't support overriding a job's configuration when starting a job execution.

Get job execution history

Each Container Apps job maintains a history of recent job executions.

Azure CLI

To get the statuses of job executions using the Azure CLI, use the `az containerapp job execution list` command. The following example returns the status of the most recent execution of a job named `my-job` in a resource group named `my-resource-group`:

Azure CLI

```
az containerapp job execution list --name "my-job" --resource-group "my-resource-group"
```

The execution history for scheduled & event-based jobs is limited to the most recent `100` successful and failed job executions.

To list all executions of a job or to get detailed output from a job, query the logs provider configured for your Container Apps environment.

Advanced job configuration

Container Apps jobs support advanced configuration options such as container settings, retries, timeouts, and parallelism.

Container settings

Container settings define the containers to run in each replica of a job execution. They include environment variables, secrets, and resource limits. For more information, see [Containers](#).

Job settings

The following table includes the job settings that you can configure:

Setting	Azure Resource Manager property	CLI parameter	Description
Job type	<code>triggerType</code>	<code>--trigger-type</code>	The type of job. (<code>Manual</code> , <code>Schedule</code> , or <code>Event</code>)
Parallelism	<code>parallelism</code>	<code>--parallelism</code>	The number of replicas to run per execution.
Replica completion count	<code>replicaCompletionCount</code>	<code>--replica-completion-count</code>	The number of replicas to complete successfully for the execution to succeed.
Replica timeout	<code>replicaTimeout</code>	<code>--replica-timeout</code>	The maximum time in seconds to wait for a replica to complete.
Replica retry limit	<code>replicaRetryLimit</code>	<code>--replica-retry-limit</code>	The maximum number of times to retry a failed replica.

Example

Azure CLI

The following example creates a job with advanced configuration options:

Azure CLI

```
az containerapp job create \
    --name "my-job" --resource-group "my-resource-group" --environment
```

```
"my-environment" \
    --trigger-type "Schedule" \
    --replica-timeout 1800 --replica-retry-limit 3 --replica-completion-
count 5 --parallelism 5 \
    --image "myregistry.azurecr.io/quickstart-jobs:latest" \
    --cpu "0.25" --memory "0.5Gi" \
    --command "/startup.sh" \
    --env-vars "MY_ENV_VAR=my-value" \
    --cron-expression "0 0 * * *" \
    --registry-server "myregistry.azurecr.io" \
    --registry-username "myregistry" \
    --registry-password "myregistrypassword"
```

Jobs preview restrictions

The following features are not supported:

- Volume mounts
- Init containers
- Dapr
- Azure Key Vault references in secrets
- Ingress and related features such as custom domains and SSL certificates

Next steps

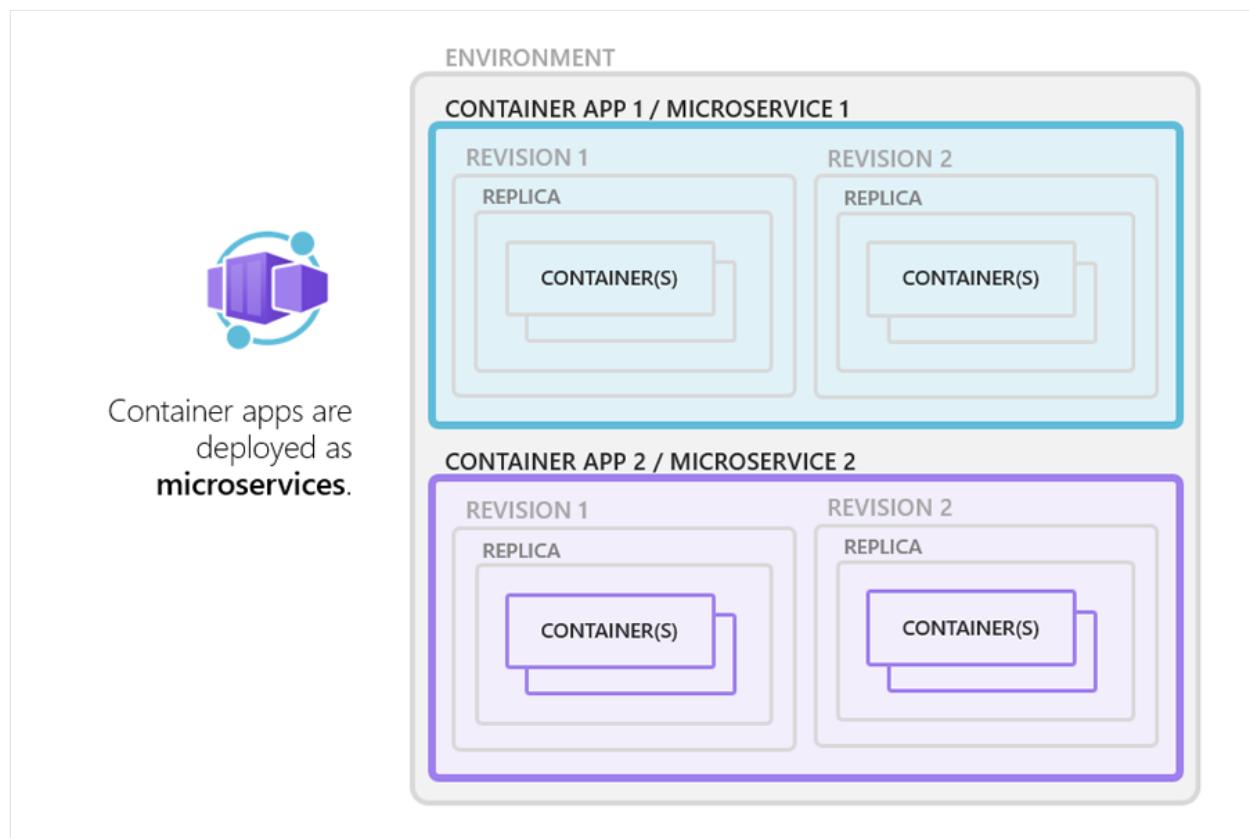
[Create a job with Azure Container Apps](#)

Microservices with Azure Container Apps

Article • 05/23/2023

[Microservice architectures](#) allow you to independently develop, upgrade, version, and scale core areas of functionality in an overall system. Azure Container Apps provides the foundation for deploying microservices featuring:

- Independent [scaling](#), [versioning](#), and [upgrades](#)
- [Service discovery](#)
- Native [Dapr integration](#)



A Container Apps [environment](#) provides a security boundary around a group of container apps. A single container app typically represents a microservice, which is composed of container apps made up of one or more [containers](#).

You can add [Azure Functions](#) and [Azure Spring Apps](#) to your Azure Container Apps environment.

Dapr integration

When implementing a system composed of microservices, function calls are spread across the network. To support the distributed nature of microservices, you need to account for failures, retries, and timeouts. While Container Apps features the building blocks for running microservices, use of [Dapr](#) provides an even richer microservices programming model. Dapr includes features like observability, pub/sub, and service-to-service invocation with mutual TLS, retries, and more.

For more information on using Dapr, see [Build microservices with Dapr](#).

Next steps

Scaling

Networking environment in Azure Container Apps

Article • 05/15/2023

Azure Container Apps run in the context of an [environment](#), which is supported by a virtual network (VNet). By default, your Container App environment is created with a VNet that is automatically generated for you. Generated VNets are inaccessible to you as they're created in Microsoft's tenant. This VNet is publicly accessible over the internet, can only reach internet accessible endpoints, and supports a limited subset of networking capabilities such as ingress IP restrictions and container app level ingress controls.

Use the Custom VNet configuration to provide your own VNet if you need more Azure networking features such as:

- Integration with Application Gateway
- Network Security Groups
- Communicating with resources behind private endpoints in your virtual network

The features available depend on your environment selection.

Environment Selection

There are two environments in Container Apps: the Consumption only environment supports only the [Consumption plan \(GA\)](#) and the workload profiles environment that supports both the [Consumption + Dedicated plan structure \(preview\)](#). The two environments share many of the same networking characteristics. However, there are some key differences.

Environment Type	Description
Workload profiles environment (preview)	Supports user defined routes (UDR) and egress through NAT Gateway. The minimum required subnet size is /27. As workload profiles are currently in preview, the number of supported regions is limited. To learn more, visit the workload profiles overview .
Consumption only environment	Doesn't support user defined routes (UDR) and egress through NAT Gateway. The minimum required subnet size is /23.

Accessibility Levels

In Container Apps, you can configure whether your container app allows public ingress or only ingress from within your VNet at the environment level.

Accessibility level	Description
External	Container Apps environments deployed as external resources are available for public requests. External environments are deployed with a virtual IP on an external, public facing IP address.
Internal	When set to internal, the environment has no public endpoint. Internal environments are deployed with a virtual IP (VIP) mapped to an internal IP address. The internal endpoint is an Azure internal load balancer (ILB) and IP addresses are issued from the custom VNet's list of private IP addresses.

Custom VNet configuration

As you create a custom VNet, keep in mind the following situations:

- If you want your container app to restrict all outside access, create an [internal Container Apps environment](#).
- When you provide your own VNet, you need to provide a subnet that is dedicated to the Container App environment you deploy. This subnet isn't available to other services.
- Network addresses are assigned from a subnet range you define as the environment is created.
 - You can define the subnet range used by the Container Apps environment.
 - You can restrict inbound requests to the environment exclusively to the VNet by deploying the environment as [internal](#).

Note

When you provide your own virtual network, additional [managed resources](#) are created, which incur billing.

As you begin to design the network around your container app, refer to [Plan virtual networks](#) for important concerns surrounding running virtual networks on Azure.

ENVIRONMENT: OPTIONAL CUSTOM VIRTUAL NETWORK



A virtual network

creates a secure boundary around your Azure Container Apps environment.



ⓘ Note

Moving VNets among different resource groups or subscriptions is not supported if the VNet is in use by a Container Apps environment.

HTTP edge proxy behavior

Azure Container Apps uses [Envoy proxy](#) as an edge HTTP proxy. TLS is terminated on the edge and requests are routed based on their traffic splitting rules and routes traffic to the correct application.

HTTP applications scale based on the number of HTTP requests and connections. Envoy routes internal traffic inside clusters. Downstream connections support HTTP1.1 and HTTP2 and Envoy automatically detects and upgrades the connection should the client connection be upgraded. Upstream connection is defined by setting the `transport` property on the [ingress](#) object.

Ingress configuration

Under the [ingress](#) section, you can configure the following settings:

- **Accessibility level:** You can set your container app as externally or internally accessible in the environment. An environment variable `CONTAINER_APP_ENV_DNS_SUFFIX` is used to automatically resolve the FQDN suffix for your environment. When communicating between Container Apps within the same environment, you may also use the app name. For more information on how to access your apps, see [ingress](#).
- **Traffic split rules:** You can define traffic splitting rules between different revisions of your application. For more information, see [Traffic splitting](#).

For more information about ingress configuration, see [Ingress in Azure Container Apps](#).

Scenarios

For more information about scenarios, see [Ingress in Azure Container Apps](#).

Portal dependencies

For every app in Azure Container Apps, there are two URLs.

Container Apps generates the first URL, which is used to access your app. See the *Application Url* in the *Overview* window of your container app in the Azure portal for the fully qualified domain name (FQDN) of your container app.

The second URL grants access to the log streaming service and the console. If necessary, you may need to add `https://azurecontainerapps.dev/` to the allowlist of your firewall or proxy.

Ports and IP addresses

The following ports are exposed for inbound connections.

Use	Port(s)
HTTP/HTTPS	80, 443

IP addresses are broken down into the following types:

Type	Description
Public inbound IP address	Used for app traffic in an external deployment, and management traffic in both internal and external deployments.

Type	Description
Outbound public IP	Used as the "from" IP for outbound connections that leave the virtual network. These connections aren't routed down a VPN. Outbound IPs aren't guaranteed and may change over time. Using a NAT gateway or other proxy for outbound traffic from a Container App environment is only supported on the workload profile environment.
Internal load balancer IP address	This address only exists in an internal deployment.
App-assigned IP-based TLS/SSL addresses	These addresses are only possible with an external deployment, and when IP-based TLS/SSL binding is configured.

Subnet

Virtual network integration depends on a dedicated subnet. How IP addresses are allocated in a subnet and what subnet sizes are supported depends on which plan you're using in Azure Container Apps. Selecting an appropriately sized subnet for the scale of your Container Apps is important as subnet sizes can't be modified post creation in Azure.

- Consumption only environment:
 - /23 is the minimum subnet size required for virtual network integration.
 - Container Apps reserves a minimum of 60 IPs for infrastructure in your VNet, and the amount may increase up to 256 addresses as your container environment scales.
 - As your app scales, a new IP address is allocated for each new replica.
- Workload profiles environment:
 - /27 is the minimum subnet size required for virtual network integration.
 - The subnet you're integrating your container app with must be delegated to [Microsoft.App/environments](#).
 - 11 IP addresses are automatically reserved for integration with the subnet. When your apps are running on workload profiles, the number of IP addresses required for infrastructure integration doesn't vary based on the scale of your container apps.
 - More IP addresses are allocated depending on your Container App's workload profile:

- When you're using Consumption workload profiles for your container app, IP address assignment behaves the same as when running on the Consumption only environment. As your app scales, a new IP address is allocated for each new replica.
- When you're using the Dedicated workload profile for your container app, each node has 1 IP address assigned.

As a Container Apps environment is created, you provide resource IDs for a single subnet.

If you're using the CLI, the parameter to define the subnet resource ID is `infrastructure-subnet-resource-id`. The subnet hosts infrastructure components and user app containers.

In addition, if you're using the Azure CLI with the Consumption only environment and the `platformReservedCidr` range is defined, both subnets must not overlap with the IP range defined in `platformReservedCidr`.

Subnet Address Range Restrictions

Subnet address ranges can't overlap with the following ranges reserved by AKS:

- 169.254.0.0/16
- 172.30.0.0/16
- 172.31.0.0/16
- 192.0.2.0/24

In addition, Container Apps on the workload profiles environment reserve the following addresses:

- 100.100.0.0/17
- 100.100.128.0/19
- 100.100.160.0/19
- 100.100.192.0/19

Routes

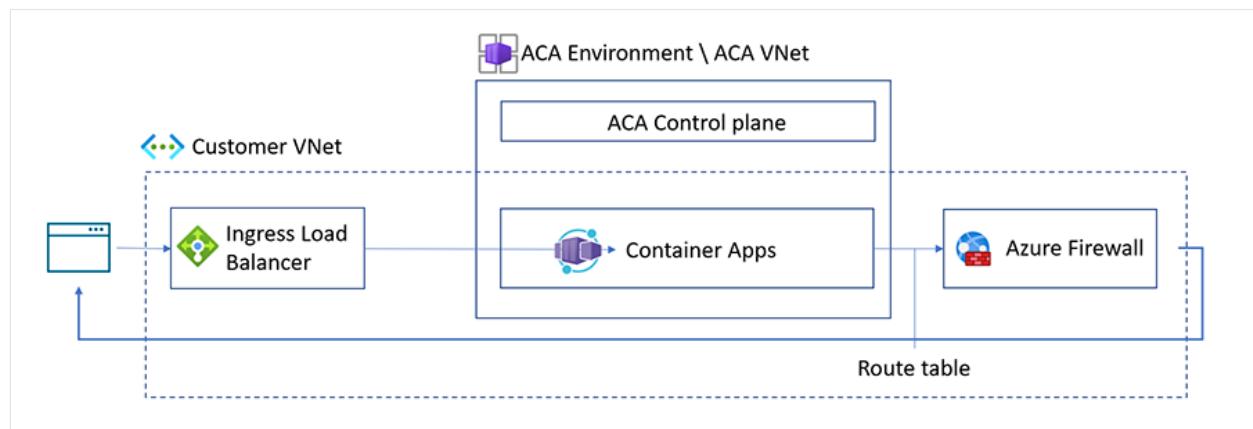
User Defined Routes (UDR) and controlled egress through NAT Gateway are supported in the workload profiles environment, which is in preview. In the Consumption only environment, these features aren't supported.

User defined routes (UDR) - preview

⚠ Note

When using UDR with Azure Firewall in Azure Container Apps, you will need to add certain FQDN's and service tags to the allowlist for the firewall. To learn more, see [configuring UDR with Azure Firewall](#).

You can use UDR on the workload profiles architecture to restrict outbound traffic from your container app through Azure Firewall or other network appliances. Configuring UDR is done outside of the Container Apps environment scope. UDR isn't supported for external environments.



Azure creates a default route table for your virtual networks upon create. By implementing a user-defined route table, you can control how traffic is routed within your virtual network. For example, you can create a UDR that routes all traffic to the firewall.

Configuring UDR with Azure Firewall - preview:

UDR is only supported on the workload profiles environment. The following application and network rules must be added to the allowlist for your firewall depending on which resources you're using.

⚠ Note

For a guide on how to setup UDR with Container Apps to restrict outbound traffic with Azure Firewall, visit the [how to for Container Apps and Azure Firewall](#).

Azure Firewall - Application Rules

Application rules allow or deny traffic based on the application layer. The following outbound firewall application rules are required based on scenario.

Scenarios	FQDNs	Description
All scenarios	<code>mcr.microsoft.com</code> , <code>*.data.mcr.microsoft.com</code>	These FQDNs for Microsoft Container Registry (MCR) are used by Azure Container Apps and either these application rules or the network rules for MCR must be added to the allowlist when using Azure Container Apps with Azure Firewall.
Azure Container Registry (ACR)	<code>Your-ACR-address</code> , <code>*.blob.windows.net</code>	These FQDNs are required when using Azure Container Apps with ACR and Azure Firewall.
Azure Key Vault	<code>Your-Azure-Key-Vault-address</code> , <code>login.microsoft.com</code>	These FQDNs are required in addition to the service tag required for the network rule for Azure Key Vault.
Docker Hub Registry	<code>hub.docker.com</code> , <code>registry-1.docker.io</code> , <code>production.cloudflare.docker.com</code>	If you're using Docker Hub registry and want to access it through the firewall, you need to add these FQDNs to the firewall.

Azure Firewall - Network Rules

Network rules allow or deny traffic based on the network and transport layer. The following outbound firewall network rules are required based on scenario.

Scenarios	Service Tag	Description
All scenarios	<code>MicrosoftContainerRegistry</code> , <code>AzureFrontDoorFirstParty</code>	These Service Tags for Microsoft Container Registry (MCR) are used by Azure Container Apps and either these network rules or the application rules for MCR must be added to the allowlist when using Azure Container Apps with Azure Firewall.
Azure Container Registry (ACR)	<code>AzureContainerRegistry</code>	When using ACR with Azure Container Apps, you'll need to configure these application rules used by Azure Container Registry.
Azure Key Vault	<code>AzureKeyVault</code> , <code>AzureActiveDirectory</code>	These service tags are required in addition to the FQDN for the application rule for Azure Key Vault.

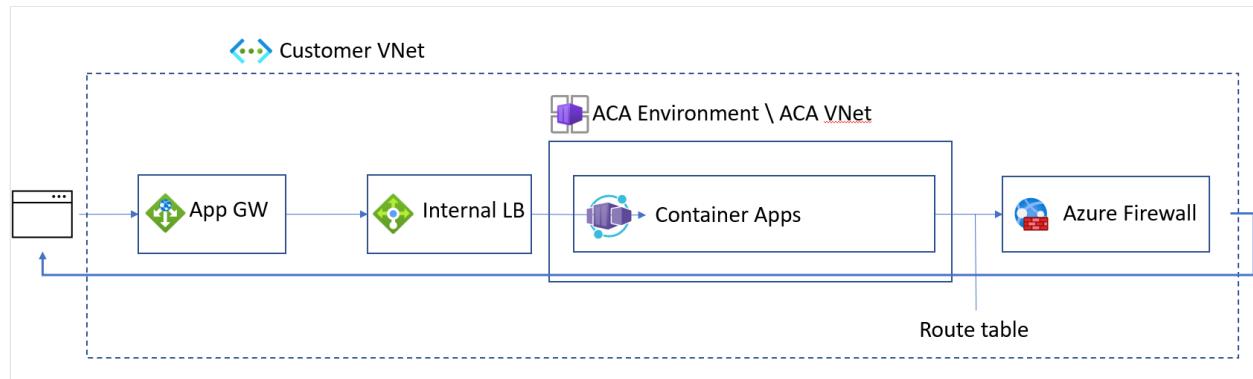
Note

For Azure resources you are using with Azure Firewall not listed in this article, please refer to the [service tags documentation](#).

NAT gateway integration - preview

You can use NAT Gateway to simplify outbound connectivity for your outbound internet traffic in your virtual network on the workload profiles environment. NAT Gateway is used to provide a static public IP address, so when you configure NAT Gateway on your Container Apps subnet, all outbound traffic from your container app is routed through the NAT Gateway's static public IP address.

Lock down your Container App environment



With the workload profiles environment (preview), you can fully secure your ingress/egress networking traffic. To do so, you should use the following features:

- Create your internal container app environment on the workload profiles environment. For steps, see [here](#).
- Integrate your Container Apps with an Application Gateway. For steps, see [here](#).
- Configure UDR to route all traffic through Azure Firewall. For steps, see [here](#).

DNS

- **Custom DNS:** If your VNet uses a custom DNS server instead of the default Azure-provided DNS server, configure your DNS server to forward unresolved DNS queries to 168.63.129.16. [Azure recursive resolvers](#) uses this IP address to resolve requests. When configuring your NSG or Firewall, don't block the 168.63.129.16 address, otherwise, your Container Apps environment won't function.
- **VNet-scope ingress:** If you plan to use VNet-scope [ingress](#) in an internal Container Apps environment, configure your domains in one of the following ways:

1. Non-custom domains: If you don't plan to use custom domains, create a private DNS zone that resolves the Container Apps environment's default domain to the static IP address of the Container Apps environment. You can use [Azure Private DNS](#) or your own DNS server. If you use Azure Private DNS, create a Private DNS Zone named as the Container App environment's default domain (`<UNIQUE_IDENTIFIER>.<REGION_NAME>.azurecontainerapps.io`), with an A record. The A record contains the name `*<DNS Suffix>` and the static IP address of the Container Apps environment.

2. Custom domains: If you plan to use custom domains, use a publicly resolvable domain to [add a custom domain and certificate](#) to the container app. Additionally, create a private DNS zone that resolves the apex domain to the static IP address of the Container Apps environment. You can use [Azure Private DNS](#) or your own DNS server. If you use Azure Private DNS, create a Private DNS Zone named as the apex domain, with an A record that points to the static IP address of the Container Apps environment.

The static IP address of the Container Apps environment can be found in the Azure portal in **Custom DNS suffix** of the container app page or using the Azure CLI `az containerapp env list` command.

Managed resources

When you deploy an internal or an external environment into your own network, a new resource group is created in the Azure subscription where your environment is hosted. This resource group contains infrastructure components managed by the Azure Container Apps platform, and it shouldn't be modified.

Consumption only environment

The name of the resource group created in the Azure subscription where your environment is hosted is prefixed with `mc_` by default, and the resource group name *can't* be customized during container app creation. The resource group contains Public IP addresses used specifically for outbound connectivity from your environment and a load balancer.

In addition to the [Azure Container Apps billing](#), you're billed for:

- One standard static public IP ↗ for egress. If you need more IPs for egress due to SNAT issues, [open a support ticket to request an override ↗](#).

- Two standard [Load Balancers](#) if using an internal environment, or one standard [Load Balancer](#) if using an external environment. Each load balancer has fewer than six rules. The cost of data processed (GB) includes both ingress and egress for management operations.

Workload profiles environment

The name of the resource group created in the Azure subscription where your environment is hosted is prefixed with `ME_` by default, and the resource group name *can* be customized during container app environment creation. For external environments, the resource group contains a public IP address used specifically for inbound connectivity to your external environment and a load balancer. For internal environments, the resource group only contains a Load Balancer.

In addition to the [Azure Container Apps billing](#), you're billed for:

- One standard static [public IP](#) for ingress in external environments and one standard [Load Balancer](#).
- The cost of data processed (GB) includes both ingress and egress for management operations.

Next steps

- [Deploy with an external environment](#)
- [Deploy with an internal environment](#)

Ingress in Azure Container Apps

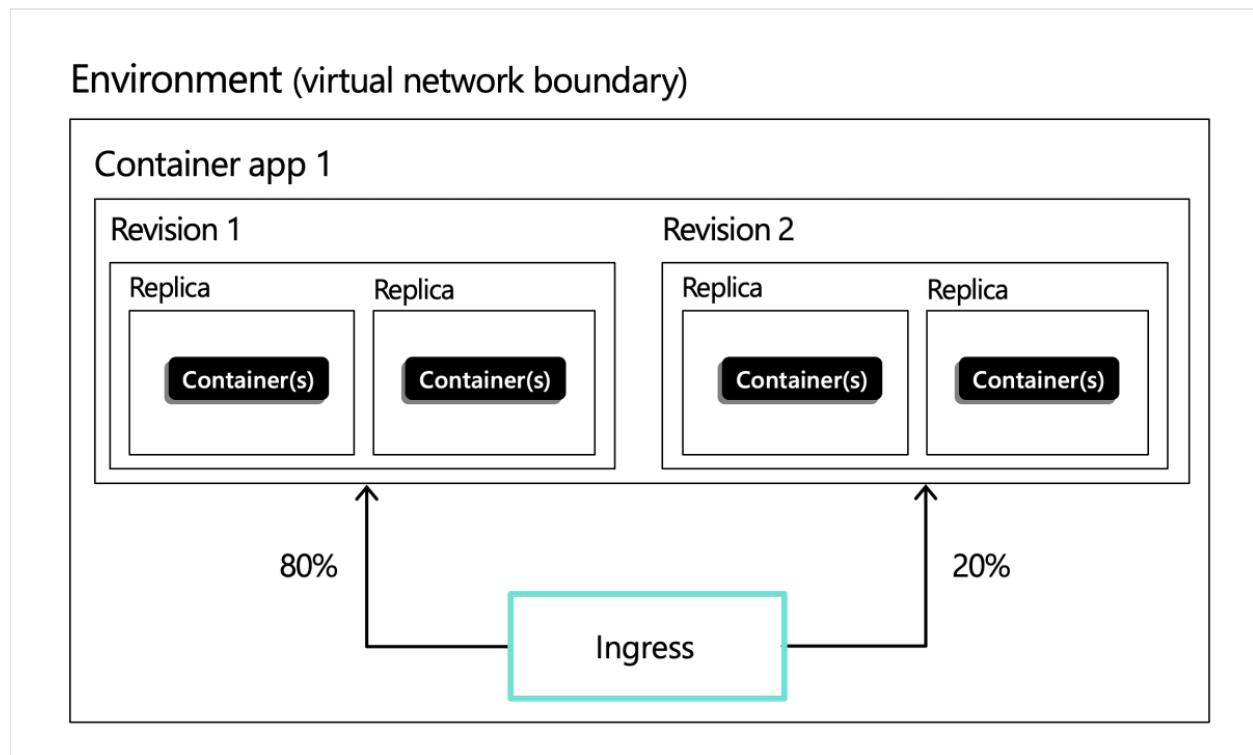
Article • 06/05/2023

Azure Container Apps allows you to expose your container app to the public web, your virtual network (VNET), and other container apps within your environment by enabling ingress. Ingress settings are enforced through a set of rules that control the routing of external and internal traffic to your container app. When you enable ingress, you don't need to create an Azure Load Balancer, public IP address, or any other Azure resources to enable incoming HTTP requests or TCP traffic.

Ingress supports:

- External and internal ingress
- HTTP and TCP ingress types
- Domain names
- IP restrictions
- Authentication
- Traffic splitting between revisions
- Session affinity

Example ingress configuration showing ingress split between two revisions:



For configuration details, see [Configure ingress](#).

External and internal ingress

When you enable ingress, you can choose between two types of ingress:

- External: Accepts traffic from both the public internet and your container app's internal environment.
- Internal: Allows only internal access from within your container app's environment.

Each container app within an environment can be configured with different ingress settings. For example, in a scenario with multiple microservice apps, to increase security you may have a single container app that receives public requests and passes the requests to a background service. In this scenario, you would configure the public-facing container app with external ingress and the internal-facing container app with internal ingress.

Protocol types

Container Apps supports two protocols for ingress: HTTP and TCP.

HTTP

With HTTP ingress enabled, your container app has:

- Support for TLS termination
- Support for HTTP/1.1 and HTTP/2
- Support for WebSocket and gRPC
- HTTPS endpoints that always use TLS 1.2, terminated at the ingress point
- Endpoints that expose ports 80 (for HTTP) and 443 (for HTTPS)
 - By default, HTTP requests to port 80 are automatically redirected to HTTPS on 443
- A fully qualified domain name (FQDN)
- Request timeout is 240 seconds

HTTP headers

HTTP ingress adds headers to pass metadata about the client request to your container app. For example, the `X-Forwarded-Proto` header is used to identify the protocol that the client used to connect with the Container Apps service. The following table lists the HTTP headers that are relevant to ingress in Container Apps:

Header	Description	Values
--------	-------------	--------

Header	Description	Values
X-Forwarded-Proto	Protocol used by the client to connect with the Container Apps service.	http or https
X-Forwarded-For	The IP address of the client that sent the request.	
X-Forwarded-Host	The host name the client used to connect with the Container Apps service.	
X-Forwarded-Cert	The client certificate if <code>clientCertificateMode</code> is set.	Semicolon seperated list of Hash, Cert, and Chain. For example: Hash=....;Cert="...";Chain="..."

TCP

Container Apps supports TCP-based protocols other than HTTP or HTTPS. For example, you can use TCP ingress to expose a container app that uses the [Redis protocol](#).

ⓘ Note

External TCP ingress is only supported for Container Apps environments that use a [custom VNET](#).

With TCP ingress enabled, your container app:

- Is accessible to other container apps in the same environment via its name (defined by the `name` property in the Container Apps resource) and exposed port number.
- Is accessible externally via its fully qualified domain name (FQDN) and exposed port number if the ingress is set to "external".

Domain names

You can access your app in the following ways:

- The default fully-qualified domain name (FQDN): Each app in a Container Apps environment is automatically assigned an FQDN based on the environment's DNS

suffix. To customize an environment's DNS suffix, see [Custom environment DNS Suffix](#).

- A custom domain name: You can configure a custom DNS domain for your Container Apps environment. For more information, see [Custom domain names and certificates](#).
- The app name: You can use the app name for communication between apps in the same environment.

To get the FQDN for your app, see [Location](#).

IP restrictions

Container Apps supports IP restrictions for ingress. You can create rules to either configure IP addresses that are allowed or denied access to your container app. For more information, see [Configure IP restrictions](#).

Authentication

Azure Container Apps provides built-in authentication and authorization features to secure your external ingress-enabled container app. For more information, see [Authentication and authorization in Azure Container Apps](#).

You can configure your app to support client certificates (mTLS) for authentication and traffic encryption. For more information, see [Configure client certificates](#)

Traffic splitting

Containers Apps allows you to split incoming traffic between active revisions. When you define a splitting rule, you assign the percentage of inbound traffic to go to different revisions. For more information, see [Traffic splitting](#).

Session affinity

Session affinity, also known as sticky sessions, is a feature that allows you to route all HTTP requests from a client to the same container app replica. This feature is useful for stateful applications that require a consistent connection to the same replica. For more information, see [Session affinity](#).

Cross origin resource sharing (CORS)

By default, any requests made through the browser from a page to a domain that doesn't match the page's origin domain are blocked. To avoid this restriction for services deployed to Container Apps, you can enable cross-origin resource sharing (CORS).

For more information, see [Configure CORS in Azure Container Apps](#).

Next steps

[Configure ingress](#)

Provide a virtual network to an external Azure Container Apps environment

Article • 04/07/2023

The following example shows you how to create a Container Apps environment in an existing virtual network.

Begin by signing in to the [Azure portal](#).

Create a container app

To create your container app, start at the Azure portal home page.

1. Search for **Container Apps** in the top search bar.
2. Select **Container Apps** in the search results.
3. Select the **Create** button.

Basics tab

In the *Basics* tab, do the following actions.

1. Enter the following values in the *Project details* section.

Setting	Action
Subscription	Select your Azure subscription.
Resource group	Select Create new and enter my-container-apps .
Container app name	Enter my-container-app .

Create an environment

Next, create an environment for your container app.

1. Select the appropriate region.

Setting	Value
Region	Select Central US .

2. In the *Create Container Apps environment* field, select the **Create new** link.

3. In the *Create Container Apps Environment* page on the **Basics** tab, enter the following values:

Setting	Value
Environment name	Enter my-environment .
Zone redundancy	Select Disabled

4. Select the **Monitoring** tab to create a Log Analytics workspace.
5. Select the **Create new** link in the *Log Analytics workspace* field and enter the following values.

Setting	Value
Name	Enter my-container-apps-logs .

The *Location* field is pre-filled with *Central US* for you.

6. Select **OK**.

 **Note**

You can use an existing virtual network, but a dedicated subnet with a CIDR range of /23 or larger is required for use with Container Apps when using the Consumption only Architecture. When using the Workload Profiles Architecture, a /27 or larger is required. To learn more about subnet sizing, see the [networking architecture overview](#).

7. Select the **Networking** tab to create a VNET.
8. Select **Yes** next to *Use your own virtual network*.
9. Next to the *Virtual network* box, select the **Create new** link and enter the following value.

Setting	Value
Name	Enter my-custom-vnet .

10. Select the **OK** button.
11. Next to the *Infrastructure subnet* box, select the **Create new** link and enter the following values:

Setting	Value
Subnet Name	Enter infrastructure-subnet .
Virtual Network Address Block	Keep the default values.
Subnet Address Block	Keep the default values.

12. Select the **OK** button.
13. Under *Virtual IP*, select **External**.
14. Select **Create**.

Deploy the container app

1. Select the **Review and create** button at the bottom of the page.

Next, the settings in the Container App are verified. If no errors are found, the **Create** button is enabled.

If there are errors, any tab containing errors is marked with a red dot. Navigate to the appropriate tab. Fields containing an error will be highlighted in red. Once all errors are fixed, select **Review and create** again.

2. Select **Create**.

A page with the message *Deployment is in progress* is displayed. Once the deployment is successfully completed, you'll see the message: *Your deployment is complete*.

Clean up resources

If you're not going to continue to use this application, you can delete the Azure Container Apps instance and all the associated services by removing the **my-container-apps** resource group. Deleting this resource group will also delete the resource group automatically created by the Container Apps service containing the custom network components.

Next steps

[Managing autoscaling behavior](#)

Provide a virtual network to an internal Azure Container Apps environment

Article • 05/15/2023

The following example shows you how to create a Container Apps environment in an existing virtual network.

Begin by signing in to the [Azure portal](#).

Create a container app

To create your container app, start at the Azure portal home page.

1. Search for **Container Apps** in the top search bar.
2. Select **Container Apps** in the search results.
3. Select the **Create** button.

Basics tab

In the *Basics* tab, do the following actions.

1. Enter the following values in the *Project details* section.

Setting	Action
Subscription	Select your Azure subscription.
Resource group	Select Create new and enter my-container-apps .
Container app name	Enter my-container-app .

Create an environment

Next, create an environment for your container app.

1. Select the appropriate region.

Setting	Value
Region	Select Central US .

2. In the *Create Container Apps environment* field, select the **Create new** link.

3. In the *Create Container Apps Environment* page on the **Basics** tab, enter the following values:

Setting	Value
Environment name	Enter my-environment .
Zone redundancy	Select Disabled

4. Select the **Monitoring** tab to create a Log Analytics workspace.
5. Select the **Create new** link in the *Log Analytics workspace* field and enter the following values.

Setting	Value
Name	Enter my-container-apps-logs .

The *Location* field is pre-filled with *Central US* for you.

6. Select **OK**.

 **Note**

You can use an existing virtual network, but a dedicated subnet with a CIDR range of **/23** or larger is required for use with Container Apps when using the Consumption only environment. When using the Workload Profiles environment, a **/27** or larger is required. To learn more about subnet sizing, see the [networking environment overview](#).

7. Select the **Networking** tab to create a VNET.
8. Select **Yes** next to *Use your own virtual network*.
9. Next to the *Virtual network* box, select the **Create new** link and enter the following value.

Setting	Value
Name	Enter my-custom-vnet .

10. Select the **OK** button.
11. Next to the *Infrastructure subnet* box, select the **Create new** link and enter the following values:

Setting	Value
Subnet Name	Enter infrastructure-subnet .
Virtual Network Address Block	Keep the default values.
Subnet Address Block	Keep the default values.

12. Select the **OK** button.

13. Under *Virtual IP*, select **Internal**.

14. Select **Create**.

Deploy the container app

1. Select the **Review and create** button at the bottom of the page.

Next, the settings in the Container App are verified. If no errors are found, the **Create** button is enabled.

If there are errors, any tab containing errors is marked with a red dot. Navigate to the appropriate tab. Fields containing an error will be highlighted in red. Once all errors are fixed, select **Review and create** again.

2. Select **Create**.

A page with the message *Deployment is in progress* is displayed. Once the deployment is successfully completed, you'll see the message: *Your deployment is complete*.

Clean up resources

If you're not going to continue to use this application, you can delete the Azure Container Apps instance and all the associated services by removing the **my-container-apps** resource group. Deleting this resource group will also delete the resource group automatically created by the Container Apps service containing the custom network components.

Additional resources

- To use VNET-scope ingress, you must set up [DNS](#).

Next steps

[Managing autoscaling behavior](#)

Securing a custom VNET in Azure Container Apps with Network Security Groups

Article • 05/15/2023

Network Security Groups (NSGs) needed to configure virtual networks closely resemble the settings required by Kubernetes.

You can lock down a network via NSGs with more restrictive rules than the default NSG rules to control all inbound and outbound traffic for the Container Apps environment at the subscription level.

In the workload profiles environment, user-defined routes (UDRs) and securing outbound traffic with a firewall are supported. Learn more in the [networking concepts document](#).

In the Consumption only environment, custom user-defined routes (UDRs) and ExpressRoutes aren't supported.

NSG allow rules

The following tables describe how to configure a collection of NSG allow rules.

ⓘ Note

The subnet associated with a Container App Environment on the Consumption only environment requires a CIDR prefix of /23 or larger. On the workload profiles environment (preview), a /27 or larger is required.

Inbound

Protocol	Port	ServiceTag	Description
Any	*	Infrastructure subnet address space	Allow communication between IPs in the infrastructure subnet. This address is passed as a parameter when you create an environment. For example, 10.0.0.0/21.
Any	*	AzureLoadBalancer	Allow the Azure infrastructure load balancer to communicate with your environment.

Outbound with service tags

The following service tags are required when using NSGs on the Consumption only environment:

Protocol	Port	ServiceTag	Description
UDP	1194	AzureCloud.<REGION>	Required for internal AKS secure connection between underlying nodes and control plane. Replace <REGION> with the region where your container app is deployed.
TCP	9000	AzureCloud.<REGION>	Required for internal AKS secure connection between underlying nodes and control plane. Replace <REGION> with the region where your container app is deployed.
TCP	443	AzureMonitor	Allows outbound calls to Azure Monitor.

The following service tags are required when using NSGs on the workload profiles environment:

ⓘ Note

If you are using Azure Container Registry (ACR) with NSGs configured on your virtual network, create a private endpoint on your ACR to allow Container Apps to pull images through the virtual network.

Protocol	Port	Service Tag	Description
TCP	443	MicrosoftContainerRegistry	This is the service tag for container registry for microsoft containers.
TCP	443	AzureFrontDoor.FirstParty	This is a dependency of the MicrosoftContainerRegistry service tag.

Outbound with wild card IP rules

The following IP rules are required when using NSGs on both the Consumption only environment and the workload profiles environment:

Protocol	Port	IP	Description
TCP	443	*	Allowing all outbound on port 443 provides a way to allow all FQDN based outbound dependencies that don't have a static IP.

Protocol	Port	IP	Description
UDP	123	*	NTP server.
TCP	5671	*	Container Apps control plane.
TCP	5672	*	Container Apps control plane.
Any	*	Infrastructure subnet address space	Allow communication between IPs in the infrastructure subnet. This address is passed as a parameter when you create an environment. For example, <code>10.0.0.0/21</code> .

Considerations

- If you're running HTTP servers, you might need to add ports `80` and `443`.
- Adding deny rules for some ports and protocols with lower priority than `65000` may cause service interruption and unexpected behavior.
- Don't explicitly deny the Azure DNS address `168.63.128.16` in the outgoing NSG rules, or your Container Apps environment won't be able to function.

Network proxying in Azure Container Apps

Article • 12/07/2022

Azure Container Apps uses [Envoy](#) as a network proxy. Network requests are proxied in Azure Container Apps to achieve the following capabilities:

- **Allow apps to scale to zero:** Running instances are required for direct calls to an application. If an app scales to zero, then a direct request would fail. With proxying, Azure Container Apps ensures calls to an app have running instances to resolve the request.
- **Achieve load balancing:** As requests come in Azure Container Apps applies load balancing rules spread requests across container replicas.

Ports and routing

In Container Apps, Envoy listens the following ports to decide which container app to route traffic.

Type	Request	IP type	Port number	Internal port number
Public	Endpoint	Public	80	8080
Public	VNET	Public	443	4430
Internal	Endpoint	Cluster	80	8081
Internal	VNET	Cluster	443	8443

Requests that come in to ports 80 and 443 are internally routed to the appropriate internal port depending on the request type.

Security

- HTTP requests are automatically redirected to HTTPS
- Envoy terminates TLS after crossing its boundary
 - Envoy sends requests to apps over HTTP in plain text
- mTLS is only available when using Dapr
 - When you use Dapr service invocation APIs, mTLS is enabled. However, because Envoy terminates mTLS, inbound calls from Envoy to Dapr-enabled container

apps isn't encrypted.

HTTPs, GRPC, and HTTP/2 all follow the same architectural model.

Timeouts

Network requests timeout after four minutes

Networking

Observability in Azure Container Apps

Article • 03/26/2023

Azure Container Apps provides several built-in observability features that together give you a holistic view of your container app's health throughout its application lifecycle. These features help you monitor and diagnose the state of your app to improve performance and respond to trends and critical problems.

These features include:

Feature	Description
Log streaming	View streaming system and console logs from a container in near real-time.
Container console	Connect to the Linux console in your containers to debug your application from inside the container.
Azure Monitor metrics	View and analyze your application's compute and network usage through metric data.
Application logging	Monitor, analyze and debug your app using log data.
Azure Monitor Log Analytics	Run queries to view and analyze your app's system and application logs.
Azure Monitor alerts	Create and manage alerts to notify you of events and conditions based on metric and log data.

ⓘ Note

While not a built-in feature, [Azure Monitor Application Insights](#) is a powerful tool to monitor your web and background applications. Although Container Apps doesn't support the Application Insights auto-instrumentation agent, you can instrument your application code using Application Insights SDKs.

Application lifecycle observability

With Container Apps observability features, you can monitor your app throughout the development-to-production lifecycle. The following sections describe the most effective monitoring features for each phase.

Development and test

During the development and test phase, real-time access to your containers' application logs and console is critical for debugging issues. Container Apps provides:

- [Log streaming](#): View real-time log streams from your containers.
- [Container console](#): Access the container console to debug your application.

Deployment

Once you deploy your container app, continuous monitoring helps you quickly identify problems that may occur around error rates, performance, and resource consumption.

Azure Monitor gives you the ability to track your app with the following features:

- [Azure Monitor metrics](#): Monitor and analyze key metrics.
- [Azure Monitor alerts](#): Receive alerts for critical conditions.
- [Azure Monitor Log Analytics](#): View and analyze application logs.

Maintenance

Container Apps manages updates to your container app by creating [revisions](#). You can run multiple revisions concurrently in blue green deployments or to perform A/B testing. These observability features will help you monitor your app across revisions:

- [Azure Monitor metrics](#): Monitor and compare key metrics for multiple revisions.
- [Azure Monitor alerts](#): Receive individual alerts per revision.
- [Azure Monitor Log Analytics](#): View, analyze and compare log data for multiple revisions.

Next steps

[Health probes in Azure Container Apps](#)

Application Logging in Azure Container Apps

Article • 03/26/2023

Azure Container Apps provides two types of application logging categories:

- [Container console logs](#): Log streams from your container console.
- [System logs](#): Logs generated by the Azure Container Apps service.

You can view the [log streams](#) in near real-time in the Azure portal or CLI. For more options to store and monitor your logs, see [Logging options](#).

Container console Logs

Container Apps captures the `stdout` and `stderr` output streams from your application containers and displays them as console logs. When you implement logging in your application, you can troubleshoot problems and monitor the health of your app.

System logs

Container Apps generates system logs to inform you of the status of service level events. Log messages include the following information:

- Successfully created dapr component
- Successfully updated dapr component
- Error creating dapr component
- Successfully mounted volume
- Error mounting volume
- Successfully bound Domain
- Auth enabled on app
- Creating authentication config
- Auth config created successfully
- Setting a traffic weight
- Creating a new revision:
- Successfully provisioned revision
- Deactivating Old revisions
- Error provisioning revision

Next steps

[Logging options](#)

Log storage and monitoring options in Azure Container Apps

Article • 03/26/2023

Azure Container Apps gives you options for storing and viewing your application logs. Logging options are configured in your Container Apps environment where you select the log destination.

Container Apps application logs consist of two different categories:

- Container console output (`stdout/stderr`) messages.
- System logs generated by Azure Container Apps.
- Spring App console logs.

You can choose between these logs destinations:

- **Log Analytics:** Azure Monitor Log Analytics is the default storage and viewing option. Your logs are stored in a Log Analytics workspace where they can be viewed and analyzed using Log Analytics queries. To learn more about Log Analytics, see [Azure Monitor Log Analytics](#).
- **Azure Monitor:** Azure Monitor routes logs to one or more destinations:
 - Log Analytics workspace for viewing and analysis.
 - Azure storage account to archive.
 - Azure event hub for data ingestion and analytic services. For more information, see [Azure Event Hubs](#).
 - An Azure partner monitoring solution such as, Datadog, Elastic, Logz.io and others. For more information, see [Partner solutions](#).
- **None:** You can disable the storage of log data. When disabled, you can still view real-time container logs via the **Logs stream** feature in your container app. For more information, see [Log streaming](#).

ⓘ Note

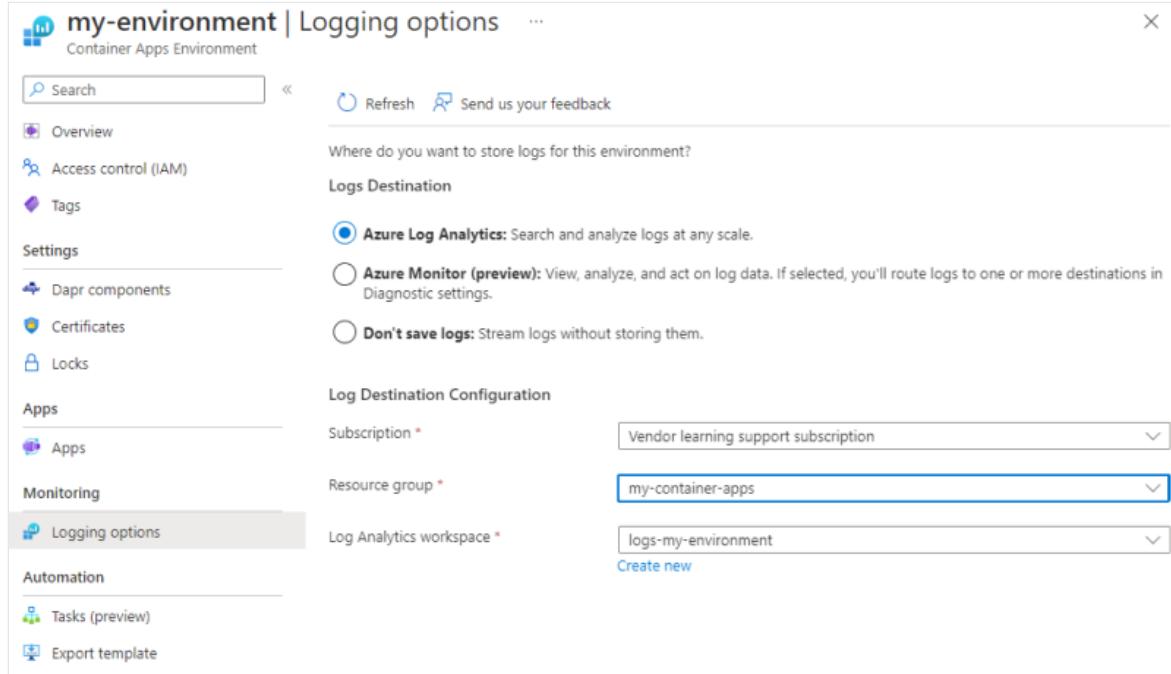
Azure Monitor is not currently supported in the Consumption + Dedicated plan structure.

When *None* or the *Azure Monitor* destination is selected, the **Logs** menu item providing the Log Analytics query editor in the Azure portal is disabled.

Configure options via the Azure portal

Use these steps to configure the logging options for your Container Apps environment in the Azure portal:

1. Go to the **Logging Options** on your Container Apps environment window in the portal.

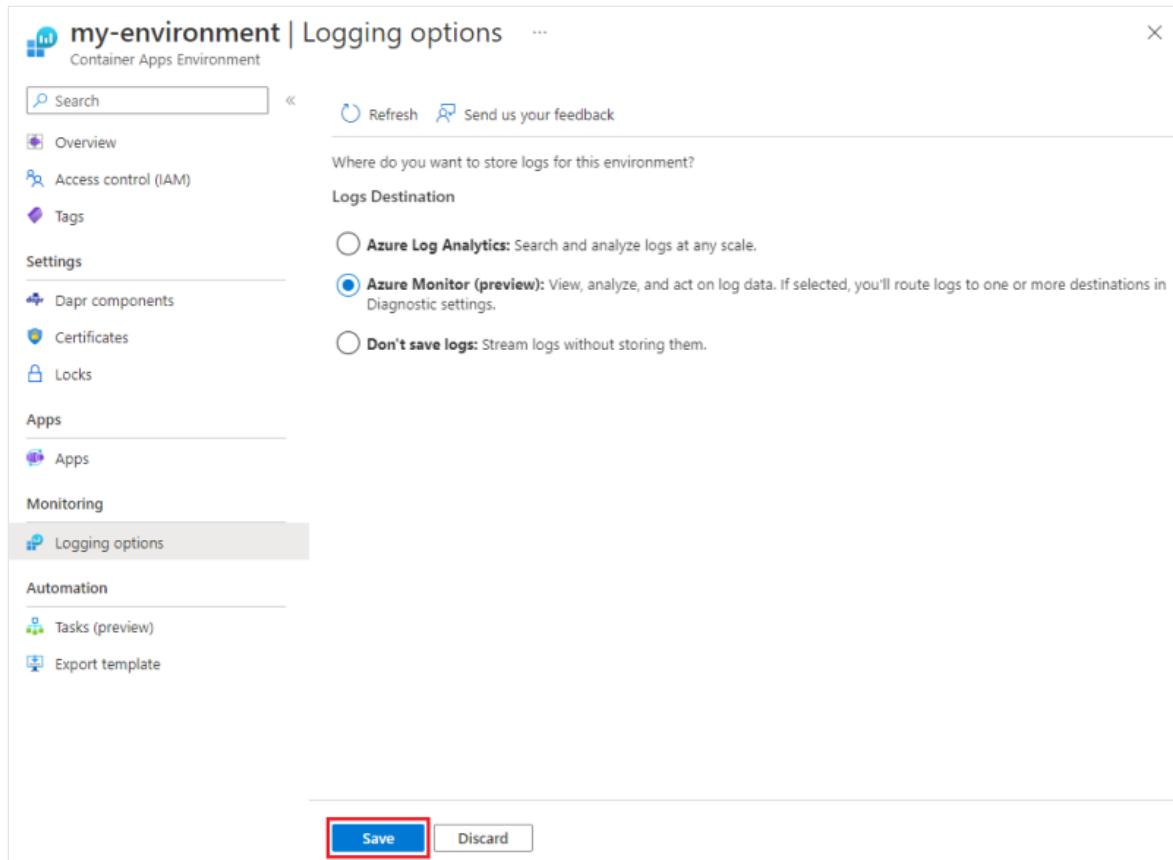


The screenshot shows the 'Logging options' section of the Azure Container Apps environment 'my-environment'. On the left, there's a sidebar with links like Overview, Access control (IAM), Tags, Settings (selected), Dapr components, Certificates, Locks, Apps (selected), Monitoring, Logging options (selected), Automation, Tasks (preview), and Export template. The main area has a heading 'Logs Destination' with three options: 'Azure Log Analytics' (selected), 'Azure Monitor (preview)', and 'Don't save logs'. Below this is a 'Log Destination Configuration' section with fields for Subscription (Vendor learning support subscription), Resource group (my-container-apps selected), and Log Analytics workspace (logs-my-environment selected). There's also a 'Create new' button.

2. You can choose from the following **Logs Destination** options:

- **Log Analytics:** With this option, you select a Log Analytics workspace to store your log data. Your logs can be viewed through Log Analytics queries. To learn more about Log Analytics, see [Azure Monitor Log Analytics](#).
- **Azure Monitor:** Azure Monitor routes your logs to a destination. When you select this option, you must select **Diagnostic settings** to complete the configuration after you select **Save** on this page.
- **None:** This option disables the storage of log data.

3. Select Save.



4. If you have selected **Azure Monitor** as your logs destination, you must configure **Diagnostic settings**. The **Diagnostic settings** item appears below the **Logging options** menu item.

Diagnostic settings

When you select **Azure Monitor** as your logs destination, you must configure the destination details. Select **Diagnostic settings** from the left side menu of the Container Apps Environment window in the portal.

my-environment | Logging options

Container Apps Environment

Search Refresh Send us your feedback

Open Diagnostic Settings and select one or more log destinations →

Where do you want to store logs for this environment?

Logs Destination

Azure Log Analytics: Search and analyze logs at any scale.

Azure Monitor (preview): View, analyze, and act on log data. If selected, you'll route logs to one or more destinations in Diagnostic settings.

Don't save logs: Stream logs without storing them.

Overview Access control (IAM) Tags

Settings

Dapr components Certificates Locks

Apps

Apps

Monitoring

Logging options **Diagnostic settings**

Automation

Tasks (preview)

Export template

Destination details are saved as *diagnostic settings*. You can create up to five diagnostic settings for your container app environment. You can configure different log categories for each diagnostic setting. For example, create one diagnostic setting to send the system logs category to one destination, and another to send the container console logs category to another destination.

To create a new *diagnostic setting*:

1. Select Add diagnostic setting.

my-environment | Diagnostic settings

Container Apps Environment

Search Refresh Feedback

Diagnostic settings are used to configure streaming export of platform logs and metrics for a resource to the destination of your choice. You may create up to five different diagnostic settings to send different logs and metrics to independent destinations. [Learn more about diagnostic settings](#)

Name	Storage account	Event hub	Log Analytics works...	Partner solution	Edit setting
No diagnostic settings defined					

+ Add diagnostic setting

Click 'Add Diagnostic setting' above to configure the collection of the following data:

- Container App console logs
- Container App system logs

Overview Access control (IAM) Tags

Settings

Dapr components Certificates Locks

Apps

Apps

Monitoring

Logging options **Diagnostic settings**

Automation

Tasks (preview)

Export template

2. Enter a name for your diagnostic setting.

The screenshot shows the 'Diagnostic setting' configuration page. At the top, there are buttons for 'Save', 'Discard', 'Delete', and 'Feedback'. Below that, a descriptive text about diagnostic settings is shown. A red box highlights the 'Diagnostic setting name *' input field, which contains a placeholder. Another red box highlights the 'Logs' section, specifically the 'Category groups' and 'Categories' sections. The 'Category groups' section has two options: 'audit' and 'allLogs'. The 'Categories' section has two options: 'Container App console logs' and 'Container App system logs'. To the right, under 'Destination details', there are four checkboxes: 'Send to Log Analytics workspace', 'Archive to a storage account', 'Stream to an event hub', and 'Send to partner solution'. The 'Send to Log Analytics workspace' checkbox is unselected.

3. Select the log **Category groups** or **Categories** you want to send to this destination. You can select one or more categories.

4. Select one or more **Destination details**:

- **Send to Log Analytics workspace:** Select from existing Log Analytics workspaces.

The screenshot shows the same 'Diagnostic setting' configuration page as before, but with different selections. The 'Diagnostic setting name *' field now contains 'console-logs-log-analytics'. In the 'Logs' section, the 'Container App console logs' checkbox is selected. In the 'Destination details' section, the 'Send to Log Analytics workspace' checkbox is selected, and its dropdown shows 'Subscription: Demo-Subscription' and 'Log Analytics workspace: workspace-mycontainerappsworKsHpe (canadacentral)'. Other destination options like 'Archive to a storage account' and 'Stream to an event hub' are unselected.

- **Archive to a storage account:** You can choose from existing storage accounts. When the individual log categories are selected, you can set the

Retention (days) for each category.

The screenshot shows the 'Diagnostic setting' configuration page. Under 'Logs', 'Container App console logs' and 'Container App system logs' are selected with retention policies of 365 and 90 days respectively. On the right, under 'Destination details', 'Archive to a storage account' is selected, and a storage account named 'mystorageaccount' is chosen. Other destination options like 'Event hub' and 'Log Analytics workspace' are also shown.

- Stream to an event hub: Select from Azure event hubs.

The screenshot shows the 'Diagnostic setting' configuration page with a diagnostic setting name 'all-logs-to-event-hub'. Under 'Logs', 'allLogs' is selected. On the right, under 'Destination details', 'Stream to an event hub' is selected, and configuration fields for 'Subscription', 'Event hub namespace', 'Event hub name', and 'Event hub policy name' are filled with 'Demo-Subscription', 'my-event-hub-name-space', 'my-event-hub', and 'my-event-hub-policy' respectively.

- Send to a partner solution: Select from Azure partner solutions.

5. Select Save.

For more information about Diagnostic settings, see [Diagnostic settings in Azure Monitor](#).

Configure options using the Azure CLI

Configure logs destination for your Container Apps environment using the Azure CLI `az containerapp create` and `az containerapp update` commands with the `--logs-destination` argument.

The destination values are: `log-analytics`, `azure-monitor`, and `none`.

For example, to create a Container Apps environment using an existing Log Analytics workspace as the logs destination, you must provide the `--logs-destination` argument with the value `log-analytics` and the `--logs-destination-id` argument with the value of the Log Analytics workspace resource ID. You can get the resource ID from the Log Analytics workspace page in the Azure portal or from the `az monitor log-analytics workspace show` command.

Replace <PLACEHOLDERS> with your values:

Azure CLI

```
az containerapp env create \
--name <ENVIRONMENT_NAME> \
--resource-group <RESOURCE_GROUP> \
--logs-destination log-analytics \
--logs-workspace-id <WORKSPACE_ID>
```

To update an existing Container Apps environment to use Azure Monitor as the logs destination:

Replace <PLACEHOLDERS> with your values:

Azure CLI

```
az containerapp env update \
--name <ENVIRONMENT_NAME> \
--resource-group <RESOURCE_GROUP> \
--logs-destination azure-monitor
```

When `--logs-destination` is set to `azure-monitor`, create diagnostic settings to configure the destination details for the log categories with the `az monitor diagnostics-settings` command.

For more information about Azure Monitor diagnostic settings commands, see [az monitor diagnostic-settings](#). Container Apps log categories are `ContainerAppConsoleLogs` and `ContainerAppSystemLogs`.

Next steps

[Monitor logs with Log Analytics](#)

View log streams in Azure Container Apps

Article • 03/26/2023

While developing and troubleshooting your container app, it's essential to see the logs for your container app in real time. Azure Container Apps lets you stream:

- system logs from the Container Apps environment and your container app.
- container console logs from your container app.

Log streams are accessible through the Azure portal or the Azure CLI.

View log streams via the Azure portal

You can view system logs and console logs in the Azure portal. System logs are generated by the container app's runtime. Console logs are generated by your container app.

Environment system log stream

To troubleshoot issues in your container app environment, you can view the system log stream from your environment page. The log stream displays the system logs for the Container Apps service and the apps actively running in the environment:

1. Go to your environment in the Azure portal.
2. Select Log stream under the *Monitoring* section on the sidebar menu.

```
Connecting...
{"TimeStamp": "2023-02-13T21:40:09Z", "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Connecting... to the events collector...", "Reason": "StartingGettingEvents", "EventSource": "ContainerAppController", "Count": 1}
("TimeStamp": "2023-02-13T21:40:10Z", "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Successfully connected to events server", "Reason": "ConnectedToEventsServer", "EventSource": "ContainerAppController", "Count": 1)
("TimeStamp": null, "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Metrics server is starting to listen", "Reason": null, "EventSource": null, "Count": 0)
("TimeStamp": null, "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Starting server", "Reason": null, "EventSource": null, "Count": 0)
("TimeStamp": null, "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Starting EventSource", "Reason": null, "EventSource": null, "Count": 0)
("TimeStamp": null, "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Starting Controller", "Reason": null, "EventSource": null, "Count": 0)
("TimeStamp": null, "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Starting workers", "Reason": null, "EventSource": null, "Count": 0)
("TimeStamp": "2023-02-13 09:05:01 \u00d70280000 UTC", "Type": "Normal", "ContainerAppName": "album-api", "RevisionName": "album-api-74e9b5", "ReplicaName": "", "Msg": "Started scalers watch", "Reason": "KEDAScalarsStarted", "EventSource": "KEDA", "Count": 1}
("TimeStamp": "2023-02-13T21:41:10Z", "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "No events since last 60 seconds", "Reason": "NoNewEvents", "EventSource": "ContainerAppController", "Count": 1)
("TimeStamp": "2023-02-13T21:42:10Z", "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "No events since last 60 seconds", "Reason": "NoNewEvents", "EventSource": "ContainerAppController", "Count": 1)
("TimeStamp": "2023-02-13T21:43:11Z", "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "No events since last 60 seconds", "Reason": "NoNewEvents", "EventSource": "ContainerAppController", "Count": 1)
("TimeStamp": "2023-02-13T21:44:11Z", "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "No events since last 60 seconds", "Reason": "NoNewEvents", "EventSource": "ContainerAppController", "Count": 1)
```

Container app log stream

You can view a log stream of your container app's system or console logs from your container app page.

1. Go to your container app in the Azure portal.
2. Select **Log stream** under the *Monitoring* section on the sidebar menu.
3. To view the console log stream, select **Console**.
 - a. If you have multiple revisions, replicas, or containers, you can select from the drop-down menus to choose a container. If your app has only one container, you can skip this step.

Home > album-api

album-api | Log stream

Container App

Search Custom domains

Dapr Identity Service Connector (preview) Locks

Application Revision management Containers Scale and replicas

Monitoring Alerts Metrics Logs **Log stream** Console Advisor recommendations

Support + troubleshooting New Support Request

Logs **Console** System

Replica

Container

Stop Copy Clear

Connecting...

```
2023-02-13T21:56:00.96290 Connecting to the container 'album-api'...
2023-02-13T21:56:00.98787 Successfully Connected to container: 'album-api' [Revision: 'album-api--vgirtv5', Replica: 'album-api--vgirtv5-56f4bb96db-5lh8t']
2023-02-13T21:54:57.1817613372 [Info]: Microsoft.Hosting.Lifetime[14]
2023-02-13T21:54:57.1818013372 Now listening on: http://[::]:3500
2023-02-13T21:54:57.1829765552 [Info]: Microsoft.Hosting.Lifetime[0]
2023-02-13T21:54:57.1829923562 Application started. Press Ctrl+C to shut down.
2023-02-13T21:54:57.1836716662 [Info]: Microsoft.Hosting.Lifetime[0]
2023-02-13T21:54:57.1836794662 Hosting environment: Production
2023-02-13T21:54:57.1836820662 [Info]: Microsoft.Hosting.Lifetime[0]
2023-02-13T21:54:57.1836847672 Content root path: /app/
2023-02-13T21:57:01.39368 No logs since last 60 seconds
2023-02-13T21:58:01.85126 No logs since last 60 seconds
2023-02-13T21:59:02.16494 No logs since last 60 seconds
2023-02-13T22:00:02.50087 No logs since last 60 seconds
```

4. To view the system log stream, select **System**. The system log stream displays the system logs for all running containers in your container app.

Home > album-api

album-api | Log stream

Container App

Search Custom domains

Dapr Identity Service Connector (preview) Locks

Application Revision management Containers Scale and replicas

Monitoring Alerts Metrics Logs **Log stream** Console Advisor recommendations

Support + troubleshooting New Support Request

Logs **System** Console

Stop Copy Clear

Connecting...

```
{"TimeStamp": "2023-02-13T22:06:40Z", "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Connecting to the events collector...", "Reason": "StartingGettingEvents", "EventSource": "ContainerAppController", "Count": 1}
{"TimeStamp": "2023-02-13T22:06:40Z", "Type": "Normal", "ContainerAppName": null, "RevisionName": null, "ReplicaName": null, "Msg": "Successfully connected to events server", "Reason": "ConnectedToEventsServer", "EventSource": "ContainerAppController", "Count": 1}
{"TimeStamp": "2023-02-13 21:54:56 \u002B0000 UTC", "Type": "Normal", "ContainerAppName": "album-api", "RevisionName": "album-api--vgirtv5", "ReplicaName": "album-api--vgirtv5-56f4bb96db-k84sn", "Msg": "Successfully pulled image \u0022ca704c3e9d38acr.azurecr.io:/album-api:20230207182936040366\u0022 in 3.271844754s", "Reason": "ImagePulled", "EventSource": "ContainerAppController", "Count": 1}
{"TimeStamp": "2023-02-13 21:54:56 \u002B0000 UTC", "Type": "Normal", "ContainerAppName": "album-api", "RevisionName": "album-api--vgirtv5", "ReplicaName": "album-api--vgirtv5-56f4bb96db-xzjs5", "Msg": "Started container album-api", "Reason": "ContainerStarted", "EventSource": "ContainerAppController", "Count": 1}
{"TimeStamp": "2023-02-13 21:54:56 \u002B0000 UTC", "Type": "Normal", "ContainerAppName": "album-api", "RevisionName": "album-api--vgirtv5", "ReplicaName": "album-api--vgirtv5-56f4bb96db-k84sn", "Msg": "Created container album-api", "Reason": "ContainerCreated", "EventSource": "ContainerAppController", "Count": 1}
{"TimeStamp": "2023-02-13 21:54:56 \u002B0000 UTC", "Type": "Normal", "ContainerAppName": "album-api", "RevisionName": "album-api--vgirtv5", "ReplicaName": "album-api--vgirtv5-56f4bb96db-5lh8t", "Msg": "Started container album-api", "Reason": "ContainerStarted", "EventSource": "ContainerAppController", "Count": 1}
{"TimeStamp": "2023-02-13 21:54:56 \u002B0000 UTC", "Type": "Normal", "ContainerAppName": "album-api", "RevisionName": "album-api--vgirtv5", "ReplicaName": "album-api--vgirtv5-56f4bb96db-k84sn", "Msg": "Started container album-api", "Reason": "ContainerStarted", "EventSource": "ContainerAppController", "Count": 1}
 {"TimeStamp": "2023-02-13 21:55:25 \u002B0000 UTC", "Type": "Normal", "ContainerAppName": "album-api", "RevisionName": "album-api--m7v4e9b", "ReplicaName": "", "Msg": "Stopped scalers watch", "Reason": "KEDAScalarsStopped", "EventSource": "KEDA", "Count": 1}
 {"TimeStamp": "2023-02-13 21:55:25 \u002B0000 UTC", "Type": "Normal", "ContainerAppName": "album-api", "RevisionName": "album-api--m7v4e9b", "ReplicaName": "", "Msg": "ScaledObject was deleted", "Reason": "ScaledObjectDeleted", "EventSource": "KEDA", "Count": 1}
```

View log streams via the Azure CLI

You can view your container app's log streams from the Azure CLI with the `az containerapp logs show` command or your container app's environment system log stream with the `az containerapp env logs show` command.

Control the log stream with the following arguments:

- `--tail` (Default) View the last n log messages. Values are 0-300 messages. The default is 20.
- `--follow` View a continuous live stream of the log messages.

Stream Container app logs

You can stream the system or console logs for your container app. To stream the container app system logs, use the `--type` argument with the value `system`. To stream the container console logs, use the `--type` argument with the value `console`. The default is `console`.

View container app system log stream

This example uses the `--tail` argument to display the last 50 system log messages from the container app. Replace the <placeholders> with your container app's values.

Bash

Azure CLI

```
az containerapp logs show \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--type system \
--tail 50
```

This example displays a continuous live stream of system log messages from the container app using the `--follow` argument. Replace the <placeholders> with your container app's values.

Bash

Azure CLI

```
az containerapp logs show \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--type system \
--follow
```

Use `Ctrl-C` or `Cmd-C` to stop the live stream.

View container console log stream

To connect to a container's console log stream in a container app with multiple revisions, replicas, and containers, include the following parameters in the `az containerapp logs show` command.

Argument	Description
<code>--revision</code>	The revision name.
<code>--replica</code>	The replica name in the revision.
<code>--container</code>	The container name to connect to.

You can get the revision names with the `az containerapp revision list` command. Replace the `<placeholders>` with your container app's values.

Bash

Azure CLI

```
az containerapp revision list \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--query "[].name"
```

Use the `az containerapp replica list` command to get the replica and container names. Replace the `<placeholders>` with your container app's values.

Bash

Azure CLI

```
az containerapp replica list \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--revision <RevisionName> \
--query "[].{Containers:properties.containers[].name, Name:name}"
```

Live stream the container console using the `az container app show` command with the `--follow` argument. Replace the <placeholders> with your container app's values.

Bash

Azure CLI

```
az containerapp logs show \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--revision <RevisionName> \
--replica <ReplicaName> \
--container <ContainerName> \
--type console \
--follow
```

Use `Ctrl-C` or `Cmd-C` to stop the live stream.

View the last 50 console log messages using the `az containerapp logs show` command with the `--tail` argument. Replace the <placeholders> with your container app's values.

Bash

Azure CLI

```
az containerapp logs show \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--revision <RevisionName> \
--replica <ReplicaName> \
--container <ContainerName> \
--type console \
--tail 50
```

View environment system log stream

Use the following command with the `--follow` argument to view the live system log stream from the Container Apps environment. Replace the <placeholders> with your environment values.

Bash

```
Azure CLI
```

```
az containerapp env logs show \
--name <ContainerAppEnvironmentName> \
--resource-group <ResourceGroup> \
--follow
```

Use `Ctrl-C` or `Cmd-C` to stop the live stream.

This example uses the `--tail` argument to display the last 50 environment system log messages. Replace the <placeholders> with your environment values.

Bash

```
Azure CLI
```

```
az containerapp env logs show \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--tail 50
```

[Log storage and monitoring options in Azure Container Apps](#)

Connect to a container console in Azure Container Apps

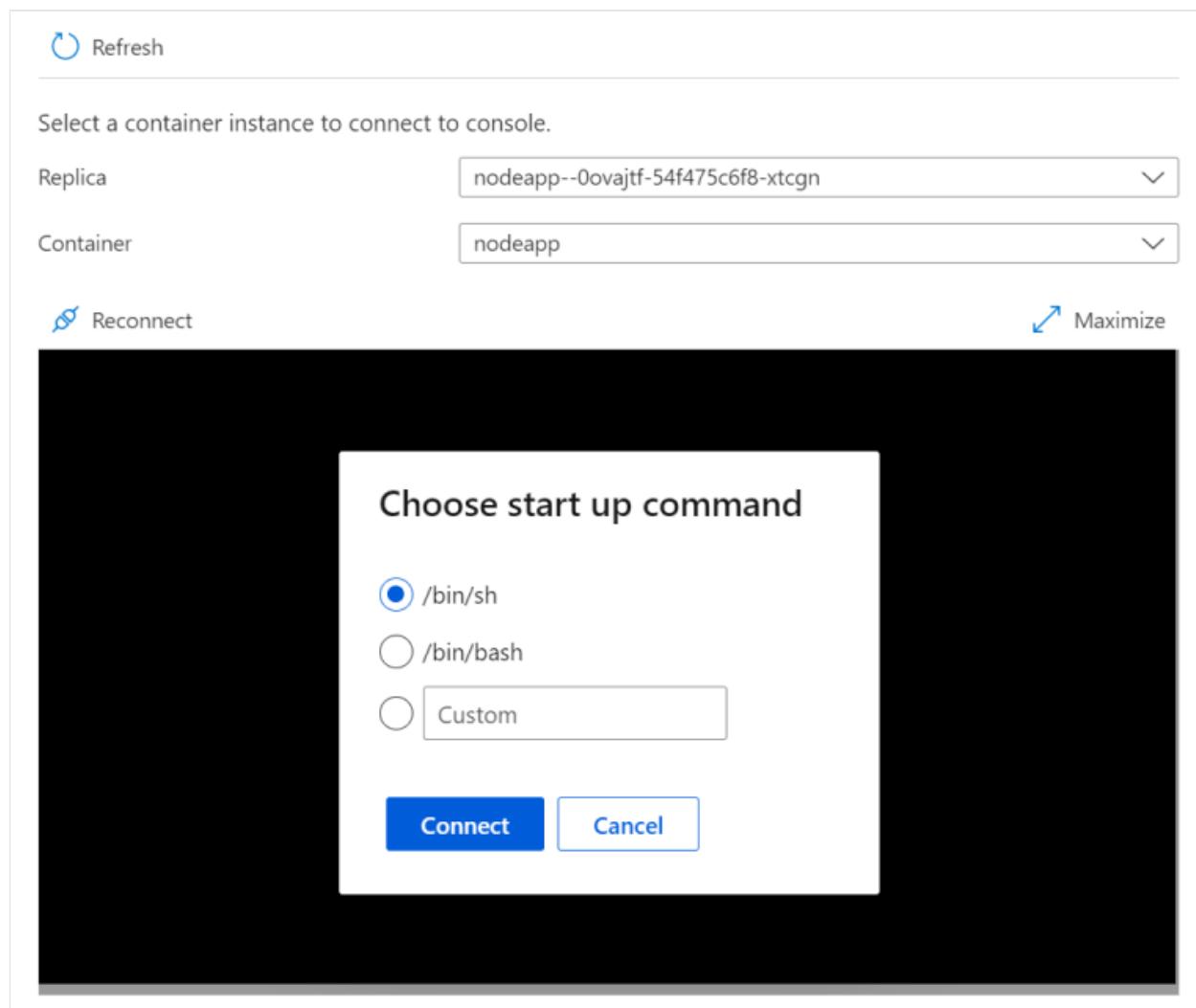
Article • 04/09/2023

Connecting to a container's console is useful when you want to troubleshoot your application inside a container. Azure Container Apps lets you connect to a container's console using the Azure portal or the Azure CLI.

Azure portal

To connect to a container's console in the Azure portal, follow these steps.

1. Select **Console** in the **Monitoring** menu group from your container app page in the Azure portal.
2. Select the revision, replica and container you want to connect to.
3. Choose to access your console via bash, sh, or a custom executable. If you choose a custom executable, it must be available in the container.



Azure CLI

Use the `az containerapp exec` command to connect to a container console. Select **Ctrl-D** to exit the console.

For example, connect to a container console in a container app with a single container using the following command. Replace the <placeholders> with your container app's values.

Bash

```
Azure CLI

az containerapp exec \
--name <ContainerAppName> \
--resource-group <ResourceGroup>
```

To connect to a container console in a container app with multiple revisions, replicas, and containers include the following parameters in the `az containerapp exec` command.

Argument	Description
<code>--revision</code>	The revision names of the container to connect to.
<code>--replica</code>	The replica name of the container to connect to.
<code>--container</code>	The container name of the container to connect to.

You can get the revision names with the `az containerapp revision list` command. Replace the <placeholders> with your container app's values.

Bash

```
Azure CLI

az containerapp revision list \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--query "[].name"
```

Use the `az containerapp replica list` command to get the replica and container names. Replace the <placeholders> with your container app's values.

Bash

Azure CLI

```
az containerapp replica list \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--revision <RevisionName> \
--query "[].{Containers:properties.containers[].name, Name:name}"
```

Connect to the container console with the `az containerapp exec` command. Replace the <placeholders> with your container app's values.

Bash

Azure CLI

```
az containerapp exec \
--name <ContainerAppName> \
--resource-group <ResourceGroup> \
--revision <RevisionName> \
--replica <ReplicaName> \
--container <ContainerName>
```

[View log streams from the Azure portal](#)

Monitor Azure Container Apps metrics

Article • 04/14/2023

Azure Monitor collects metric data from your container app at regular intervals to help you gain insights into the performance and health of your container app.

The metrics explorer in the Azure portal allows you to visualize the data. You can also retrieve raw metric data through the [Azure CLI](#) and [Azure PowerShell cmdlets](#).

Available metrics

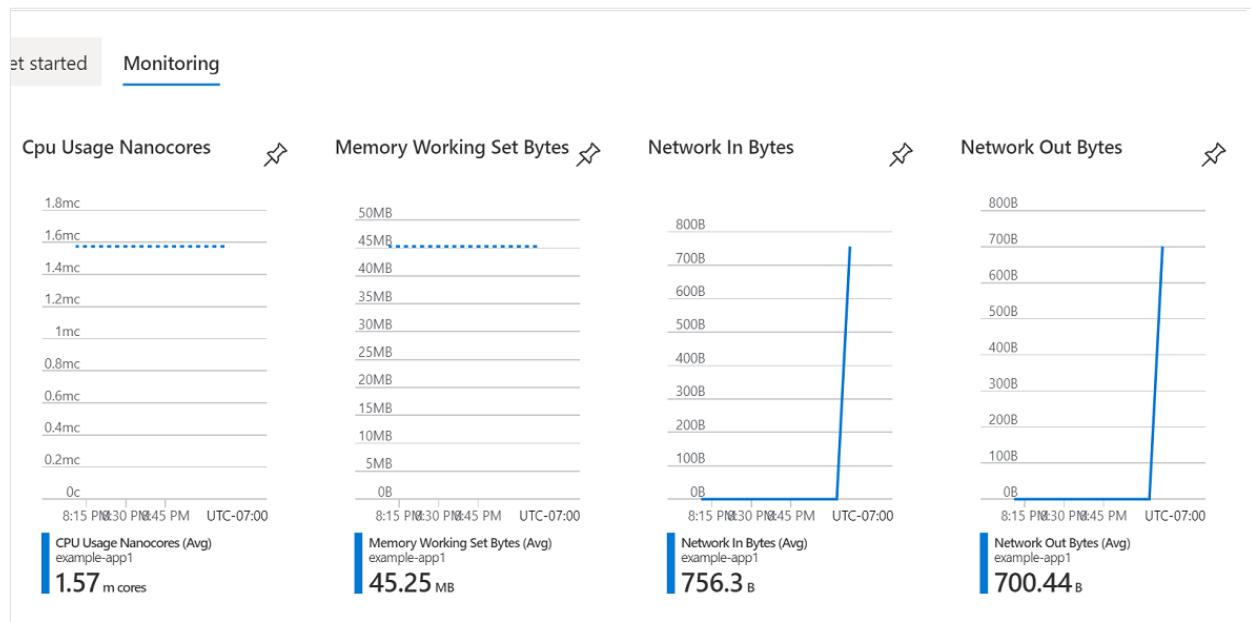
Container Apps provides these metrics.

Title	Description	Metric ID	Unit
CPU usage nanocores	CPU usage in nanocores (1,000,000,000 nanocores = 1 core)	UsageNanoCores	nanocores
Memory working set bytes	Working set memory used in bytes	WorkingSetBytes	bytes
Network in bytes	Network received bytes	RxBytes	bytes
Network out bytes	Network transmitted bytes	TxBytes	bytes
Requests	Requests processed	Requests	n/a
Replica count	Number of active replicas	Replicas	n/a
Replica Restart Count	Number of replica restarts	RestartCount	n/a

The metrics namespace is `microsoft.app/containerapps`.

Metrics snapshots

Select the **Monitoring** tab on your app's [Overview](#) page to display charts showing your container app's current CPU, memory, and network utilization.



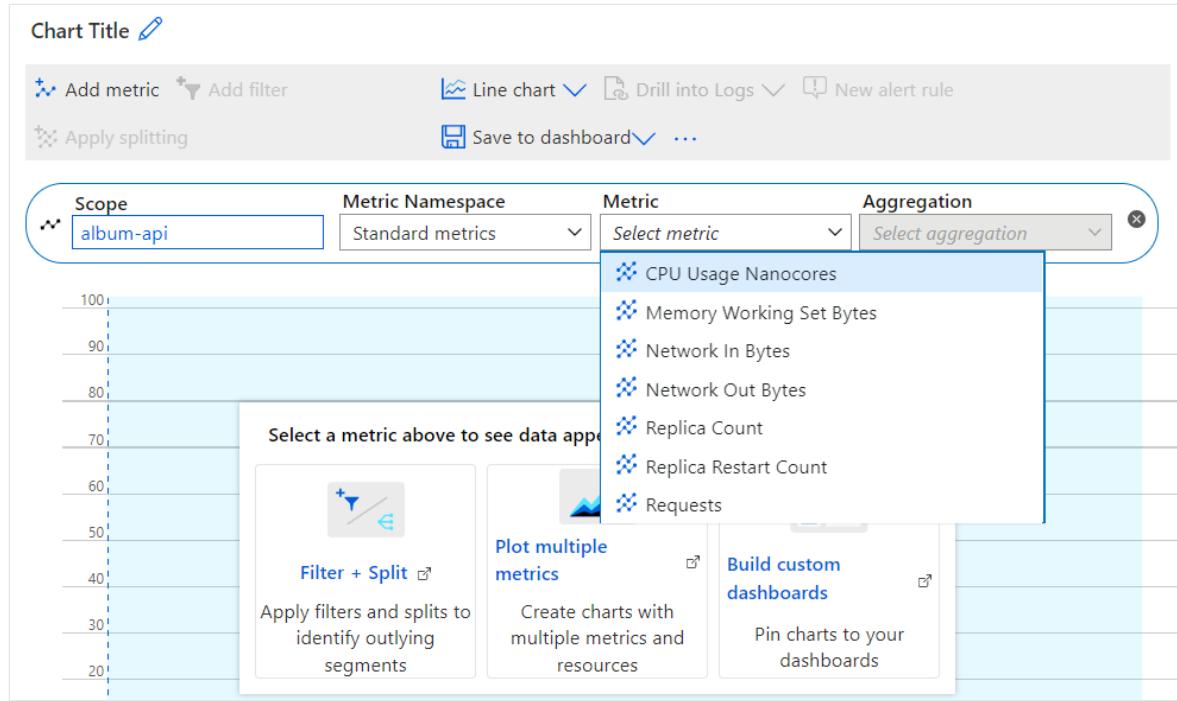
From this view, you can pin one or more charts to your dashboard or select a chart to open it in the metrics explorer.

Using metrics explorer

The Azure Monitor metrics explorer lets you create charts from metric data to help you analyze your container app's resource and network usage over time. You can pin charts to a dashboard or in a shared workbook.

1. Open the metrics explorer in the Azure portal by selecting **Metrics** from the sidebar menu on your container app's page. To learn more about metrics explorer, go to [Getting started with metrics explorer](#).
2. Create a chart by selecting **Metric**. You can modify the chart by changing aggregation, adding more metrics, changing time ranges and intervals, adding

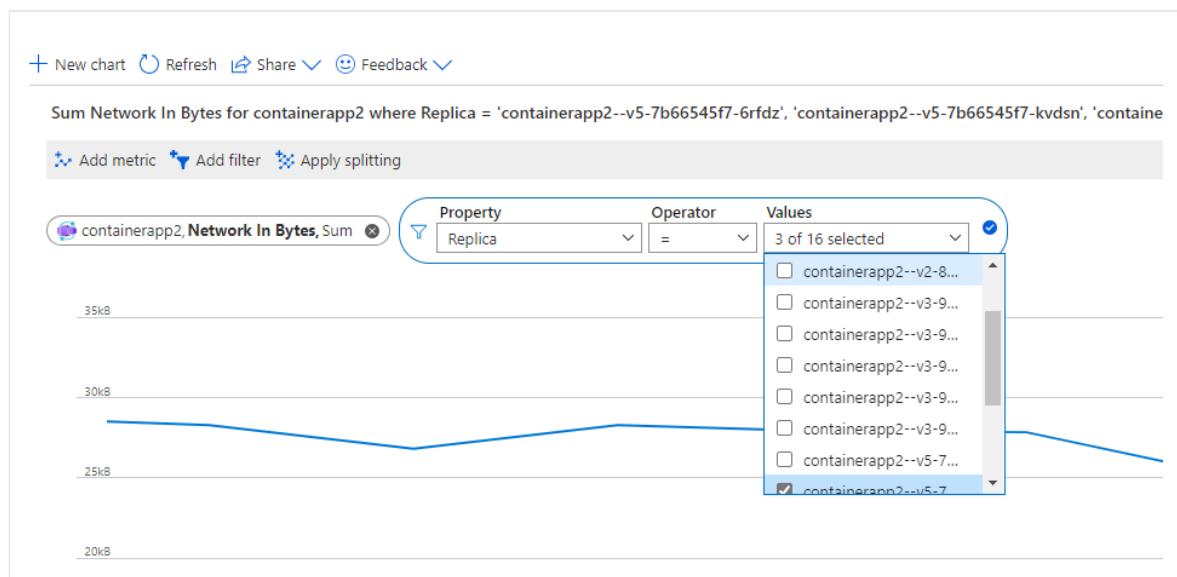
filters, and applying splitting.



Add filters

Optionally, you can create filters to limit the data shown based on revisions and replicas. To create a filter:

1. Select **Add filter**.
2. Select a revision or replica from the **Property** list.
3. Select values from the **Value** list.



Split metrics

When your chart contains a single metric, you can choose to split the metric information by revision or replica with the exceptions:

- The *Replica count* metric can only split by revision.
- The *Requests* metric can also be split by status code and status code category.

To split by revision or replica:

1. Select **Apply splitting**
2. Select **Revision or Replica** from the **Values** drop-down list.
3. You can set the limit of the number of revisions or replicas to display in the chart.
The default is 10.
4. You can set Sort order to **Ascending** or **Descending**. The default is **Descending**.

Configure signal logic

7.92 MB

Split by dimensions

Use dimensions to monitor specific time series and provide context to the fired alert. Dimensions can be either number or string columns. If you select more than one dimension value, each time series that results from the combination will trigger its own alert and will be charged separately. [About monitoring multiple time series](#) ⓘ

Dimension name	Operator	Dimension values
Replica	=	<input type="text"/> Type to start filtering... <ul style="list-style-type: none"> containerapp2--v2-8449f5bff5-fgvms containerapp2--v2-8449f5bff5-hh5xg containerapp2--v2-8449f5bff5-jj29j containerapp2--v2-8449f5bff5-tksz6 containerapp2--v3-98c586c96-fqq8n containerapp2--v3-98c586c96-kl9kr containerapp2--v3-98c586c96-n2fc6
Select dimension	=	Add custom value

Alert logic

Threshold ⓘ

Static **Dynamic**

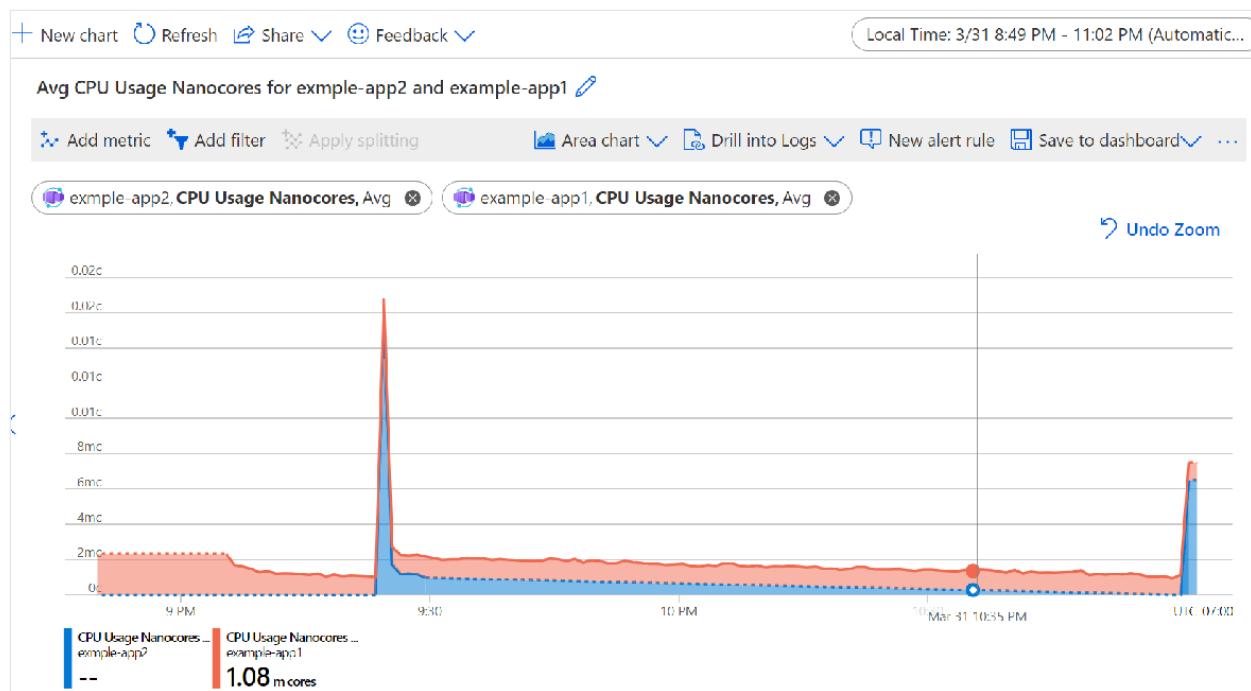
ⓘ We currently support alert rules with D

Add custom value

Monitoring 1 time series (\$0.1/time series)

Add scopes

You can add more scopes to view metrics across multiple container apps.



Set up alerts in Azure Container Apps

Monitor logs in Azure Container Apps with Log Analytics

Article • 04/14/2023

Azure Container Apps is integrated with Azure Monitor Log Analytics to monitor and analyze your container app's logs. When selected as your log monitoring solution, your Container Apps environment includes a Log Analytics workspace that provides a common place to store the system and application log data from all container apps running in the environment.

Log entries are accessible by querying Log Analytics tables through the Azure portal or a command shell using the [Azure CLI](#).

There are two types of logs for Container Apps.

- Console logs, which are emitted by your app.
- System logs, which are emitted by the Container Apps service.

System Logs

The Container Apps service provides system log messages at the container app level. System logs emit the following messages:

Source	Type	Message
Dapr	Info	Successfully created dapr component <component-name> with scope <dapr-component-scope>
Dapr	Info	Successfully updated dapr component <component-name> with scope <component-type>
Dapr	Error	Error creating dapr component <component-name>
Volume Mounts	Info	Successfully mounted volume <volume-name> for revision <revision-scope>
Volume Mounts	Error	Error mounting volume <volume-name>
Domain Binding	Info	Successfully bound Domain <domain> to the container app <container app name>
Authentication	Info	Auth enabled on app. Creating authentication config

Source	Type	Message
Authentication	Info	Auth config created successfully
Traffic weight	Info	Setting a traffic weight of <percentage>% for revision <revision-name>
Revision Provisioning	Info	Creating a new revision: <revision-name>
Revision Provisioning	Info	Successfully provisioned revision <name>
Revision Provisioning	Info	Deactivating Old revisions since 'ActiveRevisionsMode=Single'
Revision Provisioning	Error	Error provisioning revision <revision-name>. ErrorCode: <[ErrImagePull] [Timeout] [ContainerCrashing]>

The system log data is accessible by querying the `ContainerAppSystemLogs_CL` table. The most used Container Apps specific columns in the table are:

Column	Description
<code>ContainerAppName_s</code>	Container app name
<code>EnvironmentName_s</code>	Container Apps environment name
<code>Log_s</code>	Log message
<code>RevisionName_s</code>	Revision name

Console Logs

Console logs originate from the `stderr` and `stdout` messages from the containers in your container app and Dapr sidecars. You can view console logs by querying the `ContainerAppConsoleLogs_CL` table.

Tip

Instrumenting your code with well-defined log messages can help you to understand how your code is performing and to debug issues. To learn more about best practices refer to [Design for operations](#).

The most commonly used Container Apps specific columns in `ContainerAppConsoleLogs_CL` include:

Column	Description
ContainerAppName_s	Container app name
ContainerGroupName_g	Replica name
ContainerId_s	Container identifier
ContainerImage_s	Container image name
EnvironmentName_s	Container Apps environment name
Log_s	Log message
RevisionName_s	Revision name

Query Log with Log Analytics

Log Analytics is a tool in the Azure portal that you can use to view and analyze log data. Using Log Analytics, you can write Kusto queries and then sort, filter, and visualize the results in charts to spot trends and identify issues. You can work interactively with the query results or use them with other features such as alerts, dashboards, and workbooks.

Azure portal

Start Log Analytics from **Logs** in the sidebar menu on your container app page. You can also start Log Analytics from **Monitor>Logs**.

You can query the logs using the tables listed in the **CustomLogs** category **Tables** tab. The tables in this category are the `ContainerAppSystemlogs_CL` and `ContainerAppConsoleLogs_CL` tables.

New Query 1*

workspacecebca9ba3 Select scope

Tables Queries Functions ...

Search Filter Group by: Solution

Collapse all

Favorites

You can add favorites by clicking on the icon

- ▲ LogManagement
 - ▶ Usage
- ▲ Custom Logs
 - ▶ ContainerAppConsoleLogs_CL
 - ▶ ContainerAppSystemLogs_CL

Below is a Kusto query that displays console log entries for the container app named *album-api*.

Kusto

```
ContainerAppConsoleLogs_CL
| where ContainerAppName_s == 'album-api'
| project Time=TimeGenerated, AppName=ContainerAppName_s,
Revision=RevisionName_s, Container=ContainerName_s, Message=Log_s
| take 100
```

Below is a Kusto query that displays system log entries for the container app named *album-api*.

Kusto

```
ContainerAppSystemLogs_CL
| where ContainerAppName_s == 'album-api'
| project Time=TimeGenerated, EnvName=EnvironmentName_s,
AppName=ContainerAppName_s, Revision=RevisionName_s, Message=Log_s
| take 100
```

For more information regarding Log Analytics and log queries, see the [Log Analytics tutorial](#).

Azure CLI/PowerShell

Container Apps logs can be queried using the [Azure CLI](#).

These example Azure CLI queries output a table containing log records for the container app name `album-api`. The table columns are specified by the parameters after the `project` operator. The `$WORKSPACE_CUSTOMER_ID` variable contains the GUID of the Log Analytics workspace.

This example queries the `ContainerAppConsoleLogs_CL` table:

Bash

Azure CLI

```
az monitor log-analytics query --workspace $WORKSPACE_CUSTOMER_ID --  
analytics-query "ContainerAppConsoleLogs_CL | where ContainerAppName_s  
== 'album-api' | project Time=TimeGenerated, AppName=ContainerAppName_s,  
Revision=RevisionName_s, Container=ContainerName_s, Message=Log_s,  
LogLevel_s | take 5" --out table
```

This example queries the `ContainerAppSystemLogs_CL` table:

Bash

Azure CLI

```
az monitor log-analytics query --workspace $WORKSPACE_CUSTOMER_ID --  
analytics-query "ContainerAppSystemLogs_CL | where ContainerAppName_s ==  
'album-api' | project Time=TimeGenerated, AppName=ContainerAppName_s,  
Revision=RevisionName_s, Message=Log_s, LogLevel_s | take 5" --out table
```

Next steps

[View log streams from the Azure portal](#)

Set up alerts in Azure Container Apps

Article • 04/14/2023

Azure Monitor alerts notify you so that you can respond quickly to critical issues. There are two types of alerts that you can define:

- [Metric alerts](#) based on Azure Monitor metric data
- [Log alerts](#) based on Azure Monitor Log Analytics data

You can create alert rules from metric charts in the metric explorer and from queries in Log Analytics. You can also define and manage alerts from the [Monitor>Alerts](#) page. To learn more about alerts, refer to [Overview of alerts in Microsoft Azure](#).

The **Alerts** page in the **Monitoring** section on your container app page displays all of your app's alerts. You can filter the list by alert type, resource, time and severity. You can also modify and create new alert rules from this page.

Create metric alert rules

When you create alerts rules based on a metric chart in the metrics explorer, alerts are triggered when the metric data matches alert rule conditions. For more information about creating metrics charts, see [Using metrics explorer](#)

After creating a metric chart, you can create a new alert rule.

1. Select **New alert rule**. The [Create an alert rule](#) page is opened to the **Condition** tab. Here you'll find a *condition* that is populated with the metric chart settings.
2. Select the condition.

The screenshot shows the 'Create an alert rule' page with the 'Condition' tab selected. The page header includes 'Home > my-container-app | Metrics > Create an alert rule ...'. Below the tabs are buttons for 'Scope', 'Condition' (which is underlined), 'Actions', 'Details', 'Tags', and 'Review + create'. A note below the tabs says 'Configure when the alert rule should trigger by selecting a signal and defining its logic.' A table lists a single condition: 'Condition name: Whenever the average usagenanocores is greater than <logic undefined>' (with the entire row highlighted by a red box). The table has columns for 'Time series monitored' (1) and 'Estimated monthly cost (USD)' (\$ 0.10). At the bottom left is a '+ Add condition' button.

Condition name	Time series monitored	Estimated monthly cost (USD)
Whenever the average usagenanocores is greater than <logic undefined>	1	\$ 0.10
	1	Total \$ 0.10

3. Modify the **Alert logic** section to set the alert criteria. You can set the alert to trigger when the metric value is greater than, less than, or equal to a threshold value. You can also set the alert to trigger when the metric value is outside of a

range of values.

Configure signal logic

Chart period ⓘ
Over the last 6 hours

CPU Usage (Sum)
my-container-app
9.19 cores

Alert logic

Threshold ⓘ
Static Dynamic

Operator ⓘ
Greater than

Aggregation type * ⓘ
Total

Threshold value * ⓘ
10 ✓

Condition preview
Whenever the total cpu usage is greater than 10

Evaluated based on

Aggregation granularity (Period) * ⓘ
5 minutes

Frequency of evaluation ⓘ
Every 5 Minutes

Done

4. Select **Done**.
5. You can add more conditions to the alert rule by selecting **Add condition** on the **Create an alert rule** page.
6. Select the **Details** tab.
7. Enter a name and description for the alert rule.
8. Select **Review + create**.

9. Select Create.

The screenshot shows the 'Create an alert rule' dialog box. The 'Details' tab is selected. In the 'Project details' section, the subscription is set to 'Vendor learning support subscription' and the resource group is 'my-container-apps'. In the 'Alert rule details' section, the severity is '3 - Informational', the alert rule name is 'maximum replica alert', and the alert rule description is 'Alert when there are more than 10 replicas'. A red box highlights the alert rule name and description fields.

Add conditions to an alert rule

To add more conditions to your alert rule:

1. Select **Alerts** from the left side menu of your container app page.
2. Select **Alert rules** from the top menu.
3. Select an alert from the table.
4. Select **Add condition** in the **Condition** section.

5. Select from the metrics listed in the **Select a signal** pane.

The screenshot shows the 'Select a signal' pane with the following interface elements:

- Signal type:** Metrics (selected)
- Monitor service:** All
- Displaying:** 1 - 7 signals out of total 7 signals
- Search by signal name:** (empty)
- Table:** A list of 7 metrics, each with a red border around the first six rows:

Signal name	Signal type	Monitor service
Replica Count	Metrics	Platform
Requests	Metrics	Platform
Replica Restart Count	Metrics	Platform
Network In Bytes	Metrics	Platform
Network Out Bytes	Metrics	Platform
CPU Usage Nanocores	Metrics	Platform
Memory Working Set Bytes	Metrics	Platform
- Buttons:** Done

6. Configure the settings for your alert condition. For more information about configuring alerts, see [Manage metric alerts](#).

You can receive individual alerts for specific revisions or replicas by enabling alert splitting and selecting **Revision** or **Replica** from the Dimension name list.

Example of selecting a dimension to split an alert.

The screenshot shows the 'Configure signal logic' pane with the following interface elements:

- 7.92 MB** (represented by a blue bar)
- Split by dimensions:** (button)
- Dimension name:** Replica (selected)
- Operator:** =
- Dimension values:** (dropdown menu)
 - Type to start filtering...
 - containerapp2--v2-8449f5bff5-fgvm5
 - containerapp2--v2-8449f5bff5-hh5xg
 - containerapp2--v2-8449f5bff5-jj29j
 - containerapp2--v2-8449f5bff5-tksz6
 - containerapp2--v3-98c586c96-fqq8n
 - containerapp2--v3-98c586c96-kl9kr
 - containerapp2--v3-98c586c96-n2fc6
- Add custom value:** (button)
- Select dimension:** (dropdown menu)
- Alert logic:** (button)
- Threshold:** (radio buttons) Static (selected), Dynamic
- We currently support alert rules with D:** (info icon)

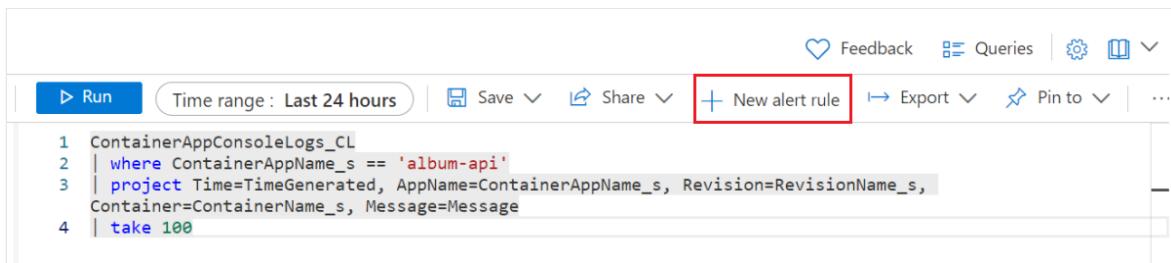
To learn more about configuring alerts, visit [Create a metric alert for an Azure resource](#)

Create log alert rules

You can create log alerts from queries in Log Analytics. When you create an alert rule from a query, the query is run at set intervals triggering alerts when the log data matches the alert rule conditions. To learn more about creating log alert rules, see [Manage log alerts](#).

To create an alert rule:

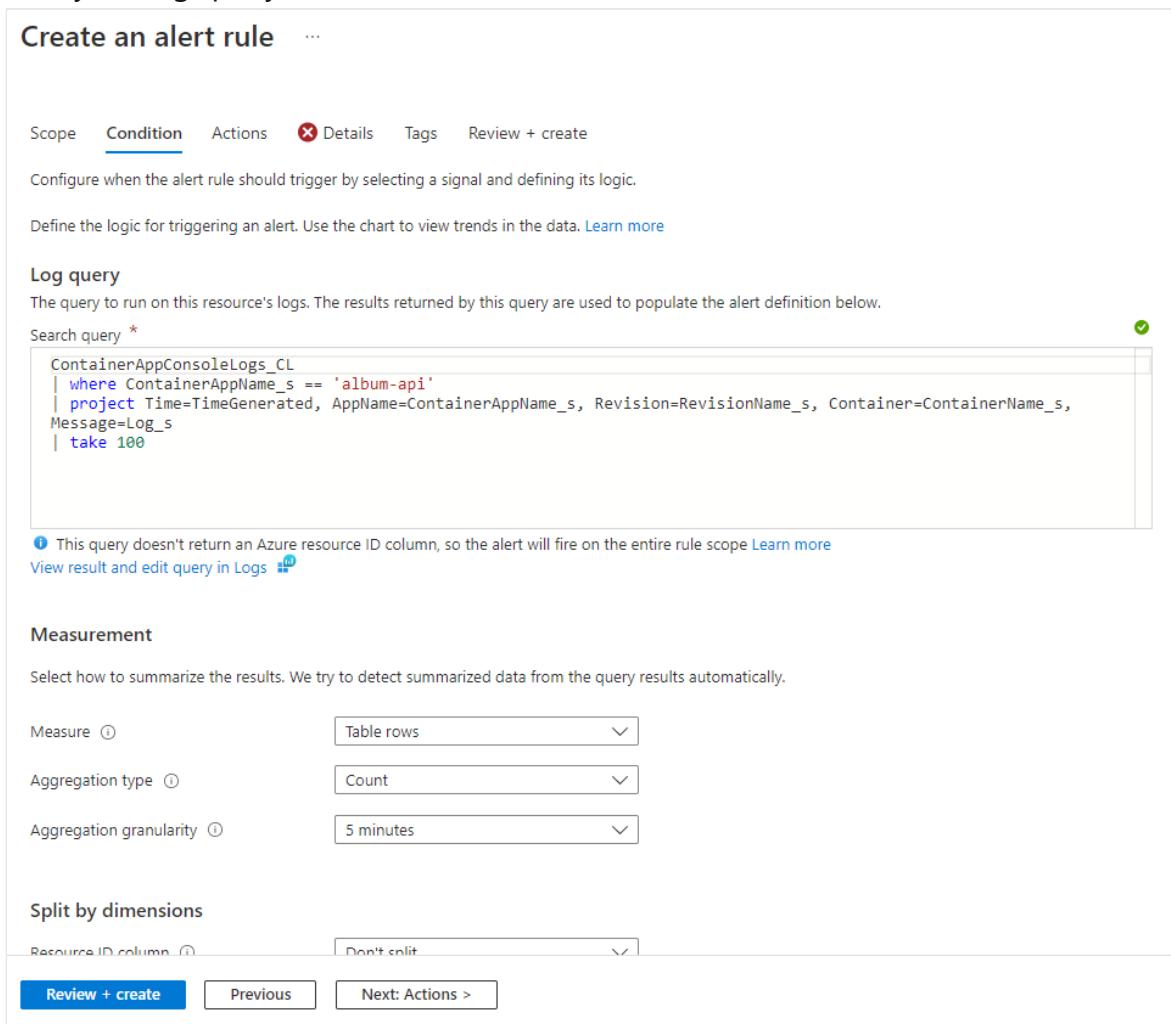
1. First, create and run a query to validate the query.
2. Select **New alert rule**.



A screenshot of the Microsoft Log Analytics workspace. At the top, there's a toolbar with various icons: Feedback, Queries, Save, Share, Export, Pin to, and a three-dot menu. A red box highlights the 'New alert rule' button, which is located next to the 'Export' icon. Below the toolbar, there's a search bar with 'Time range : Last 24 hours'. The main area contains a code editor with the following query:

```
1 ContainerAppConsoleLogs_CL
2 | where ContainerAppName_s == 'album-api'
3 | project Time=TimeGenerated, AppName=ContainerAppName_s, Revision=RevisionName_s,
   Container=ContainerName_s, Message=Message
4 | take 100
```

3. The **Create an alert rule** editor is opened to the **Condition** tab, which is populated with your log query.



The screenshot shows the 'Create an alert rule' editor with the 'Condition' tab selected. The 'Log query' section contains the same log query as the previous screenshot. Below it, the 'Measurement' section has the following settings:

- Measure: Table rows
- Aggregation type: Count
- Aggregation granularity: 5 minutes

The 'Split by dimensions' section shows a dropdown menu with 'Resource ID column' and 'Don't split' selected. At the bottom of the editor, there are navigation buttons: 'Review + create', 'Previous', and 'Next: Actions >'. A checkmark icon is visible in the top right corner of the editor window.

4. Configure the settings in the **Measurement** section

Measurement

Select how to summarize the results. We try to detect summarized data from the query results automatically.

Measure <small>(i)</small>	Table rows
Aggregation type <small>(i)</small>	Count
Aggregation granularity <small>(i)</small>	5 minutes

5. Optionally, you can enable alert splitting in the alert rule to send individual alerts for each dimension you select in the **Split by dimensions** section of the editor.

Split by dimensions

Resource ID column (i) Don't split

Use dimensions to monitor specific time series and provide context to the fired alert. Dimensions can be either number or string columns. If you select more than one dimension value, each time series that results from the combination will trigger its own alert and will be charged separately. (i)

Dimension name	Operator	Dimension values	Action
AppName	=	sample-app1	Add custom value
Revision	=	sample-app1--v2	Add custom value
Container	=	simple-hello-world-container	Add custom value
Message	=	All current and future values	Add custom value

6. Enter the threshold criteria in the **Alert logic** section.

Alert logic

Operator * (i) Greater than

Threshold value * (i)

Frequency of evaluation * (i) 5 minutes

7. Select the **Details** tab.

8. Enter a name and description for the alert rule.

The screenshot shows the 'Create an alert rule' wizard with the 'Details' tab selected. In the 'Project details' section, the subscription is set to 'Vendor learning support subscription' and the resource group is 'my-container-apps'. In the 'Alert rule details' section, the severity is '3 - Informational', the alert rule name is 'maximum replica alert', and the description is 'Alert when there are more than 10 replicas'. A red box highlights the 'Alert rule name' and 'Description' fields.

Subscription * ⓘ Vendor learning support subscription

Resource group * ⓘ my-container-apps [Create new](#)

Severity * ⓘ 3 - Informational

Alert rule name * ⓘ maximum replica alert

Alert rule description ⓘ Alert when there are more than 10 replicas

[Review + create](#) [Previous](#) [Next: Tags >](#)

9. Select **Review + create**.

10. Select **Create**.

[View log streams from the Azure portal](#)

Authentication and authorization in Azure Container Apps

Article • 03/24/2023

Azure Container Apps provides built-in authentication and authorization features (sometimes referred to as "Easy Auth"), to secure your external ingress-enabled container app with minimal or no code.

For details surrounding authentication and authorization, refer to the following guides for your choice of provider.

- [Azure Active Directory](#)
- [Facebook](#)
- [GitHub](#)
- [Google](#)
- [Twitter](#)
- [Custom OpenID Connect](#)

Why use the built-in authentication?

You're not required to use this feature for authentication and authorization. You can use the bundled security features in your web framework of choice, or you can write your own utilities. However, implementing a secure solution for authentication (signing-in users) and authorization (providing access to secure data) can take significant effort. You must make sure to follow industry best practices and standards, and keep your implementation up to date.

The built-in authentication feature for Container Apps can save you time and effort by providing out-of-the-box authentication with federated identity providers, allowing you to focus on the rest of your application.

- Azure Container Apps provides access to various built-in authentication providers.
- The built-in auth features don't require any particular language, SDK, security expertise, or even any code that you have to write.
- You can integrate with multiple providers including Azure Active Directory, Facebook, Google, and Twitter.

Identity providers

Container Apps uses [federated identity](#), in which a third-party identity provider manages the user identities and authentication flow for you. The following identity providers are available by default:

Provider	Sign-in endpoint	How-To guidance
Microsoft Identity Platform	<code>/auth/login/aad</code>	Microsoft Identity Platform
Facebook	<code>/auth/login/facebook</code>	Facebook
GitHub	<code>/auth/login/github</code>	GitHub
Google	<code>/auth/login/google</code>	Google
Twitter	<code>/auth/login/twitter</code>	Twitter
Any OpenID Connect provider	<code>/auth/login/<providerName></code>	OpenID Connect

When you use one of these providers, the sign-in endpoint is available for user authentication and authentication token validation from the provider. You can provide your users with any number of these provider options.

Considerations for using built-in authentication

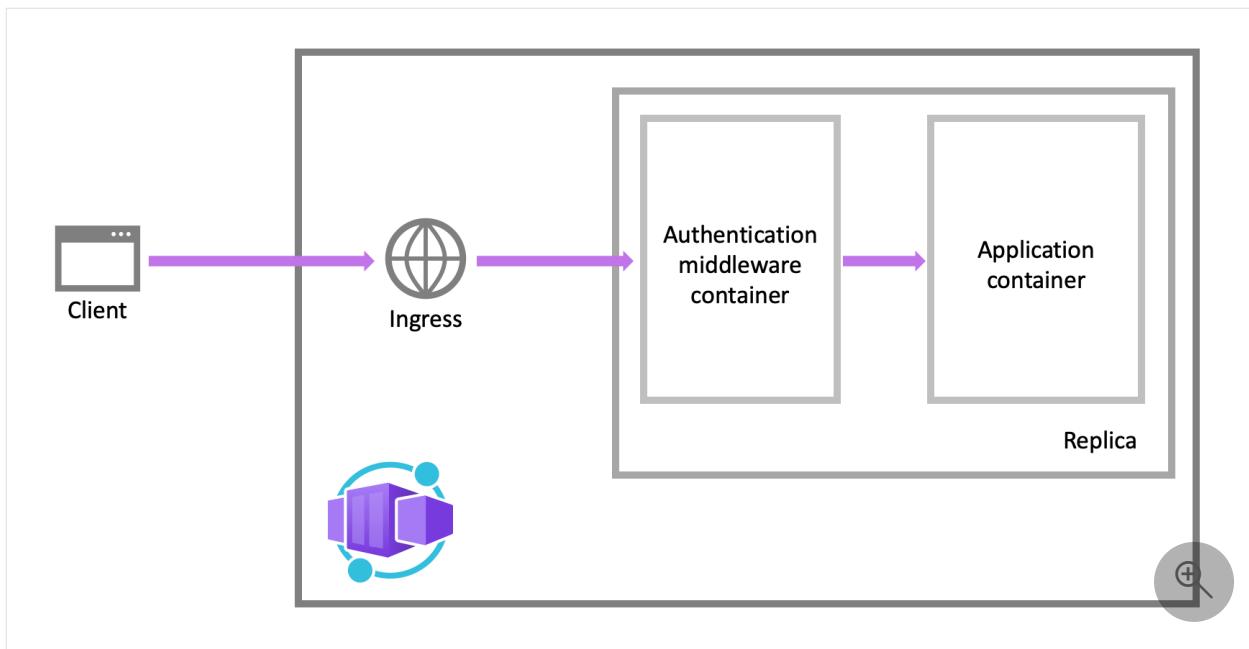
This feature should be used with HTTPS only. Ensure `allowInsecure` is disabled on your container app's ingress configuration.

You can configure your container app for authentication with or without restricting access to your site content and APIs. To restrict app access only to authenticated users, set its *Restrict access* setting to **Require authentication**. To authenticate but not restrict access, set its *Restrict access* setting to **Allow unauthenticated access**.

Each container app issues its own unique cookie or token for authentication. A client cannot use the same cookie or token provided by one container app to authenticate with another container app, even within the same container app environment.

Feature architecture

The authentication and authorization middleware component is a feature of the platform that runs as a sidecar container on each replica in your application. When enabled, every incoming HTTP request passes through the security layer before being handled by your application.



The platform middleware handles several things for your app:

- Authenticates users and clients with the specified identity provider(s)
- Manages the authenticated session
- Injects identity information into HTTP request headers

The authentication and authorization module runs in a separate container, isolated from your application code. As the security container doesn't run in-process, no direct integration with specific language frameworks is possible. However, relevant information your app needs is provided in request headers as explained below.

Authentication flow

The authentication flow is the same for all providers, but differs depending on whether you want to sign in with the provider's SDK:

- **Without provider SDK (*server-directed flow* or *server flow*):** The application delegates federated sign-in to Container Apps. Delegation is typically the case with browser apps, which presents the provider's sign-in page to the user.
- **With provider SDK (*client-directed flow* or *client flow*):** The application signs users in to the provider manually and then submits the authentication token to Container Apps for validation. This approach is typical for browser-less apps that don't present the provider's sign-in page to the user. An example is a native mobile app that signs users in using the provider's SDK.

Calls from a trusted browser app in Container Apps to another REST API in Container Apps can be authenticated using the server-directed flow. For more information, see [Customize sign-ins and sign-outs](#).

The table below shows the steps of the authentication flow.

Step	Without provider SDK	With provider SDK
1. Sign user in	Redirects client to <code>/auth/login/<PROVIDER></code> .	Client code signs user in directly with provider's SDK and receives an authentication token. For information, see the provider's documentation.
2. Post-authentication	Provider redirects client to <code>/auth/login/<PROVIDER>/callback</code> .	Client code posts token from provider to <code>/auth/login/<PROVIDER></code> for validation.
3. Establish authenticated session	Container Apps adds authenticated cookie to response.	Container Apps returns its own authentication token to client code.
4. Serve authenticated content	Client includes authentication cookie in subsequent requests (automatically handled by browser).	Client code presents authentication token in <code>X-ZUMO-AUTH</code> header.

For client browsers, Container Apps can automatically direct all unauthenticated users to `/auth/login/<PROVIDER>`. You can also present users with one or more `/auth/login/<PROVIDER>` links to sign in to your app using their provider of choice.

Authorization behavior

In the [Azure portal](#), you can edit your container app's authentication settings to configure it with various behaviors when an incoming request isn't authenticated. The following headings describe the options.

- **Allow unauthenticated access:** This option defers authorization of unauthenticated traffic to your application code. For authenticated requests, Container Apps also passes along authentication information in the HTTP headers. Your app can use information in the headers to make authorization decisions for a request.

This option provides more flexibility in handling anonymous requests. For example, it lets you [present multiple sign-in providers](#) to your users. However, you must write code.

- **Require authentication:** This option rejects any unauthenticated traffic to your application. This rejection can be a redirect action to one of the configured identity providers. In these cases, a browser client is redirected to `/auth/login/<PROVIDER>` for the provider you choose. If the anonymous request comes from a native mobile app, the returned response is an `HTTP 401 Unauthorized`. You can also configure

the rejection to be an `HTTP 401 Unauthorized` or `HTTP 403 Forbidden` for all requests.

With this option, you don't need to write any authentication code in your app. Finer authorization, such as role-specific authorization, can be handled by inspecting the user's claims (see [Access user claims](#)).

✖ Caution

Restricting access in this way applies to all calls to your app, which may not be desirable for apps wanting a publicly available home page, as in many single-page applications.

❗ Note

By default, any user in your Azure AD tenant can request a token for your application from Azure AD. You can [configure the application in Azure AD](#) if you want to restrict access to your app to a defined set of users.

Customize sign-in and sign-out

Container Apps Authentication provides built-in endpoints for sign-in and sign-out. When the feature is enabled, these endpoints are available under the `/auth` route prefix on your container app.

Use multiple sign-in providers

The portal configuration doesn't offer a turn-key way to present multiple sign-in providers to your users (such as both Facebook and Twitter). However, it isn't difficult to add the functionality to your app. The steps are outlined as follows:

First, in the **Authentication / Authorization** page in the Azure portal, configure each of the identity provider you want to enable.

In **Action to take when request is not authenticated**, select **Allow Anonymous requests (no action)**.

In the sign-in page, or the navigation bar, or any other location of your app, add a sign-in link to each of the providers you enabled (`/auth/login/<provider>`). For example:

HTML

```
<a href="/.auth/login/aad">Log in with the Microsoft Identity Platform</a>
<a href="/.auth/login/facebook">Log in with Facebook</a>
<a href="/.auth/login/google">Log in with Google</a>
<a href="/.auth/login/twitter">Log in with Twitter</a>
```

When the user selects on one of the links, the UI for the respective providers is displayed to the user.

To redirect the user post-sign-in to a custom URL, use the `post_login_redirect_uri` query string parameter (not to be confused with the Redirect URI in your identity provider configuration). For example, to navigate the user to `/Home/Index` after sign-in, use the following HTML code:

HTML

```
<a href="/.auth/login/<provider>?post_login_redirect_uri=/Home/Index">Log
in</a>
```

Client-directed sign-in

In a client-directed sign-in, the application signs in the user to the identity provider using a provider-specific SDK. The application code then submits the resulting authentication token to Container Apps for validation (see [Authentication flow](#)) using an HTTP POST request.

To validate the provider token, container app must first be configured with the desired provider. At runtime, after you retrieve the authentication token from your provider, post the token to `/.auth/login/<provider>` for validation. For example:

Console

```
POST https://<hostname>.azurecontainerapps.io/.auth/login/aad HTTP/1.1
Content-Type: application/json

{"id_token":<token>, "access_token":<token>}
```

The token format varies slightly according to the provider. See the following table for details:

Provider value	Required in request body	Comments
----------------	-----------------------------	----------

Provider value	Required in request body	Comments
aad	{"access_token": " <ACCESS_TOKEN>"}	The <code>id_token</code> , <code>refresh_token</code> , and <code>expires_in</code> properties are optional.
microsoftaccount	{"access_token": " <ACCESS_TOKEN>"} or {"authentication_token": " <TOKEN>"}	<code>authentication_token</code> is preferred over <code>access_token</code> . The <code>expires_in</code> property is optional. When requesting the token from Live services, always request the <code>wl.basic</code> scope.
google	{"id_token": " <ID_TOKEN>"}	The <code>authorization_code</code> property is optional. Providing an <code>authorization_code</code> value will add an access token and a refresh token to the token store. When specified, <code>authorization_code</code> can also optionally be accompanied by a <code>redirect_uri</code> property.
facebook	{"access_token": " <USER_ACCESS_TOKEN>"}	Use a valid user access token from Facebook.
twitter	{"access_token": " <ACCESS_TOKEN>", "access_token_secret": " <ACCES_TOKEN_SECRET>"}	

If the provider token is validated successfully, the API returns with an `authenticationToken` in the response body, which is your session token.

JSON

```
{
  "authenticationToken": "...",
  "user": {
    "userId": "sid:..."
  }
}
```

Once you have this session token, you can access protected app resources by adding the `X-ZUMO-AUTH` header to your HTTP requests. For example:

Console

```
GET https://<hostname>.azurecontainerapps.io/api/products/1
X-ZUMO-AUTH: <authenticationToken_value>
```

Sign out of a session

Users can initiate a sign-out by sending a `GET` request to the app's `/.auth/logout` endpoint. The `GET` request conducts the following actions:

- Clears authentication cookies from the current session.
- Deletes the current user's tokens from the token store.
- For Azure Active Directory and Google, performs a server-side sign-out on the identity provider.

Here's a simple sign-out link in a webpage:

HTML

```
<a href="/.auth/logout">Sign out</a>
```

By default, a successful sign-out redirects the client to the URL `/.auth/logout/done`. You can change the post-sign-out redirect page by adding the `post_logout_redirect_uri` query parameter. For example:

Console

```
GET /.auth/logout?post_logout_redirect_uri=/index.html
```

It's recommended that you [encode ↗](#) the value of `post_logout_redirect_uri`.

URL must be hosted in the same domain when using fully qualified URLs.

Access user claims in application code

For all language frameworks, Container Apps makes the claims in the incoming token available to your application code. The claims are injected into the request headers, which are present whether from an authenticated end user or a client application. External requests aren't allowed to set these headers, so they're present only if set by Container Apps. Some example headers include:

- `X-MS-CLIENT-PRINCIPAL-NAME`
- `X-MS-CLIENT-PRINCIPAL-ID`

Code that is written in any language or framework can get the information that it needs from these headers.

 **Note**

Different language frameworks may present these headers to the app code in different formats, such as lowercase or title case.

Next steps

Refer to the following articles for details on securing your container app.

- [Azure Active Directory](#)
- [Facebook](#)
- [GitHub](#)
- [Google](#)
- [Twitter](#)
- [Custom OpenID Connect](#)

Workload profiles in Consumption + Dedicated plan structure environments in Azure Container Apps (preview)

Article • 04/03/2023

Under the [Consumption + Dedicated plan structure](#), you can use different workload profiles in your environment. Workload profiles determine the amount of compute and memory resources available to container apps deployed in an environment.

Profiles are configured to fit the different needs of your applications.

Profile type	Description	Potential use
Consumption	Automatically added to any new environment.	Apps that don't require specific hardware requirements
Dedicated General purpose	Balance of memory and compute resources	Apps needing larger amounts of CPU and/or memory
Dedicated Memory optimized	Increased memory resources	Apps needing large in-memory data, in-memory machine learning models, or other high memory requirements

A Consumption workload profile is automatically added to all Consumption + Dedicated plan structure environment you create. You can optionally add dedicated workload profiles of any type or size as you create an environment or after it's created.

For each Dedicated workload profile in your environment, you can:

- Select the type and size
- Deploy multiple apps into the profile
- Use autoscaling to add and remove nodes based on the needs of the apps
- Limit scaling of the profile to for better cost control and predictability

You can configure each of your apps to run on any of the workload profiles defined in your Container Apps environment. This configuration is ideal for deploying a microservice solution where each app can run on the appropriate compute infrastructure.

Supported regions

The following regions support workload profiles during preview:

- North Central US
- North Europe
- West Europe
- East US

Profile types

There are different types and sizes of workload profiles available by region. By default each Consumption + Dedicated plan structure includes a Consumption profile, but you can also add any of the following profiles:

Display name	Name	Cores	MemoryGiB	Category	Allocation
Consumption	consumption	4	8	Consumption	per replica
Dedicated-D4	D4	4	16	General purpose	per node
Dedicated-D8	D8	8	32	General purpose	per node
Dedicated-D16	D16	16	64	General purpose	per node
Dedicated-E4	E4	4	32	Memory optimized	per node
Dedicated-E8	E8	8	64	Memory optimized	per node
Dedicated-E16	E16	16	128	Memory optimized	per node

Select a workload profile and use the *Name* field when you run `az containerapp env workload-profile set` for the `--workload-profile-type` option.

The availability of different workload profiles varies by region.

Resource consumption

You can constrain the memory and CPU usage of each app inside a workload profile, and you can run multiple apps inside a single instance of a workload profile. However, the total amount of resources available to a container app is less than what's allocated to a profile. The difference between allocated and available resources is what's reserved for the Azure Container Apps runtime.

Scaling

When demand for new apps or more replicas of an existing app exceeds the profile's current resources, profile instances may be added. Inversely, if the number of apps or replicas goes down, profile instances may be removed. You have control over the constraints on the minimum and maximum number of profile instances. Azure calculates [billing](#) largely based on the number of running profile instances.

Networking

When using workload profiles in the Consumption + Dedicated plan structure, additional networking features to fully secure your ingress/egress networking traffic such as user defined routes are available. To learn more about what networking features are supported, see [networking concepts](#), and for steps on how to secure your network with Container Apps, see the [lock down your Container App environment section](#).

Next steps

[Manage workload profiles with the CLI](#)

Dapr integration with Azure Container Apps

Article • 05/15/2023

The Distributed Application Runtime ([Dapr](#)) is a set of incrementally adoptable features that simplify the authoring of distributed, microservice-based applications. For example, Dapr provides capabilities for enabling application intercommunication, whether through messaging via pub/sub or reliable and secure service-to-service calls. Once Dapr is enabled for a container app, a secondary process is created alongside your application code that enables communication with Dapr via HTTP or gRPC.

Dapr's APIs are built on best practice industry standards, that:

- Seamlessly fit with your preferred language or framework
- Are incrementally adoptable; you can use one, several, or all dapr capabilities depending on your application's needs

Dapr is an open source, [Cloud Native Computing Foundation \(CNCF\)](#) project. The CNCF is part of the Linux Foundation and provides support, oversight, and direction for fast-growing, cloud native projects. As an alternative to deploying and managing the Dapr OSS project yourself, the Container Apps platform:

- Provides a managed and supported Dapr integration
- Handles Dapr version upgrades seamlessly
- Exposes a simplified Dapr interaction model to increase developer productivity

This guide provides insight into core Dapr concepts and details regarding the Dapr interaction model in Container Apps.

Dapr APIs

						
Service-to-service invocation	State management	Publish and subscribe	Bindings (input/output)	Actors	Observability	Secrets
Perform direct, secure, service-to-service method calls	Create long running, stateless and stateful services	Secure, scalable messaging between services	Trigger code through events from inputs Input and output bindings to external resources including databases and queues	Encapsulate code and data in reusable actor objects as a common microservices design pattern	See and measure the message calls across components and networked services	Securely access secrets from your application

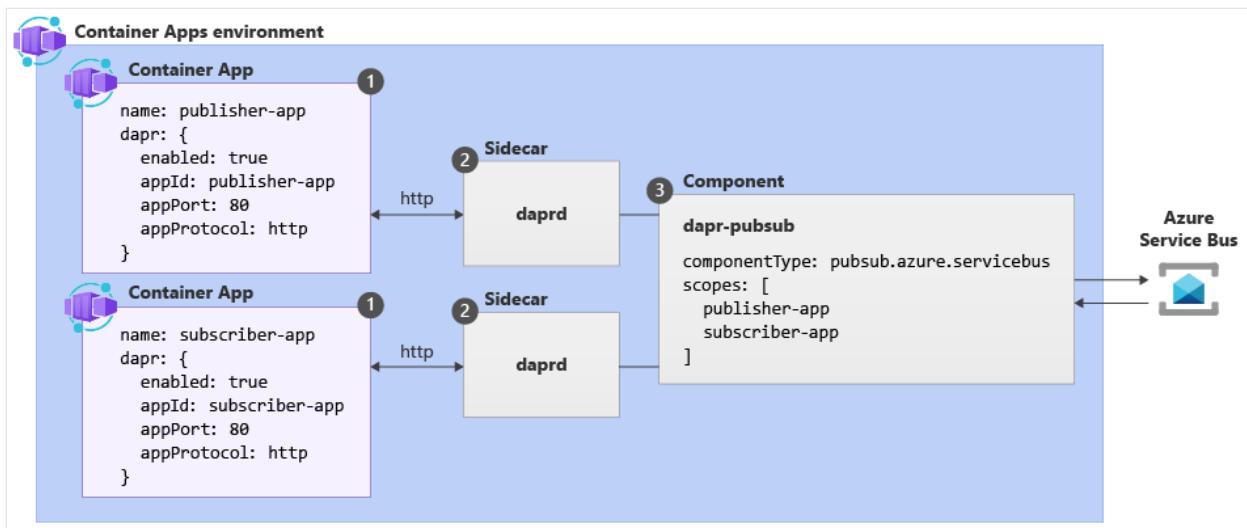
Dapr API	Description
Service-to-service invocation	Discover services and perform reliable, direct service-to-service calls with automatic mTLS authentication and encryption.
State management	Provides state management capabilities for transactions and CRUD operations.
Pub/sub	Allows publisher and subscriber container apps to intercommunicate via an intermediary message broker.
Bindings	Trigger your applications based on events
Actors	Dapr actors are message-driven, single-threaded, units of work designed to quickly scale. For example, in burst-heavy workload situations.
Observability	Send tracing information to an Application Insights backend.
Secrets	Access secrets from your application code or reference secure values in your Dapr components.

Note

The above table covers stable Dapr APIs. To learn more about using alpha APIs and features, see the [Dapr FAQ](#).

Dapr concepts overview

The following example based on the Pub/sub API is used to illustrate core concepts related to Dapr in Azure Container Apps.



Label	Dapr settings	Description
1	Container Apps with Dapr enabled	Dapr is enabled at the container app level by configuring a set of Dapr arguments. These values apply to all revisions of a given container app when running in multiple revisions mode.
2	Dapr	The fully managed Dapr APIs are exposed to each container app through a Dapr sidecar. The Dapr APIs can be invoked from your container app via HTTP or gRPC. The Dapr sidecar runs on HTTP port 3500 and gRPC port 50001.
3	Dapr component configuration	Dapr uses a modular design where functionality is delivered as a component. Dapr components can be shared across multiple container apps. The Dapr app identifiers provided in the scopes array dictate which dapr-enabled container apps will load a given component at runtime.

Dapr enablement

You can configure Dapr using various [arguments and annotations](#) based on the runtime context. Azure Container Apps provides three channels through which you can configure Dapr:

- Container Apps CLI
- Infrastructure as Code (IaC) templates, as in Bicep or Azure Resource Manager (ARM) templates
- The Azure portal

The table below outlines the currently supported list of Dapr sidecar configurations in Container Apps:

Container	Template field	Description
Apps CLI		
--enable-dapr	dapr.enabled	Enables Dapr on the container app.
--dapr-app-port	dapr.appPort	The port your application is listening on which is used by Dapr for communicating to your application
--dapr-app-protocol	dapr.appProtocol	Tells Dapr which protocol your application is using. Valid options are <code>http</code> or <code>grpc</code> . Default is <code>http</code> .
--dapr-app-id	dapr.appId	A unique Dapr identifier for your container app used for service discovery, state encapsulation and the pub/sub consumer ID.
--dapr-max-request-size	dapr.httpMaxRequestBodySize	Set the max size of request body http and grpc servers to handle uploading of large files. Default is 4 MB.
--dapr-read-buffer-size	dapr.httpReadBufferSize	Set the max size of http header read buffer in to handle when sending multi-KB headers. The default 4 KB.
--dapr-api-logging	dapr.enableApiLogging	Enables viewing the API calls from your application to the Dapr sidecar.
--dapr-log-level	dapr.logLevel	Set the log level for the Dapr sidecar. Allowed values: debug, error, info, warn. Default is <code>info</code> .

When using an IaC template, specify the following arguments in the `properties.configuration` section of the container app resource definition.

```
Bicep
```

```
Bicep
```

```
dapr: {
    enabled: true
    appId: 'nodeapp'
    appProtocol: 'http'
    appPort: 3000
}
```

The above Dapr configuration values are considered application-scope changes. When you run a container app in multiple-revision mode, changes to these settings won't

create a new revision. Instead, all existing revisions are restarted to ensure they're configured with the most up-to-date values.

Dapr components

Dapr uses a modular design where functionality is delivered as a [component ↗](#). The use of Dapr components is optional and dictated exclusively by the needs of your application.

Dapr components in container apps are environment-level resources that:

- Can provide a pluggable abstraction model for connecting to supporting external services.
- Can be shared across container apps or scoped to specific container apps.
- Can use Dapr secrets to securely retrieve configuration metadata.

Component schema

All Dapr OSS components conform to the following basic [schema ↗](#).

YAML

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: [COMPONENT-NAME]
  namespace: [COMPONENT-NAMESPACE]
spec:
  type: [COMPONENT-TYPE]
  version: v1
  initTimeout: [TIMEOUT-DURATION]
  ignoreErrors: [BOOLEAN]
  metadata:
    - name: [METADATA-NAME]
      value: [METADATA-VALUE]
```

In Container Apps, the above schema has been slightly simplified to support Dapr components and remove unnecessary fields, including `apiVersion`, `kind`, and redundant `metadata` and `spec` properties.

YAML

```
componentType: [COMPONENT-TYPE]
version: v1
initTimeout: [TIMEOUT-DURATION]
ignoreErrors: [BOOLEAN]
```

```
metadata:  
  - name: [METADATA-NAME]  
    value: [METADATA-VALUE]
```

Component scopes

By default, all Dapr-enabled container apps within the same environment load the full set of deployed components. To ensure components are loaded at runtime by only the appropriate container apps, application scopes should be used. In the example below, the component is only loaded by the two Dapr-enabled container apps with Dapr application IDs `APP-ID-1` and `APP-ID-2`:

ⓘ Note

Dapr component scopes correspond to the Dapr application ID of a container app, not the container app name.

YAML

```
componentType: [COMPONENT-TYPE]  
version: v1  
initTimeout: [TIMEOUT-DURATION]  
ignoreErrors: [BOOLEAN]  
metadata:  
  - name: [METADATA-NAME]  
    value: [METADATA-VALUE]  
scopes:  
  - [APP-ID-1]  
  - [APP-ID-2]
```

Connecting to external services via Dapr

There are a few approaches supported in container apps to securely establish connections to external services for Dapr components.

1. Using Managed Identity
2. Using a Dapr Secret Store component reference
3. Using Platform-managed Kubernetes secrets

Using managed identity

For Azure-hosted services, Dapr can use the managed identity of the scoped container apps to authenticate to the backend service provider. When using managed identity, you don't need to include secret information in a component manifest. Using managed identity is preferred as it eliminates storage of sensitive input in components and doesn't require managing a secret store.

 Note

The `azureClientId` metadata field (the client ID of the managed identity) is required for any component authenticating with user-assigned managed identity.

Using a Dapr secret store component reference

When you create Dapr components for non-AD enabled services, certain metadata fields require sensitive input values. The recommended approach for retrieving these secrets is to reference an existing Dapr secret store component that securely accesses secret information.

Here are the steps to set up a reference:

1. Create a Dapr secret store component using the Container Apps schema. The component type for all supported Dapr secret stores begins with `secretstores..`.
2. Create extra components as needed which reference this Dapr secret store component to retrieve the sensitive metadata input.

When creating a secret store component in container apps, you can provide sensitive information in the metadata section in either of the following ways:

- For an **Azure Key Vault secret store**, use managed identity to establish the connection.
- For **non-Azure secret stores**, use platform-managed Kubernetes secrets that are defined directly as part of the component manifest.

The following component showcases the simplest possible secret store configuration. In this example, publisher and subscriber applications are configured to both have a system or user-assigned managed identity with appropriate permissions on the Azure Key Vault instance.

 YAML

```
componentType: secretstores.azure.keyvault
version: v1
metadata:
```

```
- name: vaultName
  value: [your_keyvault_name]
- name: azureEnvironment
  value: "AZUREPUBLICCLOUD"
- name: azureClientId # Only required for authenticating user-assigned
  managed identity
  value: [your_managed_identity_client_id]
scopes:
- publisher-app
- subscriber-app
```

ⓘ Note

Kubernetes secrets, Local environment variables and Local file Dapr secret stores aren't supported in Container Apps. As an alternative for the upstream Dapr default Kubernetes secret store, container apps provides a platform-managed approach for creating and leveraging Kubernetes secrets.

Using Platform-managed Kubernetes secrets

This component configuration defines the sensitive value as a secret parameter that can be referenced from the metadata section. This approach can be used to connect to non-Azure services or in dev/test scenarios for quickly deploying components via the CLI without setting up a secret store or managed identity.

YAML

```
componentType: secretstores.azure.keyvault
version: v1
metadata:
- name: vaultName
  value: [your_keyvault_name]
- name: azureEnvironment
  value: "AZUREPUBLICCLOUD"
- name: azureTenantId
  value: "[your_tenant_id]"
- name: azureClientId
  value: "[your_client_id]"
- name: azureClientSecret
  secretRef: azClientSecret
secrets:
- name: azClientSecret
  value: "[your_client_secret]"
scopes:
- publisher-app
- subscriber-app
```

Referencing Dapr secret store components

Once you've created a Dapr secret store using one of the above approaches, you can reference that secret store from other Dapr components in the same environment. In the following example, the `secretStoreComponent` field is populated with the name of the secret store specified above, where the `sb-root-connectionstring` is stored.

YAML

```
componentType: pubsub.azure.servicebus.queue
version: v1
secretStoreComponent: "my-secret-store"
metadata:
  - name: connectionString
    secretRef: sb-root-connectionstring
scopes:
  - publisher-app
  - subscriber-app
```

Component examples

YAML

To create a Dapr component via the Container Apps CLI, you can use a container apps YAML manifest. When configuring multiple components, you must create and apply a separate YAML file for each component.

Azure CLI

```
az containerapp env dapr-component set --name ENVIRONMENT_NAME --
resource-group RESOURCE_GROUP_NAME --dapr-component-name pubsub --yaml
"./pubsub.yaml"
```

YAML

```
# pubsub.yaml for Azure Service Bus component
componentType: pubsub.azure.servicebus.queue
version: v1
secretStoreComponent: "my-secret-store"
metadata:
  - name: connectionString
    secretRef: sb-root-connectionstring
scopes:
  - publisher-app
  - subscriber-app
```

Limitations

Unsupported Dapr capabilities

- **Custom configuration for Dapr Observability:** Instrument your environment with Application Insights to visualize distributed tracing.
- **Dapr Configuration spec:** Any capabilities that require use of the Dapr configuration spec.
- **Declarative pub/sub subscriptions**
- **Any Dapr sidecar annotations not listed above**
- **Alpha APIs and components:** Azure Container Apps doesn't guarantee the availability of Dapr alpha APIs and features. For more information, refer to the [Dapr FAQ](#).

Known limitations

- **Actor reminders:** Require a minReplicas of 1+ to ensure reminders is always active and fires correctly.

Next Steps

Now that you've learned about Dapr and some of the challenges it solves:

- Create an Azure Dapr component via the [Azure Container Apps portal](#)
- Try [Deploying a Dapr application to Azure Container Apps using the Azure CLI or Azure Resource Manager](#).
- Walk through a tutorial [using GitHub Actions to automate changes for a multi-revision, Dapr-enabled container app](#).
- Learn how to [perform event-driven work using Dapr bindings](#).
- [Enable token authentication for Dapr requests](#).
- [Scale your Dapr applications using KEDA scalers](#)
- [Answer common questions about the Dapr integration with Azure Container Apps](#)

Scale Dapr applications with KEDA scalers

Article • 04/24/2023

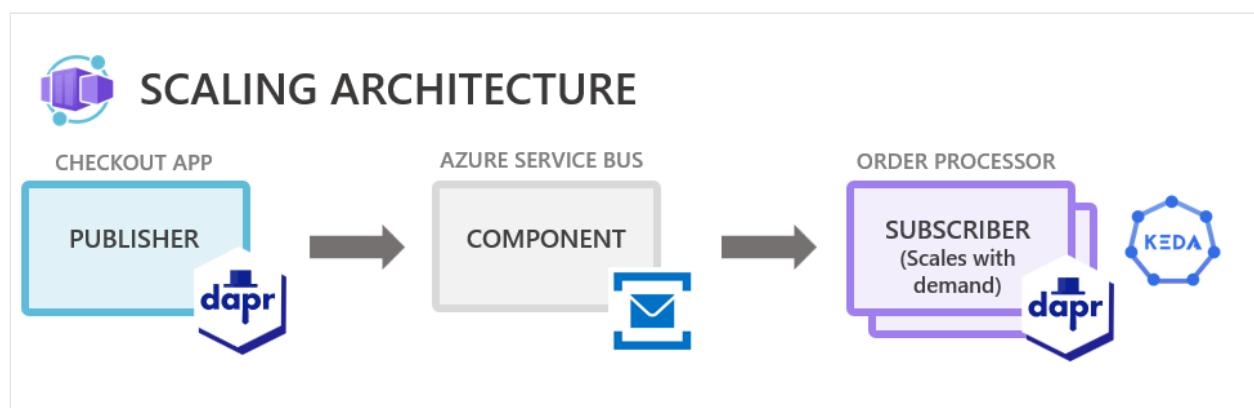
Azure Container Apps automatically scales HTTP traffic to zero. However, to scale non-HTTP traffic (like Dapr pub/sub and bindings), you can use [KEDA scalers](#) to scale your application and its Dapr sidecar up and down, based on the number of pending inbound events and messages.

This guide demonstrates how to configure the scale rules of a Dapr pub/sub application with a KEDA messaging scaler. For context, refer to the corresponding sample pub/sub applications:

- Microservice communication using pub/sub in [C#](#)
- Microservice communication using pub/sub in [JavaScript](#)
- Microservice communication using pub/sub in [Python](#)

In the above samples, the application uses the following elements:

1. The `checkout` publisher is an application that is meant to run indefinitely and never scale down to zero, despite never receiving any incoming HTTP traffic.
2. The Dapr Azure Service Bus pub/sub component.
3. An `order-processor` subscriber container app picks up messages received via the `orders` topic and processed as they arrive.
4. The scale rule for Azure Service Bus, which is responsible for scaling up the `order-processor` service and its Dapr sidecar when messages start to arrive to the `orders` topic.



Let's take a look at how to apply the scaling rules in a Dapr application.

Publisher container app

The `checkout` publisher is a headless service that runs indefinitely and never scales down to zero.

By default, [the Container Apps runtime assigns an HTTP-based scale rule to applications](#), which drives scaling based on the number of incoming HTTP requests. In the following example, `minReplicas` is set to `1`. This configuration ensures the container app doesn't follow the default behavior of scaling to zero with no incoming HTTP traffic.

Bicep

```
resource checkout 'Microsoft.App/containerApps@2022-03-01' = {
    name: 'ca-checkout-${resourceToken}'
    location: location
    identity: {
        type: 'SystemAssigned'
    }
    properties: {
        //...
        template: {
            //...
            // Scale the minReplicas to 1
            scale: {
                minReplicas: 1
                maxReplicas: 1
            }
        }
    }
}
```

Subscriber container app

The following `order-processor` subscriber app includes a custom scale rule that monitors a resource of type `azure-servicebus`. With this rule, the app (and its sidecar) scales up and down as needed based on the number of pending messages in the Bus.

Bicep

```
resource orders 'Microsoft.App/containerApps@2022-03-01' = {
    name: 'ca-orders-${resourceToken}'
    location: location
    tags: union(tags, {
        'azd-service-name': 'orders'
    })
    identity: {
        type: 'SystemAssigned'
    }
    properties: {
        managedEnvironmentId: containerAppsEnvironment.id
    }
}
```

```

configuration: {
    //...
    // Enable Dapr on the container app
    dapr: {
        enabled: true
        appId: 'orders'
        appProtocol: 'http'
        appPort: 5001
    }
    //...
}
template: {
    //...
    // Set the scale property on the order-processor resource
    scale: {
        minReplicas: 0
        maxReplicas: 10
        rules: [
            {
                name: 'topic-based-scaling'
                custom: {
                    type: 'azure-servicebus'
                    metadata: {
                        topicName: 'orders'
                        subscriptionName: 'membership-orders'
                        messageCount: '30'
                    }
                }
                auth: [
                    {
                        secretRef: 'sb-root-connectionstring'
                        triggerParameter: 'connection'
                    }
                ]
            }
        ]
    }
}
}

```

How the scaler works

Notice the `messageCount` property on the scaler's configuration in the subscriber app:

Bicep

```
{
//...
properties: {
//...
```

```
template: {
  //...
  scale: {
    //...
    rules: [
      //...
      custom: {
        //...
        metadata: {
          //...
          messageCount: '30'
        }
      }
    ]
  }
}
```

This property tells the scaler how many messages each instance of the application can process at the same time. In this example, the value is set to `30`, indicating that there should be one instance of the application created for each group of 30 messages waiting in the topic.

For example, if 150 messages are waiting, KEDA scales the app out to five instances. The `maxReplicas` property is set to `10`, meaning even with a large number of messages in the topic, the scaler never creates more than `10` instances of this application. This setting ensures you don't scale up too much and accrue too much cost.

Next steps

[Learn more about using Dapr components with Azure Container Apps.](#)

Azure Container Apps on Azure Arc (Preview)

Article • 04/27/2023

You can run Container Apps on an Azure Arc-enabled AKS or AKS-HCI cluster.

Running in an Azure Arc-enabled Kubernetes cluster allows:

- Developers to take advantage of Container Apps' features
- IT administrators to maintain corporate compliance by hosting Container Apps on internal infrastructure.

Learn to set up your Kubernetes cluster for Container Apps, via [Set up an Azure Arc-enabled Kubernetes cluster to run Azure Container Apps](#)

As you configure your cluster, you'll carry out these actions:

- **The connected cluster**, which is an Azure projection of your Kubernetes infrastructure. For more information, see [What is Azure Arc-enabled Kubernetes?](#).
- **A cluster extension**, which is a subresource of the connected cluster resource. The Container Apps extension [installs the required resources into your connected cluster](#). For more information about cluster extensions, see [Cluster extensions on Azure Arc-enabled Kubernetes](#).
- **A custom location**, which bundles together a group of extensions and maps them to a namespace for created resources. For more information, see [Custom locations on top of Azure Arc-enabled Kubernetes](#).
- **A Container Apps connected environment**, which enables configuration common across apps but not related to cluster operations. Conceptually, it's deployed into the custom location resource, and app developers create apps into this environment.

Public preview limitations

The following public preview limitations apply to Azure Container Apps on Azure Arc enabled Kubernetes.

Limitation	Details
Supported Azure regions	East US, West Europe, East Asia

Limitation	Details
Cluster networking requirement	Must support LoadBalancer service type
Feature: Managed identities	Not available
Feature: Pull images from ACR with managed identity	Not available (depends on managed identities)
Logs	Log Analytics must be configured with cluster extension; not per-site

Resources created by the Container Apps extension

When the Container Apps extension is installed on the Azure Arc-enabled Kubernetes cluster, several resources are created in the specified release namespace. These resources enable your cluster to be an extension of the [Microsoft.App](#) resource provider to support the management and operation of your apps.

Optionally, you can choose to have the extension install [KEDA](#) for event-driven scaling. However, only one KEDA installation is allowed on the cluster. If you have an existing installation, disable the KEDA installation as you install the cluster extension.

The following table describes the role of each revision created for you:

Pod	Description	Number of Instances	CPU	Memory
<code><extensionName>-k8se-activator</code>	Used as part of the scaling pipeline	2	100 millicpu	500 MB
<code><extensionName>-k8se-billing</code>	Billing record generation - Azure Container Apps on Azure Arc enabled Kubernetes is Free of Charge during preview	3	100 millicpu	100 MB
<code><extensionName>-k8se-containerapp-controller</code>	The core operator pod that creates resources on the cluster and maintains the state of components.	2	100 millicpu	1 GB
<code><extensionName>-k8se-envoy</code>	A front-end proxy layer for all data-plane http requests. It routes the inbound traffic to the correct apps.	3	1 Core	1536 MB

Pod	Description	Number of Instances	CPU	Memory
<code><extensionName>-k8se-envoy-controller</code>	Operator, which generates Envoy configuration	2	200 millicpu	500 MB
<code><extensionName>-k8se-event-processor</code>	An alternative routing destination to help with apps that have scaled to zero while the system gets the first instance available.	2	100 millicpu	500 MB
<code><extensionName>-k8se-http-scaler</code>	Monitors inbound request volume in order to provide scaling information to KEDA .	1	100 millicpu	500 MB
<code><extensionName>-k8se-keda-cosmosdb-scaler</code>	Keda Cosmos DB Scaler	1	10 m	128 MB
<code><extensionName>-k8se-keda-metrics-apiserver</code>	Keda Metrics Server	1	1 Core	1000 MB
<code><extensionName>-k8se-keda-operator</code>	Manages component updated and service endpoints for Dapr	1	100 millicpu	500 MB
<code><extensionName>-k8se-local-envoy</code>	A front-end proxy layer for all data-plane tcp requests. It routes the inbound traffic to the correct apps.	3	1 Core	1536 MB
<code><extensionName>-k8se-log-processor</code>	Gathers logs from apps and other components and sends them to Log Analytics.	2	200 millicpu	500 MB
<code><extensionName>-k8se-mdm</code>	Metrics and Logs Agent	2	500 millicpu	500 MB
dapr-metrics	Dapr metrics pod	1	100 millicpu	500 MB
dapr-operator	Manages component updates and service endpoints for Dapr	1	100 millicpu	500 MB
dapr-placement-server	Used for Actors only - creates mapping tables that map actor instances to pods	1	100 millicpu	500 MB
dapr-sentry	Manages mTLS between services and acts as a CA	2	800 millicpu	200 MB

FAQ for Azure Container Apps on Azure Arc (Preview)

- How much does it cost?
- Which Container Apps features are supported?
- Are managed identities supported?
- Are there any scaling limits?
- What logs are collected?
- What do I do if I see a provider registration error?
- Can I deploy the Container Apps extension on an ARM64 based cluster?

How much does it cost?

Azure Container Apps on Azure Arc-enabled Kubernetes is free during the public preview.

Which Container Apps features are supported?

During the preview period, certain Azure Container App features are being validated. When they're supported, their left navigation options in the Azure portal will be activated. Features that aren't yet supported remain grayed out.

Are managed identities supported?

No. Apps can't be assigned managed identities when running in Azure Arc. If your app needs an identity for working with another Azure resource, consider using an [application service principal](#) instead.

Are there any scaling limits?

All applications deployed with Azure Container Apps on Azure Arc-enabled Kubernetes are able to scale within the limits of the underlying Kubernetes cluster. If the cluster runs out of available compute resources (CPU and memory primarily), then applications scale to the number of instances of the application that Kubernetes can schedule with available resource.

What logs are collected?

Logs for both system components and your applications are written to standard output.

Both log types can be collected for analysis using standard Kubernetes tools. You can also configure the application environment cluster extension with a [Log Analytics workspace](#), and it sends all logs to that workspace.

By default, logs from system components are sent to the Azure team. Application logs aren't sent. You can prevent these logs from being transferred by setting `logProcessor.enabled=false` as an extension configuration setting. This configuration setting will also disable forwarding of application to your Log Analytics workspace. Disabling the log processor might affect the time needed for any support cases, and you'll be asked to collect logs from standard output through some other means.

What do I do if I see a provider registration error?

As you create an Azure Container Apps connected environment resource, some subscriptions might see the "No registered resource provider found" error. The error details might include a set of locations and api versions that are considered valid. If this error message is returned, the subscription must be re-registered with the `Microsoft.App` provider. Re-registering the provider has no effect on existing applications or APIs. To re-register, use the Azure CLI to run `az provider register --namespace Microsoft.App --wait`. Then reattempt the connected environment command.

Can I deploy the Container Apps extension on an ARM64 based cluster?

ARM64 based clusters aren't supported at this time.

Extension Release Notes

Container Apps extension v1.0.46 (December 2022)

- Initial public preview release of Container apps extension

Container Apps extension v1.0.47 (January 2023)

- Upgrade of Envoy to 1.0.24

Container Apps extension v1.0.48 (February 2023)

- Add probes to EasyAuth container(s)
- Increased memory limit for dapr-operator
- Added prevention of platform header overwriting

Container Apps extension v1.0.49 (February 2023)

- Upgrade of KEDA to 2.9.1
- Upgrade of Dapr to 1.9.5
- Increase Envoy Controller resource limits to 200 m CPU
- Increase Container App Controller resource limits to 1-GB memory
- Reduce EasyAuth sidecar resource limits to 50 m CPU
- Resolve KEDA error logging for missing metric values

Container Apps extension v1.0.50 (March 2023)

- Updated logging images in sync with Public Cloud

Container Apps extension v1.5.1 (April 2023)

- New versioning number format
- Upgrade of Dapr to 1.10.4
- Maintain scale of Envoy after deployments of new revisions
- Change to when default startup probes are added to a container, if developer doesn't define both startup and readiness probes, then default startup probes are added
- Adds CONTAINER_APP_REPLICA_NAME environment variable to custom containers
- Improvement in performance when multiple revisions are stopped

Next steps

[Create a Container Apps connected environment \(Preview\)](#)

Connect to services in Azure Container Apps (preview)

Article • 05/23/2023

As you develop applications in Azure Container Apps, you often need to connect to different services.

Rather than creating services ahead of time and manually connecting them to your container app, you can quickly create instances of development-grade services that are designed for nonproduction environments known as "dev services".

Dev services allow you to use OSS services without the burden of manual downloads, creation, and configuration.

Services available as dev services include:

- Open-source Redis
- Open-source PostgreSQL

Once you're ready for your app to use a production level service, you can connect your application to an Azure managed service.

Features

Dev services come with the following features:

- **Scope:** The service runs in the same environment as the connected container app.
- **Scaling:** The service can scale in to zero when there's no demand for the service.
- **Pricing:** Service billing falls under consumption-based pricing. Billing only happens when instances of the service are running.
- **Storage:** The service uses persistent storage to ensure there's no data loss as a service scales in to zero.
- **Revisions:** Anytime you change a dev service, a new revision of your container app is created.

See the service-specific features for managed services.

Binding

Both dev mode and managed services connect to a container via a "binding".

The Container Apps runtime binds a container app to a service by:

- Discovering the service
- Extracting networking and connection configuration values
- Injecting configuration and connection information into container app environment variables

Once a binding is established, the container app can read these configuration and connection values from environment variables.

Development vs production

As you move from development to production, you can move from a dev service to a managed service.

The following table shows you which service to use in development, and which service to use in production.

Functionality	dev service	Production managed service
Cache	Open-source Redis	Azure Cache for Redis
Database	N/A	Azure Cosmos DB
Database	Open-source PostgreSQL	Azure Database for PostgreSQL Flexible Service

You're responsible for data continuity between development and production environments.

Manage a service

To connect a service to an application, you first need to create the service.

Use the `service` command with `containerapp create` to create a new service.

CLI

```
az containerapp service redis create \
--name myredis \
--environment myenv
```

This command creates a new Redis service called `myredis` in a Container Apps environment called `myenv`.

To bind a service to an application, use the `--bind` argument for `containerapp create`.

CLI

```
az containerapp create \
--name myapp \
--image myimage \
--bind myredis \
--environment myenv
```

This command features the typical Container App `create` with the `--bind` argument.

The bind argument tells the Container Apps runtime to connect a service to the application.

The `--bind` argument is available to the `create` or `update` commands.

To disconnect a service from an application, use the `--unbind` argument on the `update` command

The following example shows you how to unbind a service.

CLI

```
az containerapp update --name myapp --unbind myredis
```

For a full tutorial on connecting to services, see [Connect services in Azure Container Apps](#).

For more information on the service commands and arguments, see the [az containerapp](#) reference.

Limitations

- dev services are in public preview.
- Any container app created before May 23, 2023 isn't eligible to use dev services.
- dev services come with minimal guarantees. For instance, they're automatically restarted if they crash, however there's no formal quality of service or high-availability guarantees associated with them. For production workloads, use Azure-managed services.

Next steps

[Connect services to a container app](#)

Deploy Azure Container Apps with the az containerapp up command

Article • 03/20/2023

The `az containerapp up` (or `up`) command is the fastest way to deploy an app in Azure Container Apps from an existing image, local source code or a GitHub repo. With this single command, you can have your container app up and running in minutes.

The `az containerapp up` command is a streamlined way to create and deploy container apps that primarily use default settings. However, you'll need to run other CLI commands to configure more advanced settings:

- Dapr: [az containerapp dapr enable](#)
- Secrets: [az containerapp secret set](#)
- Transport protocols: [az containerapp ingress update](#)

To customize your container app's resource or scaling settings, you can use the `up` command and then the `az containerapp update` command to change these settings.

Note that the `az containerapp up` command isn't an abbreviation of the `az containerapp update` command.

The `up` command can create or use existing resources including:

- Resource group
- Azure Container Registry
- Container Apps environment and Log Analytics workspace
- Your container app

The command can build and push a container image to an Azure Container Registry (ACR) when you provide local source code or a GitHub repo. When you're working from a GitHub repo, it creates a GitHub Actions workflow that automatically builds and pushes a new container image when you commit changes to your GitHub repo.

If you need to customize the Container Apps environment, first create the environment using the `az containerapp env create` command. If you don't provide an existing environment, the `up` command looks for one in your resource group and, if found, uses that environment. If not found, it creates an environment with a Log Analytics workspace.

To learn more about the `az containerapp up` command and its options, see [az containerapp up](#).

Prerequisites

Requirement	Instructions
Azure account	If you don't have one, create an account for free . You need the <i>Contributor</i> or <i>Owner</i> permission on the Azure subscription to proceed. Refer to Assign Azure roles using the Azure portal for details.
GitHub Account	If you use a GitHub repo, sign up for free .
Azure CLI	Install the Azure CLI .
Local source code	You need to have a local source code directory if you use local source code.
Existing Image	If you use an existing image, you'll need your registry server, image name, and tag. If you're using a private registry, you'll need your credentials.

Set up

1. Log in to Azure with the Azure CLI.

```
Azure CLI
```

```
az login
```

2. Next, install the Azure Container Apps extension for the CLI.

```
Azure CLI
```

```
az extension add --name containerapp --upgrade
```

3. Now that the current extension or module is installed, register the `Microsoft.App` namespace.

```
Azure CLI
```

```
az provider register --namespace Microsoft.App
```

4. Register the `Microsoft.OperationalInsights` provider for the Azure Monitor Log Analytics workspace.

```
Azure CLI
```

```
az provider register --namespace Microsoft.OperationalInsights
```

Deploy from an existing image

You can deploy a container app that uses an existing image in a public or private container registry. If you are deploying from a private registry, you'll need to provide your credentials using the `--registry-server`, `--registry-username`, and `--registry-password` options.

In this example, the `az containerapp up` command performs the following actions:

1. Creates a resource group.
2. Creates an environment and Log Analytics workspace.
3. Creates and deploys a container app that pulls the image from a public registry.
4. Sets the container app's ingress to external with a target port set to the specified value.

Run the following command to deploy a container app from an existing image. Replace the <Placeholders> with your values.

Azure CLI

```
az containerapp up \
--name <CONTAINER_APP_NAME> \
--image <REGISTRY_SERVER>/<IMAGE_NAME>:<TAG> \
--ingress external \
--target-port <PORT_NUMBER>
```

You can use the `up` command to redeploy a container app. If you want to redeploy with a new image, use the `--image` option to specify a new image. Ensure that the `--resource-group` and `environment` options are set to the same values as the original deployment.

Azure CLI

```
az containerapp up \
--name <CONTAINER_APP_NAME> \
--image <REGISTRY_SERVER>/<IMAGE_NAME>:<TAG> \
--resource-group <RESOURCE_GROUP_NAME> \
--environment <ENVIRONMENT_NAME> \
--ingress external \
--target-port <PORT_NUMBER>
```

Deploy from local source code

When you use the `up` command to deploy from a local source, it builds the container image, pushes it to a registry, and deploys the container app. It creates the registry in Azure Container Registry if you don't provide one.

The command can build the image with or without a Dockerfile. If building without a Dockerfile the following languages are supported:

- .NET
- Node.js
- PHP
- Python
- Ruby
- Go

The following example shows how to deploy a container app from local source code.

In the example, the `az containerapp up` command performs the following actions:

1. Creates a resource group.
2. Creates an environment and Log Analytics workspace.
3. Creates a registry in Azure Container Registry.
4. Builds the container image (using the Dockerfile if it exists).
5. Pushes the image to the registry.
6. Creates and deploys the container app.

Run the following command to deploy a container app from local source code:

Azure CLI

```
az containerapp up \
--name <CONTAINER_APP_NAME> \
--source <SOURCE_DIRECTORY> \
--ingress external
```

When the Dockerfile includes the EXPOSE instruction, the `up` command configures the container app's ingress and target port using the information in the Dockerfile.

If you've configured ingress through your Dockerfile or your app doesn't require ingress, you can omit the `ingress` option.

The output of the command includes the URL for the container app.

If there's a failure, you can run the command again with the `--debug` option to get more information about the failure. If the build fails without a Dockerfile, you can try adding a Dockerfile and running the command again.

To use the `az containerapp up` command to redeploy your container app with an updated image, include the `--resource-group` and `--environment` arguments. The following example shows how to redeploy a container app from local source code.

1. Make changes to the source code.
2. Run the following command:

Azure CLI

```
az containerapp up \
--name <CONTAINER_APP_NAME> \
--source <SOURCE_DIRECTORY> \
--resource-group <RESOURCE_GROUP_NAME> \
--environment <ENVIRONMENT_NAME>
```

Deploy from a GitHub repository

When you use the `az containerapp up` command to deploy from a GitHub repository, it generates a GitHub Actions workflow that builds the container image, pushes it to a registry, and deploys the container app. The command creates the registry in Azure Container Registry if you don't provide one.

A Dockerfile is required to build the image. When the Dockerfile includes the EXPOSE instruction, the command configures the container app's ingress and target port using the information in the Dockerfile.

The following example shows how to deploy a container app from a GitHub repository.

In the example, the `az containerapp up` command performs the following actions:

1. Creates a resource group.
2. Creates an environment and Log Analytics workspace.
3. Creates a registry in Azure Container Registry.
4. Builds the container image using the Dockerfile.
5. Pushes the image to the registry.
6. Creates and deploys the container app.
7. Creates a GitHub Actions workflow to build the container image and deploy the container app when future changes are pushed to the GitHub repository.

To deploy an app from a GitHub repository, run the following command:

Azure CLI

```
az containerapp up \
--name <CONTAINER_APP_NAME> \
--repo <GitHub repository URL> \
--ingress external
```

If you've configured ingress through your Dockerfile or your app doesn't require ingress, you can omit the `ingress` option.

Because the `up` command creates a GitHub Actions workflow, rerunning it to deploy changes to your app's image will have the unwanted effect of creating multiple workflows. Instead, push changes to your GitHub repository, and the GitHub workflow will automatically build and deploy your app. To change the workflow, edit the workflow file in GitHub.

Next steps

[Quickstart: Deploy your code to Azure Container Apps](#)

Set scaling rules in Azure Container Apps

Article • 04/09/2023

Azure Container Apps manages automatic horizontal scaling through a set of declarative scaling rules. As a container app revision scales out, new instances of the revision are created on-demand. These instances are known as replicas.

Adding or editing scaling rules creates a new revision of your container app. A revision is an immutable snapshot of your container app. See revision [change types](#) to review which types of changes trigger a new revision.

Scale definition

Scaling is defined by the combination of limits and rules.

- **Limits** are the minimum and maximum possible number of replicas per revision as your container app scales.

Scale limit	Default value	Min value	Max value
Minimum number of replicas per revision	0	0	30
Maximum number of replicas per revision	10	1	30

To request an increase in maximum replica amounts for your container app, [submit a support ticket](#).

- **Rules** are the criteria used by Container Apps to decide when to add or remove replicas.

[Scale rules](#) are implemented as HTTP, TCP, or custom.

As you define your scaling rules, keep in mind the following items:

- You aren't billed usage charges if your container app scales to zero.
- Replicas that aren't processing, but remain in memory may be billed at a lower "idle" rate. For more information, see [Billing](#).
- If you want to ensure that an instance of your revision is always running, set the minimum number of replicas to 1 or higher.

Scale rules

Scaling is driven by three different categories of triggers:

- [HTTP](#): Based on the number of concurrent HTTP requests to your revision.
- [TCP](#): Based on the number of concurrent TCP connections to your revision.
- [Custom](#): Based on CPU, memory, or supported event-driven data sources such as:
 - Azure Service Bus
 - Azure Event Hubs
 - Apache Kafka
 - Redis

If you define more than one scale rule, the container app begins to scale once the first condition of any rules is met.

HTTP

With an HTTP scaling rule, you have control over the threshold of concurrent HTTP requests that determines how your container app revision scales.

In the following example, the revision scales out up to five replicas and can scale in to zero. The scaling property is set to 100 concurrent requests per second.

Example

Define an HTTP scale rule using the `--scale-rule-http-concurrency` parameter in the [create](#) or [update](#) commands.

CLI parameter	Description	Default value	Min value	Max value
<code>--scale-rule-http-concurrency</code>	When the number of concurrent HTTP requests exceeds this value, then another replica is added. Replicas continue to add to the pool up to the <code>max-replicas</code> amount.	10	1	n/a

Azure CLI

```
az containerapp create \
--name <CONTAINER_APP_NAME> \
--resource-group <RESOURCE_GROUP> \
--environment <ENVIRONMENT_NAME> \
--image <CONTAINER_IMAGE_LOCATION>
--min-replicas 0 \
```

```
--max-replicas 5 \
--scale-rule-name azure-http-rule \
--scale-rule-type http \
--scale-rule-http-concurrency 100
```

TCP

With a TCP scaling rule, you have control over the threshold of concurrent TCP connections that determines how your app scales.

In the following example, the container app revision scales out up to five replicas and can scale in to zero. The scaling threshold is set to 100 concurrent connections per second.

Example

Define a TCP scale rule using the `--scale-rule-tcp-concurrency` parameter in the [create](#) or [update](#) commands.

CLI parameter	Description	Default value	Min value	Max value
<code>--scale-rule-tcp-concurrency</code>	When the number of concurrent TCP connections exceeds this value, then another replica is added. Replicas will continue to be added up to the <code>max-replicas</code> amount as the number of concurrent connections increase.	10	1	n/a

Azure CLI

```
az containerapp create \
--name <CONTAINER_APP_NAME> \
--resource-group <RESOURCE_GROUP> \
--environment <ENVIRONMENT_NAME> \
--image <CONTAINER_IMAGE_LOCATION>
--min-replicas 0 \
--max-replicas 5 \
--scale-rule-name azure-tcp-rule \
--scale-rule-type tcp \
--scale-rule-tcp-concurrency 100
```

Custom

You can create a custom Container Apps scaling rule based on any [ScaledObject](#) -based [KEDA scaler](#) with these defaults:

Defaults	Seconds
Polling interval	30
Cool down period	300

The following example demonstrates how to create a custom scale rule.

Example

This example shows how to convert an [Azure Service Bus scaler](#) to a Container Apps scale rule, but you use the same process for any other [ScaledObject](#)-based [KEDA scaler](#) specification.

For authentication, KEDA scaler authentication parameters convert into [Container Apps secrets](#).

1. From the KEDA scaler specification, find the `type` value.

```
yml

triggers:
- type: azure-servicebus
  metadata:
    queueName: my-queue
    namespace: service-bus-namespace
    messageCount: "5"
```

2. In the CLI command, set the `--scale-rule-type` parameter to the specification `type` value.

```
Bash

az containerapp create \
  --name <CONTAINER_APP_NAME> \
  --resource-group <RESOURCE_GROUP> \
  --environment <ENVIRONMENT_NAME> \
  --image <CONTAINER_IMAGE_LOCATION>
  --min-replicas 0 \
  --max-replicas 5 \
  --secrets "connection-string-secret=<SERVICE_BUS_CONNECTION_STRING>" \
  \
  --scale-rule-name azure-servicebus-queue-rule \
  --scale-rule-type azure-servicebus \
```

```
--scale-rule-metadata "queueName=my-queue" \
    "namespace=service-bus-namespace" \
    "messageCount=5" \
--scale-rule-auth "connection=connection-string-secret"
```

3. From the KEDA scaler specification, find the `metadata` values.

yml

```
triggers:
- type: azure-servicebus
  metadata:
    queueName: my-queue
    namespace: service-bus-namespace
    messageCount: "5"
```

4. In the CLI command, set the `--scale-rule-metadata` parameter to the metadata values.

You'll need to transform the values from a YAML format to a key/value pair for use on the command line. Separate each key/value pair with a space.

Bash

```
az containerapp create \
  --name <CONTAINER_APP_NAME> \
  --resource-group <RESOURCE_GROUP> \
  --environment <ENVIRONMENT_NAME> \
  --image <CONTAINER_IMAGE_LOCATION>
  --min-replicas 0 \
  --max-replicas 5 \
  --secrets "connection-string-secret=<SERVICE_BUS_CONNECTION_STRING>" \
  \
  --scale-rule-name azure-servicebus-queue-rule \
  --scale-rule-type azure-servicebus \
  --scale-rule-metadata "queueName=my-queue" \
    "namespace=service-bus-namespace" \
    "messageCount=5" \
  --scale-rule-auth "connection=connection-string-secret"
```

Authentication

A KEDA scaler may support using secrets in a [TriggerAuthentication](#) that is referenced by the `authenticationRef` property. You can map the `TriggerAuthentication` object to the Container Apps scale rule.

ⓘ Note

Container Apps scale rules only support secret references. Other authentication types such as pod identity are not supported.

1. Find the `TriggerAuthentication` object referenced by the KEDA `ScaledObject` specification. Identify each `secretTargetRef` of the `TriggerAuthentication` object.

yml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secrets
  namespace: my-project
type: Opaque
data:
  connection-string-secret: <SERVICE_BUS_CONNECTION_STRING>
---
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: azure-servicebus-auth
spec:
  secretTargetRef:
    - parameter: connection
      name: my-secrets
      key: connection-string-secret
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: azure-servicebus-queue-rule
  namespace: default
spec:
  scaleTargetRef:
    name: my-scale-target
  triggers:
    - type: azure-servicebus
      metadata:
        queueName: my-queue
        namespace: service-bus-namespace
        messageCount: "5"
      authenticationRef:
        name: azure-servicebus-auth
```

2. In your container app, create the `secrets` that match the `secretTargetRef` properties.

3. In the CLI command, set parameters for each `secretTargetRef` entry.

a. Create a secret entry with the `--secrets` parameter. If there are multiple secrets, separate them with a space.

b. Create an authentication entry with the `--scale-rule-auth` parameter. If there are multiple entries, separate them with a space.

Bash

```
az containerapp create \
  --name <CONTAINER_APP_NAME> \
  --resource-group <RESOURCE_GROUP> \
  --environment <ENVIRONMENT_NAME> \
  --image <CONTAINER_IMAGE_LOCATION>
  --min-replicas 0 \
  --max-replicas 5 \
  --secrets "connection-string-secret=<SERVICE_BUS_CONNECTION_STRING>" \
  \
  --scale-rule-name azure-servicebus-queue-rule \
  --scale-rule-type azure-servicebus \
  --scale-rule-metadata "queueName=my-queue" \
    "namespace=service-bus-namespace" \
    "messageCount=5" \
  --scale-rule-auth "connection=connection-string-secret"
```

Default scale rule

If you don't create a scale rule, the default scale rule is applied to your container app.

Trigger	Min replicas	Max replicas
HTTP	0	10

ⓘ Important

Make sure you create a scale rule or set `minReplicas` to 1 or more if you don't enable ingress. If ingress is disabled and you don't define a `minReplicas` or a custom scale rule, then your container app will scale to zero and have no way of starting back up.

Considerations

- In "multiple revision" mode, adding a new scale trigger creates a new revision of your application but your old revision remains available with the old scale rules. Use the [Revision management](#) page to manage traffic allocations.
- No usage charges are incurred when an application scales to zero. For more pricing information, see [Billing in Azure Container Apps](#).

Unsupported KEDA capabilities

- KEDA ScaledJobs aren't supported. For more information, see [KEDA Scaling Jobs](#).

Known limitations

- Vertical scaling isn't supported.
- Replica quantities are a target amount, not a guarantee.
- If you're using [Dapr actors](#) to manage states, you should keep in mind that scaling to zero isn't supported. Dapr uses virtual actors to manage asynchronous calls, which means their in-memory representation isn't tied to their identity or lifetime.

Next steps

[Manage secrets](#)

Manage secrets in Azure Container Apps

Article • 05/23/2023

Azure Container Apps allows your application to securely store sensitive configuration values. Once secrets are defined at the application level, secured values are available to revisions in your container apps. Additionally, you can reference secured values inside scale rules. For information on using secrets with Dapr, refer to [Dapr integration](#).

- Secrets are scoped to an application, outside of any specific revision of an application.
- Adding, removing, or changing secrets doesn't generate new revisions.
- Each application revision can reference one or more secrets.
- Multiple revisions can reference the same secret(s).

An updated or deleted secret doesn't automatically affect existing revisions in your app. When a secret is updated or deleted, you can respond to changes in one of two ways:

1. Deploy a new revision.
2. Restart an existing revision.

Before you delete a secret, deploy a new revision that no longer references the old secret. Then deactivate all revisions that reference the secret.

Defining secrets

Secrets are defined as a set of name/value pairs. The value of each secret is specified directly or as a reference to a secret stored in Azure Key Vault.

Store secret value in Container Apps

When you define secrets through the portal, or via different command line options.

Azure portal

1. Go to your container app in the [Azure portal](#).
2. Under the *Settings* section, select **Secrets**.
3. Select **Add**.
4. In the *Add secret* context pane, enter the following information:

- **Name:** The name of the secret.
- **Type:** Select **Container Apps Secret**.
- **Value:** The value of the secret.

5. Select **Add**.

Reference secret from Key Vault (preview)

When you define a secret, you create a reference to a secret stored in Azure Key Vault. Container Apps automatically retrieves the secret value from Key Vault and makes it available as a secret in your container app.

To reference a secret from Key Vault, you must first enable managed identity in your container app and grant the identity access to the Key Vault secrets.

To enable managed identity in your container app, see [Managed identities](#).

To grant access to Key Vault secrets, [create an access policy](#) in Key Vault for the managed identity you created. Enable the "Get" secret permission on this policy.

Azure portal

1. Go to your container app in the [Azure portal](#).
2. Under the *Settings* section, select **Identity**.
3. In the *System assigned* tab, select **On**.
4. Select **Save** to enable system-assigned managed identity.
5. Under the *Settings* section, select **Secrets**.
6. Select **Add**.
7. In the *Add secret* context pane, enter the following information:
 - **Name:** The name of the secret.
 - **Type:** Select **Key Vault reference**.
 - **Key Vault secret URL:** The URI of your secret in Key Vault.
 - **Identity:** The identity to use to retrieve the secret from Key Vault.
8. Select **Add**.

Note

If you're using **UDR With Azure Firewall**, you will need to add the `AzureKeyVault` service tag and the `login.microsoft.com` FQDN to the allow list for your firewall. Refer to [configuring UDR with Azure Firewall](#) to decide which additional service tags you need.

Key Vault secret URI and secret rotation

The Key Vault secret URI must be in one of the following formats:

- `https://myvault.vault.azure.net/secrets/mysecret/ec96f02080254f109c51a1f14cdb1931`: Reference a specific version of a secret.
- `https://myvault.vault.azure.net/secrets/mysecret`: Reference the latest version of a secret.

If a version isn't specified in the URI, then the app uses the latest version that exists in the key vault. When newer versions become available, the app automatically retrieves the latest version within 30 minutes. Any active revisions that reference the secret in an environment variable is automatically restarted to pick up the new value.

For full control of which version of a secret is used, specify the version in the URI.

Referencing secrets in environment variables

After declaring secrets at the application level as described in the [defining secrets](#) section, you can reference them in environment variables when you create a new revision in your container app. When an environment variable references a secret, its value is populated with the value defined in the secret.

Example

The following example shows an application that declares a connection string at the application level. This connection is referenced in a container environment variable and in a scale rule.

Azure portal

After you've [defined a secret](#) in your container app, you can reference it in an environment variable when you create a new revision.

1. Go to your container app in the [Azure portal](#).
2. Open the *Revision management* page.
3. Select **Create new revision**.
4. In the *Create and deploy new revision* page, select a container.
5. In the *Environment variables* section, select **Add**.
6. Enter the following information:
 - **Name:** The name of the environment variable.
 - **Source:** Select **Reference a secret**.
 - **Value:** Select the secret you want to reference.
7. Select **Save**.
8. Select **Create** to create the new revision.

Mounting secrets in a volume

After declaring secrets at the application level as described in the [defining secrets](#) section, you can reference them in volume mounts when you create a new revision in your container app. When you mount secrets in a volume, each secret is mounted as a file in the volume. The file name is the name of the secret, and the file contents are the value of the secret. You can load all secrets in a volume mount, or you can load specific secrets.

Example

Azure portal

After you've [defined a secret](#) in your container app, you can reference it in a volume mount when you create a new revision.

1. Go to your container app in the [Azure portal](#).
2. Open the *Revision management* page.
3. Select **Create new revision**.
4. In the *Create and deploy new revision* page.

5. Select a container and select **Edit**.
6. In the *Volume mounts* section, expand the **Secrets** section.
7. Select **Create new volume**.
8. Enter the following information:
 - **Name:** mysecrets
 - **Mount all secrets:** enabled

 **Note**

If you want to load specific secrets, disable **Mount all secrets** and select the secrets you want to load.

9. Select **Add**.
10. Under *Volume name*, select **mysecrets**.
11. Under *Mount path*, enter **/mnt/secrets**.
12. Select **Save**.
13. Select **Create** to create the new revision with the volume mount.

Next steps

[Containers](#)

Managed identities in Azure Container Apps

Article • 03/22/2023

A managed identity from Azure Active Directory (Azure AD) allows your container app to access other Azure AD-protected resources. For more about managed identities in Azure AD, see [Managed identities for Azure resources](#).

Your container app can be granted two types of identities:

- A **system-assigned identity** is tied to your container app and is deleted when your container app is deleted. An app can only have one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your container app and other resources. A container app can have multiple user-assigned identities. The identity exists until you delete them.

Why use a managed identity?

You can use a managed identity in a running container app to authenticate to any [service that supports Azure AD authentication](#).

With managed identities:

- Your app connects to resources with the managed identity. You don't need to manage credentials in your container app.
- You can use role-based access control to grant specific permissions to a managed identity.
- System-assigned identities are automatically created and managed. They're deleted when your container app is deleted.
- You can add and delete user-assigned identities and assign them to multiple resources. They're independent of your container app's life cycle.
- You can use managed identity to [authenticate with a private Azure Container Registry](#) without a username and password to pull containers for your Container App.
- You can use [managed identity to create connections for Dapr-enabled applications via Dapr components](#)

Common use cases

System-assigned identities are best for workloads that:

- are contained within a single resource
- need independent identities

User-assigned identities are ideal for workloads that:

- run on multiple resources and can share a single identity
- need pre-authorization to a secure resource

Limitations

Using managed identities in scale rules isn't supported. You'll still need to include the connection string or key in the `secretRef` of the scaling rule.

Configure managed identities

You can configure your managed identities through:

- the Azure portal
- the Azure CLI
- your Azure Resource Manager (ARM) template

When a managed identity is added, deleted, or modified on a running container app, the app doesn't automatically restart and a new revision isn't created.

 **Note**

When adding a managed identity to a container app deployed before April 11, 2022, you must create a new revision.

Add a system-assigned identity

Azure portal

1. In the left navigation of your container app's page, scroll down to the **Settings** group.
2. Select **Identity**.
3. Within the **System assigned** tab, switch **Status** to **On**. Select **Save**.

The screenshot shows the Azure portal interface for a 'Container App' named 'music-store'. In the top navigation bar, there's a search bar and several icons for account management. Below the header, the app name 'music-store' is displayed with a key icon and the text 'Container App'. On the left, a sidebar lists various settings: Overview, Access control (IAM), Tags, Diagnose and solve problems, Secrets, Ingress, Continuous deployment, Identity (which is selected and highlighted in grey), and Locks. The main content area is titled 'System assigned' and contains a note about managed identities. At the bottom right of this section is a 'Status' switch with two options: 'Off' and 'On', with 'On' being the current selection. A red box is drawn around this switch. Below the status switch are buttons for Save, Discard, Refresh, and Got feedback?.

Add a user-assigned identity

Configuring a container app with a user-assigned identity requires that you first create the identity then add its resource identifier to your container app's configuration. You can create user-assigned identities via the Azure portal or the Azure CLI. For information on creating and managing user-assigned identities, see [Manage user-assigned managed identities](#).

Azure portal

First, you'll need to create a user-assigned identity resource.

1. Create a user-assigned managed identity resource according to the steps found in [Manage user-assigned managed identities](#).
2. In the left navigation for your container app's page, scroll down to the **Settings** group.
3. Select **Identity**.
4. Within the **User assigned** tab, select **Add**.
5. Search for the identity you created earlier and select it. Select **Add**.

The screenshot shows the Azure portal interface for a 'music-store' container app. On the left, there's a sidebar with various settings like Overview, Access control (IAM), Tags, Diagnose and solve problems, and Identity (which is currently selected). The main area shows a table for 'User assigned' identities, which is currently empty. A modal window titled 'Add user assigned managed identity...' is open, allowing the selection of identities from a list. The 'music-store-user-identity' entry is highlighted with a red box.

Configure a target resource

For some resources, you'll need to configure role assignments for your app's managed identity to grant access. Otherwise, calls from your app to services, such as Azure Key Vault and Azure SQL Database, will be rejected even if you use a valid token for that identity. To learn more about Azure role-based access control (Azure RBAC), see [What is RBAC?](#). To learn more about which resources support Azure Active Directory tokens, see [Azure services that support Azure AD authentication](#).

Important

The back-end services for managed identities maintain a cache per resource URI for around 24 hours. If you update the access policy of a particular target resource and immediately retrieve a token for that resource, you may continue to get a cached token with outdated permissions until that token expires. There's currently no way to force a token refresh.

Connect to Azure services in app code

With managed identities, an app can obtain tokens to access Azure resources that use Azure Active Directory, such as Azure SQL Database, Azure Key Vault, and Azure Storage. These tokens represent the application accessing the resource, and not any specific user of the application.

Container Apps provides an internally accessible [REST endpoint](#) to retrieve tokens. The REST endpoint can be accessed from within the app with a standard HTTP GET, which can be implemented with a generic HTTP client in every language. For .NET, JavaScript, Java, and Python, the Azure Identity client library provides an abstraction over this REST endpoint. Connecting to other Azure services is as simple as adding a credential object to the service-specific client.

 **Note**

When using Azure Identity client library, the user-assigned managed identity client id must be specified.

.NET

 **Note**

When connecting to Azure SQL data sources with [Entity Framework Core](#), consider [using Microsoft.Data.SqlClient](#), which provides special connection strings for managed identity connectivity.

For .NET apps, the simplest way to work with a managed identity is through the [Azure Identity client library for .NET](#). See the respective documentation headings of the client library for information:

- [Add Azure Identity client library to your project](#)
- [Access Azure service with a system-assigned identity](#)
- [Access Azure service with a user-assigned identity](#)

The linked examples use `DefaultAzureCredential`. It's useful for most the scenarios because the same pattern works in Azure (with managed identities) and on your local machine (without managed identities).

View managed identities

You can show the system-assigned and user-assigned managed identities using the following Azure CLI command. The output shows the managed identity type, tenant IDs and principal IDs of all managed identities assigned to your container app.

Azure CLI

```
az containerapp identity show --name <APP_NAME> --resource-group  
<GROUP_NAME>
```

Remove a managed identity

When you remove a system-assigned identity, it's deleted from Azure Active Directory. System-assigned identities are also automatically removed from Azure Active Directory when you delete the container app resource itself. Removing user-assigned managed identities from your container app doesn't remove them from Azure Active Directory.

Azure portal

1. In the left navigation of your app's page, scroll down to the **Settings** group.
2. Select **Identity**. Then follow the steps based on the identity type:
 - **System-assigned identity:** Within the **System assigned** tab, switch **Status** to **Off**. Select **Save**.
 - **User-assigned identity:** Select the **User assigned** tab, select the checkbox for the identity, and select **Remove**. Select **Yes** to confirm.

Next steps

[Monitor an app](#)

Azure Container Apps image pull with managed identity

Article • 03/08/2023

You can pull images from private repositories in Microsoft Azure Container Registry using managed identities for authentication to avoid the use of administrative credentials. You can use a system-assigned or user-assigned managed identity to authenticate with Azure Container Registry.

With a system-assigned managed identity, the identity is created and managed by Azure Container Apps. The identity is tied to your container app and is deleted when your app is deleted. With a user-assigned managed identity, you create and manage the identity outside of Azure Container Apps. It can be assigned to multiple Azure resources, including Azure Container Apps.

This article describes how to use the Azure portal to configure your container app to use user-assigned and system-assigned managed identities to pull images from private Azure Container Registry repositories.

User-assigned managed identity

The following steps describe the process to configure your container app to use a user-assigned managed identity to pull images from private Azure Container Registry repositories.

1. Create a container app with a public image.
2. Add the user-assigned managed identity to the container app.
3. Create a container app revision with a private image and the system-assigned managed identity.

Prerequisites

- An Azure account with an active subscription.
 - If you don't have one, you [can create one for free](#).
- A private Azure Container Registry containing an image you want to pull.
- Create a user-assigned managed identity. For more information, see [Create a user-assigned managed identity](#).

Create a container app

Use the following steps to create a container app with the default quickstart image.

1. Navigate to the portal **Home** page.
2. Search for **Container Apps** in the top search bar.
3. Select **Container Apps** in the search results.
4. Select the **Create** button.
5. In the *Basics* tab, do the following actions.

Setting	Action
Subscription	Select your Azure subscription.
Resource group	Select an existing resource group or create a new one.
Container app name	Enter a container app name.
Location	Select a location.
Create Container App Environment	Create a new or select an existing environment.

6. Select the **Review + Create** button at the bottom of the **Create Container App** page.
7. Select the **Create** button at the bottom of the **Create Container App** window.

Allow a few minutes for the container app deployment to finish. When deployment is complete, select **Go to resource**.

Add the user-assigned managed identity

1. Select **Identity** from the left menu.
2. Select the **User assigned** tab.
3. Select the **Add user assigned managed identity** button.
4. Select your subscription.
5. Select the identity you created.
6. Select **Add**.

Create a container app revision

Create a container app revision with a private image and the system-assigned managed identity.

1. Select **Revision Management** from the left menu.
2. Select **Create new revision**.
3. Select the container image from the **Container Image** table.
4. Enter the information in the *Edit a container* dialog.

Field	Action
Name	Enter a name for the container.
Image source	Select Azure Container Registry .
Authentication	Select Managed Identity .
Identity	Select the identity you created from the drop-down menu.
Registry	Select the registry you want to use from the drop-down menu.
Image	Enter the name of the image you want to use.
Image Tag	Enter the name and tag of the image you want to pull.

Container details

You can change these settings after creating the Container App.

Name *

container-app

Image source

Azure Container Registry

Docker Hub or other registries

Authentication

Admin Credentials

Managed Identity

Identity *

container-app-mi

Registry *

containerappimages.azurecr.io

- ⓘ Failed to retrieve images for ACR 'containerappimages.azurecr.io' because admin credentials on the ACR are disabled. Please manually enter the image and tag below.

Image *

container-app

Image tag *

latest

OS type

Linux

Command override ⓘ

Example: /bin/bash, -c, echo hello; sleep 10...

Container resource allocation

Save

Cancel

ⓘ Note

If the administrative credentials are not enabled on your Azure Container Registry registry, you will see a warning message displayed and you will need to enter the image name and tag information manually.

5. Select **Save**.

6. Select **Create** from the **Create and deploy new revision** page.

A new revision will be created and deployed. The portal will automatically attempt to add the `acrpull` role to the user-assigned managed identity. If the role isn't added, you can add it manually.

You can verify that the role was added by checking the identity from the **Identity** pane of the container app page.

1. Select **Identity** from the left menu.
2. Select the **User assigned** tab.
3. Select the user-assigned managed identity.
4. Select **Azure role assignments** from the menu on the managed identity resource page.
5. Verify that the `acrpull` role is assigned to the user-assigned managed identity.

Clean up resources

If you're not going to continue to use this application, you can delete the Azure Container Apps instance and all the associated services by removing the resource group.

Warning

Deleting the resource group will delete all the resources in the group. If you have other resources in the group, they will also be deleted. If you want to keep the resources, you can delete the container app instance and the container app environment.

1. Select your resource group from the *Overview* section.
2. Select the **Delete resource group** button at the top of the resource group *Overview*.
3. Enter the resource group name in the confirmation dialog.
4. Select **Delete**.

The process to delete the resource group may take a few minutes to complete.

System-assigned managed identity

The method for configuring a system-assigned managed identity in the Azure portal is the same as configuring a user-assigned managed identity. The only difference is that you don't need to create a user-assigned managed identity. Instead, the system-assigned managed identity is created when you create the container app.

The method to configure a system-assigned managed identity in the Azure portal is:

1. Create a container app with a public image.
2. Create a container app revision with a private image and the system-assigned managed identity.

Prerequisites

- An Azure account with an active subscription.
 - If you don't have one, you [can create one for free ↗](#).
- A private Azure Container Registry containing an image you want to pull. See [Create a private Azure Container Registry](#).

Create a container app

Follow these steps to create a container app with the default quickstart image.

1. Navigate to the portal **Home** page.
2. Search for **Container Apps** in the top search bar.
3. Select **Container Apps** in the search results.
4. Select the **Create** button.
5. In the **Basics** tab, do the following actions.

Setting	Action
Subscription	Select your Azure subscription.
Resource group	Select an existing resource group or create a new one.
Container app name	Enter a container app name.
Location	Select a location.
Create Container App Environment	Create a new or select an existing environment.

6. Select the **Review + Create** button at the bottom of the **Create Container App** page.
7. Select the **Create** button at the bottom of the **Create Container App** page.

Allow a few minutes for the container app deployment to finish. When deployment is complete, select **Go to resource**.

Edit and deploy a revision

Edit the container to use the image from your private Azure Container Registry, and configure the authentication to use system-assigned identity.

1. The **Containers** from the side menu on the left.
2. Select **Edit and deploy**.
3. Select the *simple-hello-world-container* container from the list.

Setting	Action
Name	Enter the container app name.
Image source	Select Azure Container Registry .
Authentication	Select Managed identity .
Identity	Select System assigned .
Registry	Enter the Registry name.
Image	Enter the image name.
Image tag	Enter the tag.

Edit a container

X

Basics Health probes

Container details

You can change these settings after creating the Container App.

Name *

my-container-app

Image source

Azure Container Registry

Docker Hub or other registries

Authentication

Admin Credentials

Managed Identity

Identity *

System assigned

Registry *

containerappimages.azurecr.io

(i) Failed to retrieve images for ACR 'containerappimages.azurecr.io' because admin credentials on the ACR are disabled. Please manually enter the image and tag below.

Image *

container-app

Image tag *

latest

OS type

Linux

Command override *(i)*

Example: /bin/bash, -c, echo hello; sleep 10...

Container resource allocation

Save

Cancel

(!) Note

If the administrative credentials are not enabled on your Azure Container Registry registry, you will see a warning message displayed and you will need to enter the image name and tag information manually.

4. Select **Save** at the bottom of the page.
5. Select **Create** at the bottom of the **Create and deploy new revision** page
6. After a few minutes, select **Refresh** on the **Revision management** page to see the new revision.

A new revision will be created and deployed. The portal will automatically attempt to add the `acrpull` role to the system-assigned managed identity. If the role isn't added, you can add it manually.

You can verify that the role was added by checking the identity in the **Identity** pane of the container app page.

1. Select **Identity** from the left menu.
2. Select the **System assigned** tab.
3. Select **Azure role assignments**.
4. Verify that the `acrpull` role is assigned to the system-assigned managed identity.

Clean up resources

If you're not going to continue to use this application, you can delete the Azure Container Apps instance and all the associated services by removing the resource group.

Warning

Deleting the resource group will delete all the resources in the group. If you have other resources in the group, they will also be deleted. If you want to keep the resources, you can delete the container app instance and the container app environment.

1. Select your resource group from the *Overview* section.
2. Select the **Delete resource group** button at the top of the resource group *Overview*.
3. Enter the resource group name in the confirmation dialog.
4. Select **Delete**.

The process to delete the resource group may take a few minutes to complete.

Next steps

[Managed identities in Azure Container Apps](#)

Manage revisions in Azure Container Apps

Article • 03/30/2023

Supporting multiple revisions in Azure Container Apps allows you to manage the versioning of your container app. With this feature, you can activate and deactivate revisions, and control the amount of [traffic sent to each revision](#). To learn more about revisions, see [Revisions in Azure Container Apps](#)

A revision is created when you first deploy your application. New revisions are created when you [update](#) your application with [revision-scope changes](#). You can also update your container app based on a specific revision.

This article described the commands to manage your container app's revisions. For more information about Container Apps commands, see [az containerapp](#). For more information about commands to manage revisions, see [az containerapp revision](#).

Updating your container app

To update a container app, use the `az containerapp update` command. With this command you can modify environment variables, compute resources, scale parameters, and deploy a different image. If your container app update includes [revision-scope changes](#), a new revision is generated.

Bash

You may also use a YAML file to define these and other configuration options and parameters. For more information regarding this command, see [az containerapp revision copy](#).

This example updates the container image. Replace the <PLACEHOLDERS> with your values.

Azure CLI

```
az containerapp update \
--name <APPLICATION_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--image <IMAGE_NAME>
```

You can also update your container app with the [Revision copy](#) command.

Revision list

List all revisions associated with your container app with `az containerapp revision list`. For more information about this command, see [az containerapp revision list](#)

Bash

Replace the <PLACEHOLDERS> with your values.

Azure CLI

```
az containerapp revision list \
--name <APPLICATION_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
-o table
```

Revision show

Show details about a specific revision by using `az containerapp revision show`. For more information about this command, see [az containerapp revision show](#).

Bash

Replace the <PLACEHOLDERS> with your values.

Azure CLI

```
az containerapp revision show \
--name <APPLICATION_NAME> \
--revision <REVISION_NAME> \
--resource-group <RESOURCE_GROUP_NAME>
```

Revision copy

To create a new revision based on an existing revision, use the `az containerapp revision copy`. Container Apps uses the configuration of the existing revision, which you may then modify.

With this command, you can modify environment variables, compute resources, scale parameters, and deploy a different image. You may also use a YAML file to define these and other configuration options and parameters. For more information regarding this command, see [az containerapp revision copy](#).

This example copies the latest revision and sets the compute resource parameters.
(Replace the <PLACEHOLDERS> with your values.)

Bash

```
Azure CLI
```

```
az containerapp revision copy \
--name <APPLICATION_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--cpu 0.75 \
--memory 1.5Gi
```

Revision activate

Activate a revision by using `az containerapp revision activate`. For more information about this command, see [az containerapp revision activate](#).

Bash

```
Azure CLI
```

```
az containerapp revision activate \
--revision <REVISION_NAME> \
--resource-group <RESOURCE_GROUP_NAME>
```

Revision deactivate

Deactivate revisions that are no longer in use with `az containerapp revision deactivate`. Deactivation stops all running replicas of a revision. For more information, see [az containerapp revision deactivate](#).

Bash

Example: (Replace the <PLACEHOLDERS> with your values.)

Azure CLI

```
az containerapp revision deactivate \  
  --revision <REVISION_NAME> \  
  --resource-group <RESOURCE_GROUP_NAME>
```

Revision restart

This command restarts a revision. For more information about this command, see [az containerapp revision restart](#).

When you modify secrets in your container app, you need to restart the active revisions so they can access the secrets.

Bash

Example: (Replace the <PLACEHOLDERS> with your values.)

Azure CLI

```
az containerapp revision restart \  
  --revision <REVISION_NAME> \  
  --resource-group <RESOURCE_GROUP_NAME>
```

Revision set mode

The revision mode controls whether only a single revision or multiple revisions of your container app can be simultaneously active. To set your container app to support [single revision mode](#) or [multiple revision mode](#), use the `az containerapp revision set-mode` command.

The default setting is *single revision mode*. For more information about this command, see [az containerapp revision set-mode](#).

The mode values are `single` or `multiple`. Changing the revision mode doesn't create a new revision.

Example: (Replace the <placeholders> with your values.)

Bash

Example: (Replace the <PLACEHOLDERS> with your values.)

Azure CLI

```
az containerapp revision set-mode \
--name <APPLICATION_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--mode <REVISION_MODE>
```

Revision labels

Labels provide a unique URL that you can use to direct traffic to a revision. You can move a label between revisions to reroute traffic directed to the label's URL to a different revision. For more information about revision labels, see [Revision Labels](#).

You can add and remove a label from a revision. For more information about the label commands, see [az containerapp revision label](#)

Revision label add

To add a label to a revision, use the [az containerapp revision label add](#) command.

You can only assign a label to one revision at a time, and a revision can only be assigned one label. If the revision you specify has a label, the add command replaces the existing label.

This example adds a label to a revision: (Replace the <PLACEHOLDERS> with your values.)

Bash

Azure CLI

```
az containerapp revision label add \
--revision <REVISION_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--label <LABEL_NAME>
```

Revision label remove

To remove a label from a revision, use the [az containerapp revision label remove](#) command.

This example removes a label to a revision: (Replace the <PLACEHOLDERS> with your values.)

Bash

Azure CLI

```
az containerapp revision label add \
    --revision <REVISION_NAME> \
    --resource-group <RESOURCE_GROUP_NAME> \
    --label <LABEL_NAME>
```

Traffic splitting

Applied by assigning percentage values, you can decide how to balance traffic among different revisions. Traffic splitting rules are assigned by setting weights to different revisions by their name or [label](#). For more information, see, [Traffic Splitting](#).

Next steps

- [Revisions in Azure Container Apps](#)
- [Application lifecycle management in Azure Container Apps](#)

Manage workload profiles in a Consumption + Dedicated workload profiles plan structure (preview)

Article • 04/13/2023

Learn to manage a Container Apps environment with workload profile support.

Supported regions

The following regions support workload profiles during preview:

- North Central US
- North Europe
- West Europe
- East US

Create a container app in a profile

Azure Container Apps run in an environment, which uses a virtual network (VNet). By default, your Container App environment is created with a managed VNet that is automatically generated for you. Generated VNets are inaccessible to you as they're created in Microsoft's tenant.

Create a container apps environment with a [custom VNet](#) if you need any of the following features:

- [User defined routes](#)
- Integration with Application Gateway
- Network Security Groups
- Communicating with resources behind private endpoints in your virtual network

Use the following commands to create an environment with workload profile support.

1. Create *Consumption + Dedicated* environment with workload profile support

Bash

```
az containerapp env create \
--enable-workload-profiles \
--resource-group "<RESOURCE_GROUP>" \
```

```
--name "<NAME>" \
--location "<LOCATION>"
```

This command can take up to 10 minutes to complete.

2. Check status of environment. Here, you're looking to see if the environment is created successfully.

Bash

```
az containerapp env show \
--name "<ENVIRONMENT_NAME>" \
--resource-group "<RESOURCE_GROUP>"
```

The `provisioningState` needs to report `Succeeded` before moving on to the next command.

3. Create a new container app.

External environment

Azure CLI

```
az containerapp create \
--resource-group "<RESOURCE_GROUP>" \
--name "<CONTAINER_APP_NAME>" \
--target-port 80 \
--ingress external \
--image mcr.microsoft.com/azuredocs/containerapps-
helloworld:latest \
--environment "<ENVIRONMENT_NAME>" \
--workload-profile-name "Consumption"
```

This command deploys the application to the built-in Consumption workload profile. If you want to create an app in a dedicated workload profile, you first need to [add the profile to the environment](#).

This command creates the new application in the environment using a specific workload profile.

Add profiles

Add a new workload profile to an existing environment.

Azure CLI

```
az containerapp env workload-profile set \
--resource-group <RESOURCE_GROUP> \
--name <ENVIRONMENT_NAME> \
--workload-profile-type <WORKLOAD_PROFILE_TYPE> \
--workload-profile-name <WORKLOAD_PROFILE_NAME> \
--min-nodes <MIN_NODES> \
--max-nodes <MAX_NODES>
```

The value you select for the `<WORKLOAD_PROFILE_NAME>` placeholder is the workload profile "friendly name".

Using friendly names allow you to add multiple profiles of the same type to an environment. The friendly name is what you use as you deploy and maintain a container app in a workload profile.

Edit profiles

You can modify the minimum and maximum number of nodes used by a workload profile via the `set` command.

Azure CLI

```
az containerapp env workload-profile set \
--resource-group <RESOURCE_GROUP> \
--name <ENV_NAME> \
--workload-profile-type <WORKLOAD_PROFILE_TYPE> \
--workload-profile-name <WORKLOAD_PROFILE_NAME> \
--min-nodes <MIN_NODES> \
--max-nodes <MAX_NODES>
```

Delete a profile

Use the following command to delete a workload profile.

Azure CLI

```
az containerapp env workload-profile delete \
--resource-group "<RESOURCE_GROUP>" \
--name <ENVIRONMENT_NAME> \
--workload-profile-name <WORKLOAD_PROFILE_NAME>
```

ⓘ Note

The *Consumption* workload profile can't be deleted.

Inspect profiles

The following commands allow you to list available profiles in your region and ones used in a specific environment.

List available workload profiles

Use the `list-supported` command to list the supported workload profiles for your region.

The following Azure CLI command displays the results in a table

Azure CLI

```
az containerapp env workload-profile list-supported \
--location <LOCATION> \
--query "[].{Name: name, Cores: properties.cores, MemoryGiB:
properties.memoryGiB, Category: properties.category}" \
-o table
```

The response resembles a table similar to the below example:

Output

Name	Cores	MemoryGiB	Category
D4	4	16	GeneralPurpose
D8	8	32	GeneralPurpose
D16	16	64	GeneralPurpose
E4	4	32	MemoryOptimized
E8	8	64	MemoryOptimized
E16	16	128	MemoryOptimized
Consumption	4	8	Consumption

Select a workload profile and use the *Name* field when you run `az containerapp env workload-profile set` for the `--workload-profile-type` option.

Show a workload profile

Display details about a workload profile.

Azure CLI

```
az containerapp env workload-profile show \  
  --resource-group <RESOURCE_GROUP> \  
  --name <ENVIRONMENT_NAME> \  
  --workload-profile-name <WORKLOAD_PROFILE_NAME>
```

Next steps

[Workload profiles overview](#)

Manage workload profiles in a Consumption + Dedicated workload profiles plan structure (preview) in the Azure portal

Article • 04/12/2023

Learn to manage Container Apps environments with workload profile support.

Supported regions

The following regions support workload profiles during preview:

- North Central US
- North Europe
- West Europe
- East US

Create a container app in a workload profile

1. Open the Azure portal.
2. Search for *Container Apps* in the search bar, and select **Container Apps**.
3. Select **Create**.
4. Create a new container app and environment.

Create Container App

X

[Basics](#) [App settings](#) [Tags](#) [Review + create](#)

Azure Container Apps are containerized apps that scale on demand without requiring you to manage cloud infrastructure. You'll need a container and an environment for your first app. Select existing resources, or create them now. [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 *

My Subscription

Resource group *

My Resource Group

[Create new](#)

Container app name *

my-container-app

Container Apps Environment

The environment is a secure boundary around one or more container apps that can communicate with each other and share a virtual network, logging, and Dapr configuration. [Container Apps Pricing](#)

Region *

East US

Container Apps Environment *

[Create new](#)[Review + create](#)

< Previous

Next : App settings >

Enter the following values to create your new container app.

Property	Value
Subscription	Select your subscription
Resource group	Select or create a resource group
Container app name	Enter your container app name
Region	Select your region.
Container Apps Environment	Select Create New .

5. Configure the new environment.

Create Container Apps Environment

X

Basics

Workload profiles (Preview)

Monitoring

Networking

2

The environment is a secure boundary around one or more container apps that can communicate with each other and share a virtual network, logging, and Dapr configuration. [Learn more](#)

Environment details

Environment name *

my-container-apps-environment



Plan *

Consumption: Run serverless apps with support for scale-to-zero and pay only for resources your apps use.

(Preview) Consumption and Dedicated workload profiles: Run serverless apps with support for scale-to-zero and pay only for resources your apps use. Optionally, run apps with customized hardware and increased cost predictability using Dedicated workload profiles.

1

Create

Cancel

Enter the following values to create your environment.

Property	Value
Environment name	Enter an environment name.
Plan	Select (Preview) Consumption and Dedicated workload profiles

Select the new **Workload profiles** tab at the top of this section.

6. Select the **Add workload profile** button.

Create Container Apps Environment

X

Basics

Workload profiles (Preview)

Monitoring

Networking

Dedicated workload profiles allow you to run your apps on customized custom options. You can add Dedicated profiles below. An environment always has a Consumption workload profile where you can run apps that can scale-to-zero and pay only for resources you apps use. [Learn more](#)

[+ Add workload profile](#) [Delete](#)

Name	Scaling	Workload profile size
Consumption	-	Up to 4 vCPUs / 8 Gib

[Create](#) [Cancel](#)

7. For *Workload profile name*, enter a name.

8. Next to *Workload profile size*, select **Choose size**.

Add workload profile (Preview)

X

Dedicated workload profiles enable you to run your apps on custom hardware. You can control costs by adjusting the minimum and maximum workload profile instance count.

[Learn more about workload profiles](#)

Workload profile name *

Dedicated-D8

Workload profile size *

[Choose a size](#)

Autoscaling instance count range *

3



5

9. In the *Select a workload profile size* window, select a profile from the list.

Name	vCPU	RAM (GiB)
▼ General purpose D-series		
<input type="checkbox"/> Dedicated-D4	4	16
<input type="checkbox"/> Dedicated-D8	8	32
<input type="checkbox"/> Dedicated-D16	16	64
▼ Memory optimized E-series		
<input type="checkbox"/> Dedicated-E4	4	32
<input type="checkbox"/> Dedicated-E8	8	64
<input type="checkbox"/> Dedicated-E16	16	128

General purpose profiles offer a balanced mix cores vs memory for most applications.

Memory optimized profiles offer specialized hardware with increased memory capabilities.

10. Select the **Select** button.

11. For the *Autoscaling instance count range*, select the minimum and maximum number of instances you want available to this workload profile.

Dedicated workload profiles enable you to run your apps on custom hardware. You can control costs by adjusting the minimum and maximum workload profile instance count. [Learn more about workload profiles ↗](#)

Workload profile name *	Dedicated-D8
Workload profile size *	Dedicated-D8 8 vCPUs, 32 GiB included Change
Autoscaling instance count range *	<input type="text" value="3"/>  <input type="text" value="5"/>
Add Back	

12. Select **Add**.

13. Select **Create**.

14. Select **Review + Create** and wait as Azure validates your configuration options.

15. Select **Create** to create your container app and environment.

Add profiles

Add a new workload profile to an existing environment.

1. Under the *Settings* section, select **Workload profiles**.
2. Select **Add**.
3. For *Workload profile name*, enter a name.
4. Next to *Workload profile size*, select **Choose size**.
5. In the *Select a workload profile size* window, select a profile from the list.

General purpose profiles offer a balanced mix cores vs memory for most applications.

Memory optimized profiles offer specialized hardware with increased memory or compute capabilities.

6. Select the **Select** button.
7. For the *Autoscaling instance count range*, select the minimum and maximum number of instances you want available to this workload profile.

Dedicated workload profiles enable you to run your apps on custom hardware. You can control costs by adjusting the minimum and maximum workload profile instance count. [Learn more about workload profiles](#)

Workload profile name *	Dedicated-D8
Workload profile size *	Dedicated-D8 8 vCPUs, 32 GiB included Change
Autoscaling instance count range *	<div style="border: 2px solid red; padding: 5px; width: fit-content;">3 — [] — 5</div>
Add Back	

8. Select **Add**.

Edit profiles

Under the *Settings* section, select **Workload profiles**.

From this window, you can:

- Adjust the minimum and maximum number of instances available to a profile
- Add new profiles
- Delete existing profiles (except for the Consumption profile)

Delete a profile

Under the *Settings* section, select **Workload profiles**. From this window, you select a profile to delete.

ⓘ Note

The *Consumption* workload profile can't be deleted.

Next steps

[Workload profiles overview](#)

Configure Ingress for your app in Azure Container Apps

Article • 03/30/2023

This article shows you how to enable [ingress](#) features for your container app. Ingress is an application-wide setting. Changes to ingress settings apply to all revisions simultaneously, and don't generate new revisions.

Ingress settings

You can set the following ingress template properties:

Property	Description	Values	Required
<code>allowInsecure</code>	Allows insecure traffic to your container app. When set to <code>true</code> HTTP requests to port 80 aren't automatically redirected to port 443 using HTTPS, allowing insecure connections.	<code>false</code> (default), <code>true</code> enables insecure connections	No
<code>clientCertificateMode</code>	Client certificate mode for mTLS authentication. Ignore indicates server drops client certificate on forwarding. Accept indicates server forwards client certificate but doesn't require a client certificate. Require indicates server requires a client certificate.	<code>Required</code> , <code>Accept</code> , <code>Ignore</code> (default)	No
<code>customDomains</code>	Custom domain bindings for Container Apps' hostnames. See Custom domains and certificates	An array of bindings	No
<code>exposedPort</code>	(TCP ingress only) The port TCP listens on. If <code>external</code> is <code>true</code> , the value must be unique in the Container Apps environment.	A port number from 1 to 65535. (can't be 80 or 443)	No
<code>external</code>	Allow ingress to your app from outside its Container Apps environment.	<code>true</code> or <code>false</code> (default)	Yes
<code>ipSecurityRestrictions</code>	IP ingress restrictions. See Set up IP ingress restrictions	An array of rules	No

Property	Description	Values	Required
<code>stickySessions.affinity</code>	Enables session affinity .	<code>none</code> (default), <code>sticky</code>	No
<code>targetPort</code>	The port your container listens to for incoming requests.	Set this value to the port number that your container uses. For HTTP ingress, your application ingress endpoint is always exposed on port <code>443</code> .	Yes
<code>traffic</code>	Traffic splitting weights split between revisions.	An array of rules	No
<code>transport</code>	The transport protocol type.	auto (default) detects HTTP/1 or HTTP/2, <code>http</code> for HTTP/1, <code>http2</code> for HTTP/2, <code>tcp</code> for TCP.	No

Enable ingress

You can configure ingress for your container app using the Azure CLI, an ARM template, or the Azure portal.

Azure CLI

This `az containerapp ingress enable` command enables ingress for your container app. You must specify the target port, and you can optionally set the exposed port if your transport type is `tcp`.

Azure CLI

```
az containerapp ingress enable \
--name <app-name> \
--resource-group <resource-group> \
--target-port <target-port> \
--exposed-port <tcp-exposed-port> \
--transport <transport> \
--type <external>
--allow-insecure
```

`az containerapp ingress enable` ingress arguments:

Option	Property	Description	Values	Required
--type	external	Allow ingress to your app from anywhere, or limit ingress to its internal Container Apps environment.	external or internal	Yes
--allow-insecure	allowInsecure	Allow HTTP connections to your app.		No
--target-port	targetPort	The port your container listens to for incoming requests.	Set this value to the port number that your container uses. Your application ingress endpoint is always exposed on port 443.	Yes
--exposed-port	exposedPort	(TCP ingress only) A port for TCP ingress. If external is true, the value must be unique in the Container Apps environment if ingress is external.	A port number from 1 to 65535. (can't be 80 or 443)	No
--transport	transport	The transport protocol type.	auto (default) detects HTTP/1 or HTTP/2, http for HTTP/1, http2 for HTTP/2, tcp for TCP.	No

Disable ingress

Azure CLI

Disable ingress for your container app by using the `az containerapp ingress disable` command.

Azure CLI

```
az containerapp ingress disable \
  --name <app-name> \
  --resource-group <resource-group> \
```

Next steps

[Ingress in Azure Container Apps](#)

Set up IP ingress restrictions in Azure Container Apps

Article • 03/30/2023

Azure Container Apps allows you to limit inbound traffic to your container app by configuring IP ingress restrictions via ingress configuration.

There are two types of restrictions:

- *Allow*: Allow inbound traffic only from address ranges you specify in allow rules.
- *Deny*: Deny all inbound traffic only from address ranges you specify in deny rules.

when no IP restriction rules are defined, all inbound traffic is allowed.

IP restrictions rules contain the following properties:

Property	Value	Description
name	string	The name of the rule.
description	string	A description of the rule.
ipAddressRange	IP address range in CIDR format	The IP address range in CIDR notation.
action	Allow or Deny	The action to take for the rule.

The `ipAddressRange` parameter accepts IPv4 addresses. Define each IPv4 address block in [Classless Inter-Domain Routing \(CIDR\)](#) notation.

ⓘ Note

All rules must be the same type. You cannot combine allow rules and deny rules.

Manage IP ingress restrictions

You can manage IP access restrictions rules through the Azure portal or Azure CLI.

Add rules

1. Go to your container app in the Azure portal.
2. Select **Ingress** from the left side menu.

3. Select the IP Security Restrictions Mode toggle to enable IP restrictions. You can choose to allow or deny traffic from the specified IP address ranges.

4. Select Add to create the rule.

The screenshot shows the Azure Container Apps interface for an 'album-app' container app. The 'Ingress' tab is active. On the right, under 'IP Restrictions (Preview)', there's a section for 'IP Security Restrictions Mode' with three options: 'Allow all traffic (default)', 'Allow traffic from IPs configured below, deny all other traffic' (which is selected), and 'Deny traffic from IPs configured below, allow all other traffic'. A red box highlights the 'Add' button at the bottom left of the IP restrictions table. The table itself has columns for 'Source ↑', 'Name ↑', and 'Delete'.

5. Enter values in the following fields:

Field	Description
IPv4 address or range	Enter the IP address or range of IP addresses in CIDR notation. For example, to allow access from a single IP address, use the following format: 10.200.10.2/32.
Name	Enter a name for the rule.
Description	Enter a description for the rule.

6. Select Add.

7. Repeat steps 4-6 to add more rules.

8. When you have finished adding rules, select **Save**.

The screenshot shows the 'IP Restrictions' section of the Azure portal. It includes a search bar, an 'Add' button, and a table with columns for Source and Name. A single rule is listed: '192.168.0.23/32' under Source and 'single-address' under Name. To the right of each entry is a 'Delete' icon. At the bottom are 'Save' and 'Discard' buttons, with 'Save' being highlighted by a red box.

Source ↑	Name ↑	Delete
192.168.0.23/32	single-address	

Update a rule

1. Go to your container app in the Azure portal.
2. Select **Ingress** from the left side menu.
3. Select the rule you want to update.
4. Change the rule settings.
5. Select **Save** to save the updates.
6. Select **Save** on the Ingress page to save the updated rules.

Delete a rule

1. Go to your container app in the Azure portal.
2. Select **Ingress** from the left side menu.
3. Select the delete icon next to the rule you want to delete.
4. Select **Save**.

Next steps

[Configure Ingress](#)

Configure client certificate authentication in Azure Container Apps

Article • 03/30/2023

Azure Container Apps supports client certificate authentication (also known as mutual TLS or mTLS) that allows access to your container app through two-way authentication. This article shows you how to configure client certificate authorization in Azure Container Apps.

When client certificates are used, the TLS certificates are exchanged between the client and your container app to authenticate identity and encrypt traffic. Client certificates are often used in "zero trust" security models to authorize client access within an organization.

For example, you may want to require a client certificate for a container app that manages sensitive data.

Container Apps accepts client certificates in the PKCS12 format are that issued by a trusted certificate authority (CA), or are self-signed.

Configure client certificate authorization

Set the `clientCertificateMode` property in your container app template to configure support of client certificates.

The property can be set to one of the following values:

- `require`: The client certificate is required for all requests to the container app.
- `accept`: The client certificate is optional. If the client certificate isn't provided, the request is still accepted.
- `ignore`: The client certificate is ignored.

Ingress passes the client certificate to the container app if `require` or `accept` are set.

The following ARM template example configures ingress to require a client certificate for all requests to the container app.

JSON

```
{  
  "properties": {  
    "configuration": {
```

```
    "ingress": {  
      "clientCertificateMode": "require"  
    }  
  }  
}
```

Next Steps

[Configure ingress](#)

Traffic splitting in Azure Container Apps

Article • 03/30/2023

By default, when ingress is enabled, all traffic is routed to the latest deployed revision. When you enable [multiple revision mode](#) in your container app, you can split incoming traffic between active revisions.

Traffic splitting is useful for testing updates to your container app. You can use traffic splitting to gradually phase in a new revision in [blue-green deployments](#) or in [A/B testing](#).

Traffic splitting is based on the weight (percentage) of traffic that is routed to each revision. The combined weight of all traffic split rules must equal 100%. You can specify revision by revision name or [revision label](#).

This article shows you how to configure traffic splitting rules for your container app. To run the following examples, you need a container app with multiple revisions.

Configure traffic splitting

Configure traffic splitting between revisions using the [az containerapp ingress traffic set](#) command. You can specify the revisions by name with the `--revision-weight` parameter or by revision label with the `--label-weight` parameter.

The following command sets the traffic weight for each revision to 50%:

```
Azure CLI  
  
az containerapp ingress traffic set \  
  --name <APP_NAME> \  
  --resource-group <RESOURCE_GROUP> \  
  --revision-weight <REVISION_1>=50 <REVISION_2>=50
```

Make sure to replace the placeholder values surrounded by `<>` with your own values.

This command sets the traffic weight for revision `<LABEL_1>` to 80% and revision `<LABEL_2>` to 20%:

```
Azure CLI  
  
az containerapp ingress traffic set \  
  --name <APP_NAME> \  
  --resource-group <RESOURCE_GROUP> \  
  --label-weight <LABEL_1>=80 <LABEL_2>=20
```

```
--label-weight <LABEL_1>=80 <LABEL_2>=20
```

Use cases

The following scenarios describe configuration settings for common use cases. The examples are shown in JSON format, but you can also use the Azure portal or Azure CLI to configure traffic splitting.

Rapid iteration

In situations where you're frequently iterating development of your container app, you can set traffic rules to always shift all traffic to the latest deployed revision.

The following example template routes all traffic to the latest deployed revision:

JSON

```
"ingress": {  
  "traffic": [  
    {  
      "latestRevision": true,  
      "weight": 100  
    }  
  ]  
}
```

Once you're satisfied with the latest revision, you can lock traffic to that revision by updating the `ingress` settings to:

JSON

```
"ingress": {  
  "traffic": [  
    {  
      "latestRevision": false, // optional  
      "revisionName": "myapp--knowngoodrevision",  
      "weight": 100  
    }  
  ]  
}
```

Update existing revision

Consider a situation where you have a known good revision that's serving 100% of your traffic, but you want to issue an update to your app. You can deploy and test new revisions using their direct endpoints without affecting the main revision serving the app.

Once you're satisfied with the updated revision, you can shift a portion of traffic to the new revision for testing and verification.

The following template moves 20% of traffic over to the updated revision:

JSON

```
"ingress": {  
  "traffic": [  
    {  
      "revisionName": "myapp--knowngoodrevision",  
      "weight": 80  
    },  
    {  
      "revisionName": "myapp--newerrevision",  
      "weight": 20  
    }  
  ]  
}
```

Staging microservices

When building microservices, you may want to maintain production and staging endpoints for the same app. Use labels to ensure that traffic doesn't switch between different revisions.

The following example template applies labels to different revisions.

JSON

```
"ingress": {  
  "traffic": [  
    {  
      "revisionName": "myapp--knowngoodrevision",  
      "weight": 100  
    },  
    {  
      "revisionName": "myapp--98fdgt",  
      "weight": 0,  
      "label": "staging"  
    }  
  ]  
}
```

Next steps

[Configure ingress](#)

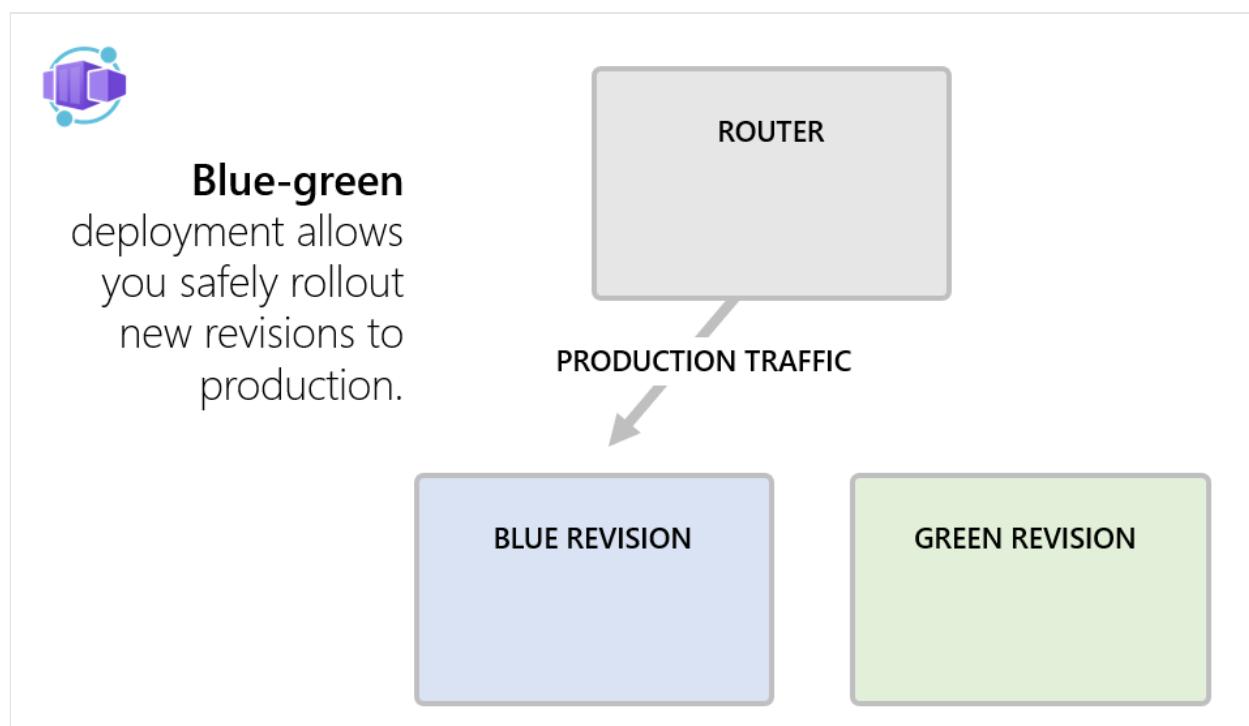
Blue-Green Deployment in Azure Container Apps

Article • 06/27/2023

[Blue-Green Deployment](#) is a software release strategy that aims to minimize downtime and reduce the risk associated with deploying new versions of an application. In a blue-green deployment, two identical environments, referred to as "blue" and "green," are set up. One environment (blue) is running the current application version and one environment (green) is running the new application version.

Once green environment is tested, the live traffic is directed to it, and the blue environment is used to deploy a new application version during next deployment cycle.

You can enable blue-green deployment in Azure Container Apps by combining [container apps revisions](#), [traffic weights](#), and [revision labels](#).



You use revisions to create instances of the blue and green versions of the application.

Revision	Description
Blue revision	The revision labeled as <i>blue</i> is the currently running and stable version of the application. This revision is the one that users interact with, and it's the target of production traffic.

Revision	Description
Green revision	The revision labeled as <i>green</i> is a copy of the <i>blue</i> revision except it uses a newer version of the app code and possibly new set of environment variables. It doesn't receive any production traffic initially but is accessible via a labeled fully qualified domain name (FQDN).

After you test and verify the new revision, you can then point production traffic to the new revision. If you encounter issues, you can easily roll back to the previous version.

Actions	Description
Testing and verification	The <i>green</i> revision is thoroughly tested and verified to ensure that the new version of the application functions as expected. This testing may involve various tasks, including functional tests, performance tests, and compatibility checks.
Traffic switch	Once the <i>green</i> revision passes all the necessary tests, a traffic switch is performed so that the <i>green</i> revision starts serving production load. This switch is done in a controlled manner, ensuring a smooth transition.
Rollback	If problems occur in the <i>green</i> revision, you can revert the traffic switch, routing traffic back to the stable <i>blue</i> revision. This rollback ensures minimal impact on users if there are issues in the new version. The <i>green</i> revision is still available for the next deployment.
Role change	The roles of the blue and green revisions change after a successful deployment to the <i>green</i> revision. During the next release cycle, the <i>green</i> revision represents the stable production environment while the new version of the application code is deployed and tested in the <i>blue</i> revision.

This article shows you how to implement blue-green deployment in a container app. To run the following examples, you need a container app environment where you can create a new app.

ⓘ Note

Refer to [containerapps-blue-green repository](#) for a complete example of a github workflow that implements blue-green deployment for Container Apps.

Create a container app with multiple active revisions enabled

The container app must have the `configuration.activeRevisionsMode` property set to `multiple` to enable traffic splitting. To get deterministic revision names, you can set the

`template.revisionSuffix` configuration setting to a string value that uniquely identifies a release. For example you can use build numbers, or git commits short hashes.

For the following commands, a set of commit hashes was used.

Azure CLI

```
export APP_NAME=<APP_NAME>
export APP_ENVIRONMENT_NAME=<APP_ENVIRONMENT_NAME>
export RESOURCE_GROUP=<RESOURCE_GROUP>

# A commitId that is assumed to correspond to the app code currently in
# production
export BLUE_COMMIT_ID=fb699ef
# A commitId that is assumed to correspond to the new version of the code to
# be deployed
export GREEN_COMMIT_ID=c6f1515

# create a new app with a new revision
az containerapp create --name $APP_NAME \
--environment $APP_ENVIRONMENT_NAME \
--resource-group $RESOURCE_GROUP \
--image mcr.microsoft.com/k8se/samples/test-app:$BLUE_COMMIT_ID \
--revision-suffix $BLUE_COMMIT_ID \
--env-vars REVISION_COMMIT_ID=$BLUE_COMMIT_ID \
--ingress external \
--target-port 80 \
--revisions-mode multiple

# Fix 100% of traffic to the revision
az containerapp ingress traffic set \
--name $APP_NAME \
--resource-group $RESOURCE_GROUP \
--revision-weight $APP_NAME-$BLUE_COMMIT_ID=100

# give that revision a label 'blue'
az containerapp revision label add \
--name $APP_NAME \
--resource-group $RESOURCE_GROUP \
--label blue \
--revision $APP_NAME-$BLUE_COMMIT_ID
```

Deploy a new revision and assign labels

The *blue* label currently refers to a revision that takes the production traffic arriving on the app's FQDN. The *green* label refers to a new version of an app that is about to be rolled out into production. A new commit hash identifies the new version of the app code. The following command deploys a new revision for that commit hash and marks it with *green* label.

Azure CLI

```
#create a second revision for green commitId
az containerapp update --name $APP_NAME \
--resource-group $RESOURCE_GROUP \
--image mcr.microsoft.com/k8se/samples/test-app:$GREEN_COMMIT_ID \
--revision-suffix $GREEN_COMMIT_ID \
--set-env-vars REVISION_COMMIT_ID=$GREEN_COMMIT_ID

#give that revision a 'green' label
az containerapp revision label add \
--name $APP_NAME \
--resource-group $RESOURCE_GROUP \
--label green \
--revision $APP_NAME-$GREEN_COMMIT_ID
```

The following example shows how the traffic section is configured. The revision with the *blue* `commitId` is taking 100% of production traffic while the newly deployed revision with *green* `commitId` doesn't take any production traffic.

JSON

```
{
  "traffic": [
    {
      "revisionName": "<APP_NAME>--fb699ef",
      "weight": 100,
      "label": "blue"
    },
    {
      "revisionName": "<APP_NAME>--c6f1515",
      "weight": 0,
      "label": "green"
    }
  ]
}
```

The newly deployed revision can be tested by using the label-specific FQDN:

Azure CLI

```
#get the containerapp environment default domain
export APP_DOMAIN=$(az containerapp env show -g $RESOURCE_GROUP -n
$APP_ENVIRONMENT_NAME --query properties.defaultDomain -o tsv | tr -d
'\r\n')

#Test the production FQDN
curl -s https://$APP_NAME.$APP_DOMAIN/api/env | jq | grep COMMIT

#Test the blue lable FQDN
```

```
curl -s https://$APP_NAME---blue.$APP_DOMAIN/api/env | jq | grep COMMIT  
  
#Test the green label FQDN  
curl -s https://$APP_NAME---green.$APP_DOMAIN/api/env | jq | grep COMMIT
```

Send production traffic to the green revision

After confirming that the app code in the *green* revision works as expected, 100% of production traffic is sent to the revision. The *green* revision now becomes the production revision.

Azure CLI

```
# set 100% of traffic to green revision  
az containerapp ingress traffic set \  
  --name $APP_NAME \  
  --resource-group $RESOURCE_GROUP \  
  --label-weight blue=0 green=100
```

The following example shows how the `traffic` section is configured after this step. The *green* revision with the new application code takes all the user traffic while *blue* revision with the old application version doesn't accept user requests.

JSON

```
{  
  "traffic": [  
    {  
      "revisionName": "<APP_NAME>--c6f1515",  
      "weight": 0,  
      "label": "blue"  
    },  
    {  
      "revisionName": "<APP_NAME>--fb699ef",  
      "weight": 100,  
      "label": "green"  
    }  
  ]  
}
```

Roll back the deployment if there were problems

If after running in production, the new revision is found to have bugs, you can roll back to the previous good state. After the rollback, 100% of the traffic is sent to the old version in the *blue* revision and that revision is designated as the production revision again.

Azure CLI

```
# set 100% of traffic to green revision
az containerapp ingress traffic set \
--name $APP_NAME \
--resource-group $RESOURCE_GROUP \
--label-weight blue=100 green=0
```

After the bugs are fixed, the new version of the application is deployed as a *green* revision again. The *green* version eventually becomes the production revision.

Next deployment cycle

Now the *green* label marks the revision currently running the stable production code.

During the next deployment cycle, the *blue* identifies the revision with the new application version being rolled out to production.

The following commands demonstrate how to prepare for the next deployment cycle.

Azure CLI

```
# set the new commitId
export BLUE_COMMIT_ID=ad1436b

# create a third revision for blue commitId
az containerapp update --name $APP_NAME \
--resource-group $RESOURCE_GROUP \
--image mcr.microsoft.com/k8se/samples/test-app:$BLUE_COMMIT_ID \
--revision-suffix $BLUE_COMMIT_ID \
--set-env-vars REVISION_COMMIT_ID=$BLUE_COMMIT_ID

# give that revision a 'blue' label
az containerapp revision label add \
--name $APP_NAME \
--resource-group $RESOURCE_GROUP \
--label blue \
--revision $APP_NAME-$BLUE_COMMIT_ID
```

Next steps

Traffic Weights

Session Affinity in Azure Container Apps (preview)

Article • 03/30/2023

Session affinity, also known as sticky sessions, is a feature that allows you to route all requests from a client to the same replica. This feature is useful for stateful applications that require a consistent connection to the same replica.

Session stickiness is enforced using HTTP cookies. This feature is available in single revision mode when HTTP ingress is enabled. A client may be routed to a new replica if the previous replica is no longer available.

If your app doesn't require session affinity, we recommend that you don't enable it. With session affinity disabled, ingress distributes requests more evenly across replicas improving the performance of your app.

ⓘ Note

Session affinity is only supported when your app is in **single revision mode** and the ingress type is HTTP.

This feature is in public preview.

Configure session affinity

You can enable session affinity when you create your container app via the Azure portal. To enable session affinity:

1. On the **Create Container App** page, select the **App settings** tab.
2. In the **Application ingress settings** section, select **Enabled** for the **Session affinity** setting.

Create Container App

Name	Value	Delete
<input type="text" value="Enter name"/>	<input type="text" value="Enter value"/>	

Application ingress settings

Enable ingress for applications that need an HTTP or TCP endpoint.

Ingress ⓘ Enabled

Ingress traffic **Limited to Container Apps Environment**
 Limited to VNet: Applies if 'internalOnly' setting is set to true on the Container Apps environment
 Accepting traffic from anywhere: Applies if 'internalOnly' setting is set to false on the Container Apps environment

Ingress type ⓘ **HTTP**
 TCP

Transport

Insecure connections Allowed

Target port * ⓘ

Session affinity ⓘ Enabled

Review + create [< Previous](#) [Next : Tags >](#)

You can also enable or disable session affinity after your container app is created. To enable session affinity:

1. Go to your app in the portal.
2. Select **Ingress**.
3. You can enable or disable **Session affinity** by selecting or deselecting **Enabled**.
4. Select **Save**.

my-container-app | Ingress

Container App

Search Refresh Send us your feedback

Overview Access control (IAM) Tags Diagnose and solve problems

Enable ingress for applications that need an HTTP or TCP endpoint.

Ingress Enabled

Limited to Container Apps Environment
 Limited to VNet: Applies if 'internalOnly' setting is set to true on the Container Apps environment
 Accepting traffic from anywhere: Applies if 'internalOnly' setting is set to false on the Container Apps environment

Ingress type HTTP TCP

Transport Auto

Insecure connections Allowed

Target port * 80

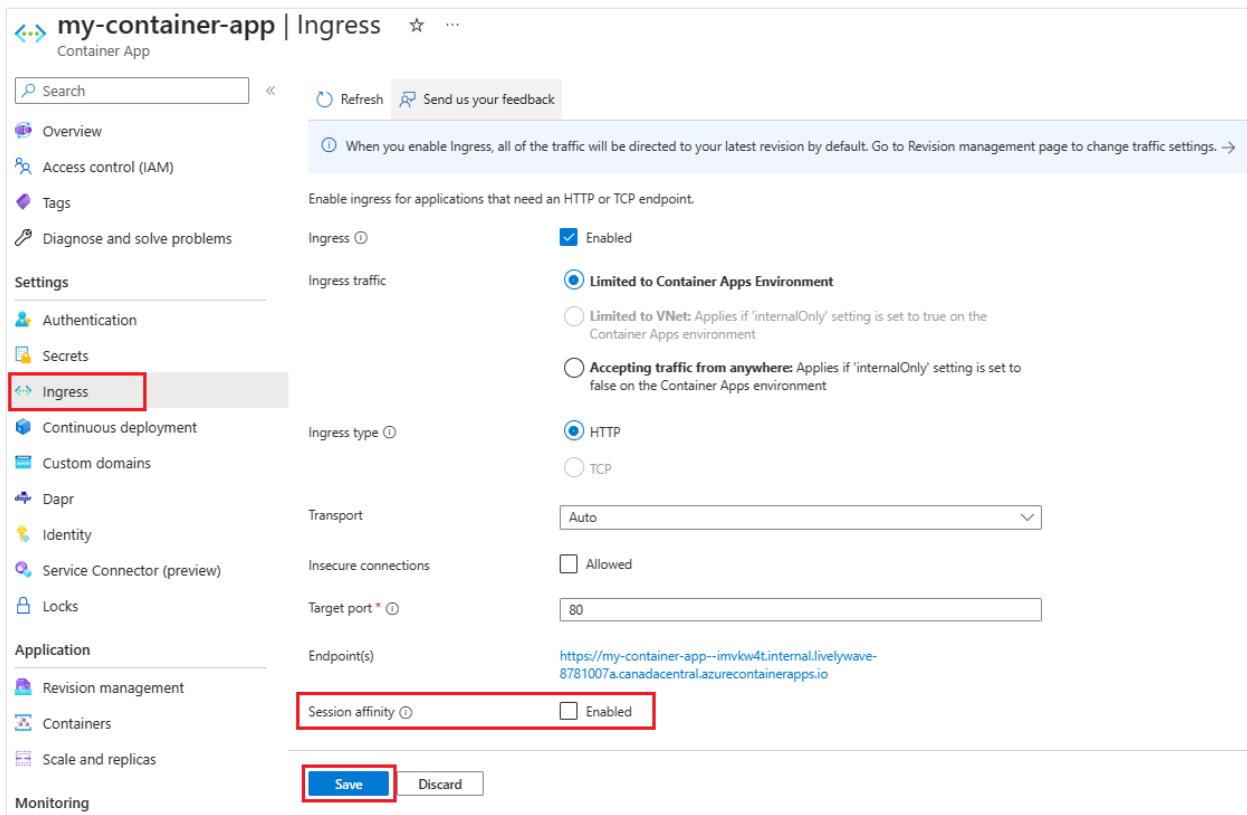
Endpoint(s) https://my-container-app--imvkw4t.internal.livelywave-8781007a.canadacentral.azurecontainerapps.io

Session affinity Enabled

Save Discard

Continuous deployment Custom domains Dapr Identity Service Connector (preview) Locks Revision management Containers Scale and replicas Monitoring

Ingress



Next steps

Configure ingress

Configure cross-origin resource sharing (CORS) for Azure Container Apps

Article • 05/04/2023

By default, requests made through the browser to a domain that doesn't match the page's origin domain are blocked. To avoid this restriction for services deployed to Container Apps, you can enable [CORS](#).

This article shows you how to enable and configure CORS in your container app.

As you enable CORS, you can configure the following settings:

Setting	Explanation
Allow credentials	Indicates whether to return the Access-Control-Allow-Credentials header.
Max age	Configures the Access-Control-Max-Age response header to indicate how long (in seconds) the results of a CORS pre-flight request can be cached.
Allowed origins	List of the origins allowed for cross-origin requests (for example, <code>https://www.contoso.com</code>). Controls the Access-Control-Allow-Origin response header. Use <code>*</code> to allow all.
Allowed methods	List of HTTP request methods allowed in cross-origin requests. Controls the Access-Control-Allow-Methods response header. Use <code>*</code> to allow all.
Allowed headers	List of the headers allowed in cross-origin requests. Controls the Access-Control-Allow-Headers response header. Use <code>*</code> to allow all.
Expose headers	By default, not all response headers are exposed to client-side JavaScript code in a cross-origin request. Exposed headers are extra headers servers can include in a response. Controls the Access-Control-Expose-Headers response header. Use <code>*</code> to expose all.

For more information, see the Web Hypertext Application Technology Working Group (WHATWG) reference on valid [HTTP responses from a fetch request](#).

Enable and configure CORS

1. Go to your container app in the Azure portal.
2. Under the settings menu, select *CORS*.

The screenshot shows the CORS configuration for a container app named 'my-container-app'. The left sidebar includes links for Search, Refresh, Locks, Application (Revision management, Containers, Scale and replicas), Monitoring (Alerts, Metrics, Logs, Log stream, Console), and CORS. The main area displays the CORS settings with 'Access-Control-Allow-Credentials' checked and 'Max Age' set to 0. Under 'Allowed Origins', there is one entry: 'https://example.com'. A text input field for adding more origins is shown below. At the bottom are 'Apply' and 'Discard' buttons.

With CORS enabled you can add, edit, and delete values for *Allowed Origins*, *Allowed Methods*, *Allowed Headers*, and *Expose Headers*.

To allow any acceptable values for methods, headers, or origins, enter `*` as the value.

Next steps

[Configure ingress](#)

Custom domain names and free managed certificates in Azure Container Apps (preview)

Article • 05/23/2023

Azure Container Apps allows you to bind one or more custom domains to a container app. You can automatically configure a free managed certificate for your custom domain.

If you want to set up a custom domain using your own certificate, see [Custom domain names and certificates in Azure Container Apps](#).

ⓘ Note

If you configure a **custom environment DNS suffix**, you cannot add a custom domain that contains this suffix to your Container App.

The managed certificates feature in Azure Container Apps is currently in preview.

Free certificate requirements

Azure Container Apps provides a free managed certificate for your custom domain. Without any action required from you, this TLS/SSL server certificate is automatically renewed as long as your app continues to meet the requirements for managed certificates.

The requirements are:

- Your container app has HTTP ingress enabled and is publicly accessible.
- For apex domains, you must have an A record pointing to your Container Apps environment's IP address.
- For subdomains, you must have a CNAME record mapped directly to the container app's automatically generated domain name. Mapping to an intermediate CNAME value blocks certificate issuance and renewal.

ⓘ Note

To ensure the certificate issuance and subsequent renewals proceed successfully, all requirements must be met at all times when the managed certificate is assigned.

Add a custom domain and managed certificate

1. Navigate to your container app in the [Azure portal](#)
2. Verify that your app has HTTP ingress enabled by selecting **Ingress** in the *Settings* section. If ingress isn't enabled, enable it with these steps:
 - a. Set *HTTP Ingress* to **Enabled**.
 - b. Select the desired *Ingress traffic* setting.
 - c. Enter the *Target port*.
 - d. Select **Save**.
3. Under the *Settings* section, select **Custom domains**.
4. Select **Add custom domain**.
5. In the *Add custom domain and certificate* window, in *TLS/SSL certificate*, select **Managed certificate**.
6. In *domain*, enter the domain you want to add.
7. Select the *Hostname record type* based on the type of your domain.

Domain type	Record type	Notes
Apex domain	A record	An apex domain is a domain at the root level of your domain. For example, if your DNS zone is <code>contoso.com</code> , then <code>contoso.com</code> is the apex domain.
Subdomain	CNAME	A subdomain is a domain that is part of another domain. For example, if your DNS zone is <code>contoso.com</code> , then <code>www.contoso.com</code> is an example of a subdomain that can be configured in the zone.

8. Using the DNS provider that is hosting your domain, create DNS records based on the *Hostname record type* you selected using the values shown in the *Domain validation* section. The records point the domain to your container app and verify that you are the owner.

- If you selected *A record*, create the following DNS records:

Record type	Host	Value
A	@	The IP address of your Container Apps environment

Record type	Host	Value
TXT	asuid	The domain verification code

- If you selected *CNAME*, create the following DNS records:

Record type	Host	Value
CNAME	The subdomain (for example, www)	The automatically generated <appname>. <region>.azurecontainerapps.io domain of your container app
TXT	asuid. followed by the subdomain (for example, asuid.www)	The domain verification code

9. Select **Validate**.

10. Once validation succeeds, select **Add**.

It may take several minutes to issue the certificate and add the domain to your container app.

11. Once the operation is complete, you see your domain name in the list of custom domains with a status of *Secured*. Navigate to your domain to verify that it's accessible.

Next steps

[Authentication in Azure Container Apps](#)

Custom domain names and bring your own certificates in Azure Container Apps

Article • 05/23/2023

Azure Container Apps allows you to bind one or more custom domains to a container app.

- Every domain name must be associated with a TLS/SSL certificate. You can upload your own certificate or use a [free managed certificate](#).
- Certificates are applied to the container app environment and are bound to individual container apps. You must have role-based access to the environment to add certificates.
- [SNI domain certificates](#) are required.
- Ingress must be enabled for the container app.

ⓘ Note

If you configure a **custom environment DNS suffix**, you cannot add a custom domain that contains this suffix to your Container App.

Add a custom domain and certificate

ⓘ Important

If you are using a new certificate, you must have an existing [SNI domain certificate](#) file available to upload to Azure.

1. Navigate to your container app in the [Azure portal](#)
2. Verify that your app has ingress enabled by selecting **Ingress** in the *Settings* section. If ingress is not enabled, enable it with these steps:
 - a. Set *HTTP Ingress* to **Enabled**.
 - b. Select the desired *Ingress traffic* setting.
 - c. Enter the *Target port*.
 - d. Select **Save**.
3. Under the *Settings* section, select **Custom domains**.

4. Select the **Add custom domain** button.
5. In the *Add custom domain and certificate* window, in *TLS/SSL certificate*, select **Bring your own certificate**.
6. In *domain*, enter the domain you want to add.
7. Select **Add a certificate**.
8. In the *Add certificate* window, in *Certificate name*, enter a name for this certificate.
9. In *Certificate file* section, browse for the certificate file you want to upload.
10. Select **Validate**.
11. Once validation succeeds, select **Add**.
12. In the *Add custom domain and certificate* window, in *Certificate*, select the certificate you just added.
13. Select the *Hostname record type* based on the type of your domain.

Domain type	Record type	Notes
Apex domain	A record	An apex domain is a domain at the root level of your domain. For example, if your DNS zone is <code>contoso.com</code> , then <code>contoso.com</code> is the apex domain.
Subdomain	CNAME	A subdomain is a domain that is part of another domain. For example, if your DNS zone is <code>contoso.com</code> , then <code>www.contoso.com</code> is an example of a subdomain that can be configured in the zone.

14. Using the DNS provider that is hosting your domain, create DNS records based on the *Hostname record type* you selected using the values shown in the *Domain validation* section. The records point the domain to your container app and verify that you own it.

- If you selected *A record*, create the following DNS records:

Record type	Host	Value
A	@	The IP address of your Container Apps environment
TXT	asuid	The domain verification code

- If you selected *CNAME*, create the following DNS records:

Record type	Host	Value
CNAME	The subdomain (for example, <code>www</code>)	The automatically generated domain of your container app
TXT	<code>asuid.</code> followed by the subdomain (for example, <code>asuid.www</code>)	The domain verification code

15. Select the **Validate** button.

16. Once validation succeeds, select the **Add** button.

17. Once the operation is complete, you see your domain name in the list of custom domains with a status of *Secured*. Navigate to your domain to verify that it's accessible.

ⓘ Note

For container apps in internal Container Apps environments, **additional configuration** is required to use custom domains with VNET-scope ingress.

Managing certificates

You can manage certificates via the Container Apps environment or through an individual container app.

Environment

The *Certificates* window of the Container Apps environment presents a table of all the certificates associated with the environment.

You can manage your certificates through the following actions:

Action	Description
Add	Select the Add certificate link to add a new certificate.
Delete	Select the trash can icon to remove a certificate.
Renew	The <i>Health status</i> field of the table indicates that a certificate is expiring soon within 60 days of the expiration date. To renew a certificate, select the Renew certificate link to upload a new certificate.

Container app

The *Custom domains* window of the container app presents a list of custom domains associated with the container app.

You can manage your certificates for an individual domain name by selecting the ellipsis (...) button, which opens the certificate binding window. From the following window, you can select a certificate to bind to the selected domain name.

Next steps

[Authentication in Azure Container Apps](#)

Custom environment DNS Suffix in Azure Container Apps (Preview)

Article • 12/06/2022

By default, an Azure Container Apps environment provides a DNS suffix in the format `<UNIQUE_IDENTIFIER>. <REGION_NAME>. azurecontainerapps.io`. Each container app in the environment generates a domain name based on this DNS suffix. You can configure a custom DNS suffix for your environment.

ⓘ Note

To configure a custom domain for individual container apps, see [Custom domain names and certificates in Azure Container Apps](#).

Add a custom DNS suffix and certificate

1. Go to your Container Apps environment in the [Azure portal](#) ↗
2. Under the *Settings* section, select **Custom DNS suffix**.
3. In **DNS suffix**, enter the custom DNS suffix for the environment.
For example, if you enter `example.com`, the container app domain names will be in the format `<APP_NAME>.example.com`.
4. In a new browser window, go to your domain provider's website and add the DNS records shown in the *Domain validation* section to your domain.

Record type	Host	Value	Description
A	<code>*</code> <code><DNS_SUFFIX></code>	Environment inbound IP address	Wildcard record configured to the IP address of the environment.
TXT	<code>asuid.</code> <code><DNS_SUFFIX></code>	Validation token	TXT record with the value of the validation token (not required for Container Apps environment with internal load balancer).

5. Back in the *Custom DNS suffix* window, in **Certificate file**, browse and select a certificate for the TLS binding.

ⓘ Important

You must use an existing wildcard certificate that's valid for the custom DNS suffix you provided.

6. In **Certificate password**, enter the password for the certificate.

7. Select **Save**.

Once the save operation is complete, the environment is updated with the custom DNS suffix and TLS certificate.

Next steps

[Custom domains in Azure Container Apps](#)

Protect Azure Container Apps with Web Application Firewall on Application Gateway

Article • 04/02/2023

When you host your apps or microservices in Azure Container Apps, you may not always want to publish them directly to the internet. Instead, you may want to expose them through a reverse proxy.

A reverse proxy is a service that sits in front of one or more services, intercepting and directing incoming traffic to the appropriate destination.

Reverse proxies allow you to place services in front of your apps that supports cross-cutting functionality including:

- Routing
- Caching
- Rate limiting
- Load balancing
- Security layers
- Request filtering

This article demonstrates how to protect your container apps using a [Web Application Firewall \(WAF\) on Azure Application Gateway](#) with an internal Container Apps environment.

For more information on networking concepts in Container Apps, see [Networking Environment in Azure Container Apps](#).

Prerequisites

- **Internal environment with custom VNet:** Have a container app that is on an internal environment and integrated with a custom virtual network. For more information on how to create a custom virtual network integrated app, see [provide a virtual network to an internal Azure Container Apps environment](#).
- **Security certificates:** If you must use TLS/SSL encryption to the application gateway, a valid public certificate that's used to bind to your application gateway is required.

Retrieve your container app's domain

Use the following steps to retrieve the values of the **default domain** and the **static IP** to set up your Private DNS Zone.

1. From the resource group's *Overview* window in the portal, select your container app.
2. On the *Overview* window for your container app resource, select the link for **Container Apps Environment**
3. On the *Overview* window for your container app environment resource, select **JSON View** in the upper right-hand corner of the page to view the JSON representation of the container apps environment.
4. Copy the values for the **defaultDomain** and **staticIp** properties and paste them into a text editor. You'll create a private DNS zone using these values for the default domain in the next section.

Create and configure an Azure Private DNS zone

1. On the Azure portal menu or the *Home* page, select **Create a resource**.
2. Search for *Private DNS Zone*, and select **Private DNS Zone** from the search results.
3. Select the **Create** button.
4. Enter the following values:

Setting	Action
Subscription	Select your Azure subscription.
Resource group	Select the resource group of your container app.
Name	Enter the defaultDomain property of the Container Apps Environment from the previous section.
Resource group location	Leave as the default. A value isn't needed as Private DNS Zones are global.

5. Select **Review + create**. After validation finishes, select **Create**.
6. After the private DNS zone is created, select **Go to resource**.

7. In the *Overview* window, select **+Record set**, to add a new record set.

8. In the *Add record set* window, enter the following values:

Setting	Action
Name	Enter *.
Type	Select A-Address Record .
TTL	Keep the default values.
TTL unit	Keep the default values.
IP address	Enter the staticIp property of the Container Apps Environment from the previous section.

9. Select **OK** to create the record set.

10. Select **+Record set** again, to add a second record set.

11. In the *Add record set* window, enter the following values:

Setting	Action
Name	Enter @.
Type	Select A-Address Record .
TTL	Keep the default values.
TTL unit	Keep the default values.
IP address	Enter the staticIp property of the Container Apps Environment from the previous section.

12. Select **OK** to create the record set.

13. Select the **Virtual network links** window from the menu on the left side of the page.

14. Select **+Add** to create a new link with the following values:

Setting	Action
Link name	Enter my-custom-vnet-pdns-link .
I know the resource ID of virtual network	Leave it unchecked.

Setting	Action
Virtual network	Select the virtual network your container app is integrated with.
Enable auto registration	Leave it unchecked.

15. Select **OK** to create the virtual network link.

Create and configure Azure Application Gateway

Basics tab

1. Enter the following values in the *Project details* section.

Setting	Action
Subscription	Select your Azure subscription.
Resource group	Select the resource group for your container app.
Application gateway name	Enter my-container-apps-agw .
Region	Select the location where your Container App was provisioned.
Tier	Select WAF V2 . You can use Standard V2 if you don't need WAF.
Enable autoscaling	Leave as default. For production environments, autoscaling is recommended. See Autoscaling Azure Application Gateway .
Availability zone	Select None . For production environments, Availability Zones are recommended for higher availability.
HTTP2	Keep the default value.
WAF Policy	Select Create new and enter my-waf-policy for the WAF Policy. Select OK . If you chose Standard V2 for the tier, skip this step.
Virtual network	Select the virtual network that your container app is integrated with.
Subnet	Select Manage subnet configuration . If you already have a subnet you wish to use, use that instead, and skip to the Frontends section .

- From within the *Subnets* window of *my-custom-vnet*, select **+Subnet** and enter the following values:

Setting	Action
Name	Enter appgateway-subnet .
Subnet address range	Keep the default values.

- For the remainder of the settings, keep the default values.
- Select **Save** to create the new subnet.
- Close the *Subnets* window to return to the *Create application gateway* window.
- Select the following values:

Setting	Action
Subnet	Select the appgateway-subnet you created.

- Select **Next: Frontends**, to proceed.

Frontends tab

- On the *Frontends* tab, enter the following values:

Setting	Action
Frontend IP address type	Select Public .
Public IP address	Select Add new . Enter my-frontend for the name of your frontend and select OK

ⓘ Note

For the Application Gateway v2 SKU, there must be a **Public** frontend IP. You can have both a public and a private frontend IP configuration, but a private-only frontend IP configuration with no public IP is currently not supported in the v2 SKU. To learn more, [read here](#).

- Select **Next: Backends**.

Backends tab

The backend pool is used to route requests to the appropriate backend servers. Backend pools can be composed of any combination of the following resources:

- NICs
- Public IP addresses
- Internal IP addresses
- Virtual Machine Scale Sets
- Fully qualified domain names (FQDN)
- Multi-tenant back-ends like Azure App Service and Container Apps

In this example, you create a backend pool that targets your container app.

1. Select **Add a backend pool**.
2. Open a new tab and navigate to your container app.
3. In the *Overview* window of the Container App, find the **Application Url** and copy it.
4. Return to the *Backends* tab, and enter the following values in the **Add a backend pool** window:

Setting	Action
Name	Enter my-agw-backend-pool .
Add backend pool without targets	Select No .
Target type	Select IP address or FQDN .
Target	Enter the Container App Application Url you copied and remove the https:// prefix. This location is the FQDN of your container app.

5. Select **Add**.
6. On the *Backends* tab, select **Next: Configuration**.

Configuration tab

On the *Configuration* tab, you connect the frontend and backend pool you created using a routing rule.

1. Select **Add a routing rule**. Enter the following values:

Setting	Action
Name	Enter my-agw-routing-rule .
Priority	Enter 1 .

2. Under Listener tab, enter the following values:

Setting	Action
Listener name	Enter my-agw-listener .
Frontend IP	Select Public .
Protocol	Select HTTPS . If you don't have a certificate you want to use, you can select HTTP
Port	Enter 443 . If you chose HTTP for your protocol, enter 80 and skip to the default/custom domain section.
Choose a Certificate	Select Upload a certificate . If your certificate is stored in key vault, you can select Choose a certificate from Key Vault .
Cert name	Enter a name for your certificate.
PFX certificate file	Select your valid public certificate.
Password	Enter your certificate password.

If you want to use the default domain, enter the following values:

Setting	Action
Listener Type	Select Basic
Error page url	Leave as No

Alternatively, if you want to use a custom domain, enter the following values:

Setting	Action
Listener Type	Select Multi site
Host type	Select Single
Host Names	Enter the Custom Domain you wish to use.

Setting	Action
Error page url	Leave as No

3. Select the **Backend targets** tab and enter the following values:

4. Toggle to the *Backend targets* tab and enter the following values:

Setting	Action
Target type	Select my-agw-backend-pool that you created earlier.
Backend settings	Select Add new .

5. In the *Add Backend setting* window, enter the following values:

Setting	Action
Backend settings name	Enter my-agw-backend-setting .
Backend protocol	Select HTTPS .
Backend port	Enter 443 .
Use well known CA certificate	Select Yes .
Override with new host name	Select Yes .
Host name override	Select Pick host name from backend target .
Create custom probes	Select No .

6. Select **Add**, to add the backend settings.

7. In the *Add a routing rule* window, select **Add** again.

8. Select **Next: Tags**.

9. Select **Next: Review + create**, and then select **Create**.

Add private link to your Application Gateway

This step is required for internal only container app environments as it allows your Application Gateway to communicate with your Container App on the backend through the virtual network.

1. Once the Application Gateway is created, select **Go to resource**.

2. From the menu on the left, select **Private link**, then select **Add**.

3. Enter the following values:

Setting	Action
Name	Enter my-agw-private-link .
Private link subnet	Select the subnet you wish to create the private link with.
Frontend IP Configuration	Select the frontend IP for your Application Gateway.

4. Under **Private IP address settings** select **Add**.

5. Select **Add** at the bottom of the window.

Verify the container app

Default domain

1. Find the public IP address for the application gateway on its *Overview* page, or you can search for the address. To search, select **All resources** and enter **my-container-apps-agw-pip** in the search box. Then, select the IP in the search results.
2. Navigate to the public IP address of the application gateway.
3. Your request is automatically routed to the container app, which verifies the application gateway was successfully created.

Clean up resources

When you no longer need the resources that you created, delete the resource group.

When you delete the resource group, you also remove all the related resources.

To delete the resource group:

1. On the Azure portal menu, select **Resource groups** or search for and select **Resource groups**.
2. On the *Resource groups* page, search for and select **my-container-apps**.

3. On the *Resource group page*, select **Delete resource group**.
4. Enter **my-container-apps** under *TYPE THE RESOURCE GROUP NAME* and then select **Delete**

Next steps

[Azure Firewall in Azure Container Apps](#)

Control outbound traffic with user defined routes (preview)

Article • 05/05/2023

ⓘ Note

This feature is in preview and is only supported for the workload profiles environment. User defined routes only work with an internal Azure Container Apps environment.

This article shows you how to use user defined routes (UDR) with [Azure Firewall](#) to lock down outbound traffic from your Container Apps to back-end Azure resources or other network resources.

Azure creates a default route table for your virtual networks on create. By implementing a user-defined route table, you can control how traffic is routed within your virtual network. In this guide, you setup UDR on the Container Apps virtual network to restrict outbound traffic with Azure Firewall.

You can also use a NAT gateway or any other third party appliances instead of Azure Firewall.

For more information on networking concepts in Container Apps, see [Networking Environment in Azure Container Apps](#).

Prerequisites

- **Internal environment:** An internal container app environment on the workload profiles environment that's integrated with a custom virtual network. When you create an internal container app environment, your container app environment has no public IP addresses, and all traffic is routed through the virtual network. For more information, see the [guide for how to create a container app environment on the workload profiles environment](#).
- **curl support:** Your container app must have a container that supports `curl` commands. In this how-to, you use `curl` to verify the container app is deployed correctly. If you don't have a container app with `curl` deployed, you can deploy the following container which supports `curl`,
`mcr.microsoft.com/k8se/quickstart:latest`.

Create the firewall subnet

A subnet called **AzureFirewallSubnet** is required in order to deploy a firewall into the integrated virtual network.

1. Open the virtual network that's integrated with your app in the [Azure portal](#).
2. From the menu on the left, select **Subnets**, then select **+ Subnet**.
3. Enter the following values:

Setting	Action
Name	Enter AzureFirewallSubnet .
Subnet address range	Use the default or specify a subnet range /26 or larger .

4. Select **Save**

Deploy the firewall

1. On the Azure portal menu or the **Home** page, select **Create a resource**.
2. Search for *Firewall*.
3. Select **Firewall**.
4. Select **Create**.
5. On the *Create a Firewall* page, configure the firewall with the following settings.

Setting	Action
Resource group	Enter the same resource group as the integrated virtual network.
Name	Enter a name of your choice
Region	Select the same region as the integrated virtual network.
Firewall policy	Create one by selecting Add new .
Virtual network	Select the integrated virtual network.
Public IP address	Select an existing address or create one by selecting Add new .

6. Select **Review + create**. After validation finishes, select **Create**. The validation step may take a few minutes to complete.

- Once the deployment completes, select **Go to Resource**.
- In the firewall's **Overview** page, copy the **Firewall private IP**. This IP address is used as the next hop address when creating the routing rule for the virtual network.

Route all traffic to the firewall

Your virtual networks in Azure have default route tables in place when you create the network. By implementing a user-defined route table, you can control how traffic is routed within your virtual network. In the following steps, you create a UDR to route all traffic to your Azure Firewall.

- On the Azure portal menu or the *Home* page, select **Create a resource**.
- Search for **Route tables**.
- Select **Route Tables**.
- Select **Create**.
- Enter the following values:

Setting	Action
Region	Select the region as your virtual network.
Name	Enter a name.
Propagate gateway routes	Select No

- Select **Review + create**. After validation finishes, select **Create**.
- Once the deployment completes, select **Go to Resource**.
- From the menu on the left, select **Routes**, then select **Add** to create a new route table
- Configure the route table with the following settings:

Setting	Action
Address prefix	Enter <i>0.0.0.0/0</i>
Next hop type	Select <i>Virtual appliance</i>
Next hop address	Enter the <i>Firewall Private IP</i> you saved in Deploy the firewall .

10. Select **Add** to create the route.
11. From the menu on the left, select **Subnets**, then select **Associate** to associate your route table with the container app's subnet.
12. Configure the *Associate subnet* with the following values:

Setting	Action
Address prefix	Select the virtual network for your container app.
Next hop type	Select the subnet your for container app.

13. Select **OK**.

Configure firewall policies

ⓘ Note

When using UDR with Azure Firewall in Azure Container Apps, you will need to add certain FQDN's and service tags to the allowlist for the firewall. Please refer to [configuring UDR with Azure Firewall](#) to determine which service tags you need.

Now, all outbound traffic from your container app is routed to the firewall. Currently, the firewall still allows all outbound traffic through. In order to manage what outbound traffic is allowed or denied, you need to configure firewall policies.

1. In your *Azure Firewall* resource on the *Overview* page, select **Firewall policy**
2. From the menu on the left of the firewall policy page, select **Application Rules**.
3. Select **Add a rule collection**.
4. Enter the following values for the **Rule Collection**:

Setting	Action
Name	Enter a collection name
Rule collection type	Select <i>Application</i>
Priority	Enter the priority such as 110
Rule collection action	Select <i>Allow</i>
Rule collection group	Select <i>DefaultApplicationRuleCollectionGroup</i>

5. Under Rules, enter the following values

Setting	Action
Name	Enter a name for the rule
Source type	Select <i>IP Address</i>
Source	Enter *
Protocol	Enter <i>http:80,https:443</i>
Destination Type	Select FQDN .
Destination	Enter <code>mcr.microsoft.com</code> , <code>*.data.mcr.microsoft.com</code> . If you're using ACR, add your <i>ACR address</i> and <code>*.blob.core.windows.net</code> .
Action	Select <i>Allow</i>

① Note

If you are using **Docker Hub registry** and want to access it through your firewall, you will need to add the following FQDNs to your rules destination list: `hub.docker.com`, `registry-1.docker.io`, and `production.cloudflare.docker.com`.

6. Select Add.

Verify your firewall is blocking outbound traffic

To verify your firewall configuration is set up correctly, you can use the `curl` command from your app's debugging console.

1. Navigate to your Container App that is configured with Azure Firewall.
2. From the menu on the left, select **Console**, then select your container that supports the `curl` command.
3. In the **Choose start up command** menu, select `/bin/sh`, and select **Connect**.
4. In the console, run `curl -s https://mcr.microsoft.com`. You should see a successful response as you added `mcr.microsoft.com` to the allowlist for your firewall policies.

5. Run `curl -s https://<FQDN_ADDRESS>` for a URL that doesn't match any of your destination rules such as `example.com`. The example command would be `curl -s https://example.com`. You should get no response, which indicates that your firewall has blocked the request.

Next steps

[Authentication in Azure Container Apps](#)

Connect a container app to a cloud service with Service Connector

Article • 04/14/2023

Azure Container Apps allows you to use Service Connector to connect to cloud services in just a few steps. Service Connector manages the configuration of the network settings and connection information between different services. To view all supported services, [learn more about Service Connector](#).

In this article, you learn to connect a container app to Azure Blob Storage.

ⓘ Important

This feature in Container Apps is currently in preview. See the [Supplemental Terms of Use for Microsoft Azure Previews](#) for legal terms that apply to Azure features that are in beta, preview, or otherwise not yet released into general availability.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An application deployed to Container Apps in a [region supported by Service Connector](#). If you don't have one yet, [create and deploy a container to Container Apps](#)
- An Azure Blob Storage account

Sign in to Azure

First, sign in to Azure.

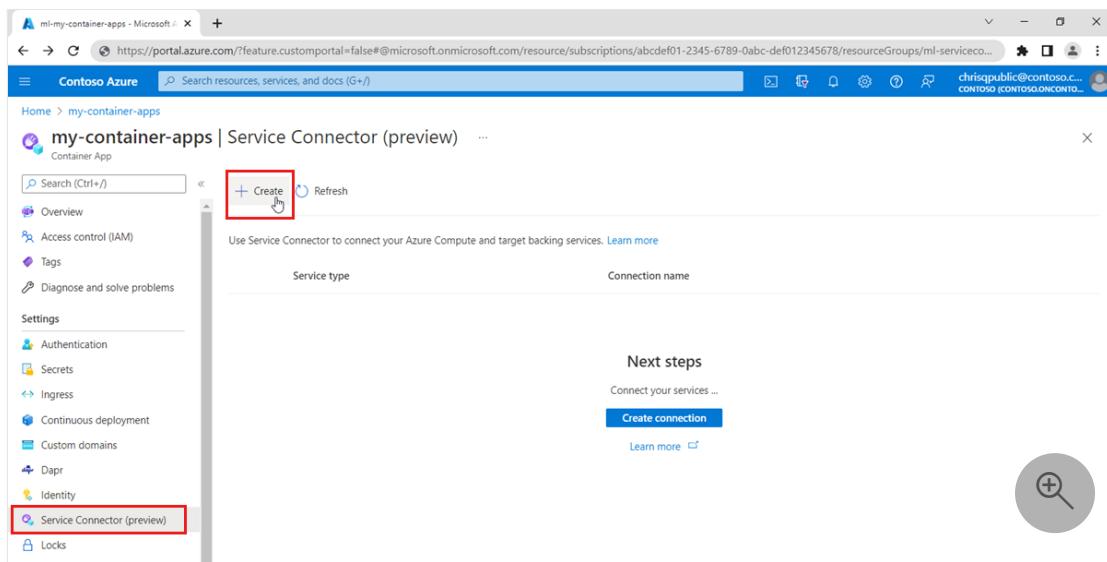
Portal

Sign in to the Azure portal at <https://portal.azure.com/> with your Azure account.

Create a new service connection

Use Service Connector to create a new service connection in Container Apps using the Azure portal or the CLI.

1. Navigate to the Azure portal.
2. Select **All resources** on the left of the Azure portal.
3. Enter **Container Apps** in the filter and select the name of the container app you want to use in the list.
4. Select **Service Connector** from the left table of contents.
5. Select **Create**.



6. Select or enter the following settings.

Setting	Suggested value	Description
Container	Your container name	Select your Container Apps.
Service type	Blob Storage	This is the target service type. If you don't have a Storage Blob container, you can create one or use another service type.
Subscription	One of your subscriptions	The subscription containing your target service. The default value is the subscription for your container app.
Connection name	Generated unique name	The connection name that identifies the connection between your container app and target service.

Setting	Suggested value	Description
Storage account	Your storage account name	The target storage account to which you want to connect. If you choose a different service type, select the corresponding target service instance.
Client type	The app stack in your selected container	Your application stack that works with the target service you selected. The default value is none , which generates a list of configurations. If you know about the app stack or the client SDK in the container you selected, select the same app stack for the client type.

7. Select **Next: Authentication** to select the authentication type. Then select **Connection string** to use access key to connect your Blob Storage account.
8. Select **Next: Network** to select the network configuration. Then select **Enable firewall settings** to update firewall allowlist in Blob Storage so that your container apps can reach the Blob Storage.
9. Then select **Next: Review + Create** to review the provided information. Running the final validation takes a few seconds. Then select **Create** to create the service connection. It might take a minute or so to complete the operation.

View service connections in Container Apps

View your existing service connections using the Azure portal or the CLI.

Portal

1. In **Service Connector**, select **Refresh** and you'll see a Container Apps connection displayed.
2. Select **>** to expand the list. You can see the environment variables required by your application code.
3. Select **...** and then **Validate**. You can see the connection validation details in the pop-up panel on the right.

The screenshot shows the Azure Service Connector configuration interface. At the top left are 'Create' and 'Refresh' buttons, with 'Refresh' highlighted by a red box and the number '1'. Below is a note about using Service Connectors to connect Azure Compute and target backing services, with a 'Learn more' link. The main area displays a table with two rows. The first row shows 'Service type' as 'Blob Storage' and 'Connection name' as 'storageblob_iscmx'. The second row shows 'AZURE_STORAGEBLOB_CONNECTION' and a note 'Hidden value. Click to show value'. To the right of the table is a context menu with three items: 'Sample code', 'Validate' (highlighted with a red box and the number '2'), and 'Edit' (highlighted with a red box and the number '3').

Next steps

[Environments in Azure Container Apps](#)

Tutorial: Connect to PostgreSQL Database from a Java Quarkus Container App without secrets using a managed identity

Article • 03/08/2023

Azure Container Apps provides a [managed identity](#) for your app, which is a turn-key solution for securing access to [Azure Database for PostgreSQL](#) and other Azure services. Managed identities in Container Apps make your app more secure by eliminating secrets from your app, such as credentials in the environment variables.

This tutorial walks you through the process of building, configuring, deploying, and scaling Java container apps on Azure. At the end of this tutorial, you'll have a [Quarkus](#) application storing data in a [PostgreSQL](#) database with a managed identity running on [Container Apps](#).

What you will learn:

- ✓ Configure a Quarkus app to authenticate using Azure Active Directory (Azure AD) with a PostgreSQL Database.
- ✓ Create an Azure container registry and push a Java app image to it.
- ✓ Create a Container App in Azure.
- ✓ Create a PostgreSQL database in Azure.
- ✓ Connect to a PostgreSQL Database with managed identity using Service Connector.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

1. Prerequisites

- [Azure CLI](#) version 2.45.0 or higher.
- [Git](#)
- [Java JDK](#)
- [Maven](#)
- [Docker](#)
- [GraalVM](#)

2. Create a container registry

Create a resource group with the [az group create](#) command. An Azure resource group is a logical container into which Azure resources are deployed and managed.

The following example creates a resource group named `myResourceGroup` in the East US Azure region.

Azure CLI

```
az group create --name myResourceGroup --location eastus
```

Create an Azure container registry instance using the [az acr create](#) command. The registry name must be unique within Azure, contain 5-50 alphanumeric characters. All letters must be specified in lower case. In the following example, `mycontainerregistry007` is used. Update this to a unique value.

Azure CLI

```
az acr create \
  --resource-group myResourceGroup \
  --name mycontainerregistry007 \
  --sku Basic
```

3. Clone the sample app and prepare the container image

This tutorial uses a sample Fruits list app with a web UI that calls a Quarkus REST API backed by [Azure Database for PostgreSQL](#). The code for the app is available [on GitHub](#). To learn more about writing Java apps using Quarkus and PostgreSQL, see the [Quarkus Hibernate ORM with Panache Guide](#) and the [Quarkus Datasource Guide](#).

Run the following commands in your terminal to clone the sample repo and set up the sample app environment.

git

```
git clone https://github.com/quarkusio/quarkus-quickstarts
cd quarkus-quickstarts/hibernate-orm-panache-quickstart
```

Modify your project

1. Add the required dependencies to your project's BOM file.

XML

```
<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-identity-providers-jdbc-postgresql</artifactId>
  <version>1.0.0-beta.1</version>
</dependency>
```

2. Configure the Quarkus app properties.

The Quarkus configuration is located in the `src/main/resources/application.properties` file. Open this file in your editor, and observe several default properties. The properties prefixed with `%prod` are only used when the application is built and deployed, for example when deployed to Azure App Service. When the application runs locally, `%prod` properties are ignored. Similarly, `%dev` properties are used in Quarkus' Live Coding / Dev mode, and `%test` properties are used during continuous testing.

Delete the existing content in `application.properties` and replace with the following to configure the database for dev, test, and production modes:

Flexible Server

properties

```
quarkus.package.type=uber-jar

quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.datasource.db-kind=postgresql
quarkus.datasource.jdbc.max-size=8
quarkus.datasource.jdbc.min-size=2
quarkus.hibernate-orm.log.sql=true
quarkus.hibernate-orm.sql-load-script=import.sql
quarkus.datasource.jdbc.acquisition-timeout = 10

%dev.quarkus.datasource.username=${AZURE_CLIENT_NAME}
%dev.quarkus.datasource.jdbc.url=jdbc:postgresql://${DBHOST}.postgre
sql.azuredatabase.com:5432/${DBNAME}?\
authenticationPluginClassName=com.azure.identity.providers.postgres
ql.AzureIdentityPostgresqlAuthenticationPlugin\
&sslmode=require\
&azure.clientId=${AZURE_CLIENT_ID}\
&azure.clientSecret=${AZURE_CLIENT_SECRET}\
&azure.tenantId=${AZURE_TENANT_ID}

%prod.quarkus.datasource.username=${AZURE_MI_NAME}
%prod.quarkus.datasource.jdbc.url=jdbc:postgresql://${DBHOST}.postgre
sql.azuredatabase.com:5432/${DBNAME}?\
authenticationPluginClassName=com.azure.identity.providers.postgres
```

```
ql.AzureIdentityPostgresqlAuthenticationPlugin\  
    &sslmode=require  
  
%dev.quarkus.class-loading.parent-first-artifacts=com.azure:azure-  
core::jar,\  
com.azure:azure-core-http-netty::jar,\  
io.projectreactor.netty:reactor-netty-core::jar,\  
io.projectreactor.netty:reactor-netty-http::jar,\  
io.netty:netty-resolver-dns::jar,\  
io.netty:netty-codec::jar,\  
io.netty:netty-codec-http::jar,\  
io.netty:netty-codec-http2::jar,\  
io.netty:netty-handler::jar,\  
io.netty:netty-resolver::jar,\  
io.netty:netty-common::jar,\  
io.netty:netty-transport::jar,\  
io.netty:netty-buffer::jar,\  
com.azure:azure-identity::jar,\  
com.azure:azure-identity-providers-core::jar,\  
com.azure:azure-identity-providers-jdbc-postgresql::jar,\  
com.fasterxml.jackson.core:jackson-core::jar,\  
com.fasterxml.jackson.core:jackson-annotations::jar,\  
com.fasterxml.jackson.core:jackson-databind::jar,\  
com.fasterxml.jackson.dataformat:jackson-dataformat-xml::jar,\  
com.fasterxml.jackson.datatype:jackson-datatype-jsr310::jar,\  
org.reactivestreams:reactive-streams::jar,\  
io.projectreactor:reactor-core::jar,\  
com.microsoft.azure:msal4j::jar,\  
com.microsoft.azure:msal4j-persistence-extension::jar,\  
org.codehaus.woodstox:stax2-api::jar,\  
com.fasterxml.woodstox:woodstox-core::jar,\  
com.nimbusds:oauth2-oidc-sdk::jar,\  
com.nimbusds:content-type::jar,\  
com.nimbusds:nimbus-jose-jwt::jar,\  
net.minidev:json-smart::jar,\  
net.minidev:accessors-smart::jar,\  
io.netty:netty-transport-native-unix-common::jar
```

Build and push a Docker image to the container registry

1. Build the container image.

Run the following command to build the Quarkus app image. You must tag it with the fully qualified name of your registry login server. The login server name is in the format *<registry-name>.azurecr.io* (must be all lowercase), for example, *mycontainerregistry007.azurecr.io*. Replace the name with your own registry name.

Bash

```
mvnw quarkus:add-extension -Dextensions="container-image-jib"
mvnw clean package -Pnative -Dquarkus.native.container-build=true -
Dquarkus.container-image.build=true -Dquarkus.container-
image.registry=mycontainerregistry007 -Dquarkus.container-
image.name=quarkus-postgres-passwordless-app -Dquarkus.container-
image.tag=v1
```

2. Log in to the registry.

Before pushing container images, you must log in to the registry. To do so, use the [az acr login][az-acr-login] command. Specify only the registry resource name when signing in with the Azure CLI. Don't use the fully qualified login server name.

Azure CLI

```
az acr login --name <registry-name>
```

The command returns a `Login Succeeded` message once completed.

3. Push the image to the registry.

Use [docker push][docker-push] to push the image to the registry instance. Replace `mycontainerregistry007` with the login server name of your registry instance. This example creates the `quarkus-postgres-passwordless-app` repository, containing the `quarkus-postgres-passwordless-app:v1` image.

Bash

```
docker push mycontainerregistry007/quarkus-postgres-passwordless-app:v1
```

4. Create a Container App on Azure

1. Create a Container Apps instance by running the following command. Make sure you replace the value of the environment variables with the actual name and location you want to use.

Azure CLI

```
RESOURCE_GROUP="myResourceGroup"
LOCATION="eastus"
CONTAINERAPPS_ENVIRONMENT="my-environment"

az containerapp env create \
--resource-group $RESOURCE_GROUP \
```

```
--name $CONTAINERAPPS_ENVIRONMENT \
--location $LOCATION
```

2. Create a container app with your app image by running the following command.

Replace the placeholders with your values. To find the container registry admin account details, see [Authenticate with an Azure container registry](#)

Azure CLI

```
CONTAINER_IMAGE_NAME=quarkus-postgres-passwordless-app:v1
REGISTRY_SERVER=mycontainerregistry007
REGISTRY_USERNAME=<REGISTRY_USERNAME>
REGISTRY_PASSWORD=<REGISTRY_PASSWORD>

az containerapp create \
--resource-group $RESOURCE_GROUP \
--name my-container-app \
--image $CONTAINER_IMAGE_NAME \
--environment $CONTAINERAPPS_ENVIRONMENT \
--registry-server $REGISTRY_SERVER \
--registry-username $REGISTRY_USERNAME \
--registry-password $REGISTRY_PASSWORD
```

5. Create and connect a PostgreSQL database with identity connectivity

Next, create a PostgreSQL Database and configure your container app to connect to a PostgreSQL Database with a system-assigned managed identity. The Quarkus app will connect to this database and store its data when running, persisting the application state no matter where you run the application.

1. Create the database service.

Flexible Server

Azure CLI

```
DB_SERVER_NAME='msdocs-quarkus-postgres-webapp-db'
ADMIN_USERNAME='demoadmin'
ADMIN_PASSWORD='<admin-password>'

az postgres flexible-server create \
--resource-group $RESOURCE_GROUP \
--name $DB_SERVER_NAME \
--location $LOCATION \
--admin-user $DB_USERNAME \
```

```
--admin-password $DB_PASSWORD \
--sku-name GP_Gen5_2
```

The following parameters are used in the above Azure CLI command:

- *resource-group* → Use the same resource group name in which you created the web app, for example `msdocs-quarkus-postgres-webapp-rg`.
- *name* → The PostgreSQL database server name. This name must be **unique across all Azure** (the server endpoint becomes `https://<name>.postgres.database.azure.com`). Allowed characters are A-Z, 0-9, and -. A good pattern is to use a combination of your company name and server identifier. (`msdocs-quarkus-postgres-webapp-db`)
- *location* → Use the same location used for the web app.
- *admin-user* → Username for the administrator account. It can't be `azure_superuser`, `admin`, `administrator`, `root`, `guest`, or `public`. For example, `demoadmin` is okay.
- *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 in this tutorial, you will create environment variables with these values where the \$ character has special meaning within the Linux container used to run Java apps.

- *public-access* → `None` which sets the server in public access mode with no firewall rules. Rules will be created in a later step.
 - *sku-name* → The name of the pricing tier and compute configuration, for example `GP_Gen5_2`. For more information, see [Azure Database for PostgreSQL pricing ↗](#).
1. Create a database named `fruits` within the PostgreSQL service with this command:

Flexible Server

Azure CLI

```
az postgres flexible-server db create \
--resource-group $RESOURCE_GROUP \
--server-name $DB_SERVER_NAME \
--database-name fruits
```

2. Install the [Service Connector](#) passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

3. Connect the database to the container app with a system-assigned managed identity, using the connection command.

Flexible Server

Azure CLI

```
az containerapp connection create postgres-flexible \
--resource-group $RESOURCE_GROUP \
--name my-container-app \
--target-resource-group $RESOURCE_GROUP \
--server $DB_SERVER_NAME \
--database fruits \
--managed-identity
```

6. Review your changes

You can find the application URL(FQDN) by using the following command:

Azure CLI

```
az containerapp list --resource-group $RESOURCE_GROUP
```

When the new webpage shows your list of fruits, your app is connecting to the database using the managed identity. You should now be able to edit fruit list as before.

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, delete the resource group by running the following command in the Cloud Shell:

Azure CLI

```
az group delete --name myResourceGroup
```

This command may take a minute to run.

Next steps

Learn more about running Java apps on Azure in the developer guide.

[Azure for Java Developers](#)

Connect applications in Azure Container Apps

Article • 03/30/2023

Azure Container Apps exposes each container app through a domain name if [ingress](#) is enabled. Ingress endpoints can be exposed either publicly to the world and to other container apps in the same environment, or ingress can be limited to only other container apps in the same [environment](#).

You can call other container apps in the same environment from your application code using one of the following methods:

- default fully qualified domain name (FQDN)
- a custom domain name
- the container app name, for instance `http://<APP_NAME>` for internal requests
- a Dapr URL

ⓘ Note

When you call another container in the same environment using the FQDN or app name, the network traffic never leaves the environment.

A sample solution showing how you can call between containers using both the FQDN Location or Dapr can be found on [Azure Samples ↗](#)

Location

A container app's location is composed of values associated with its environment, name, and region. Available through the `azurecontainerapps.io` top-level domain, the fully qualified domain name (FQDN) uses:

- the container app name
- the environment unique identifier
- region name

The following diagram shows how these values are used to compose a container app's fully qualified domain name.



Get fully qualified domain name

The `az containerapp show` command returns the fully qualified domain name of a container app.

```
Bash
```

```
Azure CLI
```

```
az containerapp show \
--resource-group <RESOURCE_GROUP_NAME> \
--name <CONTAINER_APP_NAME> \
--query properties.configuration.ingress.fqdn
```

In this example, replace the placeholders surrounded by `<>` with your values.

The value returned from this command resembles a domain name like the following example:

```
Console
```

```
myapp.happyhill-70162bb9.canadacentral.azurecontainerapps.io
```

Dapr location

Developing microservices often requires you to implement patterns common to distributed architecture. Dapr allows you to secure microservices with mutual TLS (client certificates), trigger retries when errors occur, and take advantage of distributed tracing when Azure Application Insights is enabled.

A microservice that uses Dapr is available through the following URL pattern:



CONTAINER APP ADDRESS WITH DAPR

`http://localhost:3500/v1.0/invoke/myapp`

1

1 Container app name

Next steps

[Get started](#)

Deploy to Azure Container Apps from Azure Pipelines

Article • 05/03/2023

Azure Container Apps allows you to use Azure Pipelines to publish [revisions](#) to your container app. As commits are pushed to your [Azure DevOps repository](#), a pipeline is triggered which updates the container image in the container registry. Azure Container Apps creates a new revision based on the updated container image.

The pipeline is triggered by commits to a specific branch in your repository. When creating the pipeline, you decide which branch is the trigger.

Container Apps Azure Pipelines task

The task supports the following scenarios:

- Build from a Dockerfile and deploy to Container Apps
- Build from source code without a Dockerfile and deploy to Container Apps.
Supported languages include .NET, Node.js, PHP, Python, and Ruby
- Deploy an existing container image to Container Apps

With the production release this task comes with Azure DevOps and no longer requires explicit installation. For the complete documentation please see [AzureContainerApps@1 - Azure Container Apps Deploy v1 task](#).

Usage examples

Here are some common scenarios for using the task. For more information, see the [task's documentation](#).

Build and deploy to Container Apps

The following snippet shows how to build a container image from source code and deploy it to Container Apps.

YAML

```
steps:  
- task: AzureContainerApps@1  
  inputs:  
    appSourcePath: '$(Build.SourcesDirectory)/src'
```

```
azureSubscription: 'my-subscription-service-connection'
acrName: 'myregistry'
containerAppName: 'my-container-app'
resourceGroup: 'my-container-app-rg'
```

The task uses the Dockerfile in `appSourcePath` to build the container image. If no Dockerfile is found, the task attempts to build the container image from source code in `appSourcePath`.

Deploy an existing container image to Container Apps

The following snippet shows how to deploy an existing container image to Container Apps. Note, that we're deploying a publicly available image and won't need any registry authentication as a result.

YAML

```
steps:
- task: AzureContainerApps@1
  inputs:
    azureSubscription: 'my-subscription-service-connection'
    imageToDeploy : 'mcr.microsoft.com/azuredocs/containerapps-
helloworld:latest'
    containerAppName: 'my-container-app'
    resourceGroup: 'my-container-app-rg'
    imageToDeploy: 'myregistry.azurecr.io/my-container-
app:${Build.BuildId}'
```

ⓘ Important

If you're building a container image in a separate step, make sure you use a unique tag such as the build ID instead of a stable tag like `latest`. For more information, see [Image tag best practices](#).

Authenticate with Azure Container Registry

The Azure Container Apps task needs to authenticate with your Azure Container Registry to push the container image. The container app also needs to authenticate with your Azure Container Registry to pull the container image.

To push images, the task automatically authenticates with the container registry specified in `acrName` using the service connection provided in `azureSubscription`.

To pull images, Azure Container Apps uses either managed identity (recommended) or admin credentials to authenticate with the Azure Container Registry. To use managed identity, the target container app for the task must be [configured to use managed identity](#). To authenticate with the registry's admin credentials, set the task's `acrUsername` and `acrPassword` inputs.

Configuration

Take the following steps to configure an Azure DevOps pipeline to deploy to Azure Container Apps.

- ✓ Create an Azure DevOps repository for your app
- ✓ Create a container app with managed identity enabled
- ✓ Assign the `AcrPull` role for the Azure Container Registry to the container app's managed identity
- ✓ Install the Azure Container Apps task from the Azure DevOps Marketplace
- ✓ Configure an Azure DevOps service connection for your Azure subscription
- ✓ Create an Azure DevOps pipeline

Prerequisites

Requirement	Instructions
Azure account	If you don't have one, create an account for free . You need the <i>Contributor</i> or <i>Owner</i> permission on the Azure subscription to proceed. Refer to Assign Azure roles using the Azure portal for details.
Azure Devops project	Go to Azure DevOps and select <i>Start free</i> . Then create a new project.
Azure CLI	Install the Azure CLI .

Create an Azure DevOps repository and clone the source code

Before creating a pipeline, the source code for your app must be in a repository.

1. Log in to [Azure DevOps](#) and navigate to your project.
2. Open the **Repos** page.

3. In the top navigation bar, select the repositories dropdown and select **Import repository**.

4. Enter the following information and select **Import**:

Field	Value
Repository type	Git
Clone URL	<code>https://github.com/Azure-Samples/containerapps-albumapi-csharp.git</code>
Name	<code>my-container-app</code>

5. Select **Clone** to view the repository URL and copy it.

6. Open a terminal and run the following command to clone the repository:

```
Bash
```

```
git clone <REPOSITORY_URL> my-container-app
```

Replace `<REPOSITORY_URL>` with the URL you copied.

Create a container app and configure managed identity

Create your container app using the `az containerapp up` command with the following steps. This command creates Azure resources, builds the container image, stores the image in a registry, and deploys to a container app.

After your app is created, you can add a managed identity to your app and assign the identity the `AcrPull` role to allow the identity to pull images from the registry.

1. Change into the `src` folder of the cloned repository.

```
Bash
```

```
cd my-container-app  
cd src
```

2. Create Azure resources and deploy a container app with the [az containerapp up command](#).

```
Azure CLI
```

```
az containerapp up \
--name my-container-app \
--source . \
--ingress external
```

3. In the command output, note the name of the Azure Container Registry.

4. Get the full resource ID of the container registry.

Azure CLI

```
az acr show --name <ACR_NAME> --query id --output tsv
```

Replace `<ACR_NAME>` with the name of your registry.

5. Enable managed identity for the container app.

Azure CLI

```
az containerapp identity assign \
--name my-container-app \
--resource-group my-container-app-rg \
--system-assigned \
--output tsv
```

Note the principal ID of the managed identity in the command output.

6. Assign the `AcrPull` role for the Azure Container Registry to the container app's managed identity.

Azure CLI

```
az role assignment create \
--assignee <MANAGED_IDENTITY_PRINCIPAL_ID> \
--role AcrPull \
--scope <ACR_RESOURCE_ID>
```

Replace `<MANAGED_IDENTITY_PRINCIPAL_ID>` with the principal ID of the managed identity and `<ACR_RESOURCE_ID>` with the resource ID of the Azure Container Registry.

7. Configure the container app to use the managed identity to pull images from the Azure Container Registry.

Azure CLI

```
az containerapp registry set \
--name my-container-app \
--resource-group my-container-app-rg \
--server <ACR_NAME>.azurecr.io \
--identity system
```

Replace `<ACR_NAME>` with the name of your Azure Container Registry.

Create an Azure DevOps service connection

To deploy to Azure Container Apps, you need to create an Azure DevOps service connection for your Azure subscription.

1. In Azure DevOps, select **Project settings**.
2. Select **Service connections**.
3. Select **New service connection**.
4. Select **Azure Resource Manager**.
5. Select **Service principal (automatic)** and select **Next**.
6. Enter the following information and select **Save**:

Field	Value
Subscription	Select your Azure subscription.
Resource group	Select the resource group (<code>my-container-app-rg</code>) that contains your container app and container registry.
Service connection name	<code>my-subscription-service-connection</code>

To learn more about service connections, see [Connect to Microsoft Azure](#).

Create an Azure DevOps YAML pipeline

1. In your Azure DevOps project, select **Pipelines**.
2. Select **New pipeline**.
3. Select **Azure Repos Git**.

4. Select the repo that contains your source code (`my-container-app`).

5. Select **Starter pipeline**.

6. In the editor, replace the contents of the file with the following YAML:

```
YAML

trigger:
  branches:
    include:
      - main

pool:
  vmImage: ubuntu-latest

steps:
  - task: AzureContainerApps@1
    inputs:
      appSourcePath: '$(Build.SourcesDirectory)/src'
      azureSubscription: '<AZURE_SUBSCRIPTION_SERVICE_CONNECTION>'
      acrName: '<ACR_NAME>'
      containerAppName: 'my-container-app'
      resourceGroup: 'my-container-app-rg'
```

Replace `<AZURE_SUBSCRIPTION_SERVICE_CONNECTION>` with the name of the Azure DevOps service connection (`my-subscription-service-connection`) you created in the previous step and `<ACR_NAME>` with the name of your Azure Container Registry.

7. Select **Save and run**.

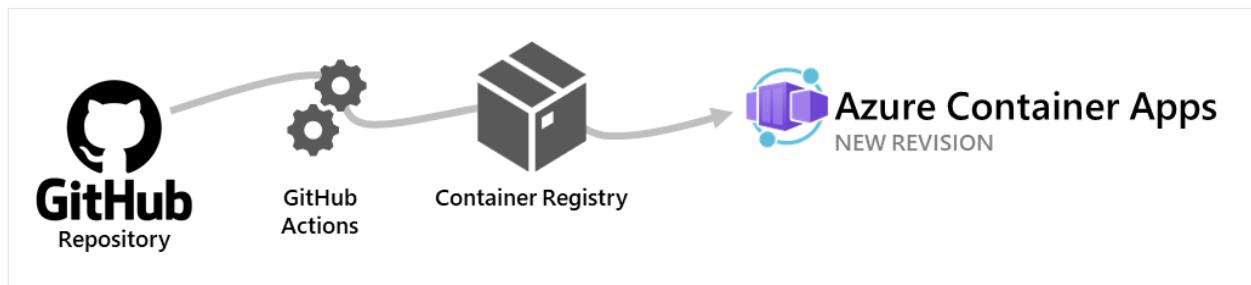
An Azure Pipelines run starts to build and deploy your container app. To check its progress, navigate to *Pipelines* and select the run. During the first pipeline run, you may be prompted to authorize the pipeline to use your service connection.

To deploy a new revision of your app, push a new commit to the *main* branch.

Deploy to Azure Container Apps with GitHub Actions

Article • 03/08/2023

Azure Container Apps allows you to use GitHub Actions to publish [revisions](#) to your container app. As commits are pushed to your GitHub repository, a workflow is triggered which updates the container image in the container registry. Azure Container Apps creates a new revision based on the updated container image.



The GitHub Actions workflow is triggered by commits to a specific branch in your repository. When creating the workflow, you decide which branch triggers the workflow.

This article shows you how to create a fully customizable workflow. To generate a starter GitHub Actions workflow with Azure CLI, see [Generate GitHub Actions workflow with Azure CLI](#).

Azure Container Apps GitHub action

To build and deploy your container app, you add the [azure/container-apps-deploy-action](#) action to your GitHub Actions workflow.

The action supports the following scenarios:

- Build from a Dockerfile and deploy to Container Apps
- Build from source code without a Dockerfile and deploy to Container Apps.
Supported languages include .NET, Node.js, PHP, Python, and Ruby
- Deploy an existing container image to Container Apps

Usage examples

Here are some common scenarios for using the action. For more information, see the action's [GitHub Marketplace page](#).

Build and deploy to Container Apps

The following snippet shows how to build a container image from source code and deploy it to Container Apps.

YAML

```
steps:  
  
  - name: Log in to Azure  
    uses: azure/login@v1  
    with:  
      creds: ${{ secrets.AZURE_CREDENTIALS }}  
  
  - name: Build and deploy Container App  
    uses: azure/container-apps-deploy-action@v1  
    with:  
      appSourcePath: ${{ github.workspace }}/src  
      acrName: myregistry  
      containerAppName: my-container-app  
      resourceGroup: my-rg
```

The action uses the Dockerfile in `appSourcePath` to build the container image. If no Dockerfile is found, the action attempts to build the container image from source code in `appSourcePath`.

Deploy an existing container image to Container Apps

The following snippet shows how to deploy an existing container image to Container Apps.

YAML

```
steps:  
  
  - name: Log in to Azure  
    uses: azure/login@v1  
    with:  
      creds: ${{ secrets.AZURE_CREDENTIALS }}  
  
  - name: Build and deploy Container App  
    uses: azure/container-apps-deploy-action@v1  
    with:  
      acrName: myregistry  
      containerAppName: my-container-app  
      resourceGroup: my-rg  
      imageToDeploy: myregistry.azurecr.io/app:${{ github.sha }}
```

ⓘ Important

If you're building a container image in a separate step, make sure you use a unique tag such as the commit SHA instead of a stable tag like `latest`. For more information, see [Image tag best practices](#).

Authenticate with Azure Container Registry

The Azure Container Apps action needs to authenticate with your Azure Container Registry to push the container image. The container app also needs to authenticate with your Azure Container Registry to pull the container image.

To push images, the action automatically authenticates with the container registry specified in `acrName` using the credentials provided to the `azure/login` action.

To pull images, Azure Container Apps uses either managed identity (recommended) or admin credentials to authenticate with the Azure Container Registry. To use managed identity, the container app the action is deploying must be [configured to use managed identity](#). To authenticate with the registry's admin credentials, set the action's `acrUsername` and `acrPassword` inputs.

Configuration

You take the following steps to configure a GitHub Actions workflow to deploy to Azure Container Apps.

- ✓ Create a GitHub repository for your app
- ✓ Create a container app with managed identity enabled
- ✓ Assign the `AcrPull` role for the Azure Container Registry to the container app's managed identity
- ✓ Configure secrets in your GitHub repository
- ✓ Create a GitHub Actions workflow

Prerequisites

Requirement	Instructions
Azure account	If you don't have one, create an account for free . You need the <i>Contributor</i> or <i>Owner</i> permission on the Azure subscription to proceed. Refer to Assign Azure roles using the Azure portal for details.

Requirement	Instructions
GitHub Account	Sign up for free ↗ .
Azure CLI	Install the Azure CLI .

Create a GitHub repository and clone source code

Before creating a workflow, the source code for your app must be in a GitHub repository.

1. Log in to Azure with the Azure CLI.

```
Azure CLI
```

```
az login
```

2. Next, install the latest Azure Container Apps extension for the CLI.

```
Azure CLI
```

```
az extension add --name containerapp --upgrade
```

3. If you do not have your own GitHub repository, create one from a sample.

- a. Navigate to the following location to create a new repository:

- [https://github.com/Azure-Samples/containerapps-albumapi-csharp/generate ↗](https://github.com/Azure-Samples/containerapps-albumapi-csharp/generate)

- b. Name your repository `my-container-app`.

4. Clone the repository to your local machine.

```
git
```

```
git clone https://github.com/<YOUR_GITHUB_ACCOUNT_NAME>/my-container-app.git
```

Create a container app with managed identity enabled

Create your container app using the `az containerapp up` command in the following steps. This command will create Azure resources, build the container image, store the

image in a registry, and deploy to a container app.

After you create your app, you can add a managed identity to the app and assign the identity the `AcrPull` role to allow the identity to pull images from the registry.

1. Change into the `src` folder of the cloned repository.

```
Bash
```

```
cd my-container-app  
cd src
```

2. Create Azure resources and deploy a container app with the [az containerapp up command](#).

```
Azure CLI
```

```
az containerapp up \  
--name my-container-app \  
--source . \  
--ingress external
```

3. In the command output, note the name of the Azure Container Registry.

4. Get the full resource ID of the container registry.

```
Azure CLI
```

```
az acr show --name <ACR_NAME> --query id --output tsv
```

Replace `<ACR_NAME>` with the name of your registry.

5. Enable managed identity for the container app.

```
Azure CLI
```

```
az containerapp identity assign \  
--name my-container-app \  
--resource-group my-container-app-rg \  
--system-assigned \  
--output tsv
```

Note the principal ID of the managed identity in the command output.

6. Assign the `AcrPull` role for the Azure Container Registry to the container app's managed identity.

Azure CLI

```
az role assignment create \
--assignee <MANAGED_IDENTITY_PRINCIPAL_ID> \
--role AcrPull \
--scope <ACR_RESOURCE_ID>
```

Replace `<MANAGED_IDENTITY_PRINCIPAL_ID>` with the principal ID of the managed identity and `<ACR_RESOURCE_ID>` with the resource ID of the Azure Container Registry.

7. Configure the container app to use the managed identity to pull images from the Azure Container Registry.

Azure CLI

```
az containerapp registry set \
--name my-container-app \
--resource-group my-container-app-rg \
--server <ACR_NAME>.azurecr.io \
--identity system
```

Replace `<ACR_NAME>` with the name of your Azure Container Registry.

Configure secrets in your GitHub repository

The GitHub workflow requires a secret named `AZURE_CREDENTIALS` to authenticate with Azure. The secret contains the credentials for a service principal with the *Contributor* role on the resource group containing the container app and container registry.

1. Create a service principal with the *Contributor* role on the resource group that contains the container app and container registry.

Azure CLI

```
az ad sp create-for-rbac \
--name my-app-credentials \
--role contributor \
--scopes /subscriptions/<SUBSCRIPTION_ID>/resourceGroups/my-
container-app-rg \
--sdk-auth \
--output json
```

Replace `<SUBSCRIPTION_ID>` with the ID of your Azure subscription. If your container registry is in a different resource group, specify both resource groups in

the `--scopes` parameter.

2. Copy the JSON output from the command.
3. In the GitHub repository, navigate to *Settings > Secrets > Actions* and select **New repository secret**.
4. Enter `AZURE_CREDENTIALS` as the name and paste the contents of the JSON output as the value.
5. Select **Add secret**.

Create a GitHub Actions workflow

1. In the GitHub repository, navigate to *Actions* and select **New workflow**.
2. Select **Set up a workflow yourself**.
3. Paste the following YAML into the editor.

```
YAML

name: Azure Container Apps Deploy

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

  steps:
    - uses: actions/checkout@v3

    - name: Log in to Azure
      uses: azure/login@v1
      with:
        creds: ${{ secrets.AZURE_CREDENTIALS }}

    - name: Build and deploy Container App
      uses: azure/container-apps-deploy-action@v1
      with:
        appSourcePath: ${{ github.workspace }}/src
        acrName: <ACR_NAME>
        containerAppName: my-container-app
        resourceGroup: my-container-app-rg
```

Replace `<ACR_NAME>` with the name of your Azure Container Registry. Confirm that the branch name under `branches` and values for `appSourcePath`, `containerAppName`, and `resourceGroup` match the values for your repository and Azure resources.

4. Commit the changes to the *main* branch.

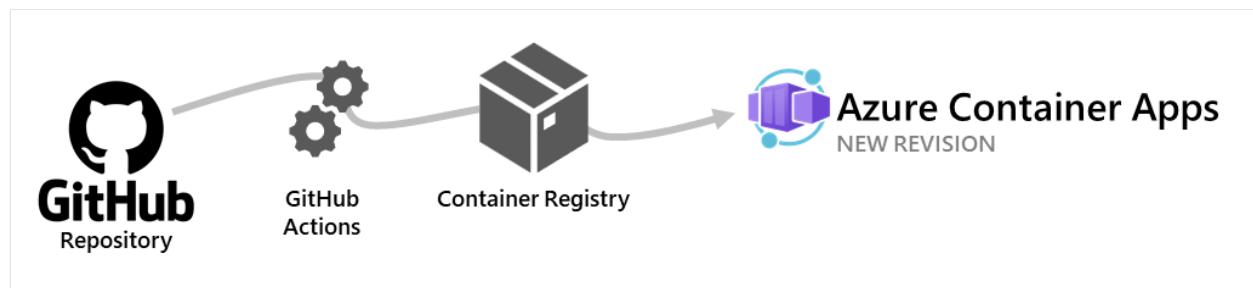
A GitHub Actions workflow run should start to build and deploy your container app. To check its progress, navigate to *Actions*.

To deploy a new revision of your app, push a new commit to the *main* branch.

Set up GitHub Actions with Azure CLI in Azure Container Apps

Article • 04/09/2023

Azure Container Apps allows you to use GitHub Actions to publish [revisions](#) to your container app. As commits are pushed to your GitHub repository, a GitHub Actions workflow is triggered which updates the [container](#) image in the container registry. Once the container is updated in the registry, Azure Container Apps creates a new revision based on the updated container image.



The GitHub Actions workflow is triggered by commits to a specific branch in your repository. When creating the workflow, you decide which branch triggers the action.

This article shows you how to generate a starter GitHub Actions workflow with Azure CLI. To create your own workflow that you can fully customize, see [Deploy to Azure Container Apps with GitHub Actions](#).

Authentication

When adding or removing a GitHub Actions integration, you can authenticate by either passing in a GitHub [personal access token](#), or using the interactive GitHub login experience. The interactive experience opens a form in your web browser and gives you the opportunity to log in to GitHub. Once successfully authenticated, then a token is passed back to the CLI that is used by GitHub for the rest of the current session.

- To pass a personal access token, use the `--token` parameter and provide a token value.
- If you choose to use interactive login, use the `--login-with-github` parameter with no value.

! Note

Your GitHub personal access token needs to have the `workflow` scope selected.

Add

The `containerapp github-action add` command creates a GitHub Actions integration with your container app.

ⓘ Note

Before you proceed with the example below, you must have your first container app already deployed.

The first time you attach GitHub Actions to your container app, you need to provide a service principal context. The following command shows you how to create a service principal.

Bash

Azure CLI

```
az ad sp create-for-rbac \
--name <SERVICE_PRINCIPAL_NAME> \
--role "contributor" \
--scopes
/subscriptions/<SUBSCRIPTION_ID>/resourceGroups/<RESOURCE_GROUP_NAME>
```

As you interact with this example, replace the placeholders surrounded by `<>` with your values.

The return values from this command include the service principal's `appId`, `password`, and `tenant`. You need to pass these values to the `az containerapp github-action add` command.

The following example shows you how to add an integration while using a personal access token.

Bash

Azure CLI

```
az containerapp github-action add \
--repo-url "https://github.com/<OWNER>/<REPOSITORY_NAME>" \
--context-path "./dockerfile" \
```

```
--branch <BRANCH_NAME> \
--name <CONTAINER_APP_NAME> \
--resource-group <RESOURCE_GROUP> \
--registry-url <URL_TO_CONTAINER_REGISTRY> \
--registry-username <REGISTRY_USER_NAME> \
--registry-password <REGISTRY_PASSWORD> \
--service-principal-client-id <appId> \
--service-principal-client-secret <password> \
--service-principal-tenant-id <tenant> \
--token <YOUR_GITHUB_PERSONAL_ACCESS_TOKEN>
```

As you interact with this example, replace the placeholders surrounded by `<>` with your values.

Show

The `containerapp github-action show` command returns the GitHub Actions configuration settings for a container app.

This example shows how to add an integration while using the personal access token.

Bash

Azure CLI

```
az containerapp github-action show \
--resource-group <RESOURCE_GROUP_NAME> \
--name <CONTAINER_APP_NAME>
```

As you interact with this example, replace the placeholders surrounded by `<>` with your values.

This command returns a JSON payload with the GitHub Actions integration configuration settings.

Delete

The `containerapp github-action delete` command removes the GitHub Actions from the container app.

Bash

Azure CLI

```
az containerapp github-action delete \
--resource-group <RESOURCE_GROUP_NAME> \
--name <CONTAINER_APP_NAME> \
--token <YOUR_GITHUB_PERSONAL_ACCESS_TOKEN>
```

As you interact with this example, replace the placeholders surrounded by `<>` with your values.

Provide a virtual network to an external Azure Container Apps environment

Article • 04/07/2023

The following example shows you how to create a Container Apps environment in an existing virtual network.

Begin by signing in to the [Azure portal](#).

Create a container app

To create your container app, start at the Azure portal home page.

1. Search for **Container Apps** in the top search bar.
2. Select **Container Apps** in the search results.
3. Select the **Create** button.

Basics tab

In the *Basics* tab, do the following actions.

1. Enter the following values in the *Project details* section.

Setting	Action
Subscription	Select your Azure subscription.
Resource group	Select Create new and enter my-container-apps .
Container app name	Enter my-container-app .

Create an environment

Next, create an environment for your container app.

1. Select the appropriate region.

Setting	Value
Region	Select Central US .

2. In the *Create Container Apps environment* field, select the **Create new** link.

3. In the *Create Container Apps Environment* page on the **Basics** tab, enter the following values:

Setting	Value
Environment name	Enter my-environment .
Zone redundancy	Select Disabled

4. Select the **Monitoring** tab to create a Log Analytics workspace.
5. Select the **Create new** link in the *Log Analytics workspace* field and enter the following values.

Setting	Value
Name	Enter my-container-apps-logs .

The *Location* field is pre-filled with *Central US* for you.

6. Select **OK**.

 **Note**

You can use an existing virtual network, but a dedicated subnet with a CIDR range of **/23** or larger is required for use with Container Apps when using the Consumption only Architecture. When using the Workload Profiles Architecture, a **/27** or larger is required. To learn more about subnet sizing, see the [networking architecture overview](#).

7. Select the **Networking** tab to create a VNET.
8. Select **Yes** next to *Use your own virtual network*.
9. Next to the *Virtual network* box, select the **Create new** link and enter the following value.

Setting	Value
Name	Enter my-custom-vnet .

10. Select the **OK** button.
11. Next to the *Infrastructure subnet* box, select the **Create new** link and enter the following values:

Setting	Value
Subnet Name	Enter infrastructure-subnet .
Virtual Network Address Block	Keep the default values.
Subnet Address Block	Keep the default values.

12. Select the **OK** button.
13. Under *Virtual IP*, select **External**.
14. Select **Create**.

Deploy the container app

1. Select the **Review and create** button at the bottom of the page.

Next, the settings in the Container App are verified. If no errors are found, the **Create** button is enabled.

If there are errors, any tab containing errors is marked with a red dot. Navigate to the appropriate tab. Fields containing an error will be highlighted in red. Once all errors are fixed, select **Review and create** again.

2. Select **Create**.

A page with the message *Deployment is in progress* is displayed. Once the deployment is successfully completed, you'll see the message: *Your deployment is complete*.

Clean up resources

If you're not going to continue to use this application, you can delete the Azure Container Apps instance and all the associated services by removing the **my-container-apps** resource group. Deleting this resource group will also delete the resource group automatically created by the Container Apps service containing the custom network components.

Next steps

[Managing autoscaling behavior](#)

Health probes in Azure Container Apps

Article • 04/06/2023

Health probes in Azure Container Apps are based on [Kubernetes health probes](#). You can set up probes using either TCP or HTTP(S) exclusively.

Container Apps support the following probes:

- **Liveness**: Reports the overall health of your replica.
- **Readiness**: Signals that a replica is ready to accept traffic.
- **Startup**: Delay reporting on a liveness or readiness state for slower apps with a startup probe.

For a full listing of the specification supported in Azure Container Apps, refer to [Azure REST API specs](#).

ⓘ Note

TCP startup probes are not supported for Consumption workload profiles in the [Consumption + Dedicated plan structure](#).

HTTP probes

HTTP probes allow you to implement custom logic to check the status of application dependencies before reporting a healthy status. Configure your health probe endpoints to respond with an HTTP status code greater than or equal to `200` and less than `400` to indicate success. Any other response code outside this range indicates a failure.

The following example demonstrates how to implement a liveness endpoint in JavaScript.

JavaScript

```
const express = require('express');
const app = express();

app.get('/liveness', (req, res) => {
  let isSystemStable = false;

  // check for database availability
  // check filesystem structure
  // etc.
```

```
// set isSystemStable to true if all checks pass

if (isSystemStable) {
    res.status(200); // Success
} else {
    res.status(503); // Service unavailable
}
})
```

TCP probes

TCP probes wait for a connection to be established with the server to indicate success. A probe failure is registered if no connection is made.

Restrictions

- You can only add one of each probe type per container.
- `exec` probes aren't supported.
- Port values must be integers; named ports aren't supported.
- gRPC is not supported.

Examples

The following code listing shows how you can define health probes for your containers.

The `...` placeholders denote omitted code. Refer to [Container Apps ARM template API specification](#) for full ARM template details.

ARM template

JSON

```
{
  ...
  "containers": [
    {
      "image": "nginx",
      "name": "web",
      "probes": [
        {
          "type": "liveness",
          "httpGet": {
            "path": "/health",
            "port": 8080,
            "httpHeaders": [
              ...
            ]
          }
        }
      ]
    }
  ]
}
```

```

        {
          "name": "Custom-Header",
          "value": "liveness probe"
        }]
      },
      "initialDelaySeconds": 7,
      "periodSeconds": 3
    },
    {
      "type": "readiness",
      "tcpSocket": {
        "port": 8081
      },
      "initialDelaySeconds": 10,
      "periodSeconds": 3
    },
    {
      "type": "startup",
      "httpGet": {
        "path": "/startup",
        "port": 8080,
        "httpHeaders": [
          {
            "name": "Custom-Header",
            "value": "startup probe"
          }
        ],
        "initialDelaySeconds": 3,
        "periodSeconds": 3
      }
    }
  ]
...
}

```

The optional `failureThreshold` setting defines the number of attempts Container Apps tries if the probe if execution fails. Attempts that exceed the `failureThreshold` amount cause different results for each probe.

Default configuration

If ingress is enabled, the following default probes are automatically added to the main app container if none is defined for each type.

Probe type	Default values

Probe type	Default values
Startup	Protocol: TCP Port: ingress target port Timeout: 1 second Period: 1 second Initial delay: 1 second Success threshold: 1 Failure threshold: <code>timeoutSeconds</code>
Readiness	Protocol: TCP Port: ingress target port Timeout: 5 seconds Period: 5 seconds Initial delay: 3 seconds Success threshold: 1 Failure threshold: <code>timeoutSeconds / 5</code>
Liveness	Protocol: TCP Port: ingress target port

If your app takes an extended amount of time to start, which is very common in Java, you often need to customize the probes so your container won't crash.

The following example demonstrates how to configure the liveness and readiness probes in order to extend the startup times.

JSON

```

"probes": [
  {
    "type": "liveness",
    "failureThreshold": 3,
    "periodSeconds": 10,
    "successThreshold": 1,
    "tcpSocket": {
      "port": 80
    },
    "timeoutSeconds": 1
  },
  {
    "type": "readiness",
    "failureThreshold": 48,
    "initialDelaySeconds": 3,
    "periodSeconds": 5,
    "successThreshold": 1,
    "tcpSocket": {
      "port": 80
    },
    "timeoutSeconds": 5
  }
]

```

Next steps

[Application logging](#)

Enable authentication and authorization in Azure Container Apps with Azure Active Directory

Article • 11/10/2022

This article shows you how to configure authentication for Azure Container Apps so that your app signs in users with the [Microsoft identity platform](#) (Azure AD) as the authentication provider.

The Container Apps Authentication feature can automatically create an app registration with the Microsoft identity platform. You can also use a registration that you or a directory admin creates separately.

- [Create a new app registration automatically](#)
- [Use an existing registration created separately](#)

Option 1: Create a new app registration automatically

This option is designed to make enabling authentication simple and requires just a few steps.

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Select **Add identity provider**.
3. Select **Microsoft** in the identity provider dropdown. The option to create a new registration is selected by default. You can change the name of the registration or the supported account types.

A client secret will be created and stored as a [secret](#) in the container app.

4. If you're configuring the first identity provider for this application, you'll also be prompted with a **Container Apps authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests, and the default selections redirect all requests to sign in with this new provider. You can customize this behavior now or adjust these settings later from

the main **Authentication** screen by choosing **Edit** next to **Authentication settings**.

To learn more about these options, see [Authentication flow](#).

5. (Optional) Select **Next: Permissions** and add any scopes needed by the application. These will be added to the app registration, but you can also change them later.

6. Select **Add**.

You're now ready to use the Microsoft identity platform for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Option 2: Use an existing registration created separately

You can also manually register your application for the Microsoft identity platform, customizing the registration and configuring Container Apps Authentication with the registration details. This approach is useful if you want to use an app registration from a different Azure AD tenant other than the one your application is defined.

Create an app registration in Azure AD for your container app

First, you'll create your app registration. As you do so, collect the following information that you'll need later when you configure the authentication in the container app:

- Client ID
- Tenant ID
- Client secret (optional)
- Application ID URI

To register the app, perform the following steps:

1. Sign in to the [Azure portal](#), search for and select **Container Apps**, and then select your app. Note your app's **URL**. You'll use it to configure your Azure Active Directory app registration.
2. From the portal menu, select **Azure Active Directory**, then go to the **App registrations** tab and select **New registration**.
3. In the **Register an application** page, enter a **Name** for your app registration.

4. In **Redirect URI**, select **Web** and type <app-url>/.auth/login/aad/callback. For example, https://<hostname>.azurecontainerapps.io/.auth/login/aad/callback.
5. Select **Register**.
6. After the app registration is created, copy the **Application (client) ID** and the **Directory (tenant) ID** for later.
7. Select **Authentication**. Under **Implicit grant and hybrid flows**, enable **ID tokens** to allow OpenID Connect user sign-ins from Container Apps. Select **Save**.
8. (Optional) Select **Branding**. In **Home page URL**, enter the URL of your container app and select **Save**.
9. Select **Expose an API**, and select **Set** next to **Application ID URI**. This value uniquely identifies the application when it's used as a resource, allowing tokens to be requested that grant access. The value is also used as a prefix for scopes you create.

For a single-tenant app, you can use the default value, which is in the form api://<application-client-id>. You can also specify a more readable URI like https://contoso.com/api based on one of the verified domains for your tenant. For a multi-tenant app, you must provide a custom URI. To learn more about accepted formats for App ID URIs, see the [app registrations best practices reference](#).
- The value is automatically saved.
10. Select **Add a scope**.
 - a. In **Add a scope**, the **Application ID URI** is the value you set in a previous step. Select **Save and continue**.
 - b. In **Scope name**, enter *user_impersonation*.
 - c. In the text boxes, enter the consent scope name and description you want users to see on the consent page. For example, enter *Access <application-name>*.
 - d. Select **Add scope**.
11. (Optional) To create a client secret, select **Certificates & secrets > Client secrets > New client secret**. Enter a description and expiration and select **Add**. Copy the client secret value shown in the page. It won't be shown again.
12. (Optional) To add multiple **Reply URLs**, select **Authentication**.

Enable Azure Active Directory in your container app

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Select **Add identity provider**.
3. Select **Microsoft** in the identity provider dropdown.
4. For **App registration type**, you can choose to **Pick an existing app registration in this directory** which will automatically gather the necessary app information. If your registration is from another tenant or you don't have permission to view the registration object, choose **Provide the details of an existing app registration**. For this option, you'll need to fill in the following configuration details:

Field	Description
Application (client) ID	Use the Application (client) ID of the app registration.
Client Secret	Use the client secret you generated in the app registration. With a client secret, hybrid flow is used and the Container Apps will return access and refresh tokens. When the client secret isn't set, implicit flow is used and only an ID token is returned. These tokens are sent by the provider and stored in the EasyAuth token store.
Issuer Url	Use <code><authentication-endpoint>/<TENANT-ID>/v2.0</code> , and replace <code><authentication-endpoint></code> with the authentication endpoint for your cloud environment (for example, " https://login.microsoftonline.com " for global Azure), also replacing <code><TENANT-ID></code> with the Directory (tenant) ID in which the app registration was created. This value is used to redirect users to the correct Azure AD tenant, and to download the appropriate metadata to determine the appropriate token signing keys and token issuer claim value for example. For applications that use Azure AD v1, omit <code>/v2.0</code> in the URL.
Allowed Token Audiences	The configured Application (client) ID is <i>always</i> implicitly considered to be an allowed audience. If this value refers to a cloud or server app and you want to accept authentication tokens from a client container app (the authentication token can be retrieved in the <code>X-MS-TOKEN-AAD-ID-TOKEN</code> header), add the Application (client) ID of the client app here.

The client secret will be stored as [secrets](#) in your container app.

5. If this is the first identity provider configured for the application, you'll also be prompted with a **Container Apps authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests, and the default selections will redirect all requests to sign in with this new provider. You can change customize this behavior now or adjust these settings

later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

6. Select **Add**.

You're now ready to use the Microsoft identity platform for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Configure client apps to access your container app

In the prior section, you registered your container app to authenticate users. This section explains how to register native client or daemon apps so that they can request access to APIs exposed by your container app on behalf of users or themselves. Completing the steps in this section isn't required if you only wish to authenticate users.

Native client application

You can register native clients to request access your container app's APIs on behalf of a signed in user.

1. In the [Azure portal](#), select **Active Directory** > **App registrations** > **New registration**.
2. In the **Register an application** page, enter a **Name** for your app registration.
3. In **Redirect URI**, select **Public client (mobile & desktop)** and type the URL `<app-url>/.auth/login/aad/callback`. For example,
`https://<hostname>.azurecontainerapps.io/.auth/login/aad/callback`.

Note

For a Microsoft Store application, use the **package SID** as the URI instead.

4. Select **Create**.
5. After the app registration is created, copy the value of **Application (client) ID**.
6. Select **API permissions** > **Add a permission** > **My APIs**.

7. Select the app registration you created earlier for your container app. If you don't see the app registration, make sure that you've added the `user_impersonation` scope in [Create an app registration in Azure AD for your container app](#).
8. Under **Delegated permissions**, select `user_impersonation`, and then select **Add permissions**.

You've now configured a native client application that can request access your container app on behalf of a user.

Daemon client application (service-to-service calls)

Your application can acquire a token to call a Web API hosted in your container app on behalf of itself (not on behalf of a user). This scenario is useful for non-interactive daemon applications that perform tasks without a logged in user. It uses the standard OAuth 2.0 [client credentials](#) grant.

1. In the [Azure portal](#), select Active Directory > App registrations > New registration.
2. In the Register an application page, enter a Name for your daemon app registration.
3. For a daemon application, you don't need a Redirect URI so you can keep that empty.
4. Select **Create**.
5. After the app registration is created, copy the value of **Application (client) ID**.
6. Select Certificates & secrets > New client secret > Add. Copy the client secret value shown in the page. It won't be shown again.

You can now [request an access token using the client ID and client secret](#) by setting the `resource` parameter to the **Application ID URI** of the target app. The resulting access token can then be presented to the target app using the standard [OAuth 2.0 Authorization header](#), and Container Apps Authentication / Authorization will validate and use the token as usual to now indicate that the caller (an application in this case, not a user) is authenticated.

This process allows *any* client application in your Azure AD tenant to request an access token and authenticate to the target app. If you also want to enforce *authorization* to allow only certain client applications, you must adjust the configuration.

1. [Define an App Role](#) in the manifest of the app registration representing the container app you want to protect.
2. On the app registration representing the client that needs to be authorized, select API permissions > Add a permission > My APIs.

3. Select the app registration you created earlier. If you don't see the app registration, make sure that you've [added an App Role](#).
4. Under **Application permissions**, select the App Role you created earlier, and then select **Add permissions**.
5. Make sure to select **Grant admin consent** to authorize the client application to request the permission.
6. Similar to the previous scenario (before any roles were added), you can now [request an access token](#) for the same target `resource`, and the access token will include a `roles` claim containing the App Roles that were authorized for the client application.
7. Within the target Container Apps code, you can now validate that the expected roles are present in the token. The validation steps aren't performed by the Container Apps auth layer. For more information, see [Access user claims](#).

You've now configured a daemon client application that can access your container app using its own identity.

Working with authenticated users

Use the following guides for details on working with authenticated users.

- [Customize sign-in and sign-out](#)
- [Access user claims in application code](#)

Next steps

[Authentication and authorization overview](#)

Enable authentication and authorization in Azure Container Apps with Facebook

Article • 05/24/2022

This article shows how to configure Azure Container Apps to use Facebook as an authentication provider.

To complete the procedure in this article, you need a Facebook account that has a verified email address and a mobile phone number. To create a new Facebook account, go to facebook.com.

Register your application with Facebook

1. Go to the [Facebook Developers](#) website and sign in with your Facebook account credentials.

If you don't have a Facebook for Developers account, select **Get Started** and follow the registration steps.

2. Select **My Apps > Add New App**.

3. In **Display Name** field:

- a. Type a unique name for your app.
- b. Provide your **Contact Email**.
- c. Select **Create App ID**.
- d. Complete the security check.

The developer dashboard for your new Facebook app opens.

4. Select **Dashboard > Facebook Login > Set up > Web**.

5. In the left navigation under **Facebook Login**, select **Settings**.

6. In the **Valid OAuth redirect URIs** field, enter

```
https://<hostname>.azurecontainerapps.io/.auth/login/facebook/callback.
```

Remember to use the hostname of your container app.

7. Select **Save Changes**.

8. In the left pane, select **Settings > Basic**.

9. In the App Secret field, select **Show**. Copy the values of App ID and App Secret.

You use them later to configure your container app in Azure.

ⓘ Important

The app secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

10. The Facebook account that you used to register the application is an administrator of the app. At this point, only administrators can sign in to this application.

To authenticate other Facebook accounts, select **App Review** and enable **Make <your-app-name> public** to enable the general public to access the app by using Facebook authentication.

Add Facebook information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.

2. Select **Authentication** in the menu on the left. Select **Add identity provider**.

3. Select **Facebook** in the identity provider dropdown. Paste in the App ID and App Secret values that you obtained previously.

The secret will be stored as a **secret** in your container app.

4. If you're configuring the first identity provider for this application, you'll be prompted with a **Container Apps authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests. The default selections redirect all requests to sign in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

5. (Optional) Select **Next: Scopes** and add any scopes needed by the application.

These scopes are requested when a user signs in for browser-based flows.

6. Select **Add**.

You're now ready to use Facebook for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Working with authenticated users

Use the following guides for details on working with authenticated users.

- [Customize sign-in and sign-out](#)
- [Access user claims in application code](#)

Next steps

[Authentication and authorization overview](#)

Enable authentication and authorization in Azure Container Apps with GitHub

Article • 06/23/2022

This article shows how to configure Azure Container Apps to use GitHub as an authentication provider.

To complete the procedure in this article, you need a GitHub account. To create a new GitHub account, go to [GitHub](#).

Register your application with GitHub

1. Sign in to the [Azure portal](#) and go to your application. Copy your **URL**. You'll use it to configure your GitHub app.
2. Follow the instructions for [creating an OAuth app on GitHub](#). In the **Authorization callback URL** section, enter the HTTPS URL of your app and append the path `/auth/login/github/callback`. For example,
`https://<hostname>.azurecontainerapps.io/.auth/login/github/callback`.
3. On the application page, make note of the **Client ID**, which you'll need later.
4. Under **Client Secrets**, select **Generate a new client secret**.
5. Make note of the client secret value, which you'll need later.

Important

The client secret is an important security credential. Do not share this secret with anyone or distribute it with your app.

Add GitHub information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Select **Add identity provider**.
3. Select **GitHub** in the identity provider dropdown. Paste in the `Client ID` and `Client secret` values that you obtained previously.

The secret will be stored as a secret in your container app.

4. If you're configuring the first identity provider for this application, you'll also be prompted with a **Container Apps authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests. The default selections redirect all requests to sign in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

5. Select **Add**.

You're now ready to use GitHub for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Working with authenticated users

Use the following guides for details on working with authenticated users.

- [Customize sign-in and sign-out](#)
- [Access user claims in application code](#)

Next steps

[Authentication and authorization overview](#)

Enable authentication and authorization in Azure Container Apps with Google

Article • 05/24/2022

This article shows you how to configure Azure Container Apps to use Google as an authentication provider.

To complete the following procedure, you must have a Google account that has a verified email address. To create a new Google account, go to accounts.google.com.

Register your application with Google

1. Follow the Google documentation at [Google Sign-In for server-side apps](#) to create a client ID and client secret. There's no need to make any code changes. Just use the following information:
 - For **Authorized JavaScript Origins**, use
`https://<hostname>.azurecontainerapps.io` with the name of your app in `<hostname>`.
 - For **Authorized Redirect URI**, use
`https://<hostname>.azurecontainerapps.io/.auth/login/google/callback`.
2. Copy the App ID and the App secret values.

Important

The App secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

Add Google information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Select **Add identity provider**.
3. Select **Google** in the identity provider dropdown. Paste in the App ID and App Secret values that you obtained previously.

The secret will be stored as a [secret](#) in your container app.

4. If you're configuring the first identity provider for this application, you'll also be prompted with a **Container Apps authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests. The default selections redirect all requests to sign in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

5. Select **Add**.

 **Note**

For adding scope: You can define what permissions your application has in the provider's registration portal. The app can request scopes at login time which leverage these permissions.

You're now ready to use Google for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Working with authenticated users

Use the following guides for details on working with authenticated users.

- [Customize sign-in and sign-out](#)
- [Access user claims in application code](#)

Next steps

[Authentication and authorization overview](#)

Enable authentication and authorization in Azure Container Apps with Twitter

Article • 05/24/2022

This article shows how to configure Azure Container Apps to use Twitter as an authentication provider.

To complete the procedure in this article, you need a Twitter account that has a verified email address and phone number. To create a new Twitter account, go to [twitter.com].

Register your application with Twitter

1. Sign in to the [Azure portal](#) and go to your application. Copy your **URL**. You'll use it to configure your Twitter app.
2. Go to the [Twitter Developers] website, sign in with your Twitter account credentials, and select **Create an app**.
3. Enter the **App name** and the **Application description** for your new app. Paste your application's **URL** into the **Website URL** field. In the **Callback URLs** section, enter the HTTPS URL of your container app and append the path `/auth/login/twitter/callback`. For example,
`https://<hostname>.azurecontainerapps.io/.auth/login/twitter/callback`.
4. At the bottom of the page, type at least 100 characters in **Tell us how this app will be used**, then select **Create**. Select **Create** again in the pop-up. The application details are displayed.
5. Select the **Keys and Access Tokens** tab.

Make a note of these values:

- API key
- API secret key

Important

The API secret key is an important security credential. Do not share this secret with anyone or distribute it with your app.

Add Twitter information to your application

1. Sign in to the [Azure portal](#) and navigate to your app.
2. Select **Authentication** in the menu on the left. Select **Add identity provider**.
3. Select **Twitter** in the identity provider dropdown. Paste in the `API key` and `API secret key` values that you obtained previously.

The secret will be stored as `secret` in your container app.

4. If you're configuring the first identity provider for this application, you'll also be prompted with a **Container Apps authentication settings** section. Otherwise, you may move on to the next step.

These options determine how your application responds to unauthenticated requests. The default selections redirect all requests to sign in with this new provider. You can change customize this behavior now or adjust these settings later from the main **Authentication** screen by choosing **Edit** next to **Authentication settings**. To learn more about these options, see [Authentication flow](#).

5. Select **Add**.

You're now ready to use Twitter for authentication in your app. The provider will be listed on the **Authentication** screen. From there, you can edit or delete this provider configuration.

Working with authenticated users

Use the following guides for details on working with authenticated users.

- [Customize sign-in and sign-out](#)
- [Access user claims in application code](#)

Next steps

[Authentication and authorization overview](#)

Enable authentication and authorization in Azure Container Apps with a Custom OpenID Connect provider

Article • 05/24/2022

This article shows you how to configure Azure Container Apps to use a custom authentication provider that adheres to the [OpenID Connect specification](#). OpenID Connect (OIDC) is an industry standard used by many identity providers (IDPs). You don't need to understand the details of the specification in order to configure your app to use an adherent IDP.

You can configure your app to use one or more OIDC providers. Each must be given a unique alphanumeric name in the configuration, and only one can serve as the default redirect target.

Register your application with the identity provider

Your provider will require you to register the details of your application with it. One of these steps involves specifying a redirect URI. This redirect URI will be of the form `<app-url>/auth/login/<provider-name>/callback`. Each identity provider should provide more instructions on how to complete these steps.

Note

Some providers may require additional steps for their configuration and how to use the values they provide. For example, Apple provides a private key which is not itself used as the OIDC client secret, and you instead must use it to craft a JWT which is treated as the secret you provide in your app config (see the "Creating the Client Secret" section of the [Sign in with Apple documentation](#))

You'll need to collect a **client ID** and **client secret** for your application.

Important

The client secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

Additionally, you'll need the OpenID Connect metadata for the provider. This information is often exposed via a [configuration metadata document](#), which is the provider's Issuer URL suffixed with `/well-known/openid-configuration`. Gather this configuration URL.

If you're unable to use a configuration metadata document, you'll need to gather the following values separately:

- The issuer URL (sometimes shown as `issuer`)
- The [OAuth 2.0 Authorization endpoint](#) (sometimes shown as `authorization_endpoint`)
- The [OAuth 2.0 Token endpoint](#) (sometimes shown as `token_endpoint`)
- The URL of the [OAuth 2.0 JSON Web Key Set](#) document (sometimes shown as `jwks_uri`)

Add provider information to your application

1. Sign in to the [Azure portal] and navigate to your app.
2. Select **Authentication** in the menu on the left. Select **Add identity provider**.
3. Select **OpenID Connect** in the identity provider dropdown.
4. Provide the unique alphanumeric name selected earlier for **OpenID provider name**.
5. If you have the URL for the **metadata document** from the identity provider, provide that value for **Metadata URL**. Otherwise, select the **Provide endpoints separately** option and put each URL gathered from the identity provider in the appropriate field.
6. Provide the earlier collected **Client ID** and **Client Secret** in the appropriate fields.
7. Specify an application setting name for your client secret. Your client secret will be stored as a `secret` in your container app.
8. Press the **Add** button to finish setting up the identity provider.

Working with authenticated users

Use the following guides for details on working with authenticated users.

- [Customize sign-in and sign-out](#)

- Access user claims in application code

Next steps

[Authentication and authorization overview](#)

Connect to Azure services via Dapr components in the Azure portal

Article • 05/24/2023

Using a combination of [Service Connector](#) and [Dapr](#), you can author Dapr components via an improved component creation feature in the Azure Container Apps portal.

With this new component creation feature, you no longer need to know or remember the Dapr open source metadata concepts. Instead, entering the component information in the easy component creation pane automatically maps your entries to the required component metadata.

By managing component creation for you, this feature:

- Simplifies the process for developers
- Reduces the likelihood for misconfiguration

This experience makes authentication easier. When using Managed Identity, Azure Container Apps, Dapr, and Service Connector ensure the selected identification is assigned to all containers apps in scope and target services.

This guide demonstrates creating a Dapr component by:

- Selecting pub/sub as component type
- Specifying Azure Service Bus as the component
- Providing required metadata to help the tool map to the right Azure Service Bus
- Providing optional metadata to customize the component

Prerequisites

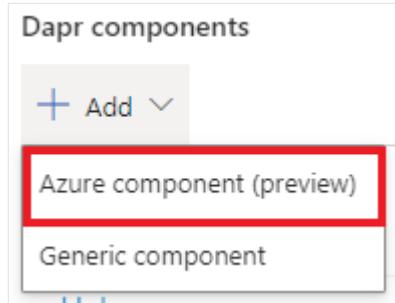
- An Azure account with an active subscription. [Create a free Azure account](#).
- [An existing Azure Container App](#).

Create a Dapr component

Start by navigating to the Dapr component creation feature.

1. In the Azure portal, navigate to your Container Apps environment.
2. In the left-side menu, under **Settings**, select **Dapr components**.

- From the top menu, select **Add > Azure component (preview)** to open the **Add Dapr Component** configuration pane.



! Note

Currently, creating Dapr components using Service Connector in the Azure portal only works with Azure services (Azure Service Bus, Azure Cosmos DB, etc.). To create non-Azure Dapr components (Redis), use the manual component creation option.

Provide required metadata

For the component creation tool to map to the required component metadata, you need to provide the required metadata from predefined dropdowns in the **Basics** tab.

For example, for a pub/sub Azure Service Bus component, you'll start with the following fields:

Field	Example	Description
Component name	mycomponent	Enter a name for your Dapr component. The name must match the component referenced in your application code.
Building block	Pub/sub	Select the building block/API for your component from the drop-down.
Component type	Service Bus	Select a component type from the drop-down.

The component creation pane populates with different fields depending on the building block and component type you select. For example, the following table and image demonstrate the fields associated with an Azure Service Bus pub/sub component type, but the fields you see may vary.

Field	Example	Description
-------	---------	-------------

Field	Example	Description
Subscription	My subscription	Select your Azure subscription
Namespace	mynamespace	Select the Service Bus namespace
Authentication	User assigned managed identity	Select the subscription that contains the component you're looking for. Recommended: User assigned managed identity.
User assigned managed identity	testidentity	Select an existing identity from the drop-down. If you don't already have one, you can create a new managed identity client ID.

Add Dapr Component

X

With Service Connector

Basics Metadata + Scopes Review + Create

Dapr component details

Component name * ⓘ

mycomponent

Building Block * ⓘ

Pubsub

Select a component type * ⓘ

Service Bus

Subscription * ⓘ

My subscription

Namespace * ⓘ

mynamespace

[Create new](#)

Authentication * ⓘ

User assigned managed identity (Recommended)

User assigned managed identity *

testidentity

[Create new](#)

[Next : Metadata & Scopes](#)

[Cancel](#)

What happened?

Now that you've filled out these required fields, they'll automatically map to the required component metadata. In this Service Bus example, the only required metadata is the connection string. The component creation tool takes the information you provided and maps the input to create a connection string in the component YAML file.

Provide optional metadata

While the component creation tool automatically populates all required metadata for the component, you can also customize the component by adding optional metadata.

1. Select **Next : Metadata + Scopes**.
2. Under **Metadata**, select **Add** to select extra, optional metadata for your Dapr component from a drop-down of supported fields.
3. Under **Scopes**, select **Add** or type in the app IDs for the container apps that you want to load this component.
 - By default, when the scope is unspecified, Dapr applies the component to all app IDs.
4. Select **Review + Create** to review the component values.
5. Select **Create**.

Save the component YAML

Once the component has been added to the Container Apps environment, the portal displays the YAML (or Bicep) for the component.

1. Copy and save the YAML file for future use.
2. Select **Done** to exit the configuration pane.

You can then check the YAML/Bicep artifact into a repo and recreate it outside of the portal experience.

Manage Dapr components

1. In your Container Apps environment, go to **Settings > Dapr components**.
2. The Dapr components that are tied to your Container Apps environment are listed on this page. Review the list and select the **Delete** icon to delete a component, or select a component's name to review or edit its details.

The screenshot shows the 'Dapr components' section of the Azure Container Apps settings. On the left, a sidebar menu includes 'Settings', 'Dapr components' (which is selected and highlighted in grey), 'Certificates', 'Custom DNS suffix (Preview)', 'Azure Files', and 'Locks'. The main area is titled 'Dapr components' and contains a table with one row. The table has columns for 'Name ↑', 'Type ↑', and 'Delete'. The single row shows 'mycomponent' in the 'Name' column and 'bindings.azure.servicebusqueues' in the 'Type' column. A blue 'Delete' button is visible in the 'Delete' column of the same row.

Next steps

Learn more about:

- [Using Dapr with Azure Container Apps](#)
- [Connecting to cloud services using Service Connector](#)

Enable token authentication for Dapr requests

Article • 05/23/2023

When [Dapr](#) is enabled for your application in Azure Container Apps, it injects the environment variable `APP_API_TOKEN` into your app's container. Dapr includes the same token in all requests sent to your app, as either:

- An HTTP header (`dapr-api-token`)
- A gRPC metadata option (`dapr-api-token[0]`)

The token is randomly generated and unique per each app and app revision. It can also change at any time. Your application should read the token from the `APP_API_TOKEN` environment variable when it starts up to ensure that it's using the correct token.

You can use this token to authenticate that calls coming into your application are actually coming from the Dapr sidecar, even when listening on public endpoints.

1. The `daprd` container reads and injects it into each call made from Dapr to your application.
2. Your application can then use that token to validate that the request is coming from Dapr.

Prerequisites

[Dapr-enabled Azure Container App](#)

Authenticate requests from Dapr

With Dapr SDKs

If you're using a [Dapr SDK](#), the Dapr SDKs automatically validates the token in all incoming requests from Dapr, rejecting calls that don't include the correct token. You don't need to perform any other action.

Incoming requests that don't include the token, or include an incorrect token, are rejected automatically.

Next steps

[Learn more about the Dapr integration with Azure Container Apps.](#)

Use storage mounts in Azure Container Apps

Article • 03/19/2023

A container app has access to different types of storage. A single app can take advantage of more than one type of storage if necessary.

Storage type	Description	Usage examples
Container file system	Temporary storage scoped to the local container	Writing a local app cache.
Ephemeral storage	Temporary storage scoped to an individual replica	Sharing files between containers in a replica. For instance, the main app container can write log files that are processed by a sidecar container.
Azure Files	Permanent storage	Writing files to a file share to make data accessible by other systems.

Container file system

A container can write to its own file system.

Container file system storage has the following characteristics:

- The storage is temporary and disappears when the container is shut down or restarted.
- Files written to this storage are only visible to processes running in the current container.
- There are no capacity guarantees. The available storage depends on the amount of disk space available in the container.

Ephemeral volume

You can mount an ephemeral, temporary volume that is equivalent to [emptyDir](#) in Kubernetes. Ephemeral storage is scoped to a single replica.

Ephemeral storage has the following characteristics:

- Files are persisted for the lifetime of the replica.

- If a container in a replica restarts, the files in the volume remain.
- Any containers in the replica can mount the same volume.
- A container can mount multiple ephemeral volumes.
- The available storage depends on the total amount of vCPUs allocated to the replica.

vCPUs	Ephemeral storage
0.25 or lower	1 GiB
0.5 or lower	2 GiB
1 or lower	4 GiB
Over 1	8 GiB

To configure ephemeral storage, first define an `EmptyDir` volume in the revision. Then define a volume mount in one or more containers in the revision.

Prerequisites

Requirement	Instructions
Azure account	If you don't have one, create an account for free .
Azure Container Apps environment	Create a container apps environment .

Configuration

When configuring ephemeral storage using the Azure CLI, you must use a YAML definition to create or update your container app.

1. To update an existing container app to use ephemeral storage, export your app's specification to a YAML file named *app.yaml*.

```
azure-cli
az containerapp show -n <APP_NAME> -g <RESOURCE_GROUP_NAME> -o yaml >
app.yaml
```

2. Make the following changes to your container app specification.

- Add a `volumes` array to the `template` section of your container app definition and define a volume. If you already have a `volumes` array, add a new volume to the array.
 - The `name` is an identifier for the volume.
 - Use `EmptyDir` as the `storageType`.
- For each container in the template that you want to mount the ephemeral volume, define a volume mount in the `volumeMounts` array of the container definition.
 - The `volumeName` is the name defined in the `volumes` array.
 - The `mountPath` is the path in the container to mount the volume.

YAML

```
properties:
  managedEnvironmentId:
    /subscriptions/<SUBSCRIPTION_ID>/resourceGroups/<RESOURCE_GROUP_NAME>/providers/Microsoft.App/managedEnvironments/<ENVIRONMENT_NAME>
  configuration:
    activeRevisionsMode: Single
  template:
    containers:
      - image: <IMAGE_NAME>
        name: my-container
        volumeMounts:
          - mountPath: /myempty
            volumeName: myempty
    volumes:
      - name: myempty
        storageType: EmptyDir
```

3. Update your container app using the YAML file.

azure-cli

```
az containerapp update --name <APP_NAME> --resource-group
<RESOURCE_GROUP_NAME> \
--yaml app.yaml
```

See the [YAML specification](#) for a full example.

Azure Files volume

You can mount a file share from [Azure Files](#) as a volume in a container.

Azure Files storage has the following characteristics:

- Files written under the mount location are persisted to the file share.
- Files in the share are available via the mount location.
- Multiple containers can mount the same file share, including ones that are in another replica, revision, or container app.
- All containers that mount the share can access files written by any other container or method.
- More than one Azure Files volume can be mounted in a single container.

To enable Azure Files storage in your container, you need to set up your container in the following ways:

- Create a storage definition in the Container Apps environment.
- Define a volume of type `AzureFile` in a revision.
- Define a volume mount in one or more containers in the revision.

Prerequisites

Requirement	Instructions
Azure account	If you don't have one, create an account for free .
Azure Storage account	Create a storage account .
Azure Container Apps environment	Create a container apps environment .

Configuration

When configuring a container app to mount an Azure Files volume using the Azure CLI, you must use a YAML definition to create or update your container app.

For a step-by-step tutorial, refer to [Create an Azure Files storage mount in Azure Container Apps](#).

1. Add a storage definition to your Container Apps environment.

```
azure-cli
az containerapp env storage set --name my-env --resource-group my-group \
  --storage-name mystorage \
  --azure-file-account-name <STORAGE_ACCOUNT_NAME> \
  --azure-file-account-key <STORAGE_ACCOUNT_KEY> \
  --azure-file-share-name <STORAGE_SHARE_NAME> \
  --access-mode ReadWrite
```

Replace <STORAGE_ACCOUNT_NAME> and <STORAGE_ACCOUNT_KEY> with the name and key of your storage account. Replace <STORAGE_SHARE_NAME> with the name of the file share in the storage account.

Valid values for --access-mode are ReadWrite and ReadOnly.

2. To update an existing container app to mount a file share, export your app's specification to a YAML file named *app.yaml*.

```
azure-cli
```

```
az containerapp show -n <APP_NAME> -g <RESOURCE_GROUP_NAME> -o yaml > app.yaml
```

3. Make the following changes to your container app specification.

- Add a volumes array to the template section of your container app definition and define a volume. If you already have a volumes array, add a new volume to the array.
 - The name is an identifier for the volume.
 - For storageType, use AzureFile.
 - For storageName, use the name of the storage you defined in the environment.
- For each container in the template that you want to mount Azure Files storage, define a volume mount in the volumeMounts array of the container definition.
 - The volumeName is the name defined in the volumes array.
 - The mountPath is the path in the container to mount the volume.

```
YAML
```

```
properties:
  managedEnvironmentId:
    /subscriptions/<SUBSCRIPTION_ID>/resourceGroups/<RESOURCE_GROUP_NAME>/providers/Microsoft.App/managedEnvironments/<ENVIRONMENT_NAME>
  configuration:
  template:
    containers:
      - image: <IMAGE_NAME>
        name: my-container
        volumeMounts:
          - volumeName: azure-files-volume
            mountPath: /my-files
    volumes:
      - name: azure-files-volume
```

```
storageType: AzureFile  
storageName: mystorage
```

4. Update your container app using the YAML file.

```
azure-cli
```

```
az containerapp update --name <APP_NAME> --resource-group  
<RESOURCE_GROUP_NAME> \  
--yaml app.yaml
```

See the [YAML specification](#) for a full example.

Disaster recovery guidance for Azure Container Apps

Article • 05/03/2023

Azure Container Apps uses [availability zones](#) in regions where they're available to provide high-availability protection for your applications and data from data center failures.

Availability zones are unique physical locations within an Azure region. Each zone is made up of one or more data centers equipped with independent power, cooling, and networking. To ensure resiliency, there's a minimum of three separate zones in all enabled regions. You can build high availability into your application architecture by co-locating your compute, storage, networking, and data resources within a zone and replicating in other zones.

By enabling Container Apps' zone redundancy feature, replicas are automatically distributed across the zones in the region. Traffic is load balanced among the replicas. If a zone outage occurs, traffic will automatically be routed to the replicas in the remaining zones.

ⓘ Note

There is no extra charge for enabling zone redundancy, but it only provides benefits when you have 2 or more replicas, with 3 or more being ideal since most regions that support zone redundancy have 3 zones.

In the unlikely event of a full region outage, you have the option of using one of two strategies:

- **Manual recovery:** Manually deploy to a new region, or wait for the region to recover, and then manually redeploy all environments and apps.
- **Resilient recovery:** First, deploy your container apps in advance to multiple regions. Next, use Azure Front Door or Azure Traffic Manager to handle incoming requests, pointing traffic to your primary region. Then, should an outage occur, you can redirect traffic away from the affected region. For more information, see [Cross-region replication in Azure](#).

ⓘ Note

Regardless of which strategy you choose, make sure your deployment configuration files are in source control so you can easily redeploy if necessary.

Additionally, the following resources can help you create your own disaster recovery plan:

- [Failure and disaster recovery for Azure applications](#)
- [Azure resiliency technical guidance](#)

Set up zone redundancy in your Container Apps environment

To take advantage of availability zones, you must enable zone redundancy when you create the Container Apps environment. The environment must include a virtual network (VNET) with an available subnet. To ensure proper distribution of replicas, you should configure your app's minimum and maximum replica count with values that are divisible by three. The minimum replica count should be at least three.

Enable zone redundancy via the Azure portal

To create a container app in an environment with zone redundancy enabled using the Azure portal:

1. Navigate to the Azure portal.
2. Search for **Container Apps** in the top search box.
3. Select **Container Apps**.
4. Select **Create New** in the *Container Apps Environment* field to open the *Create Container Apps Environment* panel.
5. Enter the environment name.
6. Select **Enabled** for the *Zone redundancy* field.

Zone redundancy requires a virtual network (VNET) with an infrastructure subnet. You can choose an existing VNET or create a new one. When creating a new VNET, you can accept the values provided for you or customize the settings.

1. Select the **Networking** tab.
2. To assign a custom VNET name, select **Create New** in the *Virtual Network* field.
3. To assign a custom infrastructure subnet name, select **Create New** in the *Infrastructure subnet* field.
4. You can select **Internal** or **External** for the *Virtual IP*.
5. Select **Create**.

Create Container Apps Environment

Basics Monitoring **Networking**

Selecting your own virtual network allows you to connect your application to other Azure resources or on-premises systems through the same network. [Learn more](#)

Virtual network

Use your own virtual network No Yes

Virtual network *

Infrastructure subnet *

Virtual IP

Internal: The endpoint is an internal load balancer
 External: Exposes the hosted apps on an internet-accessible IP address

Create **Cancel**

Enable zone redundancy with the Azure CLI

Create a VNET and infrastructure subnet to include with the Container Apps environment.

When using these commands, replace the <PLACEHOLDERS> with your values.

ⓘ Note

The subnet associated with a Container App Environment requires a CIDR prefix of /23 or larger.

Azure CLI

```
az network vnet create \
--resource-group <RESOURCE_GROUP_NAME> \
```

```
--name <VNET_NAME> \
--location <LOCATION> \
--address-prefix 10.0.0.0/16
```

Azure CLI

```
az network vnet subnet create \
--resource-group <RESOURCE_GROUP_NAME> \
--vnet-name <VNET_NAME> \
--name infrastructure \
--address-prefixes 10.0.0.0/21
```

Next, query for the infrastructure subnet ID.

Azure CLI

Azure CLI

```
INFRASTRUCTURE_SUBNET=`az network vnet subnet show --resource-group
<RESOURCE_GROUP_NAME> --vnet-name <VNET_NAME> --name infrastructure --
query "id" -o tsv | tr -d '[:space:]'`
```

Finally, create the environment with the `--zone-redundant` parameter. The location must be the same location used when creating the VNET.

Azure CLI

Azure CLI

```
az containerapp env create \
--name <CONTAINER_APP_ENV_NAME> \
--resource-group <RESOURCE_GROUP_NAME> \
--location "<LOCATION>" \
--infrastructure-subnet-resource-id $INFRASTRUCTURE_SUBNET \
--zone-redundant
```

Tutorial: Deploy your first container app

Article • 03/31/2023

The Azure Container Apps service enables you to run microservices and containerized applications on a serverless platform. With Container Apps, you enjoy the benefits of running containers while you leave behind the concerns of manually configuring cloud infrastructure and complex container orchestrators.

In this tutorial, you create a secure Container Apps environment and deploy your first container app.

ⓘ Note

You can also deploy this app using the `az containerapp up` by following the instructions in the [Quickstart: Deploy your first container app with containerapp up](#) article. The `az containerapp up` command is a fast and convenient way to build and deploy your app to Azure Container Apps using a single command. However, it doesn't provide the same level of customization for your container app.

Prerequisites

- An Azure account with an active subscription.
 - If you don't have one, you [can create one for free ↗](#).
- Install the [Azure CLI](#).

Setup

To begin, sign in to Azure. Run the following command, and follow the prompts to complete the authentication process.

Bash

Azure CLI

```
az login
```

Bash

Next, install the Azure Container Apps extension for the CLI.

Azure CLI

```
az extension add --name containerapp --upgrade
```

Now that the current extension or module is installed, register the `Microsoft.App` namespace.

ⓘ Note

Azure Container Apps resources have migrated from the `Microsoft.Web` namespace to the `Microsoft.App` namespace. Refer to [Namespace migration from Microsoft.Web to Microsoft.App in March 2022](#) for more details.

Bash

Azure CLI

```
az provider register --namespace Microsoft.App
```

Register the `Microsoft.OperationalInsights` provider for the Azure Monitor Log Analytics workspace if you have not used it before.

Bash

Azure CLI

```
az provider register --namespace Microsoft.OperationalInsights
```

Next, set the following environment variables:

Bash

Azure CLI

```
RESOURCE_GROUP="my-container-apps"  
LOCATION="canadacentral"  
CONTAINERAPPS_ENVIRONMENT="my-environment"
```

With these variables defined, you can create a resource group to organize the services related to your new container app.

Bash

Azure CLI

```
az group create \
--name $RESOURCE_GROUP \
--location $LOCATION
```

With the CLI upgraded and a new resource group available, you can create a Container Apps environment and deploy your container app.

Create an environment

An environment in Azure Container Apps creates a secure boundary around a group of container apps. Container Apps deployed to the same environment are deployed in the same virtual network and write logs to the same Log Analytics workspace.

Bash

To create the environment, run the following command:

Azure CLI

```
az containerapp env create \
--name $CONTAINERAPPS_ENVIRONMENT \
--resource-group $RESOURCE_GROUP \
--location $LOCATION
```

Create a container app

Now that you have an environment created, you can deploy your first container app. With the `containerapp create` command, deploy a container image to Azure Container Apps.

Bash

Azure CLI

```
az containerapp create \
--name my-container-app \
--resource-group $RESOURCE_GROUP \
--environment $CONTAINERAPPS_ENVIRONMENT \
--image mcr.microsoft.com/azuredocs/containerapps-helloworld:latest \
--target-port 80 \
--ingress 'external' \
--query properties.configuration.ingress.fqdn
```

ⓘ Note

Make sure the value for the `--image` parameter is in lower case.

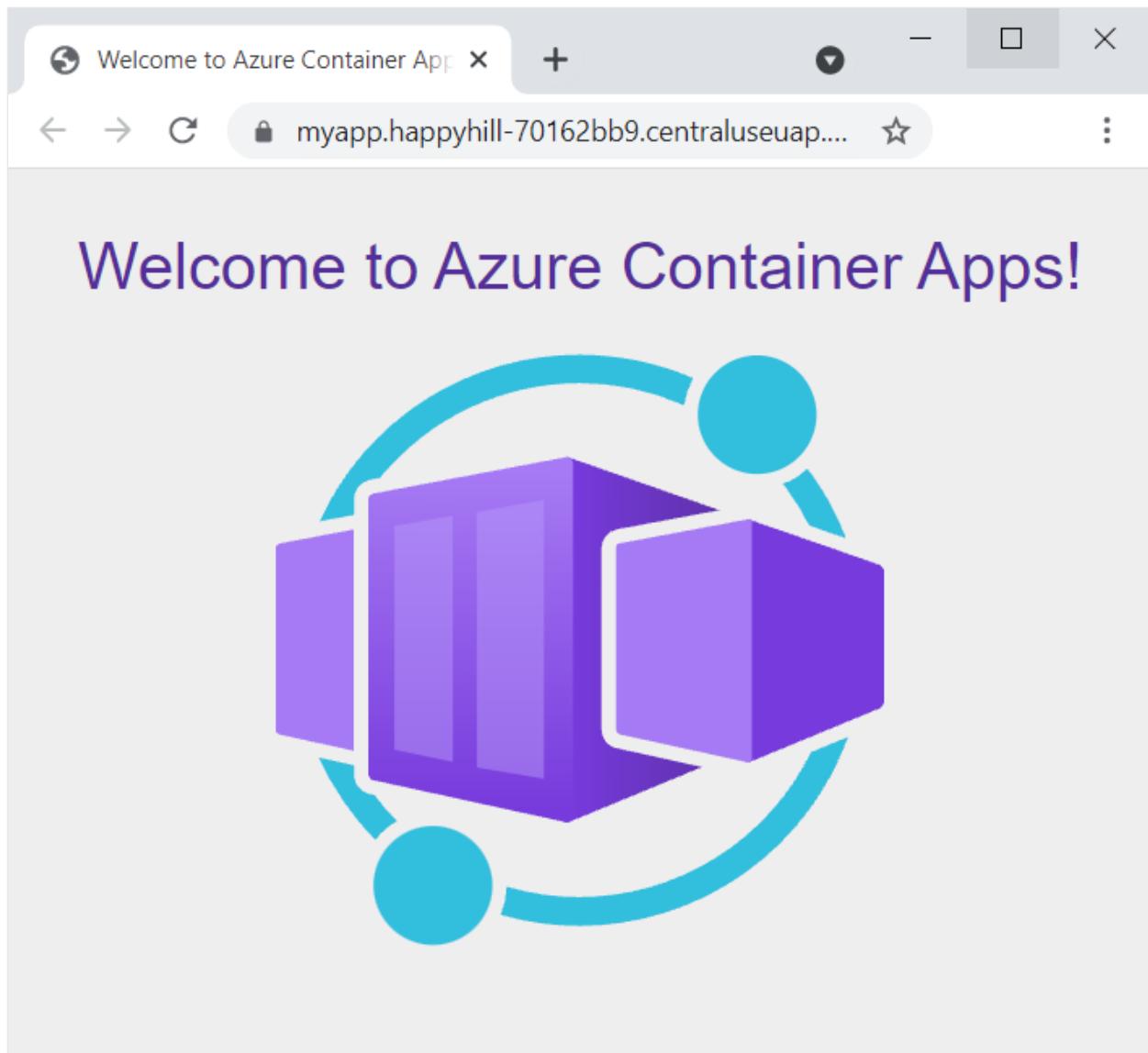
By setting `--ingress` to `external`, you make the container app available to public requests.

Verify deployment

Bash

The `create` command returns the fully qualified domain name for the container app. Copy this location to a web browser.

The following message is displayed when the container app is deployed:



Clean up resources

If you're not going to continue to use this application, run the following command to delete the resource group along with all the resources created in this tutorial.

⊗ Caution

The following command deletes the specified resource group and all resources contained within it. If resources outside the scope of this tutorial exist in the specified resource group, they will also be deleted.

Bash

Azure CLI

```
az group delete --name $RESOURCE_GROUP
```

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

[Communication between microservices](#)

Tutorial: Build and deploy your app to Azure Container Apps

Article • 03/31/2023

This article demonstrates how to build and deploy a microservice to Azure Container Apps from a source repository using the programming language of your choice.

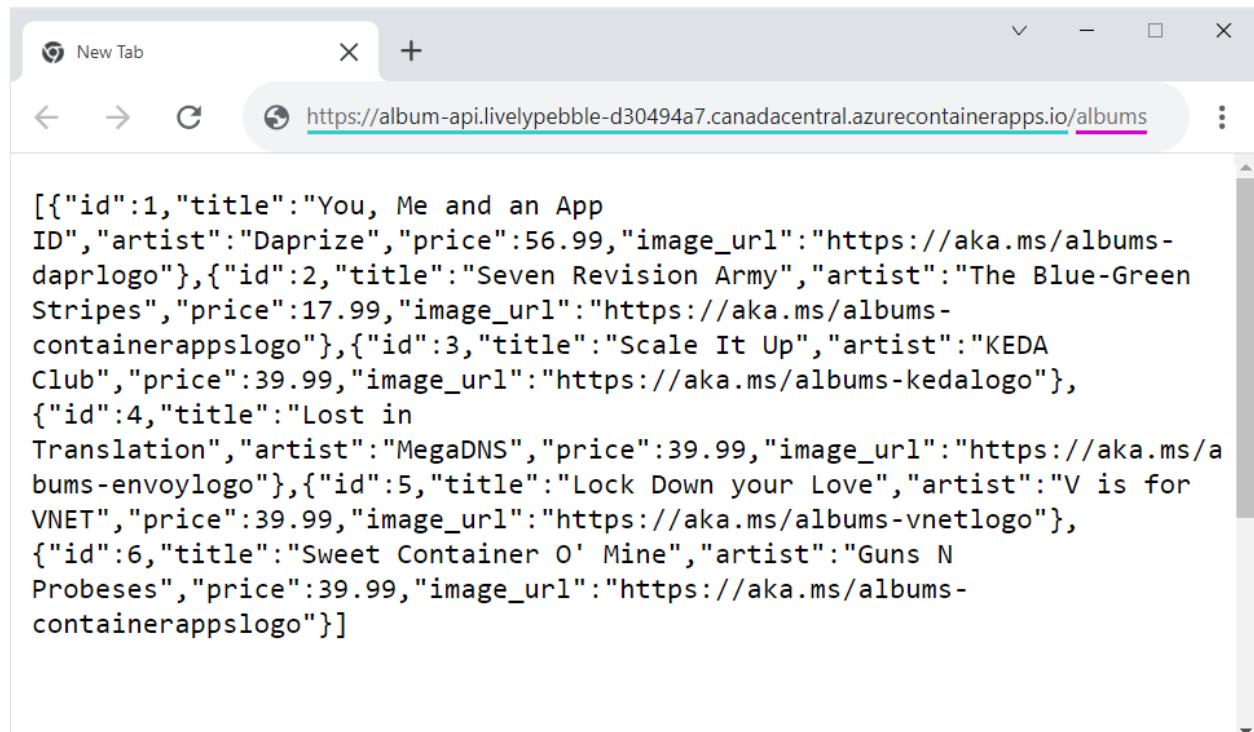
This tutorial is the first in a series of articles that walk you through how to use core capabilities within Azure Container Apps. The first step is to create a back end web API service that returns a static collection of music albums.

ⓘ Note

You can also build and deploy this app using the `az containerapp up` by following the instructions in the [Quickstart: Build and deploy an app to Azure Container Apps from a repository](#) article. The `az containerapp up` command is a fast and convenient way to build and deploy your app to Azure Container Apps using a single command. However, it doesn't provide the same level of customization for your container app.

The next tutorial in the series will build and deploy the front end web application to Azure Container Apps.

The following screenshot shows the output from the album API deployed in this tutorial.



Prerequisites

To complete this project, you need the following items:

Requirement	Instructions
Azure account	If you don't have one, create an account for free ↗ . You need the <i>Contributor</i> or <i>Owner</i> permission on the Azure subscription to proceed. Refer to Assign Azure roles using the Azure portal for details.
GitHub Account	Sign up for free ↗ .
git	Install git ↗
Azure CLI	Install the Azure CLI .

Setup

To sign in to Azure from the CLI, run the following command and follow the prompts to complete the authentication process.

```
Bash
Azure CLI
az login
```

Ensure you're running the latest version of the CLI via the upgrade command.

```
Bash
Azure CLI
az upgrade
```

```
Bash
Azure CLI
```

Next, install or update the Azure Container Apps extension for the CLI.

```
az extension add --name containerapp --upgrade
```

Register the `Microsoft.App` and `Microsoft.OperationalInsights` namespaces if you haven't already registered them in your Azure subscription.

Bash

Azure CLI

```
az provider register --namespace Microsoft.App
```

Azure CLI

```
az provider register --namespace Microsoft.OperationalInsights
```

Now that your Azure CLI setup is complete, you can define the environment variables that are used throughout this article.

Bash

Define the following variables in your bash shell.

Azure CLI

```
RESOURCE_GROUP="album-containerapps"
LOCATION="canadacentral"
ENVIRONMENT="env-album-containerapps"
API_NAME="album-api"
FRONTEND_NAME="album-ui"
GITHUB_USERNAME=<YOUR_GITHUB_USERNAME>"
```

Before you run this command, make sure to replace `<YOUR_GITHUB_USERNAME>` with your GitHub username.

Next, define a container registry name unique to you.

Azure CLI

```
ACR_NAME="acaalbums"$GITHUB_USERNAME
```

Prepare the GitHub repository

Navigate to the repository for your preferred language and fork the repository.

C#

Select the **Fork** button at the top of the [album API repo](#) to fork the repo to your account.

Now you can clone your fork of the sample repository.

Use the following git command to clone your forked repo into the *code-to-cloud* folder:

```
git
```

```
git clone https://github.com/$GITHUB_USERNAME/containerapps-albumapi-csharp.git code-to-cloud
```

Next, change the directory into the root of the cloned repo.

Console

```
cd code-to-cloud/src
```

Create an Azure resource group

Create a resource group to organize the services related to your container app deployment.

Bash

Azure CLI

```
az group create \
--name $RESOURCE_GROUP \
--location "$LOCATION"
```

Create an Azure Container Registry

Next, create an Azure Container Registry (ACR) instance in your resource group to store the album API container image once it's built.

Bash

Azure CLI

```
az acr create \
--resource-group $RESOURCE_GROUP \
--name $ACR_NAME \
--sku Basic \
--admin-enabled true
```

Build your application

With [ACR tasks](#), you can build and push the docker image for the album API without installing Docker locally.

Build the container with ACR

Run the following command to initiate the image build and push process using ACR. The `.` at the end of the command represents the docker build context, meaning this command should be run within the `src` folder where the `Dockerfile` is located.

Bash

Azure CLI

```
az acr build --registry $ACR_NAME --image $API_NAME .
```

Output from the `az acr build` command shows the upload progress of the source code to Azure and the details of the `docker build` and `docker push` operations.

Create a Container Apps environment

The Azure Container Apps environment acts as a secure boundary around a group of container apps.

Create the Container Apps environment using the following command.

Bash

Azure CLI

```
az containerapp env create \
--name $ENVIRONMENT \
--resource-group $RESOURCE_GROUP \
--location "$LOCATION"
```

Deploy your image to a container app

Now that you have an environment created, you can create and deploy your container app with the `az containerapp create` command.

Create and deploy your container app with the following command.

Bash

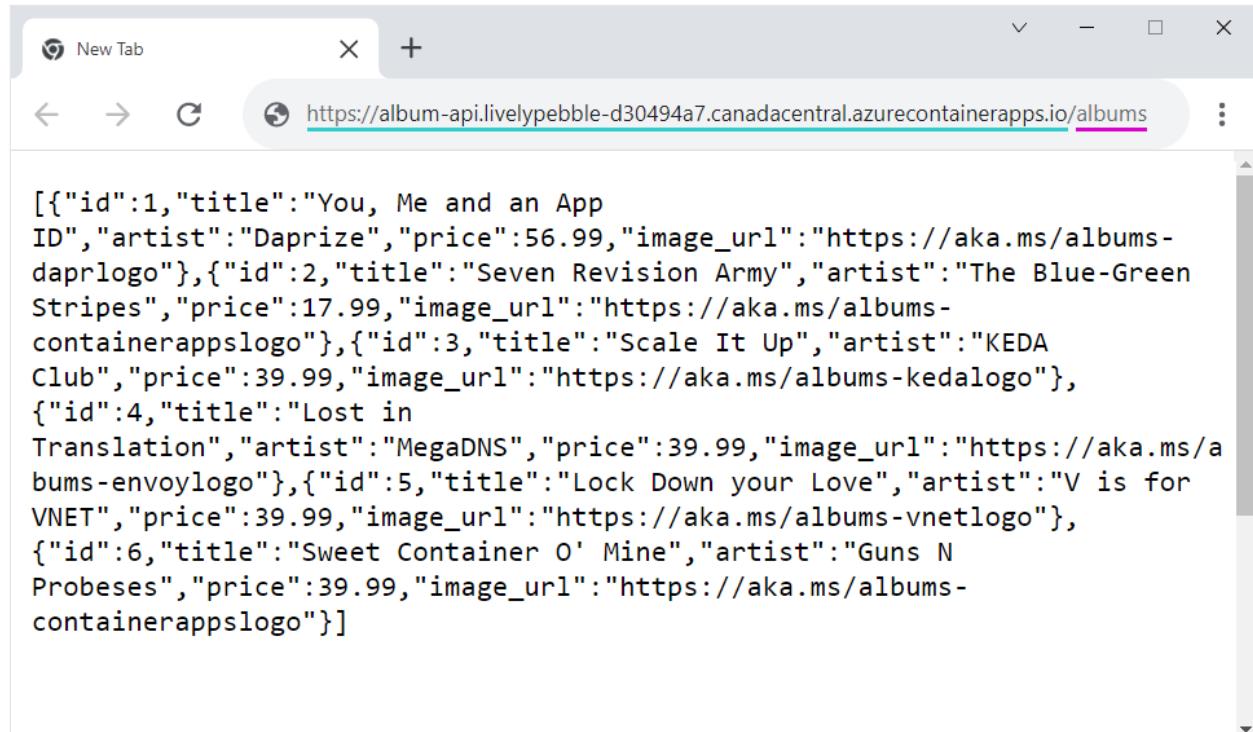
Azure CLI

```
az containerapp create \
--name $API_NAME \
--resource-group $RESOURCE_GROUP \
--environment $ENVIRONMENT \
--image $ACR_NAME.azurecr.io/$API_NAME \
--target-port 3500 \
--ingress 'external' \
--registry-server $ACR_NAME.azurecr.io \
--query properties.configuration.ingress.fqdn
```

- By setting `--ingress` to `external`, your container app is accessible from the public internet.
- The `target-port` is set to `3500` to match the port that the container is listening to for requests.
- Without a `query` property, the call to `az containerapp create` returns a JSON response that includes a rich set of details about the application. Adding a `query` parameter filters the output to just the app's fully qualified domain name (FQDN).

Verify deployment

Copy the FQDN to a web browser. From your web browser, navigate to the `/albums` endpoint of the FQDN.

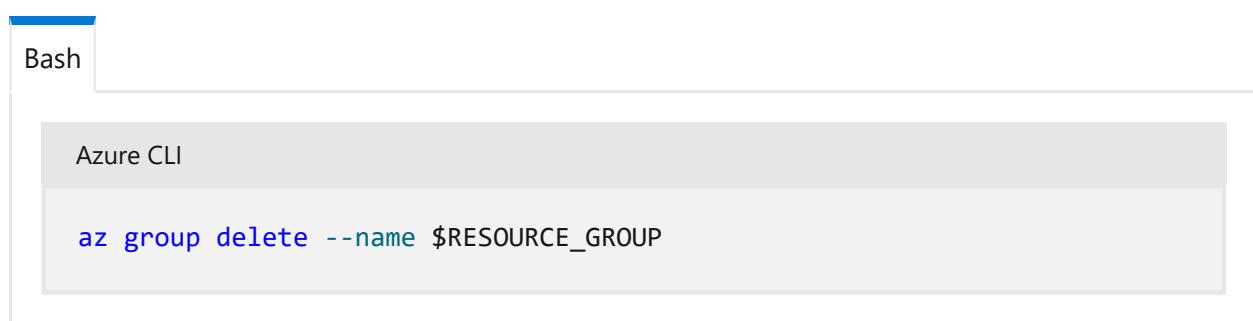


A screenshot of a Microsoft Edge browser window. The address bar shows the URL: `https://album-api.livelypebble-d30494a7.canadacentral.azurecontainerapps.io/albums`. The page content is a JSON array of album objects:

```
[{"id":1,"title":"You, Me and an App ID","artist":"Daprize","price":56.99,"image_url":"https://aka.ms/albums-daprllogo"}, {"id":2,"title":"Seven Revision Army","artist":"The Blue-Green Stripes","price":17.99,"image_url":"https://aka.ms/albums-containerappslogo"}, {"id":3,"title":"Scale It Up","artist":"KEDA Club","price":39.99,"image_url":"https://aka.ms/albums-kedalogo"}, {"id":4,"title":"Lost in Translation","artist":"MegaDNS","price":39.99,"image_url":"https://aka.ms/albums-envoylogo"}, {"id":5,"title":"Lock Down your Love","artist":"V is for VNET","price":39.99,"image_url":"https://aka.ms/albums-vnetlogo"}, {"id":6,"title":"Sweet Container O' Mine","artist":"Guns N Probeses","price":39.99,"image_url":"https://aka.ms/albums-containerappslogo"}]
```

Clean up resources

If you're not going to continue on to the [Communication between microservices](#) tutorial, you can remove the Azure resources created during this quickstart. Run the following command to delete the resource group along with all the resources created in this quickstart.



The terminal interface has two tabs: "Bash" (selected) and "Azure CLI". The "Azure CLI" tab contains the command:

```
az group delete --name $RESOURCE_GROUP
```

Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

This quickstart is the entrypoint for a set of progressive tutorials that showcase the various features within Azure Container Apps. Continue on to learn how to enable communication from a web front end that calls the API you deployed in this article.

[Tutorial: Communication between microservices](#)

Tutorial: Communication between microservices in Azure Container Apps

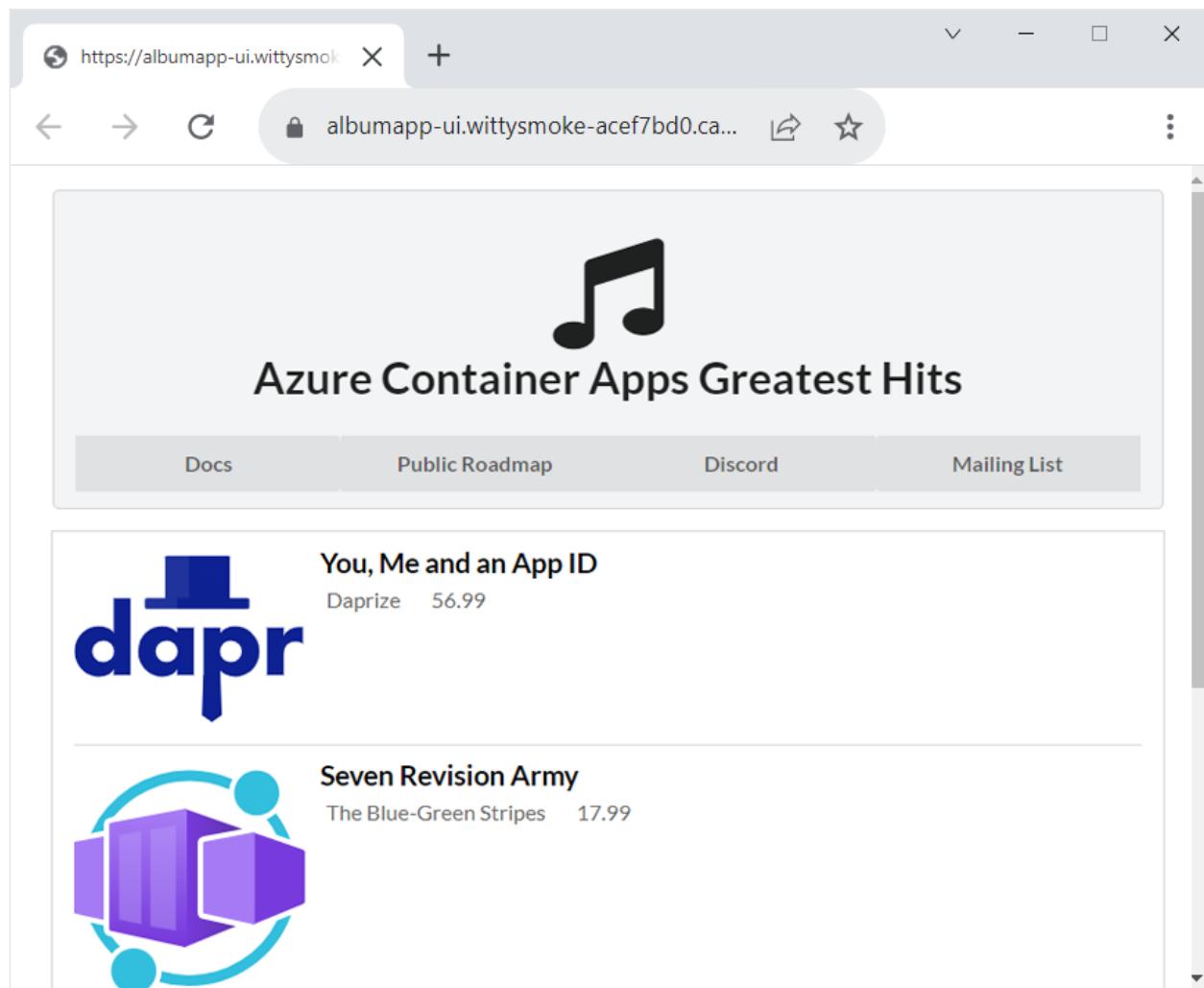
Article • 03/08/2023

Azure Container Apps exposes each container app through a domain name if [ingress](#) is enabled. Ingress endpoints for container apps within an external environment can be either publicly accessible or only available to other container apps in the same [environment](#).

Once you know the fully qualified domain name for a given container app, you can make direct calls to the service from other container apps within the shared environment.

In this tutorial, you deploy a second container app that makes a direct service call to the API deployed in the [Deploy your code to Azure Container Apps](#) quickstart.

The following screenshot shows the UI microservice deploys to container apps at the end of this article.



In this tutorial, you learn to:

- ✓ Deploy a front end application to Azure Container Apps
- ✓ Link the front end app to the API endpoint deployed in the previous quickstart
- ✓ Verify the frontend app can communicate with the back end API

Prerequisites

In the [code to cloud quickstart](#), a back end web API is deployed to return a list of music albums. If you haven't deployed the album API microservice, return to [Quickstart: Deploy your code to Azure Container Apps](#) to continue.

Setup

If you're still authenticated to Azure and still have the environment variables defined from the quickstart, you can skip the following steps and go directly to the [Prepare the GitHub repository](#) section.

Bash

Define the following variables in your bash shell.

```
Azure CLI  
  
RESOURCE_GROUP="album-containerapps"  
LOCATION="canadacentral"  
ENVIRONMENT="env-album-containerapps"  
API_NAME="album-api"  
FRONTEND_NAME="album-ui"  
GITHUB_USERNAME="<YOUR_GITHUB_USERNAME>"
```

Before you run this command, make sure to replace `<YOUR_GITHUB_USERNAME>` with your GitHub username.

Next, define a container registry name unique to you.

```
Azure CLI  
  
ACR_NAME="acaalbums"$GITHUB_USERNAME
```

Sign in to the Azure CLI.

Bash

Azure CLI

```
az login
```

Prepare the GitHub repository

1. In a new browser tab, navigate to the [repository for the UI application](#) and select the **Fork** button at the top of the page to fork the repo to your account.

Follow the prompts from GitHub to fork the repository and return here once the operation is complete.

2. Navigate to the parent of the *code-to-cloud* folder. If you're still in the *code-to-cloud/src* directory, you can use the below command to return to the parent folder.

Console

```
cd ../../
```

3. Use the following git command to clone your forked repo into the *code-to-cloud-ui* folder:

git

```
git clone https://github.com/$GITHUB_USERNAME/containerapps-albumui.git  
code-to-cloud-ui
```

⚠ Note

If the `clone` command fails, check that you have successfully forked the repository.

4. Next, change the directory into the *src* folder of the cloned repo.

Console

```
cd code-to-cloud-ui/src
```

Build the front end application

Bash

Azure CLI

```
az acr build --registry $ACR_NAME --image albumapp-ui .
```

Output from the `az acr build` command shows the upload progress of the source code to Azure and the details of the `docker build` operation.

Communicate between container apps

In the previous quickstart, the album API was deployed by creating a container app and enabling external ingress. Setting the container app's ingress to *external* made its HTTP endpoint URL publicly available.

Now you can configure the front end application to call the API endpoint by going through the following steps:

- Query the API application for its fully qualified domain name (FQDN).
- Pass the API FQDN to `az containerapp create` as an environment variable so the UI app can set the base URL for the album API call within the code.

The [UI application](#) uses the endpoint provided to invoke the album API. The following code is an excerpt from the code used in the *routes > index.js* file.

JavaScript

```
const api = axios.create({
  baseURL: process.env.API_BASE_URL,
  params: {},
  timeout: process.env.TIMEOUT || 5000,
});
```

Notice how the `baseURL` property gets its value from the `API_BASE_URL` environment variable.

Run the following command to query for the API endpoint address.

Bash

Azure CLI

```
API_BASE_URL=$(az containerapp show --resource-group $RESOURCE_GROUP --name $API_NAME --query properties.configuration.ingress.fqdn -o tsv)
```

Now that you have set the `API_BASE_URL` variable with the FQDN of the album API, you can provide it as an environment variable to the frontend container app.

Deploy front end application

Create and deploy your container app with the following command.

Bash

Azure CLI

```
az containerapp create \
--name $FRONTEND_NAME \
--resource-group $RESOURCE_GROUP \
--environment $ENVIRONMENT \
--image $ACR_NAME.azurecr.io/albumapp-ui \
--target-port 3000 \
--env-vars API_BASE_URL=https://$API_BASE_URL \
--ingress 'external' \
--registry-server $ACR_NAME.azurecr.io \
--query properties.configuration.ingress.fqdn
```

By adding the argument `--env-vars "API_BASE_URL=https://$API_ENDPOINT"` to `az containerapp create`, you define an environment variable for your front end application. With this syntax, the environment variable named `API_BASE_URL` is set to the API's FQDN.

The output from the `az containerapp create` command shows the URL of the front end application.

View website

Use the container app's FQDN to view the website. The page will resemble the following screenshot.

Docs Public Roadmap Discord Mailing List

You, Me and an App ID
Daprize 56.99

Seven Revision Army
The Blue-Green Stripes 17.99

Clean up resources

If you're not going to continue to use this application, run the following command to delete the resource group along with all the resources created in this quickstart.

⊗ Caution

This command deletes the specified resource group and all resources contained within it. If resources outside the scope of this tutorial exist in the specified resource group, they will also be deleted.

Bash

Azure CLI

```
az group delete --name $RESOURCE_GROUP
```



Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

[Environments in Azure Container Apps](#)

Tutorial: Deploy a Dapr application to Azure Container Apps using the Azure CLI

Article • 05/03/2023

Dapr [🔗](#) (Distributed Application Runtime) helps developers build resilient, reliable microservices. In this tutorial, a sample Dapr application is deployed to Azure Container Apps.

You learn how to:

- ✓ Create a Container Apps environment for your container apps
- ✓ Create an Azure Blob Storage state store for the container app
- ✓ Deploy two apps that produce and consume messages and persist them in the state store
- ✓ Verify the interaction between the two microservices.

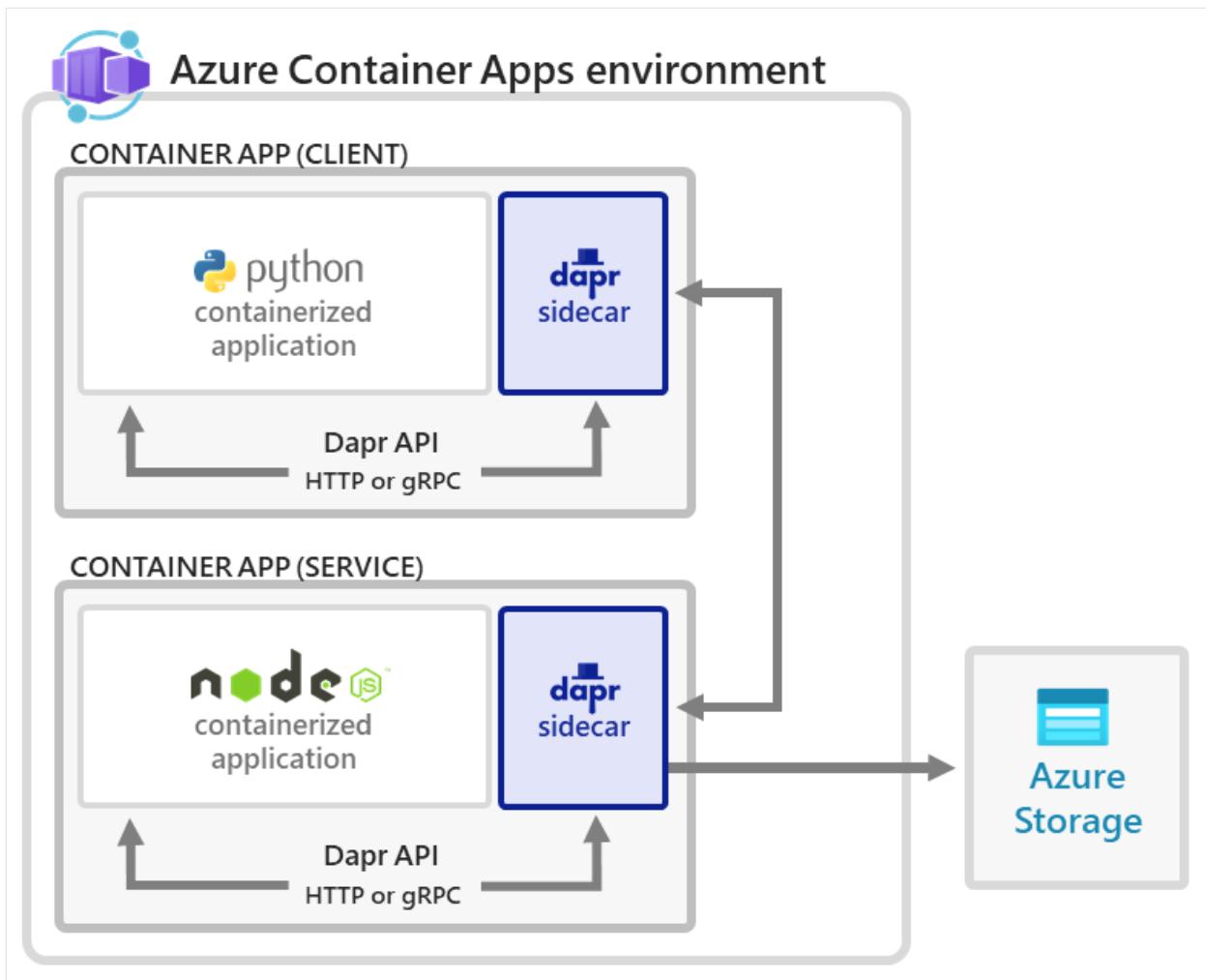
With Azure Container Apps, you get a [fully managed version of the Dapr APIs](#) when building microservices. When you use Dapr in Azure Container Apps, you can enable sidecars to run next to your microservices that provide a rich set of capabilities. Available Dapr APIs include [Service to Service calls](#) [🔗](#), [Pub/Sub](#) [🔗](#), [Event Bindings](#) [🔗](#), [State Stores](#) [🔗](#), and [Actors](#) [🔗](#).

In this tutorial, you deploy the same applications from the Dapr [Hello World](#) [🔗](#) quickstart.

The application consists of:

- A client (Python) container app to generate messages.
- A service (Node) container app to consume and persist those messages in a state store

The following architecture diagram illustrates the components that make up this tutorial:



Setup

To begin, sign in to Azure. Run the following command, and follow the prompts to complete the authentication process.

```
Bash
az login
```

```
Bash
az extension add --name containerapp --upgrade
```

Now that the current extension or module is installed, register the `Microsoft.App` namespace.

 **Note**

Azure Container Apps resources have migrated from the `Microsoft.Web` namespace to the `Microsoft.App` namespace. Refer to [Namespace migration from Microsoft.Web to Microsoft.App in March 2022](#) for more details.

Bash

Azure CLI

```
az provider register --namespace Microsoft.App
```

Register the `Microsoft.OperationalInsights` provider for the Azure Monitor Log Analytics workspace if you have not used it before.

Bash

Azure CLI

```
az provider register --namespace Microsoft.OperationalInsights
```

Next, set the following environment variables:

Bash

Azure CLI

```
RESOURCE_GROUP="my-container-apps"  
LOCATION="canadacentral"  
CONTAINERAPPS_ENVIRONMENT="my-environment"
```

With these variables defined, you can create a resource group to organize the services related to your new container app.

Bash

Azure CLI

```
az group create \
--name $RESOURCE_GROUP \
--location $LOCATION
```

With the CLI upgraded and a new resource group available, you can create a Container Apps environment and deploy your container app.

Create an environment

An environment in Azure Container Apps creates a secure boundary around a group of container apps. Container Apps deployed to the same environment are deployed in the same virtual network and write logs to the same Log Analytics workspace.

Azure CLI

Individual container apps are deployed to an Azure Container Apps environment. To create the environment, run the following command:

Azure CLI

```
az containerapp env create \
--name $CONTAINERAPPS_ENVIRONMENT \
--resource-group $RESOURCE_GROUP \
--location "$LOCATION"
```

Set up a state store

Create an Azure Blob Storage account

With the environment deployed, the next step is to deploy an Azure Blob Storage account that is used by one of the microservices to store data. Before deploying the service, you need to choose a name for the storage account. Storage account names must be *unique within Azure*, from 3 to 24 characters in length and must contain numbers and lowercase letters only.

Azure CLI

Azure CLI

```
STORAGE_ACCOUNT_NAME="<storage account name>"
```

Use the following command to create the Azure Storage account.

Azure CLI

Azure CLI

```
az storage account create \
--name $STORAGE_ACCOUNT_NAME \
--resource-group $RESOURCE_GROUP \
--location "$LOCATION" \
--sku Standard_RAGRS \
--kind StorageV2
```

Configure a user-assigned identity for the node app

While Container Apps supports both user-assigned and system-assigned managed identity, a user-assigned identity provides the Dapr-enabled node app with permissions to access the blob storage account.

1. Create a user-assigned identity.

Azure CLI

Azure CLI

```
az identity create --resource-group $RESOURCE_GROUP --name
"nodeAppIdentity" --output json
```

Retrieve the `principalId` and `id` properties and store in variables.

Azure CLI

Azure CLI

```
PRINCIPAL_ID=$(az identity show -n "nodeAppIdentity" --resource-group
$RESOURCE_GROUP --query principalId | tr -d \")
IDENTITY_ID=$(az identity show -n "nodeAppIdentity" --resource-group
```

```
$RESOURCE_GROUP --query id | tr -d \")  
CLIENT_ID=$(az identity show -n "nodeAppIdentity" --resource-group  
$RESOURCE_GROUP --query clientId | tr -d \")
```

2. Assign the `Storage Blob Data Contributor` role to the user-assigned identity

Retrieve the subscription ID for your current subscription.

Azure CLI

Azure CLI

```
SUBSCRIPTION_ID=$(az account show --query id --output tsv)
```

Azure CLI

Azure CLI

```
az role assignment create --assignee $PRINCIPAL_ID \
--role "Storage Blob Data Contributor" \
--scope
"/subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP/provider
s/Microsoft.Storage/storageAccounts/$STORAGE_ACCOUNT_NAME"
```

Configure the state store component

There are multiple ways to authenticate to external resources via Dapr. This example doesn't use the Dapr Secrets API at runtime, but uses an Azure-based state store. Therefore, you can forgo creating a secret store component and instead provide direct access from the node app to the blob store using Managed Identity. If you want to use a non-Azure state store or the Dapr Secrets API at runtime, you could create a secret store component. This component would load runtime secrets so you can reference them at runtime.

Open a text editor and create a config file named `statestore.yaml` with the properties that you sourced from the previous steps. This file helps enable your Dapr app to access your state store. The following example shows how your `statestore.yaml` file should look when configured for your Azure Blob Storage account:

YAML

```
# statestore.yaml for Azure Blob storage component
componentType: state.azure.blobstorage
version: v1
metadata:
  - name: accountName
    value: "<STORAGE_ACCOUNT_NAME>"
  - name: containerName
    value: mycontainer
  - name: azureClientId
    value: "<MANAGED_IDENTITY_CLIENT_ID>"
scopes:
  - nodeapp
```

To use this file, update the placeholders:

- Replace `<STORAGE_ACCOUNT_NAME>` with the value of the `STORAGE_ACCOUNT_NAME` variable you defined. To obtain its value, run the following command:

Azure CLI

```
Azure CLI
```

```
echo $STORAGE_ACCOUNT_NAME
```

- Replace `<MANAGED_IDENTITY_CLIENT_ID>` with the value of the `CLIENT_ID` variable you defined. To obtain its value, run the following command:

Azure CLI

```
echo $CLIENT_ID
```

Navigate to the directory in which you stored the component yaml file and run the following command to configure the Dapr component in the Container Apps environment. For more information about configuring Dapr components, see [Configure Dapr components](#).

Azure CLI

```
az containerapp env dapr-component set \
  --name $CONTAINERAPPS_ENVIRONMENT --resource-group $RESOURCE_GROUP \
  --dapr-component-name statestore \
  --yaml statestore.yaml
```

Deploy the service application (HTTP web server)

Azure CLI

```
Azure CLI

az containerapp create \
--name nodeapp \
--resource-group $RESOURCE_GROUP \
--user-assigned $IDENTITY_ID \
--environment $CONTAINERAPPS_ENVIRONMENT \
--image dapriosamples/hello-k8s-node:latest \
--min-replicas 1 \
--max-replicas 1 \
--enable-dapr \
--dapr-app-id nodeapp \
--dapr-app-port 3000 \
--env-vars 'APP_PORT=3000'
```

By default, the image is pulled from [Docker Hub](#).

Deploy the client application (headless client)

Run the following command to deploy the client container app.

Azure CLI

```
Azure CLI

az containerapp create \
--name pythonapp \
--resource-group $RESOURCE_GROUP \
--environment $CONTAINERAPPS_ENVIRONMENT \
--image dapriosamples/hello-k8s-python:latest \
--min-replicas 1 \
--max-replicas 1 \
--enable-dapr \
--dapr-app-id pythonapp
```

Verify the results

Confirm successful state persistence

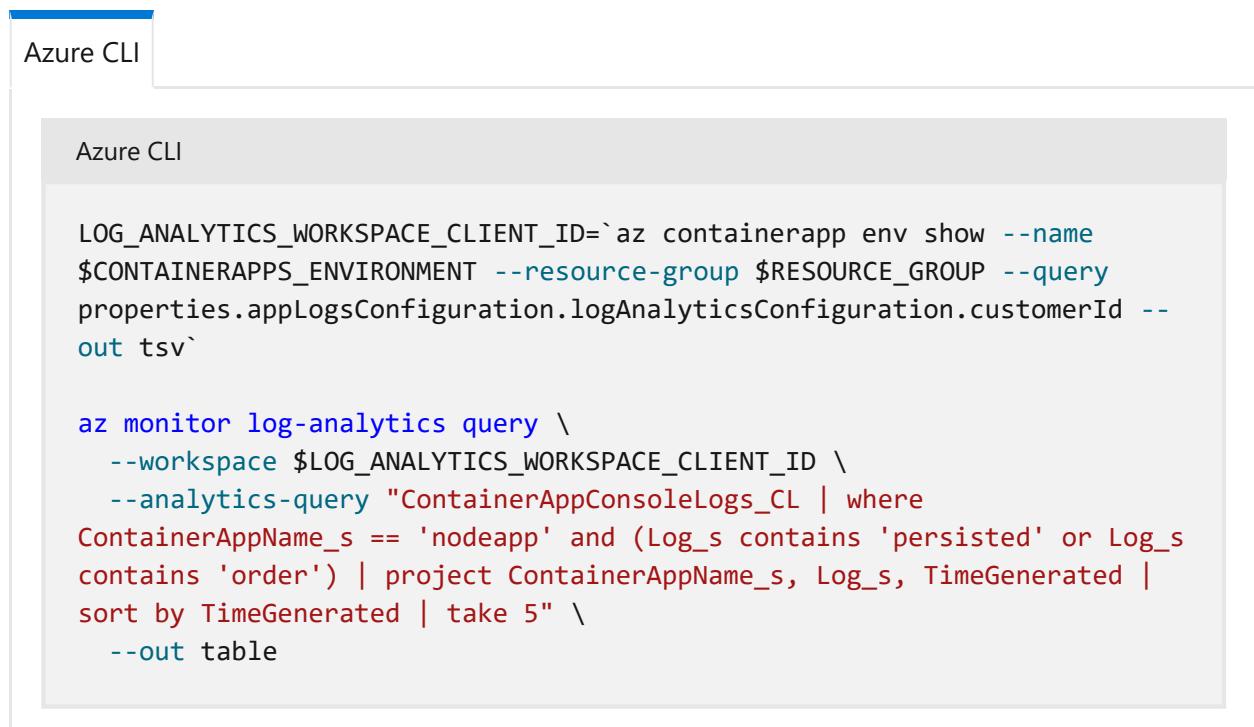
You can confirm that the services are working correctly by viewing data in your Azure Storage account.

1. Open the [Azure portal](#) in your browser and navigate to your storage account.
2. Select **Containers** left side menu.
3. Select **mycontainer**.
4. Verify that you can see the file named `order` in the container.
5. Select the file.
6. Select the **Edit** tab.
7. Select the **Refresh** button to observe how the data automatically updates.

View Logs

Logs from container apps are stored in the `ContainerAppConsoleLogs_CL` custom table in the Log Analytics workspace. You can view logs through the Azure portal or via the CLI. There may be a small delay initially for the table to appear in the workspace.

Use the following CLI command to view logs using the command line.



```
LOG_ANALYTICS_WORKSPACE_CLIENT_ID=`az containerapp env show --name $CONTAINERAPPS_ENVIRONMENT --resource-group $RESOURCE_GROUP --query properties.appLogsConfiguration.logAnalyticsConfiguration.customerId --out tsv`  
  
az monitor log-analytics query \  
--workspace $LOG_ANALYTICS_WORKSPACE_CLIENT_ID \  
--analytics-query "ContainerAppConsoleLogs_CL | where ContainerAppName_s == 'nodeapp' and (Log_s contains 'persisted' or Log_s contains 'order') | project ContainerAppName_s, Log_s, TimeGenerated | sort by TimeGenerated | take 5" \  
--out table
```

The following output demonstrates the type of response to expect from the CLI command.

Bash

ContainerAppName_s	Log_s	TableName
TimeGenerated		
nodeapp 10-22T21:31:46.184Z	Got a new order! Order ID: 61	PrimaryResult 2021-
nodeapp 10-22T21:31:46.184Z	Successfully persisted state.	PrimaryResult 2021-
nodeapp 10-22T22:01:57.174Z	Got a new order! Order ID: 62	PrimaryResult 2021-
nodeapp 10-22T22:01:57.174Z	Successfully persisted state.	PrimaryResult 2021-
nodeapp 10-22T22:45:44.618Z	Got a new order! Order ID: 63	PrimaryResult 2021-

Clean up resources

Congratulations! You've completed this tutorial. If you'd like to delete the resources created as a part of this walkthrough, run the following command.

✖ Caution

This command deletes the specified resource group and all resources contained within it. If resources outside the scope of this tutorial exist in the specified resource group, they will also be deleted.

Azure CLI

Azure CLI

```
az group delete --resource-group $RESOURCE_GROUP
```

ⓘ Note

Since `pythonapp` continuously makes calls to `nodeapp` with messages that get persisted into your configured state store, it is important to complete these cleanup steps to avoid ongoing billable operations.

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

[Application lifecycle management](#)

Tutorial: Deploy a Dapr application to Azure Container Apps with an Azure Resource Manager or Bicep template

Article • 03/08/2023

Dapr [↗](#) (Distributed Application Runtime) is a runtime that helps you build resilient stateless and stateful microservices. In this tutorial, a sample Dapr solution is deployed to Azure Container Apps via an Azure Resource Manager (ARM) or Bicep template.

You learn how to:

- ✓ Create an Azure Blob Storage for use as a Dapr state store
- ✓ Deploy a Container Apps environment to host container apps
- ✓ Deploy two dapr-enabled container apps: one that produces orders and one that consumes orders and stores them
- ✓ Assign a user-assigned identity to a container app and supply it with the appropriate role assignment to authenticate to the Dapr state store
- ✓ Verify the interaction between the two microservices.

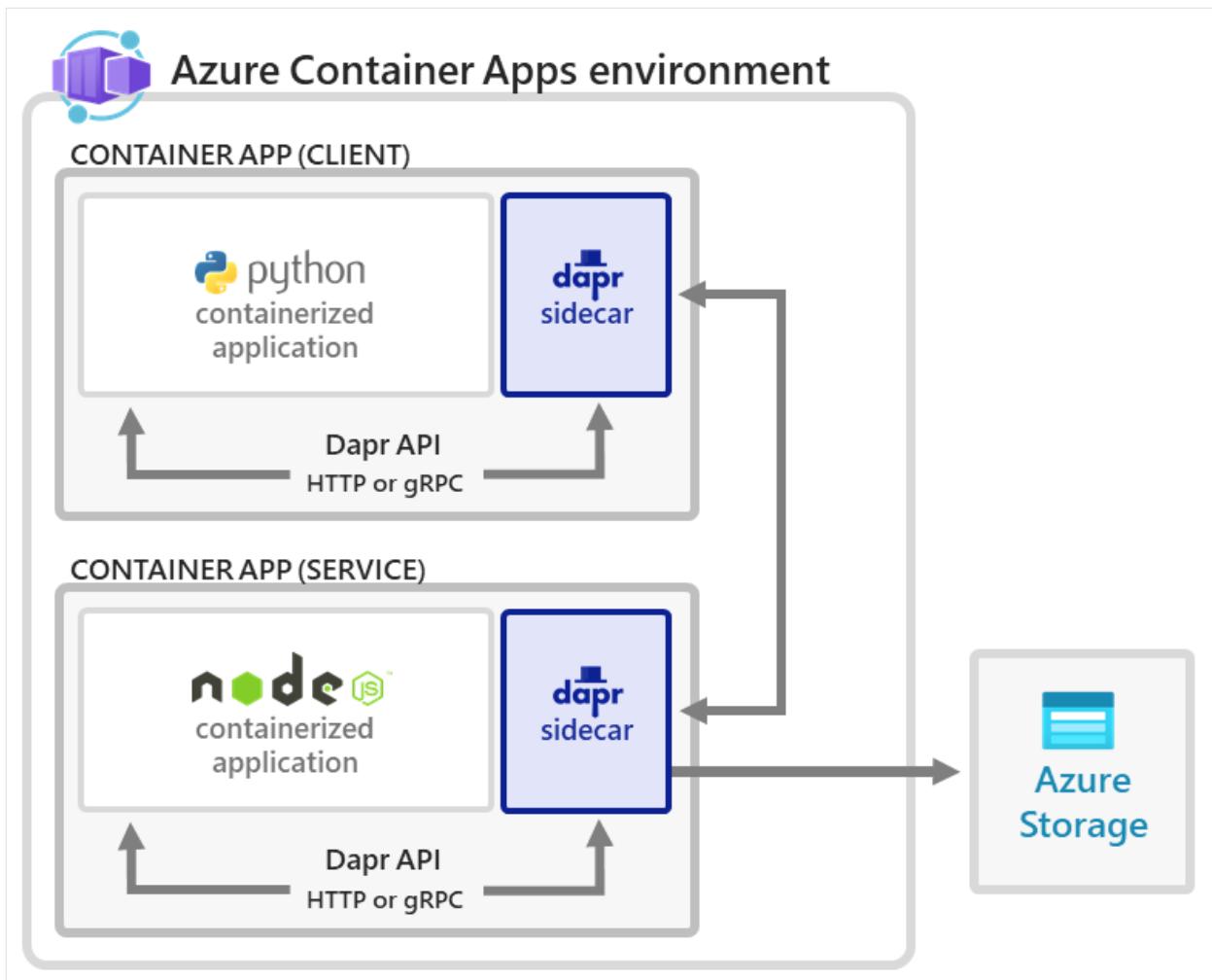
With Azure Container Apps, you get a [fully managed version of the Dapr APIs](#) when building microservices. When you use Dapr in Azure Container Apps, you can enable sidecars to run next to your microservices that provide a rich set of capabilities.

In this tutorial, you deploy the solution from the Dapr [Hello World ↗](#) quickstart.

The application consists of:

- A client (Python) container app to generate messages.
- A service (Node) container app to consume and persist those messages in a state store

The following architecture diagram illustrates the components that make up this tutorial:



Prerequisites

- Install [Azure CLI](#)
- Install [Git](#)
- An Azure account with an active subscription is required. If you don't already have one, you can [create an account for free](#).
- A GitHub Account. If you don't already have one, sign up for [free](#).

Setup

First, sign in to Azure.

```
Bash
az login
```

Bash

Ensure you're running the latest version of the CLI via the upgrade command and then install the Azure Container Apps extension for the Azure CLI.

Azure CLI

```
az upgrade
```

```
az extension add --name containerapp --upgrade
```

Now that the current extension or module is installed, register the `Microsoft.App` namespace.

Bash

Azure CLI

```
az provider register --namespace Microsoft.App
```

Next, set the following environment variables:

Bash

Azure CLI

```
RESOURCE_GROUP="my-container-apps"  
LOCATION="centralus"  
CONTAINERAPPS_ENVIRONMENT="my-environment"
```

With these variables defined, you can create a resource group to organize the services needed for this tutorial.

Bash

Azure CLI

```
az group create \  
--name $RESOURCE_GROUP \  
--location $LOCATION
```

Prepare the GitHub repository

Go to the repository holding the ARM and Bicep templates that's used to deploy the solution.

Select the **Fork** button at the top of the [repository](#) to fork the repo to your account.

Now you can clone your fork to work with it locally.

Use the following git command to clone your forked repo into the *acadapr-templates* directory.

```
git
```

```
git clone https://github.com/$GITHUB_USERNAME/Tutorial-Deploy-Dapr-Microservices-ACA.git acadapr-templates
```

Deploy

The template deploys:

- a Container Apps environment
- a Log Analytics workspace associated with the Container Apps environment
- an Application Insights resource for distributed tracing
- a blob storage account and a default storage container
- a Dapr component for the blob storage account
- the node, Dapr-enabled container app with a user-assigned managed identity: [hello-k8s-node](#)
- the python, Dapr-enabled container app: [hello-k8s-python](#)
- an Active Directory role assignment for the node app used by the Dapr component to establish a connection to blob storage

Navigate to the *acadapr-templates* directory and run the following command:

Bash

Azure CLI

```
az deployment group create \
--resource-group "$RESOURCE_GROUP" \
--template-file ./azuredeploy.json \
--parameters environment_name="$CONTAINERAPPS_ENVIRONMENT"
```

This command deploys:

- the Container Apps environment and associated Log Analytics workspace for hosting the hello world Dapr solution
- an Application Insights instance for Dapr distributed tracing
- the `nodeapp` app server running on `targetPort: 3000` with Dapr enabled and configured using: `"appId": "nodeapp"` and `"appPort": 3000`, and a user-assigned identity with access to the Azure Blob storage via a Storage Data Contributor role assignment
- A Dapr component of `"type": "state.azure.blobstorage"` scoped for use by the `nodeapp` for storing state
- the Dapr-enabled, headless `pythonapp` that invokes the `nodeapp` service using Dapr service invocation

Verify the result

Confirm successful state persistence

You can confirm that the services are working correctly by viewing data in your Azure Storage account.

1. Open the [Azure portal](#) in your browser.
2. Go to the newly created storage account in your resource group.
3. Select **Containers** from the menu on the left side.
4. Select the created container.
5. Verify that you can see the file named `order` in the container.
6. Select the file.
7. Select the **Edit** tab.
8. Select the **Refresh** button to observe updates.

View Logs

Data logged via a container app are stored in the `ContainerAppConsoleLogs_CL` custom table in the Log Analytics workspace. You can view logs through the Azure portal or

from the command line. Wait a few minutes for the analytics to arrive for the first time before you query the logged data.

Use the following command to view logs in bash or PowerShell.

Bash

Azure CLI

```
LOG_ANALYTICS_WORKSPACE_CLIENT_ID=`az containerapp env show --name $CONTAINERAPPS_ENVIRONMENT --resource-group $RESOURCE_GROUP --query properties.appLogsConfiguration.logAnalyticsConfiguration.customerId --out tsv`
```

Azure CLI

```
az monitor log-analytics query \
--workspace "$LOG_ANALYTICS_WORKSPACE_CLIENT_ID" \
--analytics-query "ContainerAppConsoleLogs_CL | where ContainerAppName_s == 'nodeapp' and (Log_s contains 'persisted' or Log_s contains 'order') | project ContainerAppName_s, Log_s, TimeGenerated | take 5" \
--out table
```

The following output demonstrates the type of response to expect from the command.

Console

ContainerAppName_s	Log_s	TableName
TimeGenerated		
-----	-----	-----
nodeapp 10-22T21:31:46.184Z	Got a new order! Order ID: 61	PrimaryResult 2021-
nodeapp 10-22T21:31:46.184Z	Successfully persisted state.	PrimaryResult 2021-
nodeapp 10-22T22:01:57.174Z	Got a new order! Order ID: 62	PrimaryResult 2021-
nodeapp 10-22T22:01:57.174Z	Successfully persisted state.	PrimaryResult 2021-
nodeapp 10-22T22:45:44.618Z	Got a new order! Order ID: 63	PrimaryResult 2021-

Clean up resources

Once you're done, run the following command to delete your resource group along with all the resources you created in this tutorial.

Bash

Azure CLI

```
az group delete \
--resource-group $RESOURCE_GROUP
```

ⓘ Note

Since `pythonapp` continuously makes calls to `nodeapp` with messages that get persisted into your configured state store, it is important to complete these cleanup steps to avoid ongoing billable operations.

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Next steps

[Application lifecycle management](#)

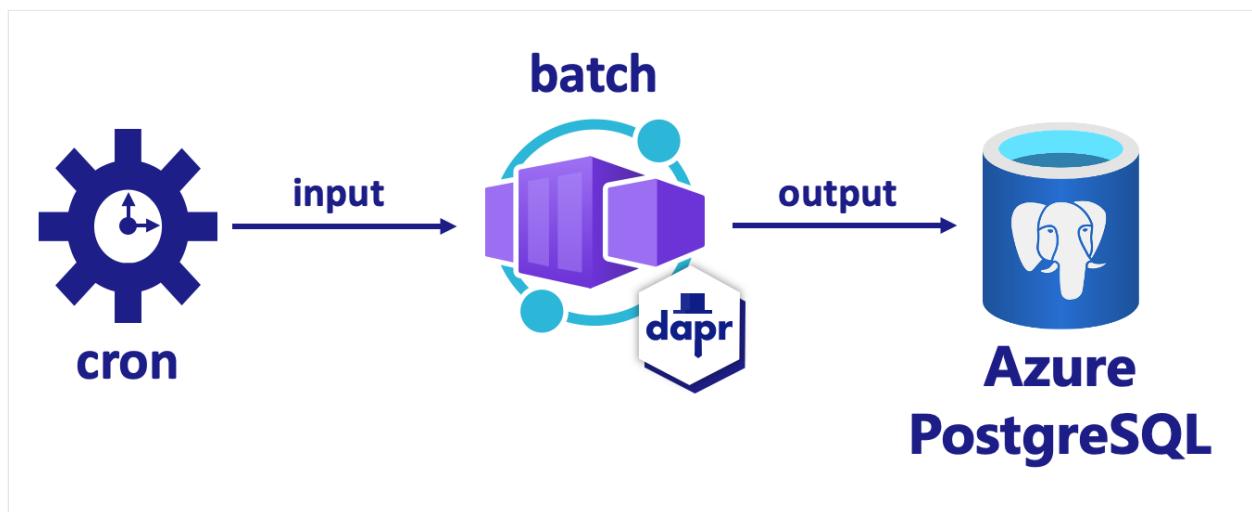
Event-driven work using Dapr Bindings

Article • 04/11/2023

In this tutorial, you create a microservice to demonstrate [Dapr's Bindings API](#) to work with external systems as inputs and outputs. You'll:

- ✓ Run the application locally.
- ✓ Deploy the application to Azure Container Apps via the Azure Developer CLI with the provided Bicep.

The service listens to input binding events from a system CRON and then outputs the contents of local data to a PostgreSQL output binding.



Prerequisites

- Install [Azure Developer CLI](#)
- [Install](#) and [init](#) Dapr
- [Docker Desktop](#)
- Install [Git](#)

Run the Node.js application locally

Before deploying the application to Azure Container Apps, start by running the PostgreSQL container and JavaScript service locally with [Docker Compose](#) and Dapr.

Prepare the project

1. Clone the [sample Dapr application](#) to your local machine.

Bash

```
git clone https://github.com/Azure-Samples/bindings-dapr-nodejs-cron-postgres.git
```

2. Navigate into the sample's root directory.

Bash

```
cd bindings-dapr-nodejs-cron-postgres
```

Run the Dapr application using the Dapr CLI

1. From the sample's root directory, change directories to `db`.

Bash

```
cd db
```

2. Run the PostgreSQL container with Docker Compose.

Bash

```
docker compose up -d
```

3. Open a new terminal window and navigate into `/batch` in the sample directory.

Bash

```
cd bindings-dapr-nodejs-cron-postgres/batch
```

4. Install the dependencies.

Bash

```
npm install
```

5. Run the JavaScript service application with Dapr.

Bash

```
dapr run --app-id batch-sdk --app-port 5002 --dapr-http-port 3500 --resources-path ../components -- node index.js
```

The `dapr run` command runs the Dapr binding application locally. Once the application is running successfully, the terminal window shows the output binding data.

Expected output

The batch service listens to input binding events from a system CRON and then outputs the contents of local data to a PostgreSQL output binding.

```
== APP == {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}  
== APP == {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}  
== APP == {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}  
== APP == Finished processing batch  
== APP == {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}  
== APP == {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}  
== APP == {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}  
== APP == Finished processing batch  
== APP == {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}  
== APP == {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}  
== APP == {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}  
== APP == Finished processing batch
```

6. In the `./db` terminal, stop the PostgreSQL container.

```
Bash
```

```
docker compose stop
```

Deploy the Dapr application template using Azure Developer CLI

Now that you've run the application locally, let's deploy the Dapr bindings application to Azure Container Apps using `azd`. During deployment, we will swap the local containerized PostgreSQL for an Azure PostgreSQL component.

Prepare the project

Navigate into the [sample's](#) root directory.

```
Bash
```

```
cd bindings-dapr-nodejs-cron-postgres
```

Provision and deploy using Azure Developer CLI

1. Run `azd init` to initialize the project.

```
Azure Developer CLI
```

```
azd init
```

2. When prompted in the terminal, provide the following parameters.

Parameter	Description
Environment Name	Prefix for the resource group created to hold all Azure resources.
Azure Location	The Azure location for your resources. Make sure you select a location available for Azure PostgreSQL.
Azure Subscription	The Azure subscription for your resources.

3. Run `azd up` to provision the infrastructure and deploy the Dapr application to Azure Container Apps in a single command.

```
Azure Developer CLI
```

```
azd up
```

This process may take some time to complete. As the `azd up` command completes, the CLI output displays two Azure portal links to monitor the deployment progress. The output also demonstrates how `azd up`:

- Creates and configures all necessary Azure resources via the provided Bicep files in the `./infra` directory using `azd provision`. Once provisioned by Azure Developer CLI, you can access these resources via the Azure portal. The files that provision the Azure resources include:

- `main.parameters.json`
- `main.bicep`
- An `app` resources directory organized by functionality
- A `core` reference library that contains the Bicep modules used by the `azd` template
- Deploys the code using `azd deploy`

Expected output

```
Azure Developer CLI

Initializing a new project (azd init)

Provisioning Azure resources (azd provision)
Provisioning Azure resources can take some time

    You can view detailed progress in the Azure Portal:
    https://portal.azure.com/#blade/HubsExtension/DeploymentDetailsBlade/overview

        (✓) Done: Resource group: resource-group-name
        (✓) Done: Log Analytics workspace: log-analytics-name
        (✓) Done: Application Insights: app-insights-name
        (✓) Done: Portal dashboard: dashboard-name
        (✓) Done: Azure Database for PostgreSQL flexible server: postgres-server
        (✓) Done: Key vault: key-vault-name
        (✓) Done: Container Apps Environment: container-apps-env-name
        (✓) Done: Container App: container-app-name

Deploying services (azd deploy)

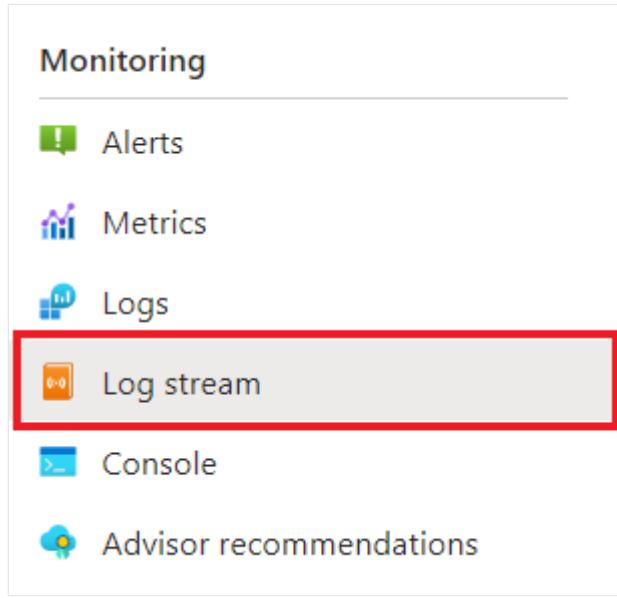
        (✓) Done: Deploying service api
            - Endpoint: https://your-container-app-endpoint.region.azurecontainerapps.io/

SUCCESS: Your Azure app has been deployed!
You can view the resources created under the resource group resource-group-name in Azure Portal:
https://portal.azure.com/#@/resource/subscriptions/your-subscription-ID/resourceGroups/your-resource-group/overview
```

Confirm successful deployment

In the Azure portal, verify the batch container app is logging each insert into Azure PostgreSQL every 10 seconds.

1. Copy the Container App name from the terminal output.
2. Navigate to the [Azure portal](#) and search for the Container App resource by name.
3. In the Container App dashboard, select **Monitoring > Log stream**.



4. Confirm the container is logging the same output as in the terminal earlier.

```
Connecting...
2023-03-16T17:43:18.39918 Connecting to the container 'batch'...
2023-03-16T17:43:18.42740 Successfully Connected to container: 'batch'
2023-03-16T17:42:29.002081843Z {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}
2023-03-16T17:42:29.002486595Z {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}
2023-03-16T17:42:29.002712521Z {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}
2023-03-16T17:42:29.002949202Z Finished processing batch
2023-03-16T17:42:39.001177919Z {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}
2023-03-16T17:42:39.001611119Z {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}
2023-03-16T17:42:39.001863501Z {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}
2023-03-16T17:42:39.002089563Z Finished processing batch
2023-03-16T17:42:49.002300437Z {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}
2023-03-16T17:42:49.002666356Z {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}
2023-03-16T17:42:49.002895265Z {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}
2023-03-16T17:42:49.003144817Z Finished processing batch
2023-03-16T17:42:59.001646495Z {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}
2023-03-16T17:42:59.001928185Z {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}
2023-03-16T17:42:59.002169928Z {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}
2023-03-16T17:42:59.002379279Z Finished processing batch
2023-03-16T17:43:09.001854673Z {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}
2023-03-16T17:43:09.002184793Z {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}
2023-03-16T17:43:09.002425682Z {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}
2023-03-16T17:43:09.002666605Z Finished processing batch
2023-03-16T17:43:19.001859295Z {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}
2023-03-16T17:43:19.002233945Z {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}
2023-03-16T17:43:19.002589029Z {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}
2023-03-16T17:43:19.002875744Z Finished processing batch
2023-03-16T17:43:29.001927907Z {"sql": "insert into orders (orderid, customer, price) values (1, 'John Smith', 100.32);"}
2023-03-16T17:43:29.0022381054Z {"sql": "insert into orders (orderid, customer, price) values (2, 'Jane Bond', 15.4);"}
2023-03-16T17:43:29.003518597Z {"sql": "insert into orders (orderid, customer, price) values (3, 'Tony James', 35.56);"}
2023-03-16T17:43:29.003551480Z Finished processing batch
```

What happened?

Upon successful completion of the `azd up` command:

- Azure Developer CLI provisioned the Azure resources referenced in the [sample project's ./infra directory](#) to the Azure subscription you specified. You can now view those Azure resources via the Azure portal.
- The app deployed to Azure Container Apps. From the portal, you can browse the fully functional app.

Clean up resources

If you're not going to continue to use this application, delete the Azure resources you've provisioned with the following command.

```
Azure Developer CLI
```

```
azd down
```

Next steps

- Learn more about [deploying Dapr applications to Azure Container Apps](#).
- [Enable token authentication for Dapr requests](#).
- Learn more about [Azure Developer CLI](#) and [making your applications compatible with azd](#).
- [Scale your Dapr applications using KEDA scalers](#)

Microservices communication using Dapr Publish and Subscribe

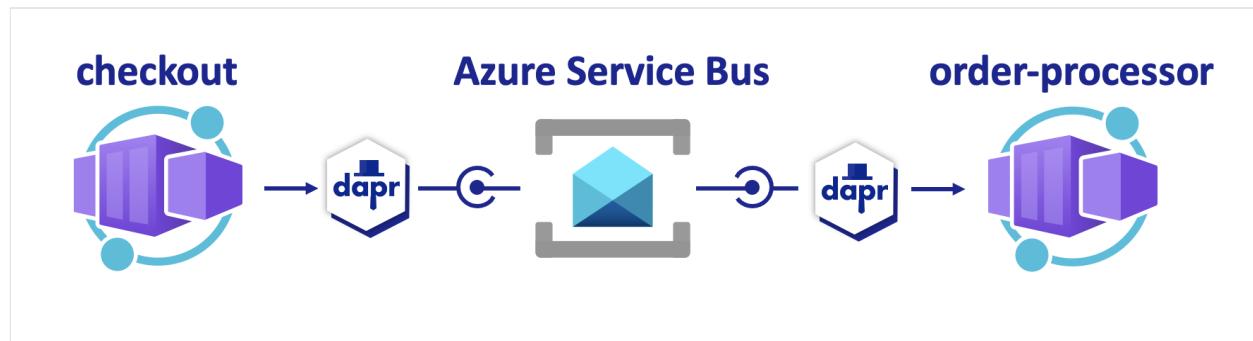
Article • 05/15/2023

In this tutorial, you'll:

- ✓ Create a publisher microservice and a subscriber microservice that leverage the [Dapr pub/sub API](#) to communicate using messages for event-driven architectures.
- ✓ Deploy the application to Azure Container Apps via the Azure Developer CLI with provided Bicep.

The sample pub/sub project includes:

1. A message generator (publisher) `checkout` service that generates messages of a specific topic.
2. An (subscriber) `order-processor` service that listens for messages from the `checkout` service of a specific topic.



Prerequisites

- Install [Azure Developer CLI](#)
- [Install](#) and [init](#) Dapr
- [Docker Desktop](#)
- Install [Git](#)

Run the Node.js applications locally

Before deploying the application to Azure Container Apps, run the `order-processor` and `checkout` services locally with Dapr and Azure Service Bus.

Prepare the project

1. Clone the [sample Dapr application](#) to your local machine.

```
Bash
```

```
git clone https://github.com/Azure-Samples/pubsub-dapr-nodejs-servicebus.git
```

2. Navigate into the sample's root directory.

```
Bash
```

```
cd pubsub-dapr-nodejs-servicebus
```

Run the Dapr applications using the Dapr CLI

Start by running the `order-processor` subscriber service with Dapr.

1. From the sample's root directory, change directories to `order-processor`.

```
Bash
```

```
cd order-processor
```

2. Install the dependencies.

```
Bash
```

```
npm install
```

3. Run the `order-processor` service with Dapr.

```
Bash
```

```
dapr run --app-port 5001 --app-id order-processing --app-protocol http --dapr-http-port 3501 --resources-path ../components -- npm run start
```

4. In a new terminal window, from the sample's root directory, navigate to the `checkout` publisher service.

```
Bash
```

```
cd checkout
```

5. Install the dependencies.

Bash

```
npm install
```

6. Run the `checkout` service with Dapr.

Bash

```
dapr run --app-id checkout --app-protocol http --resources-path  
./components -- npm run start
```

Expected output

In both terminals, the `checkout` service publishes 10 messages received by the `order-processor` service before exiting.

`checkout` output:

```
== APP == Published data: {"orderId":1}  
== APP == Published data: {"orderId":2}  
== APP == Published data: {"orderId":3}  
== APP == Published data: {"orderId":4}  
== APP == Published data: {"orderId":5}  
== APP == Published data: {"orderId":6}  
== APP == Published data: {"orderId":7}  
== APP == Published data: {"orderId":8}  
== APP == Published data: {"orderId":9}  
== APP == Published data: {"orderId":10}
```

`order-processor` output:

```
== APP == Subscriber received: {"orderId":1}  
== APP == Subscriber received: {"orderId":2}  
== APP == Subscriber received: {"orderId":3}  
== APP == Subscriber received: {"orderId":4}  
== APP == Subscriber received: {"orderId":5}  
== APP == Subscriber received: {"orderId":6}
```

```
== APP == Subscriber received: {"orderId":7}
== APP == Subscriber received: {"orderId":8}
== APP == Subscriber received: {"orderId":9}
== APP == Subscriber received: {"orderId":10}
```

7. Make sure both applications have stopped by running the following commands. In the checkout terminal:

```
sh
dapr stop --app-id checkout
```

- In the order-processor terminal:

```
sh
dapr stop --app-id order-processor
```

Deploy the Dapr application template using Azure Developer CLI

Deploy the Dapr application to Azure Container Apps using [azd](#).

Prepare the project

In a new terminal window, navigate into the [sample's](#) root directory.

```
Bash
cd pubsub-dapr-nodejs-servicebus
```

Provision and deploy using Azure Developer CLI

1. Run `azd init` to initialize the project.

```
Azure Developer CLI
azd init
```

2. When prompted in the terminal, provide the following parameters.

Parameter	Description
Environment Name	Prefix for the resource group created to hold all Azure resources.
Azure Location	The Azure location for your resources.
Azure Subscription	The Azure subscription for your resources.

3. Run `azd up` to provision the infrastructure and deploy the Dapr application to Azure Container Apps in a single command.

Azure Developer CLI

```
azd up
```

This process may take some time to complete. As the `azd up` command completes, the CLI output displays two Azure portal links to monitor the deployment progress. The output also demonstrates how `azd up`:

- Creates and configures all necessary Azure resources via the provided Bicep files in the `./infra` directory using `azd provision`. Once provisioned by Azure Developer CLI, you can access these resources via the Azure portal. The files that provision the Azure resources include:
 - `main.parameters.json`
 - `main.bicep`
 - An `app` resources directory organized by functionality
 - A `core` reference library that contains the Bicep modules used by the `azd template`
- Deploys the code using `azd deploy`

Expected output

Azure Developer CLI

```
Initializing a new project (azd init)
```

```
Provisioning Azure resources (azd provision)
Provisioning Azure resources can take some time
```

```
You can view detailed progress in the Azure Portal:
https://portal.azure.com
```

```
(✓) Done: Resource group: resource-group-name
(✓) Done: Application Insights: app-insights-name
```

```
(✓) Done: Portal dashboard: portal-dashboard-name
(✓) Done: Log Analytics workspace: log-analytics-name
(✓) Done: Key vault: key-vault-name
(✓) Done: Container Apps Environment: ca-env-name
(✓) Done: Container App: ca-checkout-name
(✓) Done: Container App: ca-orders-name
```

Deploying services (azd deploy)

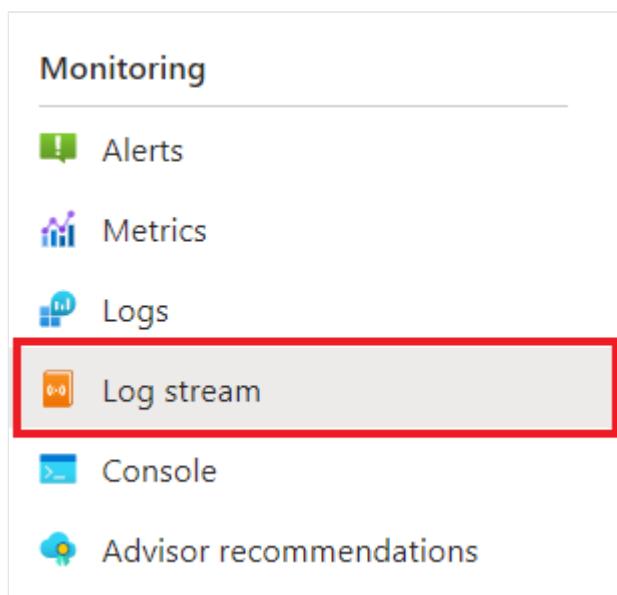
```
(✓) Done: Deploying service checkout
(✓) Done: Deploying service orders
- Endpoint: https://ca-orders-
name.endpoint.region.azurecontainerapps.io/
```

SUCCESS: Your Azure app has been deployed!
You can view the resources created under the resource group `resource-group-name` in Azure Portal:
<https://portal.azure.com/#@/resource/subscriptions/subscription-id/resourceGroups/resource-group-name/overview>

Confirm successful deployment

In the Azure portal, verify the `checkout` service is publishing messages to the Azure Service Bus topic.

1. Copy the `checkout` container app name from the terminal output.
2. Go to the [Azure portal](#) and search for the container app resource by name.
3. In the Container Apps dashboard, select **Monitoring > Log stream**.



4. Confirm the `checkout` container is logging the same output as in the terminal earlier.

```
Connecting...
2023-03-16T18:31:36.99296  Connecting to the container 'checkoutservc'...
2023-03-16T18:31:37.00856  Successfully Connected to container: 'checkoutservc'
2023-03-16T18:30:56.828036195Z Published data: {"orderId":18}
2023-03-16T18:30:57.840491575Z Published data: {"orderId":19}
2023-03-16T18:30:58.852676997Z Published data: {"orderId":20}
2023-03-16T18:31:19.889447220Z Published data: {"orderId":1}
2023-03-16T18:31:20.960174611Z Published data: {"orderId":2}
2023-03-16T18:31:21.972052331Z Published data: {"orderId":3}
2023-03-16T18:31:22.989037174Z Published data: {"orderId":4}
2023-03-16T18:31:24.005442237Z Published data: {"orderId":5}
2023-03-16T18:31:25.030923118Z Published data: {"orderId":6}
2023-03-16T18:31:26.042206386Z Published data: {"orderId":7}
2023-03-16T18:31:27.064652690Z Published data: {"orderId":8}
2023-03-16T18:31:28.085163847Z Published data: {"orderId":9}
2023-03-16T18:31:29.098112829Z Published data: {"orderId":10}
2023-03-16T18:31:30.130083890Z Published data: {"orderId":11}
2023-03-16T18:31:31.141478370Z Published data: {"orderId":12}
2023-03-16T18:31:32.152914145Z Published data: {"orderId":13}
2023-03-16T18:31:33.178869633Z Published data: {"orderId":14}
2023-03-16T18:31:34.197171578Z Published data: {"orderId":15}
2023-03-16T18:31:35.215483774Z Published data: {"orderId":16}
2023-03-16T18:31:36.239403619Z Published data: {"orderId":17}
2023-03-16T18:31:37.250161693Z Published data: {"orderId":18}
2023-03-16T18:31:38.271098752Z Published data: {"orderId":19}
2023-03-16T18:31:39.282488421Z Published data: {"orderId":20}
2023-03-16T18:32:00.299556190Z Published data: {"orderId":1}
2023-03-16T18:32:01.313715936Z Published data: {"orderId":2}
2023-03-16T18:32:02.324658618Z Published data: {"orderId":3}
2023-03-16T18:32:03.359539159Z Published data: {"orderId":4}
2023-03-16T18:32:04.370340378Z Published data: {"orderId":5}
2023-03-16T18:32:05.391758104Z Published data: {"orderId":6}
2023-03-16T18:32:06.416020741Z Published data: {"orderId":7}
2023-03-16T18:32:07.436700983Z Published data: {"orderId":8}
2023-03-16T18:32:08.451113620Z Published data: {"orderId":9}
2023-03-16T18:32:09.464197086Z Published data: {"orderId":10}
2023-03-16T18:32:10.480107097Z Published data: {"orderId":11}
2023-03-16T18:32:11.491068984Z Published data: {"orderId":12}
2023-03-16T18:32:12.501874767Z Published data: {"orderId":13}
2023-03-16T18:32:13.513070401Z Published data: {"orderId":14}
2023-03-16T18:32:14.524622724Z Published data: {"orderId":15}
2023-03-16T18:32:15.538885437Z Published data: {"orderId":16}
2023-03-16T18:32:16.550466423Z Published data: {"orderId":17}
2023-03-16T18:32:17.560761280Z Published data: {"orderId":18}
2023-03-16T18:32:18.574345037Z Published data: {"orderId":19}
2023-03-16T18:32:19.592939253Z Published data: {"orderId":20}
[]
```

5. Do the same for the `order-processor` service.

```
Connecting...
2023-03-16T18:32:36.51236 Connecting to the container 'orderssvc'...
2023-03-16T18:32:36.52492 Successfully Connected to container: 'orderssvc'
2023-03-16T18:32:00.301362583Z Subscriber received: {"orderId":1}
2023-03-16T18:32:01.318856533Z Subscriber received: {"orderId":2}
2023-03-16T18:32:02.330846489Z Subscriber received: {"orderId":3}
2023-03-16T18:32:03.377727556Z Subscriber received: {"orderId":4}
2023-03-16T18:32:04.372099637Z Subscriber received: {"orderId":5}
2023-03-16T18:32:05.394407385Z Subscriber received: {"orderId":6}
2023-03-16T18:32:06.420501724Z Subscriber received: {"orderId":7}
2023-03-16T18:32:07.441036131Z Subscriber received: {"orderId":8}
2023-03-16T18:32:08.454883619Z Subscriber received: {"orderId":9}
2023-03-16T18:32:09.466191591Z Subscriber received: {"orderId":10}
2023-03-16T18:32:10.488832713Z Subscriber received: {"orderId":11}
2023-03-16T18:32:11.492296399Z Subscriber received: {"orderId":12}
2023-03-16T18:32:12.523131890Z Subscriber received: {"orderId":13}
2023-03-16T18:32:13.518628045Z Subscriber received: {"orderId":14}
2023-03-16T18:32:14.528746484Z Subscriber received: {"orderId":15}
2023-03-16T18:32:15.547271429Z Subscriber received: {"orderId":16}
2023-03-16T18:32:16.558563837Z Subscriber received: {"orderId":17}
2023-03-16T18:32:17.561754545Z Subscriber received: {"orderId":18}
2023-03-16T18:32:18.588262841Z Subscriber received: {"orderId":19}
2023-03-16T18:32:19.642823280Z Subscriber received: {"orderId":20}
2023-03-16T18:32:40.683125039Z Subscriber received: {"orderId":1}
2023-03-16T18:32:41.721052948Z Subscriber received: {"orderId":2}
2023-03-16T18:32:42.751906853Z Subscriber received: {"orderId":3}
2023-03-16T18:32:43.769318986Z Subscriber received: {"orderId":4}
2023-03-16T18:32:44.879218480Z Subscriber received: {"orderId":5}
2023-03-16T18:32:45.926990011Z Subscriber received: {"orderId":6}
2023-03-16T18:32:46.918026086Z Subscriber received: {"orderId":7}
2023-03-16T18:32:47.927076538Z Subscriber received: {"orderId":8}
2023-03-16T18:32:48.942846717Z Subscriber received: {"orderId":9}
2023-03-16T18:32:49.965427846Z Subscriber received: {"orderId":10}
2023-03-16T18:32:51.004109254Z Subscriber received: {"orderId":11}
2023-03-16T18:32:52.009043018Z Subscriber received: {"orderId":12}
2023-03-16T18:32:53.037067302Z Subscriber received: {"orderId":13}
2023-03-16T18:32:54.047611916Z Subscriber received: {"orderId":14}
2023-03-16T18:32:55.058429174Z Subscriber received: {"orderId":15}
2023-03-16T18:32:56.069449392Z Subscriber received: {"orderId":16}
2023-03-16T18:32:57.097965149Z Subscriber received: {"orderId":17}
2023-03-16T18:32:58.094564698Z Subscriber received: {"orderId":18}
2023-03-16T18:32:59.117086229Z Subscriber received: {"orderId":19}
2023-03-16T18:33:00.146536418Z Subscriber received: {"orderId":20}
```

What happened?

Upon successful completion of the `azd up` command:

- Azure Developer CLI provisioned the Azure resources referenced in the [sample project's ./infra directory](#) to the Azure subscription you specified. You can now view those Azure resources via the Azure portal.
- The app deployed to Azure Container Apps. From the portal, you can browse to the fully functional app.

Clean up resources

If you're not going to continue to use this application, delete the Azure resources you've provisioned with the following command:

```
Azure Developer CLI
```

```
azd down
```

Next steps

- Learn more about [deploying Dapr applications to Azure Container Apps](#).
- [Enable token authentication for Dapr requests](#).
- Learn more about [Azure Developer CLI](#) and [making your applications compatible with azd](#).
- [Scale your Dapr applications using KEDA scalers](#)

Microservices communication using Dapr Service Invocation

Article • 05/15/2023

In this tutorial, you'll:

- ✓ Create and run locally two microservices that communicate securely using auto-mTLS and reliably using built-in retries via [Dapr's Service Invocation API](#).
- ✓ Deploy the application to Azure Container Apps via the Azure Developer CLI with the provided Bicep.

The sample service invocation project includes:

1. A `checkout` service that uses Dapr's HTTP proxying capability on a loop to invoke a request on the `order-processor` service.
2. A `order-processor` service that receives the request from the `checkout` service.



Prerequisites

- Install [Azure Developer CLI](#)
- [Install](#) and [init](#) Dapr
- [Docker Desktop](#)
- Install [Git](#)

Run the Node.js applications locally

Before deploying the application to Azure Container Apps, start by running the `order-processor` and `checkout` services locally with Dapr.

Prepare the project

1. Clone the [sample Dapr application](#) to your local machine.

```
Bash
```

```
git clone https://github.com/Azure-Samples/svc-invoke-dapr-nodejs.git
```

2. Navigate into the sample's root directory.

```
Bash
```

```
cd svc-invoke-dapr-nodejs
```

Run the Dapr applications using the Dapr CLI

Start by running the `order-processor` service.

1. From the sample's root directory, change directories to `order-processor`.

```
Bash
```

```
cd order-processor
```

2. Install the dependencies.

```
Bash
```

```
npm install
```

3. Run the `order-processor` service with Dapr.

```
Bash
```

```
dapr run --app-port 5001 --app-id order-processor --app-protocol http -  
-dapr-http-port 3501 -- npm start
```

4. In a new terminal window, from the sample's root directory, navigate to the `checkout` caller service.

```
Bash
```

```
cd checkout
```

5. Install the dependencies.

```
Bash
```

```
npm install
```

6. Run the `checkout` service with Dapr.

```
Bash
```

```
dapr run --app-id checkout --app-protocol http --dapr-http-port 3500 -  
- npm start
```

Expected output

In both terminals, the `checkout` service is calling orders to the `order-processor` service in a loop.

`checkout` output:

```
== APP == Order passed: {"orderId":1}  
== APP == Order passed: {"orderId":2}  
== APP == Order passed: {"orderId":3}  
== APP == Order passed: {"orderId":4}  
== APP == Order passed: {"orderId":5}  
== APP == Order passed: {"orderId":6}  
== APP == Order passed: {"orderId":7}  
== APP == Order passed: {"orderId":8}  
== APP == Order passed: {"orderId":9}  
== APP == Order passed: {"orderId":10}  
== APP == Order passed: {"orderId":11}  
== APP == Order passed: {"orderId":12}  
== APP == Order passed: {"orderId":13}  
== APP == Order passed: {"orderId":14}  
== APP == Order passed: {"orderId":15}  
== APP == Order passed: {"orderId":16}  
== APP == Order passed: {"orderId":17}  
== APP == Order passed: {"orderId":18}  
== APP == Order passed: {"orderId":19}  
== APP == Order passed: {"orderId":20}
```

`order-processor` output:

```
== APP == Order received: { orderId: 1 }  
== APP == Order received: { orderId: 2 }
```

```
== APP == Order received: { orderId: 3 }
== APP == Order received: { orderId: 4 }
== APP == Order received: { orderId: 5 }
== APP == Order received: { orderId: 6 }
== APP == Order received: { orderId: 7 }
== APP == Order received: { orderId: 8 }
== APP == Order received: { orderId: 9 }
== APP == Order received: { orderId: 10 }
== APP == Order received: { orderId: 11 }
== APP == Order received: { orderId: 12 }
== APP == Order received: { orderId: 13 }
== APP == Order received: { orderId: 14 }
== APP == Order received: { orderId: 15 }
== APP == Order received: { orderId: 16 }
== APP == Order received: { orderId: 17 }
== APP == Order received: { orderId: 18 }
== APP == Order received: { orderId: 19 }
== APP == Order received: { orderId: 20 }
```

7. Press `Cmd/Ctrl` + `c` in both terminals to exit out of the service-to-service invocation.

Deploy the Dapr application template using Azure Developer CLI

Deploy the Dapr application to Azure Container Apps using [azd](#).

Prepare the project

In a new terminal window, navigate into the sample's root directory.

Bash

```
cd svc-invoke-dapr-nodejs
```

Provision and deploy using Azure Developer CLI

1. Run `azd init` to initialize the project.

Azure Developer CLI

```
azd init
```

2. When prompted in the terminal, provide the following parameters.

Parameter	Description
Environment Name	Prefix for the resource group created to hold all Azure resources.
Azure Location	The Azure location for your resources.
Azure Subscription	The Azure subscription for your resources.

3. Run `azd up` to provision the infrastructure and deploy the Dapr application to Azure Container Apps in a single command.

Azure Developer CLI

```
azd up
```

This process may take some time to complete. As the `azd up` command completes, the CLI output displays two Azure portal links to monitor the deployment progress. The output also demonstrates how `azd up`:

- Creates and configures all necessary Azure resources via the provided Bicep files in the `./infra` directory using `azd provision`. Once provisioned by Azure Developer CLI, you can access these resources via the Azure portal. The files that provision the Azure resources include:
 - `main.parameters.json`
 - `main.bicep`
 - An `app` resources directory organized by functionality
 - A `core` reference library that contains the Bicep modules used by the `azd template`
- Deploys the code using `azd deploy`

Expected output

Azure Developer CLI

```
Initializing a new project (azd init)
```

```
Provisioning Azure resources (azd provision)
Provisioning Azure resources can take some time
```

```
You can view detailed progress in the Azure Portal:
https://portal.azure.com
```

```
(✓) Done: Resource group: resource-group-name
(✓) Done: Log Analytics workspace: log-analytics-name
(✓) Done: Application Insights: app-insights-name
```

```
(✓) Done: Portal dashboard: dashboard-name
(✓) Done: Container Apps Environment: container-apps-env-name
(✓) Done: Container App: ca-checkout-name
(✓) Done: Container App: ca-order-processor-name

Deploying services (azd deploy)

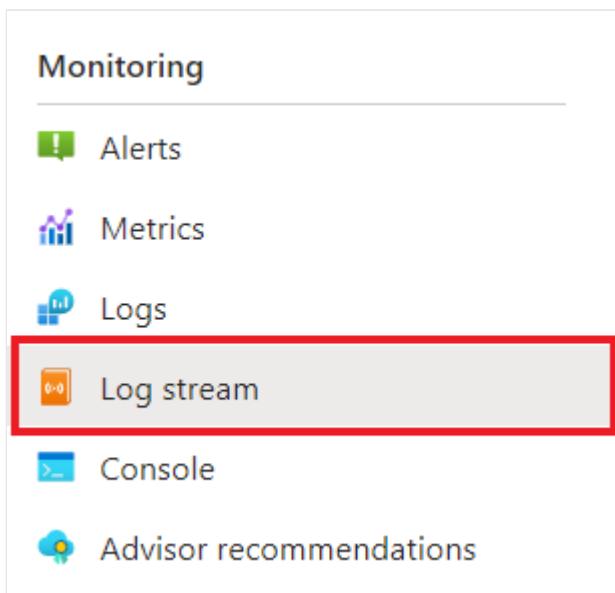
(✓) Done: Deploying service api
- Endpoint: https://ca-order-processor-
name.eastus.azurecontainerapps.io/
(✓) Done: Deploying service worker

SUCCESS: Your Azure app has been deployed!
You can view the resources created under the resource group resource-
group-name in Azure Portal:
https://portal.azure.com/#@/resource/subscriptions/<your-azure-subscription>/resourceGroups/resource-group-name/overview
```

Confirm successful deployment

In the Azure portal, verify the `checkout` service is passing orders to the `order-processor` service.

1. Copy the `checkout` container app's name from the terminal output.
2. Navigate to the [Azure portal](#) and search for the container app resource by name.
3. In the Container Apps dashboard, select **Monitoring > Log stream**.



4. Confirm the `checkout` container is logging the same output as in the terminal earlier.

```
Connecting...
2023-02-17T17:18:44.44752  Connecting to the container 'checkout'...
2023-02-17T17:18:44.46918  Successfully Connected to container: 'checkout'

2023-02-17T17:18:10.857906291Z Order passed: {"orderId":1}
2023-02-17T17:18:11.863507650Z Order passed: {"orderId":2}
2023-02-17T17:18:12.868358339Z Order passed: {"orderId":3}
2023-02-17T17:18:13.873600279Z Order passed: {"orderId":4}
2023-02-17T17:18:14.878059227Z Order passed: {"orderId":5}
2023-02-17T17:18:15.883034650Z Order passed: {"orderId":6}
2023-02-17T17:18:16.889526921Z Order passed: {"orderId":7}
2023-02-17T17:18:17.894321469Z Order passed: {"orderId":8}
2023-02-17T17:18:18.897729650Z Order passed: {"orderId":9}
2023-02-17T17:18:19.902510277Z Order passed: {"orderId":10}
2023-02-17T17:18:20.908565025Z Order passed: {"orderId":11}
2023-02-17T17:18:21.913487198Z Order passed: {"orderId":12}
2023-02-17T17:18:22.919039820Z Order passed: {"orderId":13}
2023-02-17T17:18:23.924864640Z Order passed: {"orderId":14}
2023-02-17T17:18:24.930955183Z Order passed: {"orderId":15}
2023-02-17T17:18:25.935990278Z Order passed: {"orderId":16}
2023-02-17T17:18:26.941350932Z Order passed: {"orderId":17}
2023-02-17T17:18:27.946330327Z Order passed: {"orderId":18}
2023-02-17T17:18:28.950993193Z Order passed: {"orderId":19}
2023-02-17T17:18:29.954949436Z Order passed: {"orderId":20}
2023-02-17T17:18:50.979894555Z Order passed: {"orderId":1}
2023-02-17T17:18:51.985474976Z Order passed: {"orderId":2}
2023-02-17T17:18:52.990299145Z Order passed: {"orderId":3}
2023-02-17T17:18:53.993957362Z Order passed: {"orderId":4}
2023-02-17T17:18:54.998628641Z Order passed: {"orderId":5}
```

5. Do the same for the `order-processor` service.

```
Connecting...
2023-02-17T17:24:16.58999  Connecting to the container 'order-processor'...
2023-02-17T17:24:16.61026  Successfully Connected to container: 'order-processor'
2023-02-17T17:23:36.805887872Z Order received: { orderId: 6 }
2023-02-17T17:23:37.810033680Z Order received: { orderId: 7 }
2023-02-17T17:23:38.814930622Z Order received: { orderId: 8 }
2023-02-17T17:23:39.819613502Z Order received: { orderId: 9 }
2023-02-17T17:23:40.823327552Z Order received: { orderId: 10 }
2023-02-17T17:23:41.828664748Z Order received: { orderId: 11 }
2023-02-17T17:23:42.833989666Z Order received: { orderId: 12 }
2023-02-17T17:23:43.837952569Z Order received: { orderId: 13 }
2023-02-17T17:23:44.842654830Z Order received: { orderId: 14 }
2023-02-17T17:23:45.848068174Z Order received: { orderId: 15 }
2023-02-17T17:23:46.852792769Z Order received: { orderId: 16 }
2023-02-17T17:23:47.857529824Z Order received: { orderId: 17 }
2023-02-17T17:23:48.862937841Z Order received: { orderId: 18 }
2023-02-17T17:23:49.867843896Z Order received: { orderId: 19 }
2023-02-17T17:23:50.872403776Z Order received: { orderId: 20 }
2023-02-17T17:24:11.895782198Z Order received: { orderId: 1 }
2023-02-17T17:24:12.899940853Z Order received: { orderId: 2 }
2023-02-17T17:24:13.904718259Z Order received: { orderId: 3 }
2023-02-17T17:24:14.910750405Z Order received: { orderId: 4 }
2023-02-17T17:24:15.915615615Z Order received: { orderId: 5 }
2023-02-17T17:24:16.919299658Z Order received: { orderId: 6 }
2023-02-17T17:24:17.924038953Z Order received: { orderId: 7 }
2023-02-17T17:24:18.927722608Z Order received: { orderId: 8 }
2023-02-17T17:24:19.933345036Z Order received: { orderId: 9 }
2023-02-17T17:24:20.939099248Z Order received: { orderId: 10 }
```

What happened?

Upon successful completion of the `azd up` command:

- Azure Developer CLI provisioned the Azure resources referenced in the [sample project's ./infra directory](#) to the Azure subscription you specified. You can now view those Azure resources via the Azure portal.
- The app deployed to Azure Container Apps. From the portal, you can browse the fully functional app.

Clean up resources

If you're not going to continue to use this application, delete the Azure resources you've provisioned with the following command:

```
Azure Developer CLI
```

```
azd down
```

Next steps

- Learn more about [deploying Dapr applications to Azure Container Apps](#).
- [Enable token authentication for Dapr requests](#).
- Learn more about [Azure Developer CLI](#) and [making your applications compatible with azd](#).

Tutorial: Deploy a Dapr application with GitHub Actions for Azure Container Apps

Article • 05/03/2023

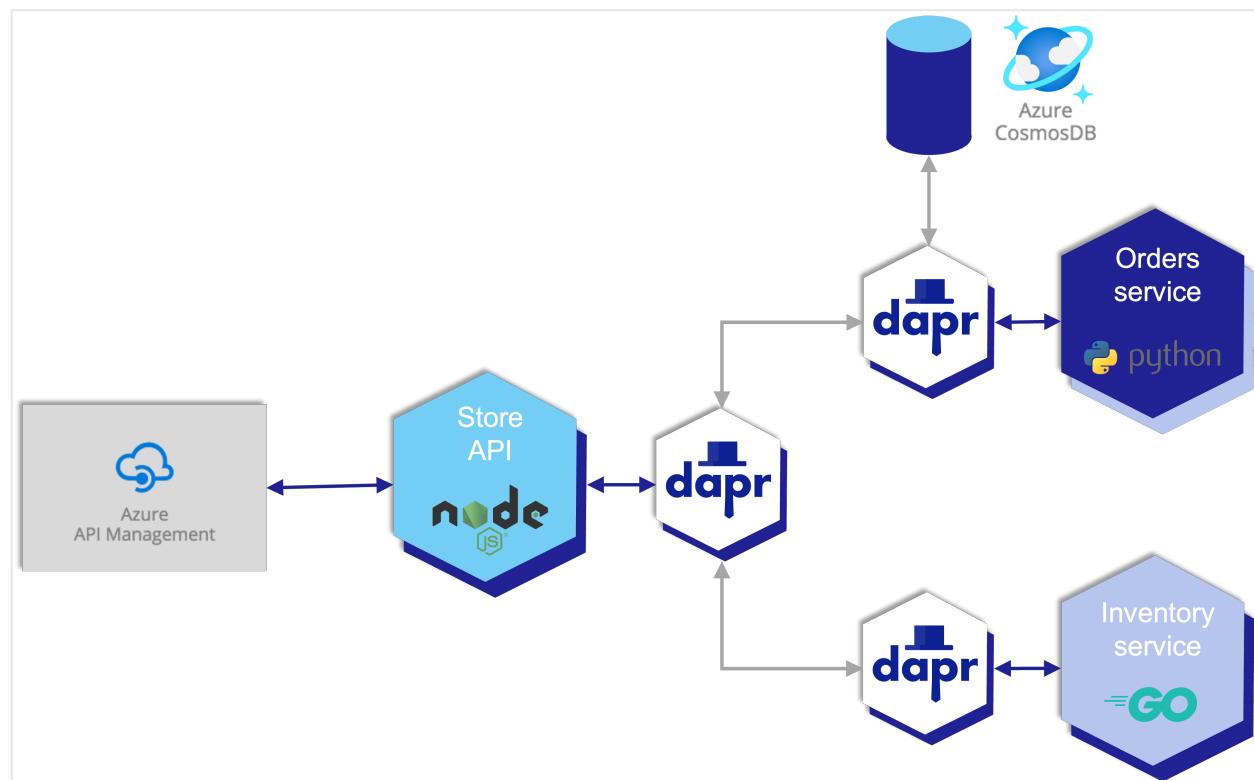
[GitHub Actions](#) gives you the flexibility to build an automated software development lifecycle workflow. In this tutorial, you'll see how revision-scope changes to a Container App using [Dapr](#) can be deployed using a GitHub Actions workflow.

Dapr is an open source project that helps developers with the inherent challenges presented by distributed applications, such as state management and service invocation. Azure Container Apps provides a managed experience of the core Dapr APIs.

In this tutorial, you'll:

- ✓ Configure a GitHub Actions workflow for deploying the end-to-end solution to Azure Container Apps.
- ✓ Modify the source code with a [revision-scope change](#) to trigger the Build and Deploy GitHub workflow.
- ✓ Learn how revisions are created for container apps in multi-revision mode.

The [sample solution](#) consists of three Dapr-enabled microservices and uses Dapr APIs for service-to-service communication and state management.



ⓘ Note

This tutorial focuses on the solution deployment outlined below. If you're interested in building and running the solution on your own, [follow the README instructions within the repo ↗](#).

Prerequisites

- An Azure account with an active subscription.
 - [Create an account for free ↗](#).
- Contributor or Owner permissions on the Azure subscription.
- A GitHub account.
 - If you don't have one, sign up for [free ↗](#).
- [Install Git ↗](#).
- [Install the Azure CLI](#).

Set up the environment

In the console, set the following environment variables:

Azure CLI

Replace <PLACEHOLDERS> with your values.

Bash

```
RESOURCE_GROUP="my-containerapp-store"
LOCATION="canadacentral"
GITHUB_USERNAME=""
SUBSCRIPTION_ID=""
```

Sign in to Azure from the CLI using the following command, and follow the prompts in your browser to complete the authentication process.

Azure CLI

Azure CLI

```
az login
```

Ensure you're running the latest version of the CLI via the upgrade command.

```
Azure CLI
Azure CLI
az upgrade
```

Now that you've validated your Azure CLI setup, bring the application code to your local machine.

Get application code

1. Navigate to the [sample GitHub repo](#) and select **Fork** in the top-right corner of the page.
2. Use the following [git](#) command with your GitHub username to clone **your fork** of the repo to your development environment:

```
Azure CLI
git
git clone https://github.com/$GITHUB_USERNAME/container-apps-store-api-microservice.git
```

3. Navigate into the cloned directory.

```
Bash
cd container-apps-store-api-microservice
```

The repository includes the following resources:

- The source code for each application
- Deployment manifests
- A GitHub Actions workflow file

Deploy Dapr solution using GitHub Actions

The GitHub Actions workflow YAML file in the `/.github/workflows/` folder executes the following steps in the background as you work through this tutorial:

Section	Tasks
Authentication	Log in to a private container registry (GitHub Container Registry)
Build	Build & push the container images for each microservice
Authentication	Log in to Azure
Deploy using bicep	1. Create a resource group 2. Deploy Azure Resources for the solution using bicep

The following resources are deployed via the bicep template in the `/deploy` path of the repository:

- Log Analytics workspace
- Application Insights
- Container apps environment
- Order service container app
- Inventory container app
- Azure Cosmos DB

Create a service principal

The workflow requires a [service principal](#) to authenticate to Azure. In the console, run the following command and replace `<SERVICE_PRINCIPAL_NAME>` with your own unique value.

Azure CLI

```
az ad sp create-for-rbac \
--name <SERVICE_PRINCIPAL_NAME> \
--role "contributor" \
--scopes /subscriptions/$SUBSCRIPTION_ID \
--sdk-auth
```

The output is the role assignment credentials that provide access to your resource. The command should output a JSON object similar to:

JSON

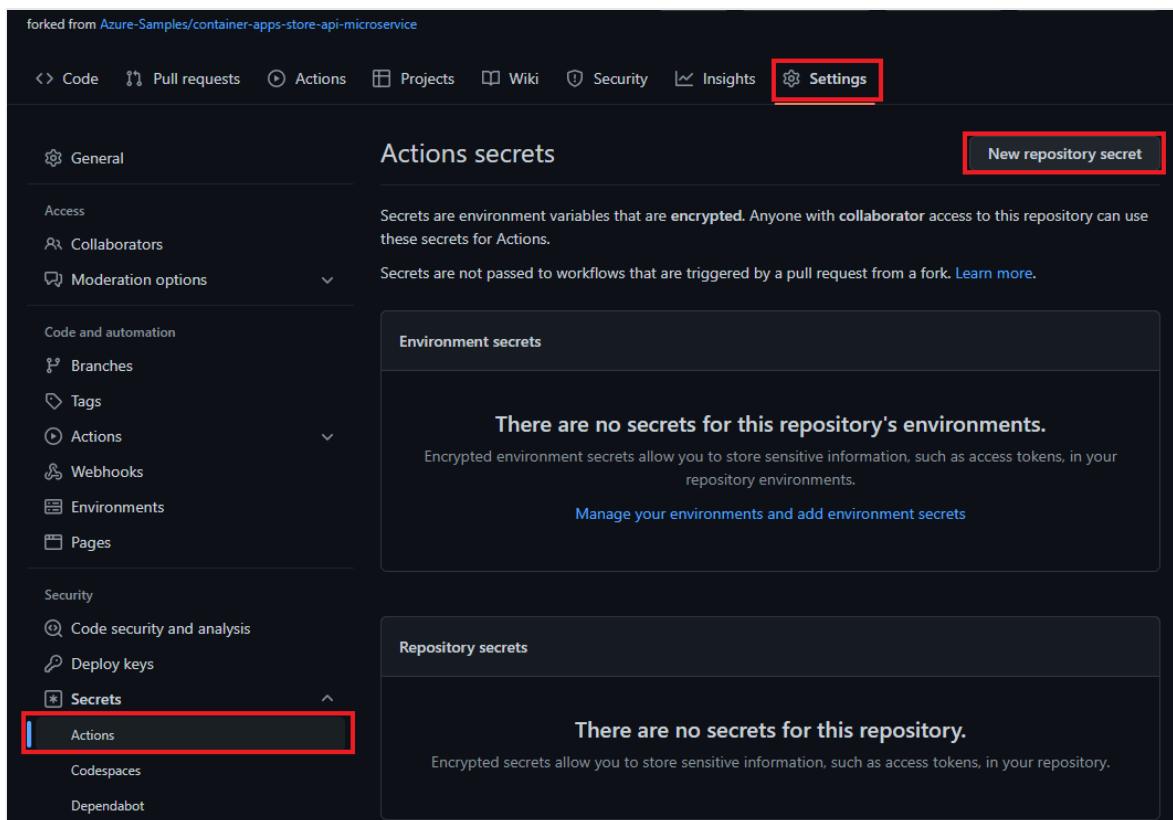
```
{
  "clientId": "<GUID>",
  "clientSecret": "<GUID>",
  "subscriptionId": "<GUID>",
  "tenantId": "<GUID>"
  (...)

}
```

Copy the JSON object output and save it to a file on your machine. You use this file as you authenticate from GitHub.

Configure GitHub Secrets

1. While in GitHub, browse to your forked repository for this tutorial.
2. Select the **Settings** tab.
3. Select **Secrets > Actions**.
4. On the **Actions secrets** page, select **New repository secret**.



5. Create the following secrets:

Name	Value
AZURE_CREDENTIALS	The JSON output you saved earlier from the service principal creation

Name	Value
RESOURCE_GROUP	Set as my-containerapp-store

The screenshot shows the 'Repository secrets' section in GitHub. It lists two secrets:

- AZURE_CREDENTIALS**: Updated 12 seconds ago. Buttons: **Update** (blue), **Remove** (red).
- RESOURCE_GROUP**: Updated now. Buttons: **Update** (blue), **Remove** (red).

Trigger the GitHub Action

To build and deploy the initial solution to Azure Container Apps, run the "Build and deploy" workflow.

1. Open the **Actions** tab in your GitHub repository.
2. In the left side menu, select the **Build and Deploy** workflow.

The screenshot shows the GitHub Actions tab for the repository 'hhunter-ms / container-apps-store-api-microservice'. The 'Build and Deploy' workflow is selected and highlighted with a red box. On the right, there is a modal window for running the workflow. The 'Run workflow' button is highlighted with a red box.

3. Select **Run workflow**.
4. In the prompt, leave the *Use workflow from* value as **Branch: main**.
5. Select **Run workflow**.

Verify the deployment

After the workflow successfully completes, verify the application is running in Azure Container Apps.

1. Navigate to the [Azure portal](#).
2. In the search field, enter **my-containerapp-store** and select the **my-containerapp-store** resource group.

The screenshot shows the Microsoft Azure portal interface. At the top, there's a search bar with the text "my-containerapp-store". Below the search bar, there are several filter buttons: "All" (selected), "Resource Groups (1)", "Documentation (6)", "Services (0)", "Resources (0)", "Marketplace (0)", and "Azure Active Directory (0)". On the left, there's a sidebar with "Azure services" and icons for "Create a resource", "API Connections", "App Services", and "Function App". The main area shows a "Resource Groups" section with a list containing "my-containerapp-store", which is highlighted with a red box.

3. Navigate to the container app called **node-app**.

The screenshot shows the "my-containerapp-store" resource group details page. At the top, there's a breadcrumb navigation "Home > my-containerapp-store". Below it, there are buttons for "Create", "Manage view", "Delete resource group", "Refresh", "Export to CSV", "Open query", and "JSON View". A "Essentials" section is expanded. The "Resources" tab is selected, showing a list of resources. The list includes "env-ppuc23jyydbya", "env-ppuc23jyydbya-ai", "env-ppuc23jyydbya-la", "go-app", "node-app" (highlighted with a red box), "python-app", and "store-api-mgmt-ppuc23jyydbya". Each resource entry shows its type, location, and three-dot more options menu.

4. Select the **Application Url**.

The screenshot shows the Azure portal interface for a Container App named 'node-app'. The 'Essentials' section displays various metadata such as Resource group, Location, Subscription ID, and Tags. The 'Application Url' field, which contains the value <https://node-app.thankfulmoss-7faf2215.canadacent...>, is highlighted with a red box.

5. Ensure the application was deployed successfully by creating a new order:

a. Enter an **Id** and **Item**.

b. Select **Create**.

The screenshot shows a web application titled 'Container App Demo'. The main heading is 'Welcome to Container App Demo'. Below it, there is a section titled 'Create new order'. It contains two input fields: 'Id' with the value '0123' and 'Item' with the value 'widget'. At the bottom is a large green button labeled 'Create'.

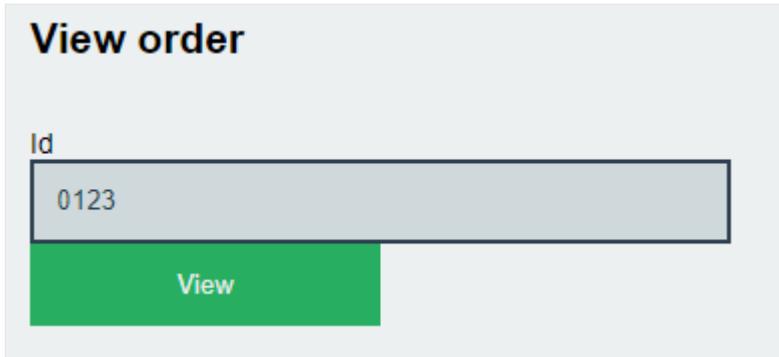
If the order is persisted, you're redirected to a page that says "Order created!"

6. Navigate back to the previous page.

7. View the item you created via the **View Order** form:

a. Enter the item **Id**.

b. Select **View**.



The page is redirected to a new page displaying the order object.

8. In the Azure portal, navigate to **Application > Revision Management** in the **node-app** container.

At this point, only one revision is available for this app.

Name	Date created	Provision Status	Traffic	Active
node-app--f413unv	9/2/2022, 2:28:...	Provisioned	100 %	<input checked="" type="checkbox"/>

Modify the source code to trigger a new revision

Container Apps run in single-revision mode by default. In the Container Apps bicep module, we explicitly set the revision mode to multiple. This means that once the source code is changed and committed, the GitHub build/deploy workflow builds and pushes a new container image to GitHub Container Registry. Changing the container image is considered a [revision-scope](#) change and results in a new container app revision.

Note

Application-scope changes do not create a new revision.

To demonstrate the inner-loop experience for creating revisions via GitHub actions, you'll make a change to the frontend application and commit this change to your repo.

1. Return to the console, and navigate into the `node-service/views` directory in the forked repository.

```
Azure CLI
Bash
cd node-service/views
```

2. Open the `index.jade` file in your editor of choice.

```
Azure CLI
Bash
code index.jade .
```

3. At the bottom of the file, uncomment the following code to enable deleting an order from the Dapr state store.

```
jade
h2= 'Delete Order'
br
br
form(action='/order/delete', method='post')
  div.input
    span.label Id
      input(type='text', name='id', placeholder='foo',
required='required')
  div.actions
    input(type='submit', value='View')
```

4. Stage the changes and push to the `main` branch of your fork using git.

```
Azure CLI
git
git add .
git commit -m '<commit message>'
git push origin main
```

View the new revision

1. In the GitHub UI of your fork, select the **Actions** tab to verify the GitHub Build and Deploy workflow is running.
2. Once the workflow is complete, navigate to the **my-containerapp-store** resource group in the Azure portal.
3. Select the **node-app** container app.
4. In the left side menu, select **Application > Revision Management**.

The screenshot shows the Azure Container Apps blade for the 'node-app' container app. The top navigation bar includes 'Home > my-containerapp-store >'. Below it is the app name 'node-app' with a purple icon and the text 'Container App'. A search bar with the placeholder 'Search (Ctrl+ /)' is present. The left sidebar menu has three items: 'Revision management' (which is highlighted with a red border), 'Containers', and 'Scale'.

Since our container app is in **multiple revision mode**, Container Apps created a new revision, and automatically sets it to **active** with 100% traffic.

The screenshot shows the 'Revision management' blade for the 'node-app' container app. At the top, there are buttons for 'Create new revision', 'Save', 'Refresh', 'Choose revision mode', and 'Send us your feedback'. Below that, a message says 'Each revision is a variation of your container app that can have different settings for traffic allocations, autoscaling or Dapr. Create a new revision to make changes to your app. Start by selecting any existing revision.' A search bar is available. On the right, there is a checkbox for 'Show inactive revisions'. The main area is a table with columns: Name, Date created, Provision Status, Label, Traffic, and Active. Two revisions are listed:

Name	Date created	Provision Status	Label	Traffic	Active
node-app--g0y1ud8	9/2/2022, 2:45...	Provisioned		100 %	<input checked="" type="checkbox"/>
node-app--f413unv	9/2/2022, 2:28...	Provisioned		0 %	<input checked="" type="checkbox"/>

5. Select each revision in the **Revision management** table to view revision details.

The screenshot shows the 'Revision details' page for the 'node-app' container app. On the left, there's a sidebar with a search bar and a table listing revisions. The main area shows detailed information for the current revision:

Revision name	node-app--g0y1ud8
Revision URL	node-app--g0y1ud8.blackdune-bebf2a1f.ca...
Status	Active
Time created	9/2/2022, 2:45:20 PM
Traffic	100%
Containers	View details
Scale	View details
Console logs	View details
System logs	View details

6. View the new revision in action by refreshing the node-app UI.
7. Test the application further by deleting the order you created in the container app.

The screenshot shows a 'Delete Order' page. It has a text input field labeled 'Id' containing '0123' and a large green button labeled 'View'.

The page is redirected to a page indicating that the order is removed.

Clean up resources

Once you've finished the tutorial, run the following command to delete your resource group, along with all the resources you created in this tutorial.

Azure CLI

```
az group delete \
--resource-group $RESOURCE_GROUP
```

Next steps

Learn more about how [Dapr integrates with Azure Container Apps](#).

Tutorial: Deploy a background processing application with Azure Container Apps

Article • 03/08/2023

Using Azure Container Apps allows you to deploy applications without requiring the exposure of public endpoints. By using Container Apps scale rules, the application can scale out and in based on the Azure Storage queue length. When there are no messages on the queue, the container app scales in to zero.

You learn how to:

- ✓ Create a Container Apps environment to deploy your container apps
- ✓ Create an Azure Storage Queue to send messages to the container app
- ✓ Deploy your background processing application as a container app
- ✓ Verify that the queue messages are processed by the container app

Setup

To begin, sign in to Azure. Run the following command, and follow the prompts to complete the authentication process.

Bash

Azure CLI

```
az login
```

Bash

Next, install the Azure Container Apps extension for the CLI.

Azure CLI

```
az extension add --name containerapp --upgrade
```

Now that the current extension or module is installed, register the `Microsoft.App` namespace.

 **Note**

Azure Container Apps resources have migrated from the `Microsoft.Web` namespace to the `Microsoft.App` namespace. Refer to [Namespace migration from Microsoft.Web to Microsoft.App in March 2022](#) for more details.

Bash

Azure CLI

```
az provider register --namespace Microsoft.App
```

Register the `Microsoft.OperationalInsights` provider for the Azure Monitor Log Analytics workspace if you have not used it before.

Bash

Azure CLI

```
az provider register --namespace Microsoft.OperationalInsights
```

Next, set the following environment variables:

Bash

Azure CLI

```
RESOURCE_GROUP="my-container-apps"  
LOCATION="canadacentral"  
CONTAINERAPPS_ENVIRONMENT="my-environment"
```

With these variables defined, you can create a resource group to organize the services related to your new container app.

Bash

Azure CLI

```
az group create \
--name $RESOURCE_GROUP \
--location $LOCATION
```

With the CLI upgraded and a new resource group available, you can create a Container Apps environment and deploy your container app.

Create an environment

An environment in Azure Container Apps creates a secure boundary around a group of container apps. Container Apps deployed to the same environment are deployed in the same virtual network and write logs to the same Log Analytics workspace.

Individual container apps are deployed to an Azure Container Apps environment. To create the environment, run the following command:

Bash

Azure CLI

```
az containerapp env create \
--name $CONTAINERAPPS_ENVIRONMENT \
--resource-group $RESOURCE_GROUP \
--location "$LOCATION"
```

Set up a storage queue

Begin by defining a name for the storage account. Storage account names must be *unique within Azure* and be from 3 to 24 characters in length containing numbers and lowercase letters only.

Bash

Bash

```
STORAGE_ACCOUNT_NAME=<STORAGE_ACCOUNT_NAME>
```

Create an Azure Storage account.

Bash

Azure CLI

```
az storage account create \
--name $STORAGE_ACCOUNT_NAME \
--resource-group $RESOURCE_GROUP \
--location "$LOCATION" \
--sku Standard_RAGRS \
--kind StorageV2
```

Next, get the connection string for the queue.

Bash

Azure CLI

```
QUEUE_CONNECTION_STRING=`az storage account show-connection-string -g
$RESOURCE_GROUP --name $STORAGE_ACCOUNT_NAME --query connectionString --
out json | tr -d '"'`
```

Now you can create the message queue.

Bash

Azure CLI

```
az storage queue create \
--name "myqueue" \
--account-name $STORAGE_ACCOUNT_NAME \
--connection-string $QUEUE_CONNECTION_STRING
```

Finally, you can send a message to the queue.

Bash

Azure CLI

```
az storage message put \
--content "Hello Queue Reader App" \
--queue-name "myqueue" \
--connection-string $QUEUE_CONNECTION_STRING
```

Deploy the background application

Create a file named *queue.json* and paste the following configuration code into the file.

JSON

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2019-08-  
01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "location": {  
            "defaultValue": "canadacentral",  
            "type": "String"  
        },  
        "environment_name": {  
            "type": "String"  
        },  
        "queueconnection": {  
            "type": "secureString"  
        }  
    },  
    "variables": {},  
    "resources": [  
        {  
            "name": "queuereader",  
            "type": "Microsoft.App/containerApps",  
            "apiVersion": "2022-03-01",  
            "kind": "containerapp",  
            "location": "[parameters('location')]",  
            "properties": {  
                "managedEnvironmentId": "  
[resourceId('Microsoft.App/managedEnvironments',  
parameters('environment_name'))]",  
                "configuration": {  
                    "activeRevisionsMode": "single",  
                    "secrets": [  
                        {  
                            "name": "queueconnection",  
                            "value": "[parameters('queueconnection')]"  
                        }]  
                },  
                "template": {  
                    "containers": [  
                        {  
                            "image": "mcr.microsoft.com/azuredocs/containerapps-  
queuereader",  
                            "name": "queuereader",  
                            "env": [  
                                {  
                                    "name": "QueueName",  
                                    "value": "[parameters('queueconnection')]"  
                                }]  
                        }]  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```
        "value": "myqueue"
    },
{
    "name": "QueueConnectionString",
    "secretRef": "queueconnection"
}
]
}
],
"scale": {
    "minReplicas": 1,
    "maxReplicas": 10,
    "rules": [
        {
            "name": "myqueuerule",
            "azureQueue": {
                "queueName": "myqueue",
                "queueLength": 100,
                "auth": [
                    {
                        "secretRef": "queueconnection",
                        "triggerParameter": "connection"
                    }
                ]
            }
        }
    ]
}
}
}
}]
```

Now you can create and deploy your container app.

Bash

Azure CLI

```
az deployment group create --resource-group "$RESOURCE_GROUP" \
--template-file ./queue.json \
--parameters \
environment_name="$CONTAINERAPPS_ENVIRONMENT" \
queueconnection="$QUEUE_CONNECTION_STRING" \
location="$LOCATION"
```

This command deploys the demo application from the public container image called mcr.microsoft.com/azuredocs/containerapps-queuereader and sets secrets and

environments variables used by the application.

The application scales out to 10 replicas based on the queue length as defined in the `scale` section of the ARM template.

Verify the result

The container app runs as a background process. As messages arrive from the Azure Storage Queue, the application creates log entries in Log analytics. You must wait a few minutes for the analytics to arrive for the first time before you're able to query the logged data.

Run the following command to see logged messages. This command requires the Log analytics extension, so accept the prompt to install extension when requested.

Bash

Azure CLI

```
LOG_ANALYTICS_WORKSPACE_CLIENT_ID=`az containerapp env show --name $CONTAINERAPPS_ENVIRONMENT --resource-group $RESOURCE_GROUP --query properties.appLogsConfiguration.logAnalyticsConfiguration.customerId --out tsv`  
  
az monitor log-analytics query \  
  --workspace $LOG_ANALYTICS_WORKSPACE_CLIENT_ID \  
  --analytics-query "ContainerAppConsoleLogs_CL | where ContainerAppName_s == 'queuereader' and Log_s contains 'Message ID' | project Time=TimeGenerated, AppName=ContainerAppName_s, Revision=RevisionName_s, Container=ContainerName_s, Message=Log_s | take 5" \  
  --out table
```



Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Clean up resources

Once you're done, run the following command to delete the resource group that contains your Container Apps resources.

 **Caution**

The following command deletes the specified resource group and all resources contained within it. If resources outside the scope of this tutorial exist in the specified resource group, they will also be deleted.

Bash

Azure CLI

```
az group delete \
--resource-group $RESOURCE_GROUP
```

Tutorial: Deploy an event-driven job with Azure Container Apps

Article • 05/23/2023

Azure Container Apps [jobs](#) allow you to run containerized tasks that execute for a finite duration and exit. You can trigger a job execution manually, on a schedule, or based on events. Jobs are best suited to for tasks such as data processing, machine learning, or any scenario that requires serverless ephemeral compute resources.

In this tutorial, you learn how to work with [event-driven jobs](#).

- ✓ Create a Container Apps environment to deploy your container apps
- ✓ Create an Azure Storage Queue to send messages to the container app
- ✓ Build a container image that runs a job
- ✓ Deploy the job to the Container Apps environment
- ✓ Verify that the queue messages are processed by the container app

The job you create starts an execution for each message that is sent to an Azure Storage Queue. Each job execution runs a container that performs the following steps:

1. Dequeues one message from the queue.
2. Logs the message to the job execution logs.
3. Deletes the message from the queue.
4. Exits.

The source code for the job you run in this tutorial is available in an Azure Samples [GitHub repository](#) ↗.

Prerequisites

- An Azure account with an active subscription.
 - If you don't have one, you [can create one for free](#) ↗.
- Install the [Azure CLI](#).
- See [Jobs preview limitations](#) for a list of limitations.

Setup

1. To sign in to Azure from the CLI, run the following command and follow the prompts to complete the authentication process.

```
Azure CLI
```

```
az login
```

2. Ensure you're running the latest version of the CLI via the upgrade command.

```
Azure CLI
```

```
az upgrade
```

3. Install the latest version of the Azure Container Apps CLI extension.

```
Azure CLI
```

```
az extension add --name containerapp --upgrade
```

4. Register the `Microsoft.App` and `Microsoft.OperationalInsights` namespaces if you haven't already registered them in your Azure subscription.

```
Azure CLI
```

```
az provider register --namespace Microsoft.App  
az provider register --namespace Microsoft.OperationalInsights
```

5. Now that your Azure CLI setup is complete, you can define the environment variables that are used throughout this article.

```
Azure CLI
```

```
RESOURCE_GROUP="jobs-quickstart"  
LOCATION="northcentralus"  
ENVIRONMENT="env-jobs-quickstart"  
JOB_NAME="my-job"
```

Create a Container Apps environment

The Azure Container Apps environment acts as a secure boundary around container apps and jobs so they can share the same network and communicate with each other.

1. Create a resource group using the following command.

```
Azure CLI
```

```
az group create \
--name "$RESOURCE_GROUP" \
--location "$LOCATION"
```

2. Create the Container Apps environment using the following command.

Azure CLI

```
az containerapp env create \
--name "$ENVIRONMENT" \
--resource-group "$RESOURCE_GROUP" \
--location "$LOCATION"
```

Set up a storage queue

The job uses an Azure Storage queue to receive messages. In this section, you create a storage account and a queue.

1. Define a name for your storage account.

Bash

```
STORAGE_ACCOUNT_NAME=<STORAGE_ACCOUNT_NAME>
QUEUE_NAME="myqueue"
```

Replace `<STORAGE_ACCOUNT_NAME>` with a unique name for your storage account.

Storage account names must be *unique within Azure* and be from 3 to 24 characters in length containing numbers and lowercase letters only.

2. Create an Azure Storage account.

Azure CLI

```
az storage account create \
--name "$STORAGE_ACCOUNT_NAME" \
--resource-group "$RESOURCE_GROUP" \
--location "$LOCATION" \
--sku Standard_LRS \
--kind StorageV2
```

3. Save the queue's connection string into a variable.

Bash

```
QUEUE_CONNECTION_STRING=`az storage account show-connection-string -g $RESOURCE_GROUP --name $STORAGE_ACCOUNT_NAME --query connectionString --output tsv`
```

4. Create the message queue.

Azure CLI

```
az storage queue create \
--name "$QUEUE_NAME" \
--account-name "$STORAGE_ACCOUNT_NAME" \
--connection-string "$QUEUE_CONNECTION_STRING"
```

Build and deploy the job

To deploy the job, you must first build a container image for the job and push it to a registry. Then, you can deploy the job to the Container Apps environment.

1. Define a name for your container image and registry.

Bash

```
CONTAINER_IMAGE_NAME="queue-reader-job:1.0"
CONTAINER_REGISTRY_NAME=<CONTAINER_REGISTRY_NAME>
```

Replace `<CONTAINER_REGISTRY_NAME>` with a unique name for your container registry. Container registry names must be *unique within Azure* and be from 5 to 50 characters in length containing numbers and lowercase letters only.

2. Create a container registry.

Azure CLI

```
az acr create \
--name "$CONTAINER_REGISTRY_NAME" \
--resource-group "$RESOURCE_GROUP" \
--location "$LOCATION" \
--sku Basic \
--admin-enabled true
```

3. The source code for the job is available on [GitHub](#). Run the following command to clone the repository and build the container image in the cloud using the `az acr build` command.

Azure CLI

```
az acr build \
    --registry "$CONTAINER_REGISTRY_NAME" \
    --image "$CONTAINER_IMAGE_NAME" \
    "https://github.com/Azure-Samples/container-apps-event-driven-jobs-tutorial.git"
```

The image is now available in the container registry.

4. Create a job in the Container Apps environment.

Azure CLI

```
az containerapp job create \
    --name "$JOB_NAME" \
    --resource-group "$RESOURCE_GROUP" \
    --environment "$ENVIRONMENT" \
    --trigger-type "Event" \
    --replica-timeout "60" \
    --replica-retry-limit "1" \
    --replica-completion-count "1" \
    --parallelism "1" \
    --min-executions "0" \
    --max-executions "10" \
    --polling-interval "60" \
    --scale-rule-name "queue" \
    --scale-rule-type "azure-queue" \
    --scale-rule-metadata "accountName=$STORAGE_ACCOUNT_NAME"
    "queueName=$QUEUE_NAME" "queueLength=1" \
    --scale-rule-auth "connection=connection-string-secret" \
    --image "$CONTAINER_REGISTRY_NAME.azurecr.io/$CONTAINER_IMAGE_NAME"
    \
    --cpu "0.5" \
    --memory "1Gi" \
    --secrets "connection-string-secret=$QUEUE_CONNECTION_STRING" \
    --registry-server "$CONTAINER_REGISTRY_NAME.azurecr.io" \
    --env-vars "AZURE_STORAGE_QUEUE_NAME=$QUEUE_NAME"
    "AZURE_STORAGE_CONNECTION_STRING=secretref:connection-string-secret"
```

The following table describes the key parameters used in the command.

Parameter	Description
--replica-timeout	The maximum duration a replica can execute.
--replica-retry-limit	The number of times to retry a replica.

Parameter	Description
--replica-completion-count	The number of replicas to complete successfully before a job execution is considered successful.
--parallelism	The number of replicas to start per job execution.
--min-executions	The minimum number of job executions to run per polling interval.
--max-executions	The maximum number of job executions to run per polling interval.
--polling-interval	The polling interval at which to evaluate the scale rule.
--scale-rule-name	The name of the scale rule.
--scale-rule-type	The type of scale rule to use.
--scale-rule-metadata	The metadata for the scale rule.
--scale-rule-auth	The authentication for the scale rule.
--secrets	The secrets to use for the job.
--registry-server	The container registry server to use for the job. For an Azure Container Registry, the command automatically configures authentication.
--env-vars	The environment variables to use for the job.

The scale rule configuration defines the event source to monitor. It is evaluated on each polling interval and determines how many job executions to trigger. To learn more, see [Set scaling rules](#).

The event-driven job is now created in the Container Apps environment.

Verify the deployment

The job is configured to evaluate the scale rule every 60 seconds, which checks the number of messages in the queue. For each evaluation period, it starts a new job execution for each message in the queue, up to a maximum of 10 executions.

To verify the job was configured correctly, you can send some messages to the queue, confirm that job executions are started, and the messages are logged to the job execution logs.

1. Send a message to the queue.

Azure CLI

```
az storage message put \
--content "Hello Queue Reader Job" \
--queue-name "$QUEUE_NAME" \
--connection-string "$QUEUE_CONNECTION_STRING"
```

2. List the executions of a job.

Azure CLI

```
az containerapp job execution list \
--name "$JOB_NAME" \
--resource-group "$RESOURCE_GROUP" \
--output json
```

Since the job is configured to evaluate the scale rule every 60 seconds, it may take up to a full minute for the job execution to start. Repeat the command until you see the job execution and its status is `Succeeded`.

3. Run the following commands to see logged messages. These commands require the Log analytics extension, so accept the prompt to install extension when requested.

Azure CLI

```
LOG_ANALYTICS_WORKSPACE_ID=`az containerapp env show --name
$ENVIRONMENT --resource-group $RESOURCE_GROUP --query
properties.appLogsConfiguration.logAnalyticsConfiguration.customerId --
out tsv` 

az monitor log-analytics query \
--workspace "$LOG_ANALYTICS_WORKSPACE_ID" \
--analytics-query "ContainerAppConsoleLogs_CL | where
ContainerJobName_s == '$JOB_NAME' | order by _timestamp_d asc"
```

Until the `ContainerAppConsoleLogs_CL` table is ready, the command returns an error: `BadArgumentError: The request had some invalid properties`. Wait a few minutes and try again.

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Clean up resources

Once you're done, run the following command to delete the resource group that contains your Container Apps resources.

⊗ Caution

The following command deletes the specified resource group and all resources contained within it. If resources outside the scope of this tutorial exist in the specified resource group, they will also be deleted.

Azure CLI

```
az group delete \
--resource-group $RESOURCE_GROUP
```

Next steps

[Container Apps jobs](#)

Tutorial: Create an Azure Files volume mount in Azure Container Apps

Article • 03/08/2023

Learn to write to permanent storage in a container app using an Azure Files storage mount. For more information about storage mounts, see [Use storage mounts in Azure Container Apps](#).

In this tutorial, you learn how to:

- ✓ Create a Container Apps environment
- ✓ Create an Azure Storage account
- ✓ Define a file share in the storage account
- ✓ Link the environment to the storage file share
- ✓ Mount the storage share in an individual container
- ✓ Verify the storage mount by viewing the website access log

Prerequisites

- Install the latest version of the [Azure CLI](#).

Set up the environment

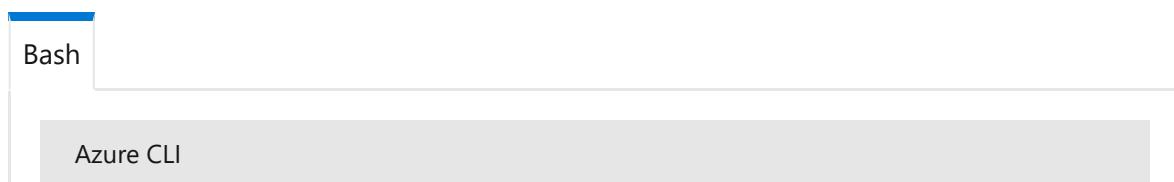
The following commands help you define variables and ensure your Container Apps extension is up to date.

1. Sign in to the Azure CLI.



A screenshot of a terminal window titled "Bash". The window shows the Azure CLI interface. In the command line area, the command `az login` is typed and highlighted in blue, indicating it is the current active command.

2. Set up environment variables used in various commands to follow.



A screenshot of a terminal window titled "Bash". The window shows the Azure CLI interface. The command line area is empty, showing only the prompt.

```
RESOURCE_GROUP="my-container-apps-group"
ENVIRONMENT_NAME="my-storage-environment"
LOCATION="canadacentral"
```

3. Ensure you have the latest version of the Container Apps Azure CLI extension.

Bash

Azure CLI

```
az extension add -n containerapp --upgrade
```

4. Register the `Microsoft.App` namespace.

Bash

Azure CLI

```
az provider register --namespace Microsoft.App
```

5. Register the `Microsoft.OperationalInsights` provider for the Azure Monitor Log Analytics workspace if you haven't used it before.

Bash

Azure CLI

```
az provider register --namespace Microsoft.OperationalInsights
```

Create an environment

The following steps create a resource group and a Container Apps environment.

1. Create a resource group.

Bash

Azure CLI

```
az group create \
--name $RESOURCE_GROUP \
--location $LOCATION \
--query "properties.provisioningState"
```

Once created, the command returns a "Succeeded" message.

At the end of this tutorial, you can delete the resource group to remove all the services created during this article.

2. Create a Container Apps environment.

Bash

Azure CLI

```
az containerapp env create \
--name $ENVIRONMENT_NAME \
--resource-group $RESOURCE_GROUP \
--location "$LOCATION" \
--query "properties.provisioningState"
```

Once created, the command returns a "Succeeded" message.

Storage mounts are associated with a Container Apps environment and configured within individual container apps.

Set up a storage account

Next, create a storage account and establish a file share to mount to the container app.

1. Define a storage account name.

This command generates a random suffix to the storage account name to ensure uniqueness.

Bash

Azure CLI

```
STORAGE_ACCOUNT_NAME="myacastorageaccount$RANDOM"
```

2. Create an Azure Storage account.

Bash

Azure CLI

```
az storage account create \
--resource-group $RESOURCE_GROUP \
--name $STORAGE_ACCOUNT_NAME \
--location "$LOCATION" \
--kind StorageV2 \
--sku Standard_LRS \
--enable-large-file-share \
--query provisioningState
```

Once created, the command returns a "Succeeded" message.

3. Define a file share name.

Bash

Bash

```
STORAGE_SHARE_NAME="myfileshare"
```

4. Create the Azure Storage file share.

Bash

Azure CLI

```
az storage share-rm create \
--resource-group $RESOURCE_GROUP \
--storage-account $STORAGE_ACCOUNT_NAME \
--name $STORAGE_SHARE_NAME \
--quota 1024 \
--enabled-protocols SMB \
--output table
```

5. Get the storage account key.

Bash

Bash

```
STORAGE_ACCOUNT_KEY=`az storage account keys list -n  
$STORAGE_ACCOUNT_NAME --query "[0].value" -o tsv`
```

The storage account key is required to create the storage link in your Container Apps environment.

6. Define the storage mount name.

Bash

Bash

```
STORAGE_MOUNT_NAME="mystoragemount"
```

This value is the name used to define the storage mount link from your Container Apps environment to your Azure Storage account.

Create the storage mount

Now you can update the container app configuration to support the storage mount.

1. Create the storage link in the environment.

Bash

Azure CLI

```
az containerapp env storage set \  
--access-mode ReadWrite \  
--azure-file-account-name $STORAGE_ACCOUNT_NAME \  
--azure-file-account-key $STORAGE_ACCOUNT_KEY \  
--azure-file-share-name $STORAGE_SHARE_NAME \  
--storage-name $STORAGE_MOUNT_NAME \  
--name $ENVIRONMENT_NAME \  
--resource-group $RESOURCE_GROUP \  
--output table
```

This command creates a link between container app environment and the file share created with the `az storage share rm` command.

Now that the storage account and environment are linked, you can create a container app that uses the storage mount.

2. Define the container app name.

```
Bash
Bash
CONTAINER_APP_NAME="my-container-app"
```

3. Create the container app.

```
Bash
Azure CLI
az containerapp create \
--name $CONTAINER_APP_NAME \
--resource-group $RESOURCE_GROUP \
--environment $ENVIRONMENT_NAME \
--image nginx \
--min-replicas 1 \
--max-replicas 1 \
--target-port 80 \
--ingress external \
--query properties.configuration.ingress.fqdn
```

This command displays the URL of your new container app.

4. Copy the URL and paste into your web browser to navigate to the website.

Once the page loads, you'll see the "Welcome to nginx!" message. Keep this browser tab open. You'll return to the website during the storage mount verification steps.

Now that you've confirmed the container app is configured, you can update the app to with a storage mount definition.

5. Export the container app's configuration.

```
Bash
Azure CLI
```

```
az containerapp show \
--name $CONTAINER_APP_NAME \
--resource-group $RESOURCE_GROUP \
--output yaml > app.yaml
```

ⓘ Note

While this application doesn't have secrets, many apps do feature secrets. By default, when you export an app's configuration, the values for secrets aren't included in the generated YAML.

If you don't need to change secret values, then you can remove the `secrets` section and your secrets remain unaltered. Alternatively, if you need to change a secret's value, make sure to provide both the `name` and `value` for all secrets in the file before attempting to update the app. Omitting a secret from the `secrets` section deletes the secret.

6. Open `app.yaml` in a code editor.

7. Replace the `volumes: null` definition in the `template` section with a `volumes:` definition referencing the storage volume. The template section should look like the following:

```
yml

template:
  volumes:
    - name: my-azure-file-volume
      storageName: mystoragemount
      storageType: AzureFile
  containers:
    - image: nginx
      name: my-container-app
      volumeMounts:
        - volumeName: my-azure-file-volume
          mountPath: /var/log/nginx
  resources:
    cpu: 0.5
    ephemeralStorage: 3Gi
    memory: 1Gi
  initContainers: null
  revisionSuffix: ''
  scale:
    maxReplicas: 1
    minReplicas: 1
    rules: null
```

The new `template.volumes` section includes the following properties.

Property	Description
<code>name</code>	This value matches the volume created by calling the <code>az containerapp env storage set</code> command.
<code>storageName</code>	This value defines the name used by containers in the environment to access the storage volume.
<code>storageType</code>	This value determines the type of storage volume defined for the environment. In this case, an Azure Files mount is declared.

The `volumes` section defines volumes at the app level that your application container or sidecar containers can reference via a `volumeMounts` section associated with a container.

8. Add a `volumeMounts` section to the `nginx` container in the `containers` section.

```
yml

containers:
  - image: nginx
    name: my-container-app
    volumeMounts:
      - volumeName: my-azure-file-volume
        mountPath: /var/log/nginx
```

The new `volumeMounts` section includes the following properties:

Property	Description
<code>volumeName</code>	This value must match the name defined in the <code>volumes</code> definition.
<code>mountPath</code>	This value defines the path in your container where the storage is mounted.

9. Update the container app with the new storage mount configuration.

Bash

Azure CLI

```
az containerapp update \
--name $CONTAINER_APP_NAME \
--resource-group $RESOURCE_GROUP \
```

```
--yaml app.yaml \
--output table
```

Verify the storage mount

Now that the storage mount is established, you can manipulate files in Azure Storage from your container. Use the following commands to observe the storage mount at work.

1. Open an interactive shell inside the container app to execute commands inside the running container.

Bash

Azure CLI

```
az containerapp exec \
--name $CONTAINER_APP_NAME \
--resource-group $RESOURCE_GROUP
```

This command may take a moment to open the remote shell. Once the shell is ready, you can interact with the storage mount via file system commands.

2. Change into the nginx `/var/log/nginx` folder.

Bash

Bash

```
cd /var/log/nginx
```

3. Return to the browser and navigate to the website and refresh the page a few times.

The requests made to the website create a series of log stream entries.

4. Return to your terminal and list the values of the `/var/log/nginx` folder.

Bash

```
Bash
```

```
ls
```

Note how the *access.log* and *error.log* files appear in this folder. These files are written to the Azure Files mount in your Azure Storage share created in the previous steps.

5. View the contents of the *access.log* file.

```
Bash
```

```
Bash
```

```
cat access.log
```

6. Exit out of the container's interactive shell to return to your local terminal session.

```
Bash
```

```
Bash
```

```
exit
```

7. Now, you can view the files in the Azure portal to verify they exist in your Azure Storage account. Print the name of your randomly generated storage account.

```
Bash
```

```
Bash
```

```
echo $STORAGE_ACCOUNT_NAME
```

8. Navigate to the Azure portal and open up the storage account created in this procedure.

9. Under **Data Storage** select **File shares**.

10. Select **myshare** to view the *access.log* and *error.log* files.

Clean up resources

If you're not going to continue to use this application, run the following command to delete the resource group along with all the resources created in this article.

Bash

Azure CLI

```
az group delete \
--name $RESOURCE_GROUP
```

Next steps

[Connect container apps together](#)

Tutorial: Deploy self-hosted CI/CD runners and agents with Azure Container Apps jobs

Article • 06/15/2023

GitHub Actions and Azure Pipelines allow you to run CI/CD workflows with self-hosted runners and agents. You can run self-hosted runners and agents using event-driven Azure Container Apps [jobs](#).

Self-hosted runners are useful when you need to run workflows that require access to local resources or tools that aren't available to a cloud-hosted runner. For example, a self-hosted runner in a Container Apps job allows your workflow to access resources inside the job's virtual network that isn't accessible to a cloud-hosted runner.

Running self-hosted runners as event-driven jobs allows you to take advantage of the serverless nature of Azure Container Apps. Jobs execute automatically when a workflow is triggered and exit when the job completes.

You only pay for the time that the job is running.

In this tutorial, you learn how to run Azure Pipelines agents as an [event-driven Container Apps job](#).

- ✓ Create a Container Apps environment to deploy your self-hosted agent
- ✓ Create an Azure DevOps organization and project
- ✓ Build a container image that runs an Azure Pipelines agent
- ✓ Use a manual job to create a placeholder agent in the Container Apps environment
- ✓ Deploy the agent as a job to the Container Apps environment
- ✓ Create a pipeline that uses the self-hosted agent and verify that it runs

Important

Self-hosted agents are only recommended for *private* projects. Using them with public projects can allow dangerous code to execute on your self-hosted agent. For more information, see [Self-hosted agent security](#).

Note

Container apps and jobs don't support running Docker in containers. Any steps in your workflows that use Docker commands will fail when run on a self-hosted runner or agent in a Container Apps job.

Prerequisites

- **Azure account:** If you don't have one, you [can create one for free](#).
- **Azure CLI:** Install the [Azure CLI](#).
- **Azure DevOps organization:** If you don't have a DevOps organization with an active subscription, you [can create one for free](#).

Refer to [jobs preview limitations](#) for a list of limitations.

Setup

1. To sign in to Azure from the CLI, run the following command and follow the prompts to complete the authentication process.

```
Bash
Bash
az login
```

2. Ensure you're running the latest version of the CLI via the `upgrade` command.

```
Bash
Bash
az upgrade
```

3. Install the latest version of the Azure Container Apps CLI extension.

```
Bash
Bash
```

```
az extension add --name containerapp --upgrade
```

4. Register the `Microsoft.App` and `Microsoft.OperationalInsights` namespaces if you haven't already registered them in your Azure subscription.

Bash

Bash

```
az provider register --namespace Microsoft.App  
az provider register --namespace Microsoft.OperationalInsights
```

5. Define the environment variables that are used throughout this article.

Bash

Bash

```
RESOURCE_GROUP="jobs-sample"  
LOCATION="northcentralus"  
ENVIRONMENT="env-jobs-sample"  
JOB_NAME="azure-pipelines-agent-job"  
PLACEHOLDER_JOB_NAME="placeholder-agent-job"
```

Create a Container Apps environment

The Azure Container Apps environment acts as a secure boundary around container apps and jobs so they can share the same network and communicate with each other.

1. Create a resource group using the following command.

Bash

Bash

```
az group create \  
--name "$RESOURCE_GROUP" \  
--location "$LOCATION"
```

2. Create the Container Apps environment using the following command.

Bash

Bash

```
az containerapp env create \
--name "$ENVIRONMENT" \
--resource-group "$RESOURCE_GROUP" \
--location "$LOCATION"
```

Create an Azure DevOps project and repository

To execute a pipeline, you need an Azure DevOps project and repository.

1. Navigate to [Azure DevOps](#) and sign in to your account.
2. Select an existing organization or create a new one.
3. In the organization overview page, select **New project** and enter the following values.

Setting	Value
<i>Project name</i>	Enter a name for your project.
<i>Visibility</i>	Select Private .

4. Select **Create**.
5. From the side navigation, select **Repos**.
6. Under *Initialize main branch with a README or .gitignore*, select **Add a README**.
7. Leave the rest of the values as defaults and select **Initialize**.

Create a new agent pool

Create a new agent pool to run the self-hosted runner.

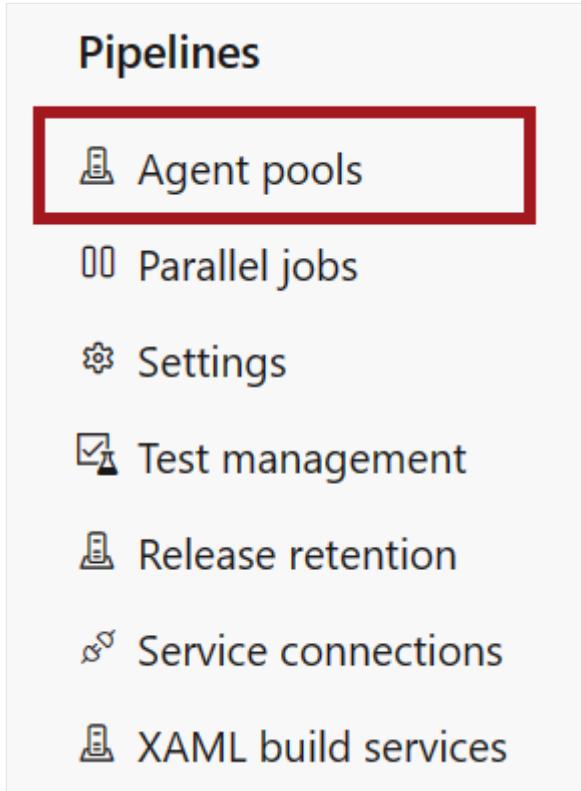
1. In your Azure DevOps project, expand the left navigation bar and select **Project settings**.



Project settings



2. Under the *Pipelines* section in the *Project settings* navigation menu, select **Agent pools**.



3. Select **Add pool** and enter the following values.

Setting	Value
Pool to link	Select New .
Pool type	Select Self-hosted .
Name	Enter container-apps .
Grant access permission to all pipelines	Select this checkbox.

4. Select **Create**.

Get an Azure DevOps personal access token

To run a self-hosted runner, you need to create a personal access token (PAT) in Azure DevOps. The PAT is used to authenticate the runner with Azure DevOps. It's also used by the scale rule to determine the number of pending pipeline runs and trigger new job executions.

1. In Azure DevOps, select *User settings* next to your profile picture in the upper-right corner.

2. Select **Personal access tokens**.

3. In the *Personal access tokens* page, select **New Token** and enter the following values.

Setting	Value
Name	Enter a name for your token.
Organization	Select the organization you chose or created earlier.
Scopes	Select Custom defined .
Show all scopes	Select Show all scopes .
Agent Pools (Read & manage)	Select Agent Pools (Read & manage) .

Leave all other scopes unselected.

4. Select **Create**.

5. Copy the token value to a secure location.

You can't retrieve the token after you leave the page.

6. Define variables that are used to configure the Container Apps jobs later.

The screenshot shows a terminal window with a blue header bar. Below it, there is a dropdown menu with 'Bash' selected. The main area of the terminal contains the following text:

```
AZP_TOKEN=<AZP_TOKEN>
ORGANIZATION_URL=<ORGANIZATION_URL>
AZP_POOL="container-apps"
```

Replace the placeholders with the following values:

Placeholder	Value	Comments
<AZP_TOKEN>	The Azure DevOps PAT you generated.	
<ORGANIZATION_URL>	The URL of your Azure DevOps organization.	For example, https://dev.azure.com/myorg or https://myorg.visualstudio.com .

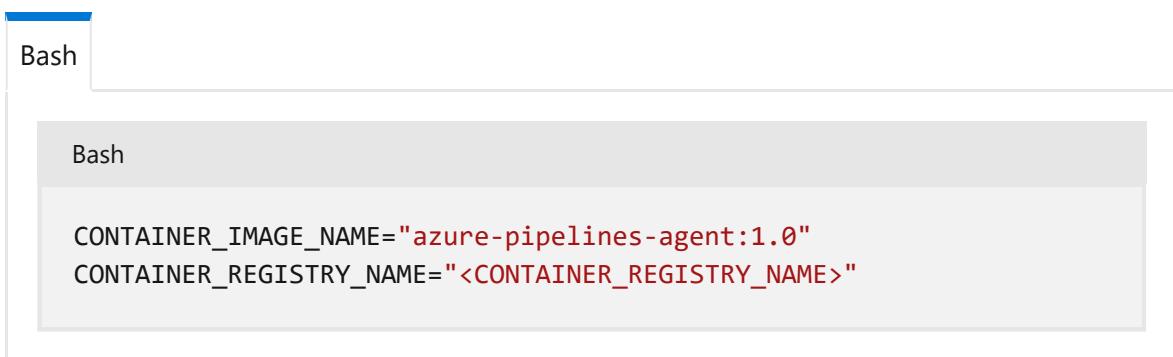
Build the Azure Pipelines agent container image

To create a self-hosted agent, you need to build a container image that runs the agent. In this section, you build the container image and push it to a container registry.

ⓘ Note

The image you build in this tutorial contains a basic self-hosted agent that's suitable for running as a Container Apps job. You can customize it to include additional tools or dependencies that your pipelines require.

1. Back in your terminal, define a name for your container image and registry.

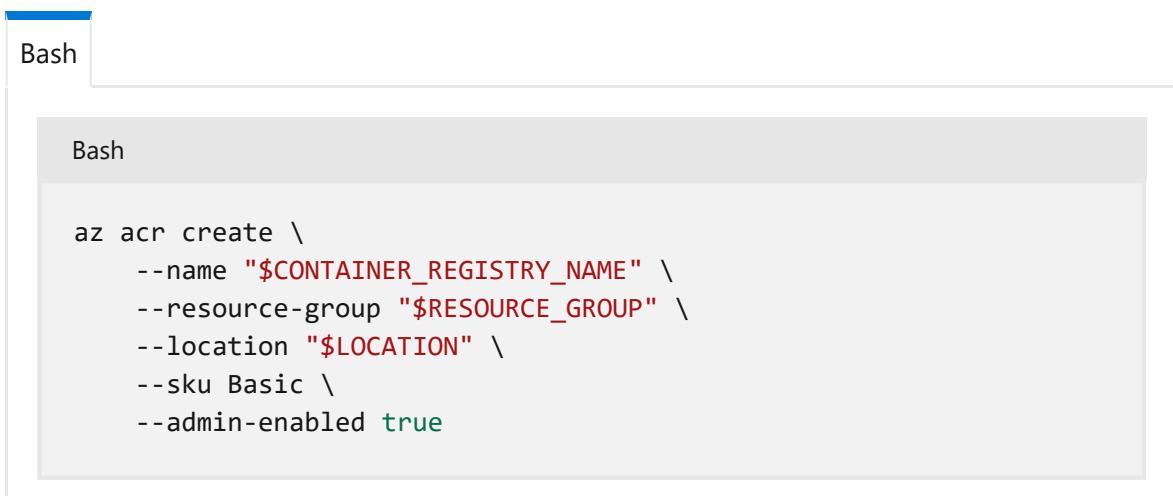


```
CONTAINER_IMAGE_NAME="azure-pipelines-agent:1.0"
CONTAINER_REGISTRY_NAME=""
```

Replace `<CONTAINER_REGISTRY_NAME>` with a unique name for creating a container registry.

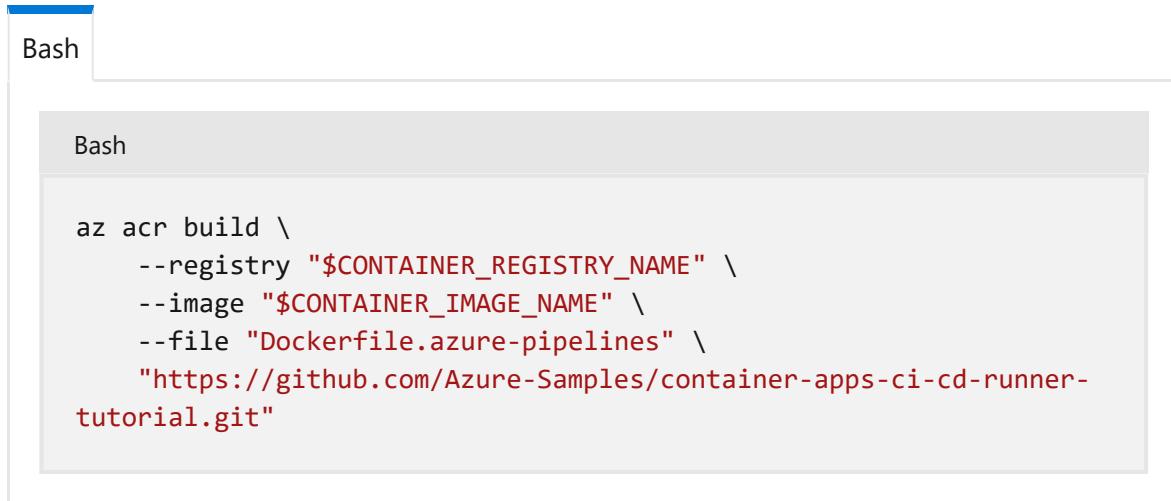
Container registry names must be *unique within Azure* and be from 5 to 50 characters in length containing numbers and lowercase letters only.

2. Create a container registry.



```
az acr create \
--name "$CONTAINER_REGISTRY_NAME" \
--resource-group "$RESOURCE_GROUP" \
--location "$LOCATION" \
--sku Basic \
--admin-enabled true
```

3. The Dockerfile for creating the runner image is available on [GitHub](#). Run the following command to clone the repository and build the container image in the cloud using the `az acr build` command.



A screenshot of a terminal window titled "Bash". The command `az acr build` is being typed in, with the registry URL and Dockerfile path highlighted in red. The command is as follows:

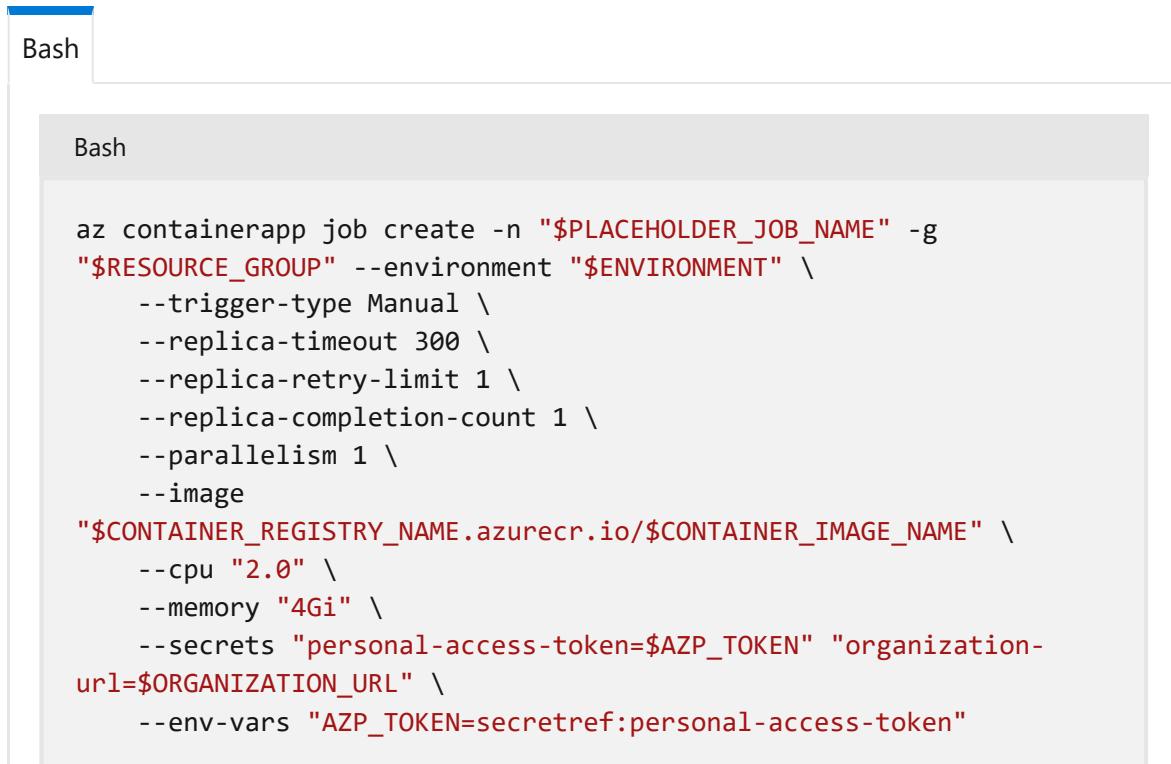
```
az acr build \
    --registry "$CONTAINER_REGISTRY_NAME" \
    --image "$CONTAINER_IMAGE_NAME" \
    --file "Dockerfile.azure-pipelines" \
    "https://github.com/Azure-Samples/container-apps-ci-cd-runner-tutorial.git"
```

The image is now available in the container registry.

Create a placeholder self-hosted agent

Before you can run a self-hosted agent in your new agent pool, you need to create a placeholder agent. Pipelines that use the agent pool fail when there's no placeholder agent. You can create a placeholder agent by running a job that registers an offline placeholder agent.

1. Create a manual job in the Container Apps environment that creates the placeholder agent.



A screenshot of a terminal window titled "Bash". The command `az containerapp job create` is being typed in, with various parameters highlighted in red. The command is as follows:

```
az containerapp job create -n "$PLACEHOLDER_JOB_NAME" -g "$RESOURCE_GROUP" --environment "$ENVIRONMENT" \
    --trigger-type Manual \
    --replica-timeout 300 \
    --replica-retry-limit 1 \
    --replica-completion-count 1 \
    --parallelism 1 \
    --image
    "$CONTAINER_REGISTRY_NAME.azurecr.io/$CONTAINER_IMAGE_NAME" \
    --cpu "2.0" \
    --memory "4Gi" \
    --secrets "personal-access-token=$AZP_TOKEN" "organization-url=$ORGANIZATION_URL" \
    --env-vars "AZP_TOKEN=secretref:personal-access-token"
```

```
"AZP_URL=secretref:organization-url" "AZP_POOL=$AZP_POOL"  
"AZP_PLACEHOLDER=1" "AZP_AGENT_NAME=placeholder-agent" \  
--registry-server "$CONTAINER_REGISTRY_NAME.azurecr.io"
```

The following table describes the key parameters used in the command.

Parameter	Description
--replica-timeout	The maximum duration a replica can execute.
--replica-retry-limit	The number of times to retry a failed replica.
--replica-completion-count	The number of replicas to complete successfully before a job execution is considered successful.
--parallelism	The number of replicas to start per job execution.
--secrets	The secrets to use for the job.
--env-vars	The environment variables to use for the job.
--registry-server	The container registry server to use for the job. For an Azure Container Registry, the command automatically configures authentication.

Setting the `AZP_PLACEHOLDER` environment variable configures the agent container to register as an offline placeholder agent without running a job.

2. Execute the manual job to create the placeholder agent.

Bash

Bash

```
az containerapp job start -n "$PLACEHOLDER_JOB_NAME" -g  
"$RESOURCE_GROUP"
```

3. List the executions of the job to confirm a job execution was created and completed successfully.

Bash

Bash

```
az containerapp job execution list \
    --name "$PLACEHOLDER_JOB_NAME" \
    --resource-group "$RESOURCE_GROUP" \
    --output table \
    --query '[].{Status: properties.status, Name: name, StartTime: properties.startTime}'
```

4. Verify the placeholder agent was created in Azure DevOps.
 - a. In Azure DevOps, navigate to your project.
 - b. Select **Project settings > Agent pools > container-apps > Agents**.
 - c. Confirm that a placeholder agent named `placeholder-agent` is listed.

Create a self-hosted agent as an event-driven job

Now that you have a placeholder agent, you can create a self-hosted agent. In this section, you create an event-driven job that runs a self-hosted agent when a pipeline is triggered.

Bash

Bash

```
az containerapp job create -n "$JOB_NAME" -g "$RESOURCE_GROUP" --environment "$ENVIRONMENT" \
    --trigger-type Event \
    --replica-timeout 300 \
    --replica-retry-limit 1 \
    --replica-completion-count 1 \
    --parallelism 1 \
    --image "$CONTAINER_REGISTRY_NAME.azurecr.io/$CONTAINER_IMAGE_NAME"
\
    --min-executions 0 \
    --max-executions 10 \
    --polling-interval 30 \
    --scale-rule-name "azure-pipelines" \
    --scale-rule-type "azure-pipelines" \
    --scale-rule-metadata "poolName=container-apps"
"targetPipelinesQueueLength=1" \
    --scale-rule-auth "personalAccessToken=personal-access-token"
"organizationURL=organization-url" \
    --cpu "2.0" \
    --memory "4Gi" \
    --secrets "personal-access-token=$AZP_TOKEN" "organization-
url=$ORGANIZATION_URL" \
    --env-vars "AZP_TOKEN=secretref:personal-access-token"
```

```
"AZP_URL=secretref:organization-url" "AZP_POOL=$AZP_POOL" \  
--registry-server "$CONTAINER_REGISTRY_NAME.azurecr.io"
```

The following table describes the scale rule parameters used in the command.

Parameter	Description
--min-executions	The minimum number of job executions to run per polling interval.
--max-executions	The maximum number of job executions to run per polling interval.
--polling-interval	The polling interval at which to evaluate the scale rule.
--scale-rule-name	The name of the scale rule.
--scale-rule-type	The type of scale rule to use. To learn more about the Azure Pipelines scaler, see the KEDA documentation .
--scale-rule-metadata	The metadata for the scale rule.
--scale-rule-auth	The authentication for the scale rule.

The scale rule configuration defines the event source to monitor. It's evaluated on each polling interval and determines how many job executions to trigger. To learn more, see [Set scaling rules](#).

The event-driven job is now created in the Container Apps environment.

Run a pipeline and verify the job

Now that you've configured a self-hosted agent job, you can run a pipeline and verify it's working correctly.

1. In the left-hand navigation of your Azure DevOps project, navigate to **Pipelines**.
2. Select **Create pipeline**.
3. Select **Azure Repos Git** as the location of your code.
4. Select the repository you created earlier.

5. Select Starter pipeline.

6. In the pipeline YAML, change the `pool` from `vmImage: ubuntu-latest` to `name: container-apps`.

YAML

```
pool:  
  name: container-apps
```

7. Select Save and run.

The pipeline runs and uses the self-hosted agent job you created in the Container Apps environment.

8. List the executions of the job to confirm a job execution was created and completed successfully.

Bash

Bash

```
az containerapp job execution list \  
  --name "$JOB_NAME" \  
  --resource-group "$RESOURCE_GROUP" \  
  --output table \  
  --query '[].{Status: properties.status, Name: name, StartTime: properties.startTime}'
```

💡 Tip

Having issues? Let us know on GitHub by opening an issue in the [Azure Container Apps repo](#).

Clean up resources

Once you're done, run the following command to delete the resource group that contains your Container Apps resources.

⊗ Caution

The following command deletes the specified resource group and all resources contained within it. If resources outside the scope of this tutorial exist in the specified resource group, they will also be deleted.

Bash

Bash

```
az group delete \  
  --resource-group $RESOURCE_GROUP
```

To delete your GitHub repository, see [Deleting a repository ↗](#).

Next steps

[Container Apps jobs](#)

Tutorial: Create and use an Apache Kafka service for development

Article • 06/22/2023

Azure Container Apps allows you to connect to development and production-grade services to provide a wide variety of functionality to your applications.

In this tutorial, you learn to create and use a development Apache Kafka service.

Azure CLI commands and Bicep template fragments are featured in this tutorial. If you use Bicep, you can add all the fragments to a single Bicep file and [deploy the template all at once](#).

- ✓ Create a Container Apps environment to deploy your service and container app
- ✓ Create an Apache Kafka service
- ✓ Set up a command line app to use the dev Apache Kafka service
- ✓ Deploy a *kafka-ui* app to view application data
- ✓ Compile a final bicep template to deploy all resources using a consistent and predictable template deployment
- ✓ Use an `azd` template for a one command deployment of all resources

Prerequisites

- Install the [Azure CLI](#).
- Optional: [Azure Developer CLI](#) for following AZD instructions.

ⓘ Note

For a one command deployment, skip to the last `azd template step`.

Setup

1. Define variables for common values.



```
RESOURCE_GROUP="kafka-dev"
LOCATION="northcentralus"
ENVIRONMENT="aca-env"
KAFKA_SVC="kafka01"
KAFKA_CLI_APP="kafka-cli-app"
KAFKA_UI_APP="kafka-ui-app"
```

2. Log in to Azure.

```
Bash
```

```
az login
```

3. Upgrade the CLI to the latest version.

```
Bash
```

```
az upgrade
```

4. Upgrade Bicep to the latest version.

```
Bash
```

```
az bicep upgrade
```

5. Add the `containerapp` extension.

```
Bash
```

```
az extension add --name containerapp --upgrade
```

6. Register required namespaces.

```
Bash
```

```
az provider register --namespace Microsoft.App
```

```
Bash
```

```
az provider register --namespace Microsoft.OperationalInsights
```

Create a Container Apps environment

1. Create a resource group.

```
Bash
Bash

az group create \
--name "$RESOURCE_GROUP" \
--location "$LOCATION"
```

2. Create a Container Apps environment.

```
Bash
Bash

az containerapp env create \
--name "$ENVIRONMENT" \
--resource-group "$RESOURCE_GROUP" \
--location "$LOCATION"
```

Create an Apache Kafka service

1. Create an Apache Kafka service.

```
Bash
Bash

ENVIRONMENT_ID=$(az containerapp env show \
--name "$ENVIRONMENT" \
--resource-group "$RESOURCE_GROUP" \
--output tsv \
--query id)
```

2. Deploy the template.

```
Bash
```

Bash

```
az rest \
--method PUT \
--url "/subscriptions/${(az account show --output tsv --query id)/resourceGroups/$RESOURCE_GROUP/providers/Microsoft.App/containers/$KAFKA_SVC?api-version=2023-04-01-preview" \
--body "{\"location\": \"$LOCATION\", \"properties\": {\"environmentId\": \"$ENVIRONMENT_ID\", \"configuration\": {\"service\": {\"type\": \"kafka\"}}}}
```

3. View log output from the Kafka instance

Bash

Use the `logs` command to view log messages.

Bash

```
az containerapp logs show \
--name $KAFKA_SVC \
--resource-group $RESOURCE_GROUP \
--follow --tail 30
```

```
{"TimeStamp": "2023-06-07T02:08:05.5473813+00:00", "Log": "= 604800000"}  
{"TimeStamp": "2023-06-07T02:08:05.5473851+00:00", "Log": "= false"}  
{"TimeStamp": "2023-06-07T02:08:05.5473894+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5473934+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5473963+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474017+00:00", "Log": "= 10"}  
{"TimeStamp": "2023-06-07T02:08:05.5474049+00:00", "Log": "= false"}  
{"TimeStamp": "2023-06-07T02:08:05.5474084+00:00", "Log": "= 18000"}  
{"TimeStamp": "2023-06-07T02:08:05.5474111+00:00", "Log": "= false"}  
{"TimeStamp": "2023-06-07T02:08:05.5474154+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474194+00:00", "Log": "= false"}  
{"TimeStamp": "2023-06-07T02:08:05.5474226+00:00", "Log": "= false"}  
{"TimeStamp": "2023-06-07T02:08:05.5474261+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474293+00:00", "Log": "= HTTPS"}  
{"TimeStamp": "2023-06-07T02:08:05.5474329+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474364+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474392+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474425+00:00", "Log": "= false"}  
{"TimeStamp": "2023-06-07T02:08:05.5474464+00:00", "Log": "= TLSv1.2"}  
{"TimeStamp": "2023-06-07T02:08:05.54745+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474545+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474578+00:00", "Log": "= null"}  
{"TimeStamp": "2023-06-07T02:08:05.5474624+00:00", "Log": "= [kafka.server.KafkaConfig]}"}  
{"TimeStamp": "2023-06-07T02:08:05.6102429+00:00", "Log": "= 02:08:05,610] INFO [SocketServer listenerType=BROKER, nodeId=1] Enabling request processing. (kafka.network.SocketServer)"}  
{"TimeStamp": "2023-06-07T02:08:05.6466283+00:00", "Log": "= 02:08:05,646] INFO [BrokerLifecycleManager id=1] The broker has been unfenced. Transitioning from RECOVERY to RUNNING. (kafka.server.BrokerLifecycleManager)"}  
{"TimeStamp": "2023-06-07T02:08:05.6468583+00:00", "Log": "= 02:08:05,646] INFO [BrokerServer id=1] Transition from STARTING to STARTED (kafka.server.BrokerServer)"}  
{"TimeStamp": "2023-06-07T02:08:05.6473756+00:00", "Log": "= 02:08:05,647] INFO Kafka version: 3.4.0 (org.apache.kafka.common.utils.AppInfoParser)"}  
{"TimeStamp": "2023-06-07T02:08:05.6474318+00:00", "Log": "= 02:08:05,647] INFO Kafka commitId: 2e1947d240607d53 (org.apache.kafka.common.utils.AppInfoParser)"}  
{"TimeStamp": "2023-06-07T02:08:05.6474952+00:00", "Log": "= 02:08:05,647] INFO Kafka startTimeMs: 1686103685646 (org.apache.kafka.common.utils.AppInfoParser)"}  
{"TimeStamp": "2023-06-07T02:08:05.6484361+00:00", "Log": "= 02:08:05,648] INFO [KafkaRaftServer nodeId=1] Kafka Server started (kafka.server.KafkaRaftServer)"}
```

Create an app to test the service

When you create the app, you'll set it up to use `./kafka-topics.sh`, `./kafka-console-producer.sh`, and `kafka-console-consumer.sh` to connect to the Kafka instance.

1. Create a `kafka-cli-app` app that binds to the PostgreSQL service.

```
Bash

Bash

az containerapp create \
    --name "$KAFKA_CLI_APP" \
    --image mcr.microsoft.com/k8se/services/kafka:3.4 \
    --environment "$ENVIRONMENT" \
    --resource-group "$RESOURCE_GROUP" \
    --min-replicas 1 \
    --max-replicas 1 \
    --command "/bin/sleep" "infinity"

az rest \
    --method PATCH \
    --headers "Content-Type=application/json" \
    --url "/subscriptions/$(az account show --output tsv --query id)/resourceGroups/$RESOURCE_GROUP/providers/Microsoft.App/containerApps/$KAFKA_CLI_APP?api-version=2023-04-01-preview" \
    --body "{\"properties\": {\"template\": {\"serviceBinds\": [{\"serviceId\": \"/subscriptions/$(az account show --output tsv --query id)/resourceGroups/$RESOURCE_GROUP/providers/Microsoft.App/containerApps/$KAFKA_SVC\"}]}}}"
```

2. Run the CLI `exec` command to connect to the test app.

```
Bash

Bash

az containerapp exec \
    --name $KAFKA_CLI_APP \
    --resource-group $RESOURCE_GROUP \
    --command /bin/bash
```

When you use `--bind` or `serviceBinds` on the test app, the connection information is injected into the application environment. Once you connect to the test container, you can inspect the values using the `env` command.

Bash

```
env | grep "^KAFKA_"

KAFKA_SECURITYPROTOCOL=SASL_PLAINTEXT
KAFKA_BOOTSTRAPSERVER=kafka01:9092
KAFKA_HOME=/opt/kafka
KAFKA_PROPERTIES_SASL_JAAS_CONFIG=org.apache.kafka.common.security.plain.PlainLoginModule required username="kafka-user" password="7dw..."
user_kafka-user="7dw..." ;
KAFKA_BOOTSTRAP_SERVERS=kafka01:9092
KAFKA_SASLUSERNAME=kafka-user
KAFKA_SASL_USER=kafka-user
KAFKA_VERSION=3.4.0
KAFKA_SECURITY_PROTOCOL=SASL_PLAINTEXT
KAFKA_SASL_PASSWORD=7dw...
KAFKA_SASLPASSWORD=7dw...
KAFKA_SASL_MECHANISM=PLAIN
KAFKA_SASLMECHANISM=PLAIN
```

3. Use `kafka-topics.sh` to create an event topic.

Create a `kafka.props` file.

Bash

```
echo "security.protocol=$KAFKA_SECURITY_PROTOCOL" >> kafka.props && \
echo "sasl.mechanism=$KAFKA_SASL_MECHANISM" >> kafka.props && \
echo "sasl.jaas.config=$KAFKA_PROPERTIES_SASL_JAAS_CONFIG" >>
kafka.props
```

Create a `quickstart-events` event topic.

Bash

```
/opt/kafka/bin/kafka-topics.sh \
--create --topic quickstart-events \
--bootstrap-server $KAFKA_BOOTSTRAP_SERVERS \
--command-config kafka.props
# Created topic quickstart-events.

/opt/kafka/bin/kafka-topics.sh \
--describe --topic quickstart-events \
--bootstrap-server $KAFKA_BOOTSTRAP_SERVERS \
--command-config kafka.props
# Topic: quickstart-events  TopicId: lCKTKmvZSgSUCHzhhvz1Q
PartitionCount: 1  ReplicationFactor: 1    Configs:
segment.bytes=1073741824
# Topic: quickstart-events  Partition: 0      Leader: 1    Replicas: 1
Isr: 1
```

4. Use `kafka-console-producer.sh` to write events to the topic.

Bash

```
/opt/kafka/bin/kafka-console-producer.sh \
  --topic quickstart-events \
  --bootstrap-server $KAFKA_BOOTSTRAP_SERVERS \
  --producer.config kafka.props

> this is my first event
> this is my second event
> this is my third event
> CTRL-C
```

① Note

The `./kafka-console-producer.sh` command prompts you to write events with `>`. Write some events as shown, then hit `CTRL-C` to exit.

5. Use `kafka-console-consumer.sh` to read events from the topic.

Bash

```
/opt/kafka/bin/kafka-console-consumer.sh \
  --topic quickstart-events \
  --bootstrap-server $KAFKA_BOOTSTRAP_SERVERS \
  --from-beginning \
  --consumer.config kafka.props

# this is my first event
# this is my second event
# this is my third event
```

```

root@kafka-cli-app--keu370t-7cfb95dc5d-nwpfm:/# /opt/kafka/bin/kafka-topics.sh \
  --create --topic quickstart-events \
  --bootstrap-server $KAFKA_BOOTSTRAP_SERVERS \
  --command-config kafka.props
Created topic quickstart-events.
root@kafka-cli-app--keu370t-7cfb95dc5d-nwpfm:/# /opt/kafka/bin/kafka-topics.sh \
  --describe --topic quickstart-events \
  --bootstrap-server $KAFKA_BOOTSTRAP_SERVERS \
  --command-config kafka.props
Topic: quickstart-events          TopicId: lD6GKnEAQ626f2kvK5u4JA PartitionCount: 1      ReplicationFactor: 1   C
onfigs: segment.bytes=1073741824
  Topic: quickstart-events      Partition: 0      Leader: 1      Replicas: 1      Isr: 1
root@kafka-cli-app--keu370t-7cfb95dc5d-nwpfm:/# /opt/kafka/bin/kafka-console-producer.sh \
  --topic quickstart-events \
  --bootstrap-server $KAFKA_BOOTSTRAP_SERVERS \
  --producer.config kafka.props
>this is my first event
>this is my second event
>this is my third event
>"Croot@kafka-cli-app--keu370t-7cfb95dc5d-nwpfm:/#
root@kafka-cli-app--keu370t-7cfb95dc5d-nwpfm:/# /opt/kafka/bin/kafka-console-consumer.sh \
  --topic quickstart-events \
  --bootstrap-server $KAFKA_BOOTSTRAP_SERVERS \
  --from-beginning \
  --consumer.config kafka.props
this is my first event
this is my second event
this is my third event
"CProcessed a total of 3 messages
root@kafka-cli-app--keu370t-7cfb95dc5d-nwpfm:/# ■

```

Using a dev service with an existing app

If you already have an app that uses Apache Kafka, you can change how connection information is loaded.

First, create the following environment variables.

Bash

```

KAFKA_HOME=/opt/kafka
KAFKA_PROPERTIES_SASL_JAAS_CONFIG=org.apache.kafka.common.security.plain.Pla
inLoginModule required username="kafka-user" password="7dw..." user_kafka-
user="7dw..." ;
KAFKA_BOOTSTRAP_SERVERS=kafka01:9092
KAFKA_SASL_USER=kafka-user
KAFKA_VERSION=3.4.0
KAFKA_SECURITY_PROTOCOL=SASL_PLAINTEXT
KAFKA_SASL_PASSWORD=7dw...
KAFKA_SASL_MECHANISM=PLAIN

```

Using the CLI (or Bicep) you can update the app to add `--bind $KAFKA_SVC` to use the dev service.

Binding to the dev service

Deploy [kafka-ui](#) to view and manage the Kafka instance.

Bash

See Bicep or azd example.

DONE 81 ms 67 Bytes 3 messages consumed

Offset	Partition	Timestamp	Key	Preview	Value
0	0	6/6/2023, 19:40:34			this is my first event
1	0	6/6/2023, 19:40:38			this is my second event
2	0	6/6/2023, 19:40:46			this is my third event

Deploy all resources

Use the following examples to if you want to deploy all resources at once.

Bicep

The following Bicep template contains all the resources in this tutorial.

You can create a `postgres-dev.bicep` file with this content.

```
Bicep

targetScope = 'resourceGroup'
param location string = resourceGroup().location
param appEnvironmentName string = 'aca-env'
param kafkaSvcName string = 'kafka01'
param kafkaCliAppName string = 'kafka-cli-app'
param kafkaUiAppName string = 'kafka-ui'

resource logAnalytics 'Microsoft.OperationalInsights/workspaces@2022-10-01'
= {
  name: '${appEnvironmentName}-log-analytics'
  location: location
  properties: {
    sku: {
      name: 'PerGB2018'
    }
  }
}
```

```

    }

}

resource appEnvironment 'Microsoft.App/managedEnvironments@2023-04-01-preview' = {
    name: appEnvironmentName
    location: location
    properties: {
        appLogsConfiguration: {
            destination: 'log-analytics'
            logAnalyticsConfiguration: {
                customerId: logAnalytics.properties.customerId
                sharedKey: logAnalytics.listKeys().primarySharedKey
            }
        }
    }
}

resource kafka 'Microsoft.App/containerApps@2023-04-01-preview' = {
    name: kafkaSvcName
    location: location
    properties: {
        environmentId: appEnvironment.id
        configuration: {
            service: {
                type: 'kafka'
            }
        }
    }
}

resource kafkaCli 'Microsoft.App/containerApps@2023-04-01-preview' = {
    name: kafkaCliAppName
    location: location
    properties: {
        environmentId: appEnvironment.id
        template: {
            serviceBinds: [
                {
                    serviceId: kafka.id
                }
            ]
            containers: [
                {
                    name: 'kafka-cli'
                    image: 'mcr.microsoft.com/k8se/services/kafka:3.4'
                    command: [ '/bin/sleep', 'infinity' ]
                }
            ]
            scale: {
                minReplicas: 1
                maxReplicas: 1
            }
        }
    }
}

```

```

}

resource kafkaUi 'Microsoft.App/containerApps@2023-04-01-preview' = {
  name: kafkaUiAppName
  location: location
  properties: {
    environmentId: appEnvironment.id
    configuration: {
      ingress: {
        external: true
        targetPort: 8080
      }
    }
    template: {
      serviceBinds: [
        {
          serviceId: kafka.id
          name: 'kafka'
        }
      ]
      containers: [
        {
          name: 'kafka-ui'
          image: 'docker.io/provectuslabs/kafka-ui:latest'
          command: [
            '/bin/sh'
          ]
          args: [
            '-c'
            '''export
KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS="$KAFKA_BOOTSTRAP_SERVERS" && \
            export
KAFKA_CLUSTERS_0_PROPERTIES_SASL_JAAS_CONFIG="$KAFKA_PROPERTIES_SASL_JAAS_CO
NFIG" && \
            export
KAFKA_CLUSTERS_0_PROPERTIES_SASL_MECHANISM="$KAFKA_SASL_MECHANISM" && \
            export
KAFKA_CLUSTERS_0_PROPERTIES_SECURITY_PROTOCOL="$KAFKA_SECURITY_PROTOCOL" &&
\
            java $JAVA_OPTS -jar kafka-ui-api.jar'''"
          ]
          resources: {
            cpu: json('1.0')
            memory: '2.0Gi'
          }
        }
      ]
    }
  }
}

output kafkaUiUrl string =
'https://${kafkaUi.properties.configuration.ingress.fqdn}'

output kafkaCliExec string = 'az containerapp exec -n ${kafkaCli.name} -g

```

```
 ${resourceGroup().name} --command /bin/bash'

output kafkaLogs string = 'az containerapp logs show -n ${kafka.name} -g
${resourceGroup().name} --follow --tail 30'
```

Use the Azure CLI to deploy it the template.

Bash

```
RESOURCE_GROUP="kafka-dev"
LOCATION="northcentralus"

az group create \
    --name "$RESOURCE_GROUP" \
    --location "$LOCATION"

az deployment group create -g $RESOURCE_GROUP \
    --query 'properties.outputs.*.value' \
    --template-file kafka-dev.bicep
```

Azure Developer CLI

A [final template](#) is available on GitHub.

Use `azd up` to deploy the template.

Bash

```
git clone https://github.com/Azure-Samples/aca-dev-service-kafka-azd
cd aca-dev-service-kafka-azd
azd up
```

Clean up resources

Once you're done, run the following command to delete the resource group that contains your Container Apps resources.

⊗ Caution

The following command deletes the specified resource group and all resources contained within it. If resources outside the scope of this tutorial exist in the specified resource group, they will also be deleted.

```
az group delete \  
  --resource-group $RESOURCE_GROUP
```

Tutorial: Use a PostgreSQL service for development

Article • 06/22/2023

Azure Container Apps allows you to connect to development and production-grade services to provide a wide variety of functionality to your applications.

In this tutorial, you learn to use a development PostgreSQL service with Container Apps.

Azure CLI commands and Bicep template fragments are featured in this tutorial. If you use Bicep, you can add all the fragments to a single Bicep file and [deploy the template all at once](#).

- ✓ Create a Container Apps environment to deploy your service and container apps
- ✓ Create a PostgreSQL service
- ✓ Create and use a command line app to use the dev PostgreSQL service
- ✓ Create a *pgweb* app
- ✓ Write data to the PostgreSQL database

Prerequisites

- Install the [Azure CLI](#).
- Optional: [Azure Developer CLI](#) for following AZD instructions.

ⓘ Note

For a one command deployment, skip to the last `azd template step`.

Setup

1. Define variables for common values.

```
Bash
Bash
RESOURCE_GROUP="postgres-dev"
LOCATION="northcentralus"
ENVIRONMENT="aca-env"
```

```
PG_SVC="postgres01"
PSQL_CLI_APP="psql-cloud-cli-app"
```

2. Log in to Azure.

```
Bash
```

```
az login
```

3. Upgrade the CLI to the latest version.

```
Bash
```

```
az upgrade
```

4. Upgrade Bicep to the latest version.

```
Bash
```

```
az bicep upgrade
```

5. Add the `containerapp` extension.

```
Bash
```

```
az extension add --name containerapp --upgrade
```

6. Register required namespaces.

```
Bash
```

```
az provider register --namespace Microsoft.App
```

```
Bash
```

```
az provider register --namespace Microsoft.OperationalInsights
```

Create a Container Apps environment

1. Create a resource group.



Bash

Bash

```
az group create \
--name "$RESOURCE_GROUP" \
--location "$LOCATION"
```

2. Create a Container Apps environment.

Bash

Bash

```
az containerapp env create \
--name "$ENVIRONMENT" \
--resource-group "$RESOURCE_GROUP" \
--location "$LOCATION"
```

Create a PostgreSQL service

1. Create a PostgreSQL service.

Bash

Bash

```
az containerapp service postgres create \
--name "$PG_SVC" \
--resource-group "$RESOURCE_GROUP" \
--environment "$ENVIRONMENT"
```

2. View log output from the Postgres instance

Bash

Use the `logs` command to view log messages.

Bash

```
az containerapp logs show \
    --name $PG_SVC \
    --resource-group $RESOURCE_GROUP \
    --follow --tail 30
```

```
+ az containerapp logs show -n postgres-01 -g postgres-dev --revision postgres-01--plus9v8 --follow --tail 30
{"TimeStamp": "2023-06-06T20:05:45.44013", "Log": "Connecting to the container 'postgres' ..."}
{"TimeStamp": "2023-06-06T20:05:45.45931", "Log": "Successfully Connected to container: 'postgres' [Revision: 'postgres-01--plus9v8', Replica: 'postgres-01--plus9v8-74d9fd878b-x6rhw']"}
{"TimeStamp": "2023-06-06T20:04:15.8704564+00:00", "Log": ""}
{"TimeStamp": "2023-06-06T20:04:15.8704813+00:00", "Log": "You can now start the database server using:"}
{"TimeStamp": "2023-06-06T20:04:15.8704858+00:00", "Log": ""}
{"TimeStamp": "2023-06-06T20:04:15.8704892+00:00", "Log": "pg_ctl -D /mnt/data/pgdata -l logfile start"}
 {"TimeStamp": "2023-06-06T20:04:15.8704911+00:00", "Log": ""}
 {"TimeStamp": "2023-06-06T20:04:15.8704936+00:00", "Log": "For server to start... 2023-06-06 20:04:15.996 UTC [49] LOG: starting PostgreSQL 14.8 (Debian 14.8-1.pgdg110+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 10.2.1-20210110, 64-bit)"}
 {"TimeStamp": "2023-06-06T20:04:15.9972697+00:00", "Log": "20:04:15.997 UTC [49] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432""}
 {"TimeStamp": "2023-06-06T20:04:16.1091987+00:00", "Log": "20:04:16.109 UTC [50] LOG: database system was shut down at 2023-06-06 20:04:05 UTC"}
 {"TimeStamp": "2023-06-06T20:04:16.2238047+00:00", "Log": "20:04:16.223 UTC [49] LOG: database system is ready to accept connections"}
 {"TimeStamp": "2023-06-06T20:04:16.3259918+00:00", "Log": "done"}
 {"TimeStamp": "2023-06-06T20:04:16.3260101+00:00", "Log": "started"}
 {"TimeStamp": "2023-06-06T20:04:16.5341094+00:00", "Log": "done"}
 {"TimeStamp": "2023-06-06T20:04:16.5341344+00:00", "Log": "ignoring /docker-entrypoint-initdb.d/*"}
 {"TimeStamp": "2023-06-06T20:04:16.5540668+00:00", "Log": ""}
 {"TimeStamp": "2023-06-06T20:04:16.600639+00:00", "Log": "for server to shut down ... 2023-06-06 20:04:16.600 UTC [49] LOG: aborting any active transactions"}
 {"TimeStamp": "2023-06-06T20:04:16.645991+00:00", "Log": "20:04:16.645 UTC [49] LOG: background worker \"logical replication launcher\" (PID 56) exited with exit code 1"}
 {"TimeStamp": "2023-06-06T20:04:16.77022+00:00", "Log": "20:04:16.770 UTC [51] LOG: shutting down"}
 {"TimeStamp": "2023-06-06T20:04:17.0714615+00:00", "Log": "20:04:17.071 UTC [49] LOG: database system is shut down"}
 {"TimeStamp": "2023-06-06T20:04:17.1287345+00:00", "Log": "done"}
 {"TimeStamp": "2023-06-06T20:04:17.1287626+00:00", "Log": "stopped"}
 {"TimeStamp": "2023-06-06T20:04:17.1297693+00:00", "Log": ""}
 {"TimeStamp": "2023-06-06T20:04:17.1297933+00:00", "Log": "init process complete; ready for start up."}
 {"TimeStamp": "2023-06-06T20:04:17.1992697+00:00", "Log": "20:04:17.199 UTC [1] LOG: starting PostgreSQL 14.8 (Debian 14.8-1.pgdg110+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 10.2.1-20210110, 64-bit)"}
 {"TimeStamp": "2023-06-06T20:04:17.1993774+00:00", "Log": "20:04:17.199 UTC [1] LOG: listening on IPv4 address \"0.0.0.0\", port 5432"}
 {"TimeStamp": "2023-06-06T20:04:17.1993986+00:00", "Log": "20:04:17.199 UTC [1] LOG: listening on IPv6 address \"::\", port 5432"}
 {"TimeStamp": "2023-06-06T20:04:17.2092406+00:00", "Log": "20:04:17.209 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432""}
 {"TimeStamp": "2023-06-06T20:04:17.2679896+00:00", "Log": "20:04:17.266 UTC [62] LOG: database system was shut down at 2023-06-06 20:04:16 UTC"}
 {"TimeStamp": "2023-06-06T20:04:17.3182285+00:00", "Log": "20:04:17.317 UTC [1] LOG: database system is ready to accept connections"}]
```

Create an app to test the service

When you create the app, you begin by creating a debug app to use the `psql` CLI to connect to the PostgreSQL instance.

1. Create a `psql` app that binds to the PostgreSQL service.

```
Bash
az containerapp create \
    --name "$PSQL_CLI_APP" \
    --image mcr.microsoft.com/k8se/services/postgres:14 \
    --bind "$PG_SVC" \
    --environment "$ENVIRONMENT" \
    --resource-group "$RESOURCE_GROUP" \
    --min-replicas 1 \
    --max-replicas 1 \
    --command "/bin/sleep" "infinity"
```

2. Run the CLI `exec` command to connect to the test app.

Bash

Bash

```
az containerapp exec \
--name $PSQL_CLI_APP \
--resource-group $RESOURCE_GROUP \
--command /bin/bash
```

When you use `--bind` or `serviceBinds` on the test app, the connection information is injected into the application environment. Once you connect to the test container, you can inspect the values using the `env` command.

Bash

```
env | grep "^\$POSTGRES_"

POSTGRES_HOST=postgres01
POSTGRES_PASSWORD=AiSf...
POSTGRES_SSL=disable
POSTGRES_URL=postgres://postgres:AiSf...@postgres01:5432/postgres?
sslmode=disable
POSTGRES_DATABASE=postgres
POSTGRES_PORT=5432
POSTGRES_USERNAME=postgres
POSTGRES_CONNECTION_STRING=host=postgres01 database=postgres
user=postgres password=AiSf...
```

3. Use `psql` to connect to the service

Bash

```
psql $POSTGRES_URL
```

```
[+ ~ az containerapp exec -n $PSQL_CLI_APP -g $RESOURCE_GROUP --command /bin/bash           29.07s
INFO: Connecting to the container 'pgsql-cloud-cli-app' ...
Use ctrl + D to exit.
INFO: Successfully connected to container: 'pgsql-cloud-cli-app'. [ Revision: 'pgsql-cloud-cli-app--km806fv', Replica: 'pgsql-cloud-cli-app--km806fv--6597cc7696-fwd9w' ]
root@pgsql-cloud-cli-app--km806fv-6597cc7696-fwd9w:/# psql $POSTGRES_URL
psql (14.8 (Debian 14.8-1.pgdg110+1))
Type "help" for help.

postgres=# \list
              List of databases
   Name    |  Owner   | Encoding | Collate | Ctype | Access privileges
postgres | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres
template0 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres=CTc/postgres
template1 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres=CTc/postgres
(3 rows)

postgres=#
```

4. Create a table named `accounts` and insert data.

SQL

```
postgres=# CREATE TABLE accounts (
    user_id serial PRIMARY KEY,
    username VARCHAR ( 50 ) UNIQUE NOT NULL,
    email VARCHAR ( 255 ) UNIQUE NOT NULL,
    created_on TIMESTAMP NOT NULL,
    last_login TIMESTAMP
);

postgres=# INSERT INTO accounts (username, email, created_on)
VALUES
('user1', 'user1@example.com', current_timestamp),
('user2', 'user2@example.com', current_timestamp),
('user3', 'user3@example.com', current_timestamp);

postgres=# SELECT * FROM accounts;
```

```
+ - az containerapp exec -n $PGSQL_CLI_APP -g $RESOURCE_GROUP --command /bin/bash
INFO: Connecting to the container 'pgsql-cloud-cli-app' ...
Use ctrl + D to exit.
INFO: Successfully connected to container: 'pgsql-cloud-cli-app'. [ Revision: 'pgsql-cloud-cli-app--km806fv', Replica: 'pgsql-cloud-cli-app--km806fv-6597cc7696-fwd9w' ]
root@pgsql-cloud-cli-app--km806fv-6597cc7696-fwd9w:/# psql $POSTGRES_URL
psql (14.8 (Debian 14.8-1.pgdg110+1))
Type "help" for help.

postgres=# CREATE TABLE accounts (
    user_id serial PRIMARY KEY,
    username VARCHAR ( 50 ) UNIQUE NOT NULL,
    email VARCHAR ( 255 ) UNIQUE NOT NULL,
    created_on TIMESTAMP NOT NULL,
    last_login TIMESTAMP
);
CREATE TABLE
postgres=# INSERT INTO accounts (username, email, created_on)
VALUES
('user1', 'user1@example.com', current_timestamp),
('user2', 'user2@example.com', current_timestamp),
('user3', 'user3@example.com', current_timestamp);
INSERT 0 3
postgres=# SELECT * FROM accounts;
user_id | username |      email       |      created_on      | last_login
-----+-----+-----+-----+-----+
  1 | user1  | user1@example.com | 2023-06-06 21:28:53.309114 |
  2 | user2  | user2@example.com | 2023-06-06 21:28:53.309114 |
  3 | user3  | user3@example.com | 2023-06-06 21:28:53.309114 |
(3 rows)

postgres=#
2m20s
```

Using a dev service with an existing app

If you already have an app that uses PostgreSQL, you can change how connection information is loaded.

First, create the following environment variables.

Bash

```
POSTGRES_HOST=postgres01
POSTGRES_PASSWORD=AiSf...
POSTGRES_SSL=disable
POSTGRES_URL=postgres://postgres:AiSf...@postgres01:5432/postgres?
sslmode=disable
POSTGRES_DATABASE=postgres
POSTGRES_PORT=5432
POSTGRES_USERNAME=postgres
```

```
POSTGRES_CONNECTION_STRING=host=postgres01 database=postgres user=postgres  
password=AiSf...
```

Using the CLI (or Bicep) you can update the app to add `--bind $PG_SVC` to use the dev service.

Binding to the dev service

Deploy [pgweb](#) to view and manage the PostgreSQL instance.

Bash

See Bicep or `azd` example.

The screenshot shows the pgweb interface for a PostgreSQL database named 'postgres'. The left sidebar shows the schema structure with a 'public' schema containing a 'accounts' table. The main area displays the contents of the 'accounts' table:

	user_id	username	email	created_on	last_login
1	1	user1	user1@example.com	2023-06-06T21:28:53.309114Z	NULL
2	2	user2	user2@example.com	2023-06-06T21:28:53.309114Z	NULL
3	3	user3	user3@example.com	2023-06-06T21:28:53.309114Z	NULL

Below the table, there is a 'TABLE INFORMATION' section providing details about the table's size and structure.

Deploy all resources

Use the following examples to if you want to deploy all resources at once.

Bicep

The following Bicep template contains all the resources in this tutorial.

You can create a `postgres-dev.bicep` file with this content.

Bicep

```

targetScope = 'resourceGroup'
param location string = resourceGroup().location
param appEnvironmentName string = 'aca-env'
param pgSvcName string = 'postgres01'
param pgsqlCliAppName string = 'pgsql-cloud-cli-app'

resource logAnalytics 'Microsoft.OperationalInsights/workspaces@2022-10-01'
= {
  name: '${appEnvironmentName}-log-analytics'
  location: location
  properties: {
    sku: {
      name: 'PerGB2018'
    }
  }
}

resource appEnvironment 'Microsoft.App/managedEnvironments@2023-04-01-preview' = {
  name: appEnvironmentName
  location: location
  properties: {
    appLogsConfiguration: {
      destination: 'log-analytics'
      logAnalyticsConfiguration: {
        customerId: logAnalytics.properties.customerId
        sharedKey: logAnalytics.listKeys().primarySharedKey
      }
    }
  }
}

resource postgres 'Microsoft.App/containerApps@2023-04-01-preview' = {
  name: pgSvcName
  location: location
  properties: {
    environmentId: appEnvironment.id
    configuration: {
      service: {
        type: 'postgres'
      }
    }
  }
}

resource pgsqlCli 'Microsoft.App/containerApps@2023-04-01-preview' = {
  name: pgsqlCliAppName
  location: location
  properties: {
    environmentId: appEnvironment.id
    template: {
      serviceBinds: [
        {
          serviceId: postgres.id
        }
      ]
    }
  }
}

```

```

        }
    ]
  containers: [
    {
      name: 'pgsql'
      image: 'mcr.microsoft.com/k8se/services/postgres:14'
      command: [ '/bin/sleep', 'infinity' ]
    }
  ]
  scale: {
    minReplicas: 1
    maxReplicas: 1
  }
}
}

resource pgweb 'Microsoft.App/containerApps@2023-04-01-preview' = {
  name: 'pgweb'
  location: location
  properties: {
    environmentId: appEnvironment.id
    configuration: {
      ingress: {
        external: true
        targetPort: 8081
      }
    }
    template: {
      serviceBinds: [
        {
          serviceId: postgres.id
          name: 'postgres'
        }
      ]
      containers: [
        {
          name: 'pgweb'
          image: 'docker.io/sosedoff/pgweb:latest'
          command: [
            '/bin/sh'
          ]
          args: [
            '-c'
            'PGWEB_DATABASE_URL=$POSTGRES_URL /usr/bin/pgweb --bind=0.0.0.0
--listen=8081'
          ]
        }
      ]
    }
  }
}

output pgsqlCliExec string = 'az containerapp exec -n ${pgsqlCli.name} -g
${resourceGroup().name} --revision ${pgsqlCli.properties.latestRevisionName}'

```

```
--command /bin/bash'

output postgresLogs string = 'az containerapp logs show -n ${postgres.name}
-g ${resourceGroup().name} --follow --tail 30'

output pgwebUrl string =
'https://${pgweb.properties.configuration.ingress fqdn}'
```

Use the Azure CLI to deploy it the template.

Bash

```
RESOURCE_GROUP="postgres-dev"
LOCATION="northcentralus"

az group create \
--name "$RESOURCE_GROUP" \
--location "$LOCATION"

az deployment group create -g $RESOURCE_GROUP \
--query 'properties.outputs.*.value' \
--template-file postgres-dev.bicep
```

Azure Developer CLI

A [final template](#) is available on GitHub.

Use `azd up` to deploy the template.

Bash

```
git clone https://github.com/Azure-Samples/aca-dev-service-postgres-azd
cd aca-dev-service-postgres-azd
azd up
```

Clean up resources

Once you're done, run the following command to delete the resource group that contains your Container Apps resources.

⊗ Caution

The following command deletes the specified resource group and all resources contained within it. If resources outside the scope of this tutorial exist in the specified resource group, they will also be deleted.

Azure CLI

```
az group delete \
--resource-group $RESOURCE_GROUP
```

Tutorial: Connect services in Azure Container Apps (preview)

Article • 06/13/2023

Azure Container Apps allows you to connect to services that support your app that run in the same environment as your container app.

When in development, your application can quickly create and connect to [dev services](#). These services are easy to create and are development-grade services designed for nonproduction environments.

As you move to production, your application can connect production-grade managed services.

This tutorial shows you how to connect both dev and production grade services to your container app.

In this tutorial, you learn to:

- ✓ Create a new Redis development service
- ✓ Connect a container app to the Redis dev service
- ✓ Disconnect the service from the application
- ✓ Inspect the service running an in-memory cache

Prerequisites

Resource	Description
Azure account	An active subscription is required. If you don't have one, you can create one for free .
Azure CLI	Install the Azure CLI if you don't have it on your machine.
Azure resource group	Create a resource group named my-services-resource-group in the East US region.
Azure Cache for Redis	Create an instance of Azure Cache for Redis in the my-services-resource-group .

Set up

1. Sign in to the Azure CLI.

Azure CLI

```
az login
```

2. Upgrade the Container Apps CLI extension.

Azure CLI

```
az extension add --name containerapp --upgrade
```

3. Register the `Microsoft.App` namespace.

Azure CLI

```
az provider register --namespace Microsoft.App
```

4. Register the `Microsoft.ServiceLinker` namespace.

Azure CLI

```
az provider register --namespace Microsoft.ServiceLinker
```

5. Set up the resource group variable.

Azure CLI

```
RESOURCE_GROUP="my-services-resource-group"
```

6. Create a variable for the Azure Cache for Redis DNS name.

To display a list of the Azure Cache for Redis instances, run the following command.

Azure CLI

```
az redis list --resource-group "$RESOURCE_GROUP" --query "[].name" -o table
```

7. Create a variable to hold your environment name.

Replace `<MY_ENVIRONMENT_NAME>` with the name of your container apps environment.

Azure CLI

```
ENVIRONMENT=<MY_ENVIRONMENT_NAME>
```

8. Set up the location variable.

Azure CLI

```
LOCATION="eastus"
```

9. Create a new environment.

Azure CLI

```
az containerapp env create \
--location "$LOCATION" \
--resource-group "$RESOURCE_GROUP" \
--name "$ENVIRONMENT"
```

With the CLI configured and an environment created, you can now create an application and dev service.

Create a dev service

The sample application manages a set of strings, either in-memory, or in Redis cache.

Create the Redis dev service and name it `myredis`.

Azure CLI

```
az containerapp service redis create \
--name myredis \
--resource-group "$RESOURCE_GROUP" \
--environment "$ENVIRONMENT"
```

Create a container app

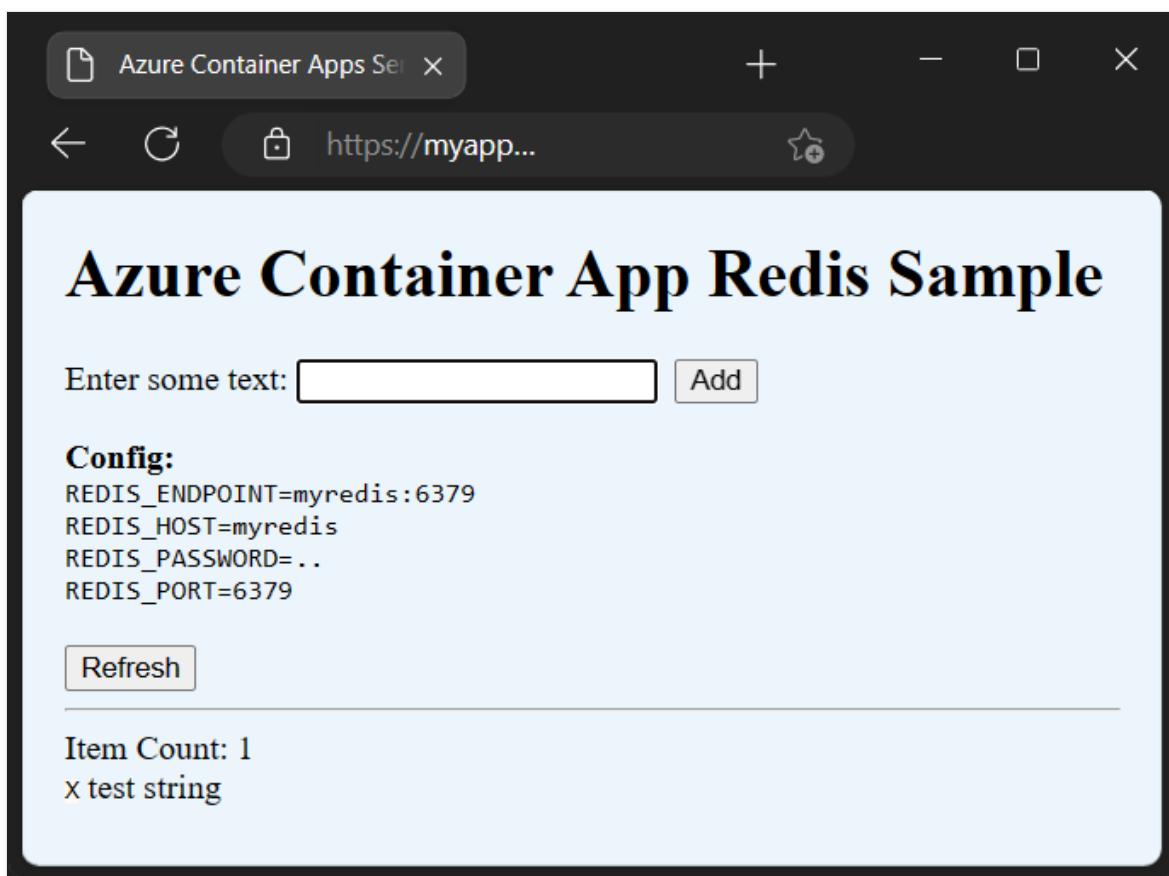
Next, create your internet-accessible container app.

1. Create a new container app and bind it to the Redis service.

Azure CLI

```
az containerapp create \
--name myapp \
--image mcr.microsoft.com/k8se/samples/sample-service-redis:latest \
--ingress external \
--target-port 8080 \
--bind myredis \
--environment "$ENVIRONMENT" \
--resource-group "$RESOURCE_GROUP" \
--query properties.configuration.ingress.fqdn
```

This command returns the fully qualified domain name (FQDN). Paste this location into a web browser so you can inspect the application's behavior throughout this tutorial.



The `containerapp create` command uses the `--bind` option to create a link between the container app and the Redis dev service.

The bind request gathers connection information, including credentials and connection strings, and injects it into the application as environment variables. These values are now available to the application code to use in order to create a connection to the service.

In this case, the following environment variables are available to the application:

Bash

```
REDIS_ENDPOINT=myredis:6379  
REDIS_HOST=myredis  
REDIS_PASSWORD=...  
REDIS_PORT=6379
```

If you access the application via a browser, you can add and remove strings from the Redis database. The Redis cache is responsible for storing application data, so data is available even after the application is restarted after scaling to zero.

You can also remove a binding from your application.

2. Unbind the Redis dev service.

To remove a binding from a container app, use the `--unbind` option.

Azure CLI

```
az containerapp update \  
--name myapp \  
--unbind myredis \  
--resource-group "$RESOURCE_GROUP"
```

The application is written so that if the environment variables aren't defined, then the text strings are stored in memory.

In this state, if the application scales to zero, then data is lost.

You can verify this change by returning to your web browser and refreshing the web application. You can now see the configuration information displayed indicates data is stored in-memory.

Now you can rebind the application to the Redis service, to see your previously stored data.

3. Rebind the Redis dev service.

Azure CLI

```
az containerapp update \  
--name myapp \  
--bind myredis \  
--resource-group "$RESOURCE_GROUP"
```

With the service reconnected, you can refresh the web application to see data stored in Redis.

Connecting to a managed service

When your application is ready to move to production, you can bind your application to a managed service instead of a dev service.

The following steps bind your application to an existing instance of Azure Cache for Redis.

1. Create a variable for your DNS name.

Make sure to replace `<YOUR_DNS_NAME>` with the DNS name of your instance of Azure Cache for Redis.

```
Azure CLI
```

```
AZURE_REDISHOST=<YOUR_DNS_NAME>
```

2. Bind to Azure Cache for Redis.

```
Azure CLI
```

```
az containerapp update \
--name myapp \
--unbind myredis \
--bind "$AZURE_REDISHOST" \
--resource-group "$RESOURCE_GROUP"
```

This command simultaneously removes the development binding and establishes the binding to the production-grade managed service.

3. Return to your browser and refresh the page.

The console prints up values like the following example.

```
Bash
```

```
AZURE_REDISHOST=azureRedis.redis.cache.windows.net
AZURE_REDISHOSTNAME=azureRedis.redis.cache.windows.net
AZURE_REDISHOSTPORT=6380
AZURE_REDISHOSTSSL=true
```

 Note

Environment variable names used for dev mode services and managed service vary slightly.

If you'd like to see the sample code used for this tutorial please see <https://github.com/Azure-Samples/sample-service-redis>.

Now when you add new strings, the values are stored in an instance Azure Cache for Redis instead of the dev service.

Clean up resources

If you don't plan to continue to use the resources created in this tutorial, you can delete the application and the Redis service.

The application and the service are independent. This independence means the service can be connected to any number of applications in the environment and exists until explicitly deleted, even if all applications are disconnect from it.

Run the following commands to delete your container app and the dev service.

Azure CLI

```
az containerapp delete --name myapp  
az containerapp service redis delete --name myredis
```

Alternatively you can delete the resource group to remove the container app and all services.

Azure CLI

```
az group delete \  
--resource-group "$RESOURCE_GROUP"
```

Next steps

[Application lifecycle management](#)

Tutorial: Enable Azure Container Apps on Azure Arc-enabled Kubernetes (Preview)

Article • 05/03/2023

With [Azure Arc-enabled Kubernetes clusters](#), you can create a [Container Apps enabled custom location](#) in your on-premises or cloud Kubernetes cluster to deploy your Azure Container Apps applications as you would any other region.

This tutorial will show you how to enable Azure Container Apps on your Arc-enabled Kubernetes cluster. In this tutorial you will:

- ✓ Create a connected cluster.
- ✓ Create a Log Analytics workspace.
- ✓ Install the Container Apps extension.
- ✓ Create a custom location.
- ✓ Create the Azure Container Apps connected environment.

ⓘ Note

During the preview, Azure Container Apps on Arc are not supported in production configurations. This article provides an example configuration for evaluation purposes only.

This tutorial uses [Azure Kubernetes Service \(AKS\)](#) to provide concrete instructions for setting up an environment from scratch. However, for a production workload, you may not want to enable Azure Arc on an AKS cluster as it is already managed in Azure.

Prerequisites

- An Azure account with an active subscription.
 - If you don't have one, you [can create one for free ↗](#).
- Install the [Azure CLI](#).
- Access to a public or private container registry, such as the [Azure Container Registry](#).

Setup

Install the following Azure CLI extensions.

```
Azure CLI

az extension add --name connectedk8s --upgrade --yes
az extension add --name k8s-extension --upgrade --yes
az extension add --name customlocation --upgrade --yes
az extension remove --name containerapp
az extension add --source https://aka.ms/acaarccli/containerapp-latest-
py2.py3-none-any.whl --yes
```

Register the required namespaces.

```
Azure CLI

az provider register --namespace Microsoft.ExtendedLocation --wait
az provider register --namespace Microsoft.KubernetesConfiguration --
wait
az provider register --namespace Microsoft.App --wait
az provider register --namespace Microsoft.OperationalInsights --wait
```

Set environment variables based on your Kubernetes cluster deployment.

```
Azure CLI

Bash

GROUP_NAME="my-arc-cluster-group"
AKS_CLUSTER_GROUP_NAME="my-aks-cluster-group"
AKS_NAME="my-aks-cluster"
LOCATION="eastus"
```

Create a connected cluster

The following steps help you get started understanding the service, but for production deployments, they should be viewed as illustrative, not prescriptive. See [Quickstart](#):

Connect an existing Kubernetes cluster to Azure Arc for general instructions on creating an Azure Arc-enabled Kubernetes cluster.

1. Create a cluster in Azure Kubernetes Service.

Azure CLI

Azure CLI

```
az group create --name $AKS_CLUSTER_GROUP_NAME --location $LOCATION
az aks create \
    --resource-group $AKS_CLUSTER_GROUP_NAME \
    --name $AKS_NAME \
    --enable-aad \
    --generate-ssh-keys
```

2. Get the [kubeconfig](#) file and test your connection to the cluster. By default, the kubeconfig file is saved to `~/.kube/config`.

Azure CLI

```
az aks get-credentials --resource-group $AKS_CLUSTER_GROUP_NAME --name
$AKS_NAME --admin

kubectl get ns
```

3. Create a resource group to contain your Azure Arc resources.

Azure CLI

Azure CLI

```
az group create --name $GROUP_NAME --location $LOCATION
```

4. Connect the cluster you created to Azure Arc.

Azure CLI

Azure CLI

```
CLUSTER_NAME="${GROUP_NAME}-cluster" # Name of the connected
cluster resource
```

```
az connectedk8s connect --resource-group $GROUP_NAME --name  
$CLUSTER_NAME
```

5. Validate the connection with the following command. It should show the `provisioningState` property as `Succeeded`. If not, run the command again after a minute.

Azure CLI

```
az connectedk8s show --resource-group $GROUP_NAME --name $CLUSTER_NAME
```

Create a Log Analytics workspace

A [Log Analytics workspace](#) provides access to logs for Container Apps applications running in the Azure Arc-enabled Kubernetes cluster. A Log Analytics workspace is optional, but recommended.

1. Create a Log Analytics workspace.

Azure CLI

```
WORKSPACE_NAME="$GROUP_NAME-workspace" # Name of the Log Analytics workspace

az monitor log-analytics workspace create \
    --resource-group $GROUP_NAME \
    --workspace-name $WORKSPACE_NAME
```

2. Run the following commands to get the encoded workspace ID and shared key for an existing Log Analytics workspace. You need them in the next step.

Azure CLI

```
LOG_ANALYTICS_WORKSPACE_ID=$(az monitor log-analytics workspace show \
    --resource-group $GROUP_NAME \
    --workspace-name $WORKSPACE_NAME \
    --query customerId \
    --output tsv)
```

```
LOG_ANALYTICS_WORKSPACE_ID_ENC=$(printf %s  
$LOG_ANALYTICS_WORKSPACE_ID | base64 -w0) # Needed for the next  
step  
LOG_ANALYTICS_KEY=$(az monitor log-analytics workspace get-shared-  
keys \  
--resource-group $GROUP_NAME \  
--workspace-name $WORKSPACE_NAME \  
--query primarySharedKey \  
--output tsv)  
LOG_ANALYTICS_KEY_ENC=$(printf %s $LOG_ANALYTICS_KEY | base64 -w0)  
# Needed for the next step
```

Install the Container Apps extension

1. Set the following environment variables to the desired name of the [Container Apps extension](#), the cluster namespace in which resources should be provisioned, and the name for the Azure Container Apps connected environment. Choose a unique name for <connected-environment-name>. The connected environment name will be part of the domain name for app you'll create in the Azure Container Apps connected environment.

Azure CLI

Bash

```
EXTENSION_NAME="appenv-ext"  
NAMESPACE="appplat-ns"  
CONNECTED_ENVIRONMENT_NAME="<connected-environment-name>"
```

2. Install the Container Apps extension to your Azure Arc-connected cluster with Log Analytics enabled. Log Analytics can't be added to the extension later.

Azure CLI

```
az k8s-extension create \  
--resource-group $GROUP_NAME \  
--name $EXTENSION_NAME \  
--cluster-type connectedClusters \  
--cluster-name $CLUSTER_NAME \  
--extension-type 'Microsoft.App.Environment' \  
--release-train stable \  
--auto-upgrade-minor-version true \  
--enable-log-analytics
```

```

--scope cluster \
--release-namespace $NAMESPACE \
--configuration-settings
"Microsoft.CustomLocation.ServiceAccount=default" \
--configuration-settings "appsNamespace=${NAMESPACE}" \
--configuration-settings
"clusterName=${CONNECTED_ENVIRONMENT_NAME}" \
--configuration-settings
"envoy.annotations.service.beta.kubernetes.io/azure-load-balancer-
resource-group=${AKS_CLUSTER_GROUP_NAME}" \
--configuration-settings "logProcessor.appLogs.destination=log-
analytics" \
--configuration-protected-settings
"logProcessor.appLogs.logAnalyticsConfig.customerId=${LOG_ANALYTICS_
_WORKSPACE_ID_ENC}" \
--configuration-protected-settings
"logProcessor.appLogs.logAnalyticsConfig.sharedKey=${LOG_ANALYTICS_
KEY_ENC}"

```

Note

To install the extension without Log Analytics integration, remove the last three `--configuration-settings` parameters from the command.

The following table describes the various `--configuration-settings` parameters when running the command:

Parameter	Description
<code>Microsoft.CustomLocation.ServiceAccount</code>	The service account created for the custom location. It's recommended that it's set to the value <code>default</code> .
<code>appsNamespace</code>	The namespace used to create the app definitions and revisions. It must match that of the extension release namespace.
<code>clusterName</code>	The name of the Container Apps extension Kubernetes environment that will be created against this extension.

Parameter	Description
<code>logProcessor.appLogs.destination</code>	Optional. Destination for application logs. Accepts <code>log-analytics</code> or <code>none</code> , choosing none disables platform logs.
<code>logProcessor.appLogs.logAnalyticsConfig.customerId</code>	Required only when <code>logProcessor.appLogs.destination</code> is set to <code>log-analytics</code> . The base64-encoded Log analytics workspace ID. This parameter should be configured as a protected setting.
<code>logProcessor.appLogs.logAnalyticsConfig.sharedKey</code>	Required only when <code>logProcessor.appLogs.destination</code> is set to <code>log-analytics</code> . The base64-encoded Log analytics workspace shared key. This parameter should be configured as a protected setting.
<code>envoy.annotations.service.beta.kubernetes.io/azure-load-balancer-resource-group</code>	The name of the resource group in which the Azure Kubernetes Service cluster resides. Valid and required only when the underlying cluster is Azure Kubernetes Service.

- Save the `id` property of the Container Apps extension for later.

Azure CLI

```
Azure CLI

EXTENSION_ID=$(az k8s-extension show \
--cluster-type connectedClusters \
--cluster-name $CLUSTER_NAME \
--resource-group $GROUP_NAME \
--name $EXTENSION_NAME \
--query id \
--output tsv)
```

- Wait for the extension to fully install before proceeding. You can have your terminal session wait until it completes by running the following command:

Azure CLI

```
az resource wait --ids $EXTENSION_ID --custom  
"properties.provisioningState!='Pending'" --api-version "2020-07-01-  
preview"
```

You can use `kubectl` to see the pods that have been created in your Kubernetes cluster:

Bash

```
kubectl get pods -n $NAMESPACE
```

To learn more about these pods and their role in the system, see [Azure Arc overview](#).

Create a custom location

The [custom location](#) is an Azure location that you assign to the Azure Container Apps connected environment.

1. Set the following environment variables to the desired name of the custom location and for the ID of the Azure Arc-connected cluster.

Azure CLI

Azure CLI

```
CUSTOM_LOCATION_NAME="my-custom-location" # Name of the custom  
location  
CONNECTED_CLUSTER_ID=$(az connectedk8s show --resource-group  
$GROUP_NAME --name $CLUSTER_NAME --query id --output tsv)
```

2. Create the custom location:

Azure CLI

```
az customlocation create \  
--resource-group $GROUP_NAME \  
--name $CUSTOM_LOCATION_NAME \  
--host-resource-id $CONNECTED_CLUSTER_ID \  
--namespace $NAMESPACE \  
--cluster-extension-ids $EXTENSION_ID
```

⚠ Note

If you experience issues creating a custom location on your cluster, you may need to **enable the custom location feature on your cluster**. This is required if logged into the CLI using a Service Principal or if you are logged in with an Azure Active Directory user with restricted permissions on the cluster resource.

3. Validate that the custom location is successfully created with the following command. The output should show the `provisioningState` property as `Succeeded`. If not, rerun the command after a minute.

Azure CLI

```
az customlocation show --resource-group $GROUP_NAME --name  
$CUSTOM_LOCATION_NAME
```

4. Save the custom location ID for the next step.

Azure CLI

```
CUSTOM_LOCATION_ID=$(az customlocation show \  
--resource-group $GROUP_NAME \  
--name $CUSTOM_LOCATION_NAME \  
--query id \  
--output tsv)
```

Create the Azure Container Apps connected environment

Before you can start creating apps in the custom location, you need an [Azure Container Apps connected environment](#).

1. Create the Container Apps connected environment:

Azure CLI

Azure CLI

```
az containerapp connected-env create \
--resource-group $GROUP_NAME \
--name $CONNECTED_ENVIRONMENT_NAME \
--custom-location $CUSTOM_LOCATION_ID \
--location $LOCATION
```

2. Validate that the Container Apps connected environment is successfully created with the following command. The output should show the `provisioningState` property as `Succeeded`. If not, run it again after a minute.

Azure CLI

```
az containerapp connected-env show --resource-group $GROUP_NAME --name
$CONNECTED_ENVIRONMENT_NAME
```

Next steps

[Create a container app on Azure Arc](#)

Tutorial: Create an Azure Container App on Azure Arc-enabled Kubernetes (Preview)

Article • 03/20/2023

In this tutorial, you create a Container app to an Azure Arc-enabled Kubernetes cluster (Preview) and learn to:

- ✓ Create a container app on Azure Arc
- ✓ View your application's diagnostics

Prerequisites

Before you proceed to create a container app, you first need to set up an [Azure Arc-enabled Kubernetes cluster to run Azure Container Apps](#).

Add Azure CLI extensions

Launch the Bash environment in [Azure Cloud Shell](#).



Next, add the required Azure CLI extensions.

⚠️ Warning

The following command installs a custom Container Apps extension that can't be used with the public cloud service. You need to uninstall the extension if you switch back to the Azure public cloud.

Azure CLI

```
az extension add --upgrade --yes --name customlocation  
az extension remove --name containerapp  
az extension add -s https://aka.ms/acaarccli/containerapp-1latest-py2.py3-  
none-any.whl --yes
```

Create a resource group

Create a resource group for the services created in this tutorial.

Azure CLI

```
myResourceGroup="my-container-apps-resource-group"
az group create --name $myResourceGroup --location eastus
```

Get custom location information

Get the following location group, name, and ID from your cluster administrator. See [Create a custom location](#) for details.

Azure CLI

```
customLocationGroup="<RESOURCE_GROUP_CONTAINING_CUSTOM_LOCATION>"
```

Azure CLI

```
customLocationName="<NAME_OF_CUSTOM_LOCATION>"
```

Get the custom location ID.

Azure CLI

```
customLocationId=$(az customlocation show \
    --resource-group $customLocationGroup \
    --name $customLocationName \
    --query id \
    --output tsv)
```

Retrieve connected environment ID

Now that you have the custom location ID, you can query for the connected environment.

A connected environment is largely the same as a standard Container Apps environment, but network restrictions are controlled by the underlying Arc-enabled Kubernetes cluster.

azure

```
myContainerApp="my-container-app"
myConnectedEnvironment=$(az containerapp connected-env list --custom-
```

```
location $customLocationId -o tsv --query '[] .id')
```

Create an app

The following example creates a Node.js app.

Azure CLI

```
az containerapp create \
    --resource-group $myResourceGroup \
    --name $myContainerApp \
    --environment $myConnectedEnvironment \
    --environment-type connected \
    --image mcr.microsoft.com/azuredocs/containerapps-helloworld:latest \
    --target-port 80 \
    --ingress 'external'

az containerapp browse --resource-group $myResourceGroup --name
$myContainerApp
```

Get diagnostic logs using Log Analytics

ⓘ Note

A Log Analytics configuration is required as you [install the Container Apps extension](#) to view diagnostic information. If you installed the extension without Log Analytics, skip this step.

Navigate to the [Log Analytics workspace that's configured with your Container Apps extension](#), then select **Logs** in the left navigation.

Run the following sample query to show logs over the past 72 hours.

If there's an error when running a query, try again in 10-15 minutes. There may be a delay for Log Analytics to start receiving logs from your application.

Kusto

```
let StartTime = ago(72h);
let EndTime = now();
ContainerAppConsoleLogs_CL
| where TimeGenerated between (StartTime .. EndTime)
| where ContainerAppName_s =~ "my-container-app"
```

The application logs for all the apps hosted in your Kubernetes cluster are logged to the Log Analytics workspace in the custom log table named `ContainerAppConsoleLogs_CL`.

- `Log_s` contains application logs for a given Container Apps extension
- `AppName_s` contains the Container App app name. In addition to logs you write via your application code, the `Log_s` column also contains logs on container startup and shutdown.

You can learn more about log queries in [getting started with Kusto](#).

Next steps

- [Communication between microservices](#)

Container Apps ARM template API specification

Article • 03/10/2023

Azure Container Apps deployments are powered by an Azure Resource Manager (ARM) template. Some Container Apps CLI commands also support using a YAML template to specify a resource.

ⓘ Note

Azure Container Apps resources have migrated from the `Microsoft.Web` namespace to the `Microsoft.App` namespace. Refer to [Namespace migration from Microsoft.Web to Microsoft.App in March 2022](#) for more details.

Container Apps environment

The following tables describe the properties available in the Container Apps environment resource.

Resource

A container app resource of the ARM template has the following properties:

Property	Description	Data type
<code>name</code>	The Container Apps environment name.	string
<code>location</code>	The Azure region where the Container Apps environment is deployed.	string
<code>type</code>	<code>Microsoft.App/managedEnvironments</code> – the ARM resource type	string

properties

A resource's `properties` object has the following properties:

Property	Description	Data type	Read only
<code>daprAIInstrumentationKey</code>	The Application Insights instrumentation key used by Dapr.	string	No

Property	Description	Data type	Read only
appLogsConfiguration	The environment's logging configuration.	Object	No

Examples

The following example ARM template deploys a Container Apps environment.

ⓘ Note

The commands to create container app environments don't support YAML configuration input.

JSON

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "location": {
      "defaultValue": "canadacentral",
      "type": "String"
    },
    "dapr_ai_instrumentation_key": {
      "defaultValue": "",
      "type": "String"
    },
    "environment_name": {
      "defaultValue": "myenvironment",
      "type": "String"
    },
    "log_analytics_customer_id": {
      "type": "String"
    },
    "log_analytics_shared_key": {
      "type": "SecureString"
    },
    "storage_account_name": {
      "type": "String"
    },
    "storage_account_key": {
      "type": "SecureString"
    },
    "storage_share_name": {
      "type": "String"
    }
  }
},
```

```

"variables": {},
"resources": [
{
  "type": "Microsoft.App/managedEnvironments",
  "apiVersion": "2022-03-01",
  "name": "[parameters('environment_name')]",
  "location": "[parameters('location')]",
  "properties": {
    "daprAIInstrumentationKey": "[parameters('dapr_ai_instrumentation_key')]",
    "appLogsConfiguration": {
      "destination": "log-analytics",
      "logAnalyticsConfiguration": {
        "customerId": "[parameters('log_analytics_customer_id')]",
        "sharedKey": "[parameters('log_analytics_shared_key')]"
      }
    }
  },
  "resources": [
    {
      "type": "storages",
      "name": "myazurefiles",
      "apiVersion": "2022-03-01",
      "dependsOn": [
        "[resourceId('Microsoft.App/managedEnvironments',
parameters('environment_name'))]"
      ],
      "properties": {
        "azureFile": {
          "accountName": "[parameters('storage_account_name')]",
          "accountKey": "[parameters('storage_account_key')]",
          "shareName": "[parameters('storage_share_name')]",
          "accessMode": "ReadWrite"
        }
      }
    }
  ]
}
]
}

```

Container app

The following tables describe the properties available in the container app resource.

Resource

A container app resource of the ARM template has the following properties:

Property	Description	Data type
----------	-------------	-----------

Property	Description	Data type
name	The Container Apps application name.	string
location	The Azure region where the Container Apps instance is deployed.	string
tags	Collection of Azure tags associated with the container app.	array
type	Microsoft.App/containerApps – the ARM resource type	string

In this example, you put your values in place of the placeholder tokens surrounded by <> brackets.

properties

A resource's `properties` object has the following properties:

Property	Description	Data type	Read only
provisioningState	The state of a long running operation, for example when new container revision is created. Possible values include: provisioning, provisioned, failed. Check if app is up and running.	string	Yes
environmentId	The environment ID for your container app. This is a required property.	string	No
latestRevisionName	The name of the latest revision.	string	Yes
latestRevisionFqdn	The latest revision's URL.	string	Yes

The `environmentId` value takes the following form:

Console

```
/subscriptions/<SUBSCRIPTION_ID>/resourcegroups/<RESOURCE_GROUP_NAME>/providers/Microsoft.App/environmentId/<ENVIRONMENT_NAME>
```

In this example, you put your values in place of the placeholder tokens surrounded by <> brackets.

properties.configuration

A resource's `properties.configuration` object has the following properties:

Property	Description	Data type
<code>activeRevisionsMode</code>	Setting to <code>single</code> automatically deactivates old revisions, and only keeps the latest revision active. Setting to <code>multiple</code> allows you to maintain multiple revisions.	string
<code>secrets</code>	Defines secret values in your container app.	object
<code>ingress</code>	Object that defines public accessibility configuration of a container app.	object
<code>registries</code>	Configuration object that references credentials for private container registries. Entries defined with <code>secretref</code> reference the secrets configuration object.	object
<code>dapr</code>	Configuration object that defines the Dapr settings for the container app.	object

Changes made to the `configuration` section are [application-scope changes](#), which doesn't trigger a new revision.

properties.template

A resource's `properties.template` object has the following properties:

Property	Description	Data type
<code>revisionSuffix</code>	A friendly name for a revision. This value must be unique as the runtime rejects any conflicts with existing revision name suffix values.	string
<code>containers</code>	Configuration object that defines what container images are included in the container app.	object
<code>scale</code>	Configuration object that defines scale rules for the container app.	object

Changes made to the `template` section are [revision-scope changes](#), which triggers a new revision.

Examples

For details on health probes, refer to [Health probes in Azure Container Apps](#).

ARM template

The following example ARM template deploys a container app.

JSON

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2015-01-  
01/deploymentTemplate.json#",  
  "contentVersion": "1.0.0.0",  
  "parameters": {  
    "containerappName": {  
      "defaultValue": "mycontainerapp",  
      "type": "String"  
    },  
    "location": {  
      "defaultValue": "canadacentral",  
      "type": "String"  
    },  
    "environment_name": {  
      "defaultValue": "myenvironment",  
      "type": "String"  
    },  
    "container_image": {  
      "type": "String"  
    },  
    "registry_password": {  
      "type": "SecureString"  
    }  
  },  
  "variables": {},  
  "resources": [  
    {  
      "apiVersion": "2022-11-01-preview",  
      "type": "Microsoft.App/containerApps",  
      "name": "[parameters('containerappName')]",  
      "location": "[parameters('location')]",  
      "identity": {  
        "type": "None"  
      },  
      "properties": {  
        "managedEnvironmentId": "  
[resourceId('Microsoft.App/managedEnvironments',  
parameters('environment_name'))]",  
        "configuration": {  
          "secrets": [  
            {  
              "name": "mysecret",  
              "value": "thisismysecret"  
            },  
            {  
              "name": "myregistrypassword",  
              "value": "[parameters('registry_password')]"  
            }  
          ],  
        }  
      }  
    }  
  ]  
}
```

```
"ingress": {
    "external": true,
    "targetPort": 80,
    "allowInsecure": false,
    "traffic": [
        {
            "latestRevision": true,
            "weight": 100
        }
    ]
},
"registries": [
    {
        "server": "myregistry.azurecr.io",
        "username": "[parameters('containerappName')]",
        "passwordSecretRef": "myregistrypassword"
    }
],
"dapr": {
    "appId": "[parameters('containerappName')]",
    "appPort": 80,
    "appProtocol": "http",
    "enabled": true
},
"template": {
    "revisionSuffix": "myrevision",
    "containers": [
        {
            "name": "main",
            "image": "[parameters('container_image')]",
            "env": [
                {
                    "name": "HTTP_PORT",
                    "value": "80"
                },
                {
                    "name": "SECRET_VAL",
                    "secretRef": "mysecret"
                }
            ],
            "command": [
                "npm",
                "start"
            ],
            "resources": {
                "cpu": 0.5,
                "memory": "1Gi"
            },
            "probes": [
                {
                    "type": "liveness",
                    "httpGet": {
                        "path": "/health",
                        "port": 8080,
                        "interval": "10s",
                        "timeout": "1s"
                    }
                }
            ]
        }
    ]
}
```

```
        "httpHeaders": [
            {
                "name": "Custom-Header",
                "value": "liveness probe"
            }
        ],
        "initialDelaySeconds": 7,
        "periodSeconds": 3
    },
    {
        "type": "readiness",
        "tcpSocket": {
            "port": 8081
        },
        "initialDelaySeconds": 10,
        "periodSeconds": 3
    },
    {
        "type": "startup",
        "httpGet": {
            "path": "/startup",
            "port": 8080,
            "httpHeaders": [
                {
                    "name": "Custom-Header",
                    "value": "startup probe"
                }
            ]
        },
        "initialDelaySeconds": 3,
        "periodSeconds": 3
    }
],
"volumeMounts": [
    {
        "mountPath": "/myempty",
        "volumeName": "myempty"
    },
    {
        "mountPath": "/myfile",
        "volumeName": "azure-files-volume"
    },
    {
        "mountPath": "/mysecrets",
        "volumeName": "mysecrets"
    }
]
],
"scale": {
    "minReplicas": 1,
    "maxReplicas": 3
},
"volumes": [
```

```
{  
    "name": "myempty",  
    "storageType": "EmptyDir"  
},  
{  
    "name": "azure-files-volume",  
    "storageType": "AzureFile",  
    "storageName": "myazurerefiles"  
},  
{  
    "name": "mysecrets",  
    "storageType": "Secret",  
    "secrets": [  
        {  
            "secretRef": "mysecret",  
            "path": "mysecret.txt"  
        }  
    ]  
}  
]  
}  
]
```

az containerapp

Reference

Manage Azure Container Apps.

Commands

az containerapp auth	Manage containerapp authentication and authorization.
az containerapp auth apple	Manage containerapp authentication and authorization of the Apple identity provider.
az containerapp auth apple show	Show the authentication settings for the Apple identity provider.
az containerapp auth apple update	Update the client id and client secret for the Apple identity provider.
az containerapp auth facebook	Manage containerapp authentication and authorization of the Facebook identity provider.
az containerapp auth facebook show	Show the authentication settings for the Facebook identity provider.
az containerapp auth facebook update	Update the app id and app secret for the Facebook identity provider.
az containerapp auth github	Manage containerapp authentication and authorization of the GitHub identity provider.
az containerapp auth github show	Show the authentication settings for the GitHub identity provider.
az containerapp auth github update	Update the client id and client secret for the GitHub identity provider.
az containerapp auth google	Manage containerapp authentication and authorization of the Google identity provider.
az containerapp auth google show	Show the authentication settings for the Google identity provider.
az containerapp auth google update	Update the client id and client secret for the Google identity provider.
az containerapp auth microsoft	Manage containerapp authentication and authorization of the Microsoft identity provider.

<code>az containerapp auth microsoft show</code>	Show the authentication settings for the Azure Active Directory identity provider.
<code>az containerapp auth microsoft update</code>	Update the client id and client secret for the Azure Active Directory identity provider.
<code>az containerapp auth openid-connect</code>	Manage containerapp authentication and authorization of the custom OpenID Connect identity providers.
<code>az containerapp auth openid-connect add</code>	Configure a new custom OpenID Connect identity provider.
<code>az containerapp auth openid-connect remove</code>	Removes an existing custom OpenID Connect identity provider.
<code>az containerapp auth openid-connect show</code>	Show the authentication settings for the custom OpenID Connect identity provider.
<code>az containerapp auth openid-connect update</code>	Update the client id and client secret setting name for an existing custom OpenID Connect identity provider.
<code>az containerapp auth show</code>	Show the authentication settings for the containerapp.
<code>az containerapp auth twitter</code>	Manage containerapp authentication and authorization of the Twitter identity provider.
<code>az containerapp auth twitter show</code>	Show the authentication settings for the Twitter identity provider.
<code>az containerapp auth twitter update</code>	Update the consumer key and consumer secret for the Twitter identity provider.
<code>az containerapp auth update</code>	Update the authentication settings for the containerapp.
<code>az containerapp browse</code>	Open a containerapp in the browser, if possible.
<code>az containerapp compose</code>	Commands to create Azure Container Apps from Compose specifications.
<code>az containerapp compose create</code>	Create one or more Container Apps in a new or existing Container App Environment from a Compose specification.
<code>az containerapp connection</code>	Commands to manage containerapp connections.
<code>az containerapp connection create</code>	Create a connection between a containerapp and a target resource.
<code>az containerapp connection create appconfig</code>	Create a containerapp connection to appconfig.
<code>az containerapp connection create confluent-cloud</code>	Create a containerapp connection to confluent-cloud.

az containerapp connection create cosmos-cassandra	Create a containerapp connection to cosmos-cassandra.
az containerapp connection create cosmos-gremlin	Create a containerapp connection to cosmos-gremlin.
az containerapp connection create cosmos-mongo	Create a containerapp connection to cosmos-mongo.
az containerapp connection create cosmos-sql	Create a containerapp connection to cosmos-sql.
az containerapp connection create cosmos-table	Create a containerapp connection to cosmos-table.
az containerapp connection create eventhub	Create a containerapp connection to eventhub.
az containerapp connection create keyvault	Create a containerapp connection to keyvault.
az containerapp connection create mysql	Create a containerapp connection to mysql.
az containerapp connection create mysql-flexible	Create a containerapp connection to mysql-flexible.
az containerapp connection create postgres	Create a containerapp connection to postgres.
az containerapp connection create postgres-flexible	Create a containerapp connection to postgres-flexible.
az containerapp connection create redis	Create a containerapp connection to redis.
az containerapp connection create redis-enterprise	Create a containerapp connection to redis-enterprise.
az containerapp connection create servicebus	Create a containerapp connection to servicebus.
az containerapp connection create signalr	Create a containerapp connection to signalr.
az containerapp connection create sql	Create a containerapp connection to sql.
az containerapp connection create storage-blob	Create a containerapp connection to storage-blob.
az containerapp connection create storage-file	Create a containerapp connection to storage-file.

az containerapp connection create storage-queue	Create a containerapp connection to storage-queue.
az containerapp connection create storage-table	Create a containerapp connection to storage-table.
az containerapp connection create webpubsub	Create a containerapp connection to webpubsub.
az containerapp connection delete	Delete a containerapp connection.
az containerapp connection list	List connections of a containerapp.
az containerapp connection list-configuration	List source configurations of a containerapp connection.
az containerapp connection list-support-types	List client types and auth types supported by containerapp connections.
az containerapp connection show	Get the details of a containerapp connection.
az containerapp connection update	Update a containerapp connection.
az containerapp connection update appconfig	Update a containerapp to appconfig connection.
az containerapp connection update confluent-cloud	Update a containerapp to confluent-cloud connection.
az containerapp connection update cosmos-cassandra	Update a containerapp to cosmos-cassandra connection.
az containerapp connection update cosmos-gremlin	Update a containerapp to cosmos-gremlin connection.
az containerapp connection update cosmos-mongo	Update a containerapp to cosmos-mongo connection.
az containerapp connection update cosmos-sql	Update a containerapp to cosmos-sql connection.
az containerapp connection update cosmos-table	Update a containerapp to cosmos-table connection.
az containerapp connection update eventhub	Update a containerapp to eventhub connection.
az containerapp connection	Update a containerapp to keyvault connection.

update keyvault	
az containerapp connection update mysql	Update a containerapp to mysql connection.
az containerapp connection update mysql-flexible	Update a containerapp to mysql-flexible connection.
az containerapp connection update postgres	Update a containerapp to postgres connection.
az containerapp connection update postgres-flexible	Update a containerapp to postgres-flexible connection.
az containerapp connection update redis	Update a containerapp to redis connection.
az containerapp connection update redis-enterprise	Update a containerapp to redis-enterprise connection.
az containerapp connection update servicebus	Update a containerapp to servicebus connection.
az containerapp connection update signalr	Update a containerapp to signalr connection.
az containerapp connection update sql	Update a containerapp to sql connection.
az containerapp connection update storage-blob	Update a containerapp to storage-blob connection.
az containerapp connection update storage-file	Update a containerapp to storage-file connection.
az containerapp connection update storage-queue	Update a containerapp to storage-queue connection.
az containerapp connection update storage-table	Update a containerapp to storage-table connection.
az containerapp connection update webpubsub	Update a containerapp to webpubsub connection.
az containerapp connection validate	Validate a containerapp connection.
az containerapp connection wait	Place the CLI in a waiting state until a condition of the connection is met.
az containerapp create	Create a container app.
az containerapp dapr	Commands to manage Dapr. To manage Dapr components, see

```
az containerapp env dapr-component.
```

az containerapp dapr disable	Disable Dapr for a container app. Removes existing values.
az containerapp dapr enable	Enable Dapr for a container app. Updates existing values.
az containerapp delete	Delete a container app.
az containerapp env	Commands to manage Container Apps environments.
az containerapp env certificate	Commands to manage certificates for the Container Apps environment.
az containerapp env certificate create	Create a managed certificate.
az containerapp env certificate delete	Delete a certificate from the Container Apps environment.
az containerapp env certificate list	List certificates for an environment.
az containerapp env certificate upload	Add or update a certificate.
az containerapp env create	Create a Container Apps environment.
az containerapp env dapr-component	Commands to manage Dapr components for the Container Apps environment.
az containerapp env dapr-component list	List Dapr components for an environment.
az containerapp env dapr-component remove	Remove a Dapr component from an environment.
az containerapp env dapr-component set	Create or update a Dapr component.
az containerapp env dapr-component show	Show the details of a Dapr component.
az containerapp env delete	Delete a Container Apps environment.
az containerapp env list	List Container Apps environments by subscription or resource group.
az containerapp env logs	Show container app environment logs.
az containerapp env logs show	Show past environment logs and/or print logs in real time (with the --follow parameter).
az containerapp env show	Show details of a Container Apps environment.

<code>az containerapp env storage</code>	Commands to manage storage for the Container Apps environment.
<code>az containerapp env storage list</code>	List the storages for an environment.
<code>az containerapp env storage remove</code>	Remove a storage from an environment.
<code>az containerapp env storage set</code>	Create or update a storage.
<code>az containerapp env storage show</code>	Show the details of a storage.
<code>az containerapp env update</code>	Update a Container Apps environment.
<code>az containerapp env workload-profile</code>	Manage the workload profiles of a Container Apps environment.
<code>az containerapp env workload-profile add</code>	Create a workload profile in a Container Apps environment.
<code>az containerapp env workload-profile delete</code>	Delete a workload profile from a Container Apps environment.
<code>az containerapp env workload-profile list</code>	List the workload profiles from a Container Apps environment.
<code>az containerapp env workload-profile list-supported</code>	List the supported workload profiles in a region.
<code>az containerapp env workload-profile set</code>	Create or update an existing workload profile in a Container Apps environment.
<code>az containerapp env workload-profile show</code>	Show a workload profile from a Container Apps environment.
<code>az containerapp env workload-profile update</code>	Update an existing workload profile in a Container Apps environment.
<code>az containerapp exec</code>	Open an SSH-like interactive shell within a container app replica.
<code>az containerapp github-action</code>	Commands to manage GitHub Actions.
<code>az containerapp github-action add</code>	Add a GitHub Actions workflow to a repository to deploy a container app.
<code>az containerapp github-action delete</code>	Remove a previously configured Container Apps GitHub Actions workflow from a repository.
<code>az containerapp github-action</code>	Show the GitHub Actions configuration on a container app.

show	
az containerapp hostname	Commands to manage hostnames of a container app.
az containerapp hostname add	Add the hostname to a container app without binding.
az containerapp hostname bind	Add or update the hostname and binding with a certificate.
az containerapp hostname delete	Delete hostnames from a container app.
az containerapp hostname list	List the hostnames of a container app.
az containerapp identity	Commands to manage managed identities.
az containerapp identity assign	Assign managed identity to a container app.
az containerapp identity remove	Remove a managed identity from a container app.
az containerapp identity show	Show managed identities of a container app.
az containerapp ingress	Commands to manage ingress and traffic-splitting.
az containerapp ingress access-restriction	Commands to manage IP access restrictions.
az containerapp ingress access-restriction list	List IP access restrictions for a container app.
az containerapp ingress access-restriction remove	Remove IP access restrictions from a container app.
az containerapp ingress access-restriction set	Configure IP access restrictions for a container app.
az containerapp ingress cors	Commands to manage CORS policy for a container app.
az containerapp ingress cors disable	Disable CORS policy for a container app.
az containerapp ingress cors enable	Enable CORS policy for a container app.
az containerapp ingress cors show	Show CORS policy for a container app.
az containerapp ingress cors update	Update CORS policy for a container app.

<code>az containerapp ingress disable</code>	Disable ingress for a container app.
<code>az containerapp ingress enable</code>	Enable or update ingress for a container app.
<code>az containerapp ingress show</code>	Show details of a container app's ingress.
<code>az containerapp ingress sticky-sessions</code>	Commands to set Sticky session affinity for a container app.
<code>az containerapp ingress sticky-sessions set</code>	Configure Sticky session for a container app.
<code>az containerapp ingress sticky-sessions show</code>	Show the Affinity for a container app.
<code>az containerapp ingress traffic</code>	Commands to manage traffic-splitting.
<code>az containerapp ingress traffic set</code>	Configure traffic-splitting for a container app.
<code>az containerapp ingress traffic show</code>	Show traffic-splitting configuration for a container app.
<code>az containerapp ingress update</code>	Update ingress for a container app.
<code>az containerapp job</code>	Commands to manage Container Apps jobs.
<code>az containerapp job create</code>	Create a container apps job.
<code>az containerapp job delete</code>	Delete a Container Apps Job.
<code>az containerapp job execution</code>	Commands to view executions of a Container App Job.
<code>az containerapp job execution list</code>	Get list of all executions of a Container App Job.
<code>az containerapp job execution show</code>	Get execution of a Container App Job.
<code>az containerapp job identity</code>	Commands to manage managed identities for container app job.
<code>az containerapp job identity assign</code>	Assign managed identity to a container app job.
<code>az containerapp job identity remove</code>	Remove a managed identity from a container app job.
<code>az containerapp job identity show</code>	Show managed identities of a container app job.

az containerapp job list	List Container Apps Job by subscription or resource group.
az containerapp job secret	Commands to manage secrets.
az containerapp job secret list	List the secrets of a container app job.
az containerapp job secret remove	Remove secrets from a container app job.
az containerapp job secret set	Create/update secrets.
az containerapp job secret show	Show details of a secret.
az containerapp job show	Show details of a Container Apps Job.
az containerapp job start	Start a Container Apps Job execution.
az containerapp job stop	Stops a Container Apps Job execution.
az containerapp job update	Update a Container Apps Job.
az containerapp list	List container apps.
az containerapp logs	Show container app logs.
az containerapp logs show	Show past logs and/or print logs in real time (with the --follow parameter). Note that the logs are only taken from one revision, replica, and container (for non-system logs).
az containerapp patch	Patch Azure Container Apps. Patching is only available for the apps built using the source to cloud feature. See https://aka.ms/aca-local-source-to-cloud .
az containerapp patch apply	List and apply container apps to be patched. Patching is only available for the apps built using the source to cloud feature. See https://aka.ms/aca-local-source-to-cloud .
az containerapp patch interactive	List and select container apps to be patched in an interactive way. Patching is only available for the apps built using the source to cloud feature. See https://aka.ms/aca-local-source-to-cloud .
az containerapp patch list	List container apps that can be patched. Patching is only available for the apps built using the source to cloud feature. See https://aka.ms/aca-local-source-to-cloud .
az containerapp registry	Commands to manage container registry information.
az containerapp registry list	List container registries configured in a container app.
az containerapp registry remove	Remove a container registry's details.

az containerapp registry set	Add or update a container registry's details.
az containerapp registry show	Show details of a container registry.
az containerapp replica	Manage container app replicas.
az containerapp replica list	List a container app revision's replica.
az containerapp replica show	Show a container app replica.
az containerapp revision	Commands to manage revisions.
az containerapp revision activate	Activate a revision.
az containerapp revision copy	Create a revision based on a previous revision.
az containerapp revision deactivate	Deactivate a revision.
az containerapp revision label	Manage revision labels assigned to traffic weights.
az containerapp revision label add	Set a revision label to a revision with an associated traffic weight.
az containerapp revision label remove	Remove a revision label from a revision with an associated traffic weight.
az containerapp revision label swap	Swap a revision label between two revisions with associated traffic weights.
az containerapp revision list	List a container app's revisions.
az containerapp revision restart	Restart a revision.
az containerapp revision set-mode	Set the revision mode of a container app.
az containerapp revision show	Show details of a revision.
az containerapp secret	Commands to manage secrets.
az containerapp secret list	List the secrets of a container app.
az containerapp secret remove	Remove secrets from a container app.
az containerapp secret set	Create/update secrets.
az containerapp secret show	Show details of a secret.
az containerapp service	Commands to manage services available within the environment.

az containerapp service kafka	Commands to manage the kafka service for the Container Apps environment.
az containerapp service kafka create	Command to create the kafka service.
az containerapp service kafka delete	Command to delete the kafka service.
az containerapp service list	List all services within the environment.
az containerapp service postgres	Commands to manage the postgres service for the Container Apps environment.
az containerapp service postgres create	Command to create the postgres service.
az containerapp service postgres delete	Command to delete the postgres service.
az containerapp service redis	Commands to manage the redis service for the Container Apps environment.
az containerapp service redis create	Command to create the redis service.
az containerapp service redis delete	Command to delete the redis service.
az containerapp show	Show details of a container app.
az containerapp ssl	Upload certificate to a managed environment, add hostname to an app in that environment, and bind the certificate to the hostname.
az containerapp ssl upload	Upload certificate to a managed environment, add hostname to an app in that environment, and bind the certificate to the hostname.
az containerapp up	Create or update a container app as well as any associated resources (ACR, resource group, container apps environment, GitHub Actions, etc.).
az containerapp update	Update a container app. In multiple revisions mode, create a new revision based on the latest revision.

az containerapp browse

Open a containerapp in the browser, if possible.

Azure CLI

```
az containerapp browse [--ids]
                      [--name]
                      [--resource-group]
                      [--subscription]
```

Examples

open a containerapp in the browser

Azure CLI

```
az containerapp browse -n MyContainerapp -g MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--name -n

The name of the Containerapp. A name must consist of lower case alphanumeric characters or '-', start with a letter, end with an alphanumeric character, cannot have '--', and must be less than 32 characters.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az containerapp create

Create a container app.

Azure CLI

```
az containerapp create --name
                      --resource-group
                      [--args]
                      [--bind]
                      [--command]
                      [--container-name]
```

```
[--cpu]
[--dal]
[--dapr-app-id]
[--dapr-app-port]
[--dapr-app-protocol {grpc, http}]
[--dapr-http-max-request-size]
[--dapr-http-read-buffer-size]
[--dapr-log-level {debug, error, info, warn}]
[--enable-dapr {false, true}]
[--env-vars]
[--environment]
[--exposed-port]
[--image]
[--ingress {external, internal}]
[--max-replicas]
[--memory]
[--min-replicas]
[--no-wait]
[--registry-identity]
[--registry-password]
[--registry-server]
[--registry-username]
[--revision-suffix]
[--revisions-mode {multiple, single}]
[--scale-rule-auth]
[--scale-rule-http-concurrency]
[--scale-rule-metadata]
[--scale-rule-name]
[--scale-rule-type]
[--secret-volume-mount]
[--secrets]
[--system-assigned]
[--tags]
[--target-port]
[--transport {auto, http, http2, tcp}]
[--user-assigned]
[--workload-profile-name]
[--yaml]
```

Examples

Create a container app and retrieve its fully qualified domain name.

Azure CLI

```
az containerapp create -n MyContainerapp -g MyResourceGroup \
    --image myregistry.azurecr.io/my-app:v1.0 --environment
    MyContainerappEnv \
        --ingress external --target-port 80 \
        --registry-server myregistry.azurecr.io --registry-username myregistry \
        --registry-password $REGISTRY_PASSWORD \
        --query properties.configuration.ingress.fqdn
```

Create a container app with resource requirements and replica count limits.

Azure CLI

```
az containerapp create -n MyContainerapp -g MyResourceGroup \
--image nginx --environment MyContainerappEnv \
--cpu 0.5 --memory 1.0Gi \
--min-replicas 4 --max-replicas 8
```

Create a container app with secrets and environment variables.

Azure CLI

```
az containerapp create -n MyContainerapp -g MyResourceGroup \
--image my-app:v1.0 --environment MyContainerappEnv \
--secrets mysecret=secretvalue1 anothersecret="secret value 2" \
--env-vars GREETING="Hello, world" SECRETENV=secretref:anothersecret
```

Create a container app using a YAML configuration. Example YAML configuration -
<https://aka.ms/azure-container-apps-yaml>

Azure CLI

```
az containerapp create -n MyContainerapp -g MyResourceGroup \
--environment MyContainerappEnv \
--yaml "path/to/yaml/file.yml"
```

Create a container app with an http scale rule

Azure CLI

```
az containerapp create -n myapp -g mygroup --environment myenv --image nginx \
--scale-rule-name my-http-rule \
--scale-rule-http-concurrency 50
```

Create a container app with a custom scale rule

Azure CLI

```
az containerapp create -n MyContainerapp -g MyResourceGroup \
--image my-queue-processor --environment MyContainerappEnv \
--min-replicas 4 --max-replicas 8 \
--scale-rule-name queue-based-autoscaling \
--scale-rule-type azure-queue \
--scale-rule-metadata "accountName=mystorageaccountname" \
```

```
"cloud=AzurePublicCloud" \
"queueLength": "5" "queueName": "foo" \
--scale-rule-auth "connection=my-connection-string-secret-name"
```

Create a container app with secrets and mounts them in a volume.

Azure CLI

```
az containerapp create -n MyContainerapp -g MyResourceGroup \
--image my-app:v1.0 --environment MyContainerappEnv \
--secrets mysecret=secretvalue1 anothersecret="secret value 2" \
--secret-volume-mount "mnt/secrets"
```

Required Parameters

--name -n

The name of the Containerapp. A name must consist of lower case alphanumeric characters or '-', start with a letter, end with an alphanumeric character, cannot have '--', and must be less than 32 characters.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

Optional Parameters

--args

A list of container startup command argument(s). Space-separated values e.g. "-c" "mycommand". Empty string to clear existing values.

--bind

Space separated list of services(bindings) to be connected to this app. e.g.
SVC_NAME1[:BIND_NAME1] SVC_NAME2[:BIND_NAME2]...

--command

A list of supported commands on the container that will executed during startup.

Space-separated values e.g. "/bin/queue" "mycommand". Empty string to clear existing values.

--container-name

Name of the container.

--cpu

Required CPU in cores from 0.25 - 2.0, e.g. 0.5.

--dal --dapr-enable-api-logging

Enable API logging for the Dapr sidecar.

default value: False

--dapr-app-id

The Dapr application identifier.

--dapr-app-port

The port Dapr uses to talk to the application.

--dapr-app-protocol

The protocol Dapr uses to talk to the application.

accepted values: grpc, http

--dapr-http-max-request-size --dhmrs

Increase max size of request body http and grpc servers parameter in MB to handle uploading of big files.

--dapr-http-read-buffer-size --dhrbs

Dapr max size of http header read buffer in KB to handle when sending multi-KB headers..

--dapr-log-level

Set the log level for the Dapr sidecar.

accepted values: debug, error, info, warn

--enable-dapr

Boolean indicating if the Dapr side car is enabled.

accepted values: false, true

default value: False

--env-vars

A list of environment variable(s) for the container. Space-separated values in 'key=value' format. Empty string to clear existing values. Prefix value with 'secretref:' to reference a secret.

--environment

Name or resource ID of the container app's environment.

--exposed-port

Additional exposed port. Only supported by tcp transport protocol. Must be unique per environment if the app ingress is external.

--image -i

Container image, e.g. publisher/image-name:tag.

--ingress

The ingress type.

accepted values: external, internal

--max-replicas

The maximum number of replicas.

--memory

Required memory from 0.5 - 4.0 ending with "Gi", e.g. 1.0Gi.

--min-replicas

The minimum number of replicas.

--no-wait

Do not wait for the long-running operation to finish.

default value: False

--registry-identity

A Managed Identity to authenticate with the registry server instead of username/password. Use a resource ID or 'system' for user-defined and system-defined identities, respectively. The registry must be an ACR. If possible, an 'acrpull' role assignment will be created for the identity automatically.

--registry-password

The password to log in to container registry. If stored as a secret, value must start with 'secretref:' followed by the secret name.

--registry-server

The container registry server hostname, e.g. myregistry.azurecr.io.

--registry-username

The username to log in to container registry.

--revision-suffix

User friendly suffix that is appended to the revision name.

--revisions-mode

The active revisions mode for the container app.

accepted values: multiple, single

default value: single

--scale-rule-auth --sra

Scale rule auth parameters. Auth parameters must be in format " = = ...".

--scale-rule-http-concurrency --scale-rule-tcp-concurrency --srhc --srtc

The maximum number of concurrent requests before scale out. Only supported for http and tcp scale rules.

--scale-rule-metadata --srm

Scale rule metadata. Metadata must be in format " = = ...".

--scale-rule-name --srn

The name of the scale rule.

--scale-rule-type --srt

The type of the scale rule. Default: http. For more information please visit <https://learn.microsoft.com/azure/container-apps/scale-app#scale-triggers>.

--secret-volume-mount

Path to mount all secrets e.g. mnt/secrets.

--secrets -s

A list of secret(s) for the container app. Space-separated values in 'key=value' format.

--system-assigned

Boolean indicating whether to assign system-assigned identity.

default value: False

--tags

Space-separated tags: key[=value] [key[=value] ...]. Use "" to clear existing tags.

--target-port

The application port used for ingress traffic.

--transport

The transport protocol used for ingress traffic.

accepted values: auto, http, http2, tcp

default value: auto

--user-assigned

Space-separated user identities to be assigned.

--workload-profile-name -w

Name of the workload profile to run the app on.

--yaml

Path to a .yaml file with the configuration of a container app. All other parameters will be ignored. For an example, see <https://docs.microsoft.com/azure/container-apps/azure-resource-manager-api-spec#examples>.

▽ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az containerapp delete

Delete a container app.

Azure CLI

```
az containerapp delete [--ids]
                      [--name]
                      [--no-wait]
                      [--resource-group]
                      [--subscription]
                      [--yes]
```

Examples

Delete a container app.

Azure CLI

```
az containerapp delete -g MyResourceGroup -n MyContainerapp
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--name -n

The name of the Containerapp. A name must consist of lower case alphanumeric characters or '-', start with a letter, end with an alphanumeric character, cannot have '--', and must be less than 32 characters.

--no-wait

Do not wait for the long-running operation to finish.

default value: False

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--yes -y

Do not prompt for confirmation.

default value: False

▽ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az containerapp exec

Open an SSH-like interactive shell within a container app replica.

Azure CLI

```
az containerapp exec --name  
    --resource-group  
    [--command]  
    [--container]  
    [--replica]  
    [--revision]
```

Examples

exec into a container app

Azure CLI

```
az containerapp exec -n MyContainerapp -g MyResourceGroup
```

exec into a particular container app replica and revision

Azure CLI

```
az containerapp exec -n MyContainerapp -g MyResourceGroup --replica  
MyReplica --revision MyRevision
```

open a bash shell in a containerapp

Azure CLI

```
az containerapp exec -n MyContainerapp -g MyResourceGroup --command bash
```

Required Parameters

--name -n

The name of the Containerapp.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

Optional Parameters

--command

The startup command (bash, zsh, sh, etc.).

default value: sh

--container

The name of the container to ssh into.

--replica

The name of the replica to ssh into. List replicas with 'az containerapp replica list'. A replica may not exist if there is not traffic to your app.

--revision

The name of the container app revision to ssh into. Defaults to the latest revision.

Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use `--debug` for full debug logs.

az containerapp list

List container apps.

Azure CLI

```
az containerapp list [--environment]
                     [--resource-group]
```

Examples

List container apps in the current subscription.

Azure CLI

```
az containerapp list
```

List container apps by resource group.

Azure CLI

```
az containerapp list -g MyResourceGroup
```

Optional Parameters

--environment

Name or resource ID of the container app's environment.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

▽ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az containerapp show

Show details of a container app.

Azure CLI

```
az containerapp show [--ids]
                     [--name]
                     [--resource-group]
                     [--show-secrets]
                     [--subscription]
```

Examples

Show the details of a container app.

Azure CLI

```
az containerapp show -n MyContainerapp -g MyResourceGroup
```

Optional Parameters

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--name -n

The name of the Containerapp. A name must consist of lower case alphanumeric characters or '-', start with a letter, end with an alphanumeric character, cannot have '--', and must be less than 32 characters.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--show-secrets

Show Containerapp secrets.

default value: False

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az containerapp up

Create or update a container app as well as any associated resources (ACR, resource group, container apps environment, GitHub Actions, etc.).

Azure CLI

```
az containerapp up --name
    [--branch]
    [--browse]
    [--context-path]
    [--env-vars]
    [--environment]
    [--image]
    [--ingress {external, internal}]
    [--location]
    [--logs-workspace-id]
    [--logs-workspace-key]
    [--registry-password]
    [--registry-server]
    [--registry-username]
    [--repo]
    [--resource-group]
    [--service-principal-client-id]
    [--service-principal-client-secret]
    [--service-principal-tenant-id]
    [--source]
    [--target-port]
    [--token]
    [--workload-profile-name]
```

Examples

Create a container app from a dockerfile in a GitHub repo (setting up github actions)

Azure CLI

```
az containerapp up -n MyContainerapp --repo
https://github.com/myAccount/myRepo
```

Create a container app from a dockerfile in a local directory (or autogenerated a container if no dockerfile is found)

Azure CLI

```
az containerapp up -n MyContainerapp --source .
```

Create a container app from an image in a registry

Azure CLI

```
az containerapp up -n MyContainerapp --image  
myregistry.azurecr.io/myImage:myTag
```

Create a container app from an image in a registry with ingress enabled and a specified environment

Azure CLI

```
az containerapp up -n MyContainerapp --image  
myregistry.azurecr.io/myImage:myTag --ingress external --target-port 80 --  
environment MyEnv
```

Required Parameters

--name -n

The name of the Containerapp. A name must consist of lower case alphanumeric characters or '-', start with a letter, end with an alphanumeric character, cannot have '--', and must be less than 32 characters.

Optional Parameters

--branch -b

The branch of the Github repo. Assumed to be the Github repo's default branch if not specified.

--browse

Open the app in a web browser after creation and deployment, if possible.
default value: False

--context-path

Path in the repo from which to run the docker build. Defaults to "./". Dockerfile is assumed to be named "Dockerfile" and in this directory.

--env-vars

A list of environment variable(s) for the container. Space-separated values in 'key=value' format. Empty string to clear existing values. Prefix value with 'secretref:' to reference a secret.

--environment

Name or resource ID of the container app's environment.

--image -i

Container image, e.g. publisher/image-name:tag.

--ingress

The ingress type.

accepted values: external, internal

--location -l

Location. Values from: `az account list-locations`. You can configure the default location using `az configure --defaults location=<location>`.

--logs-workspace-id

Workspace ID of the Log Analytics workspace to send diagnostics logs to. You can use "az monitor log-analytics workspace create" to create one. Extra billing may apply.

--logs-workspace-key

Log Analytics workspace key to configure your Log Analytics workspace. You can use "az monitor log-analytics workspace get-shared-keys" to retrieve the key.

--registry-password

The password to log in to container registry. If stored as a secret, value must start with 'secretref:' followed by the secret name.

--registry-server

The container registry server hostname, e.g. myregistry.azurecr.io.

--registry-username

The username to log in to container registry.

--repo

Create an app via Github Actions. In the format: <https://github.com/> or /.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--service-principal-client-id --sp-cid

The service principal client ID. Used by Github Actions to authenticate with Azure.

--service-principal-client-secret --sp-sec

The service principal client secret. Used by Github Actions to authenticate with Azure.

--service-principal-tenant-id --sp-tid

The service principal tenant ID. Used by Github Actions to authenticate with Azure.

--source

Local directory path containing the application source and Dockerfile for building the container image. Preview: If no Dockerfile is present, a container image is generated using buildpacks. If Docker is not running or buildpacks cannot be used, Oryx will be used to generate the image. See the supported Oryx runtimes here: <https://github.com/microsoft/Oryx/blob/main/doc/supportedRuntimeVersions.md>.

--target-port

The application port used for ingress traffic.

--token

A Personal Access Token with write access to the specified repository. For more information: <https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line>. If not provided or not found in the cache (and using --repo), a browser page will be opened to authenticate with Github.

--workload-profile-name -w

The friendly name for the workload profile.

▽ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use --debug for full debug logs.

az containerapp update

Update a container app. In multiple revisions mode, create a new revision based on the latest revision.

Azure CLI

```
az containerapp update [--args]
                      [--bind]
                      [--command]
                      [--container-name]
                      [--cpu]
                      [--ids]
                      [--image]
                      [--max-replicas]
                      [--memory]
                      [--min-replicas]
                      [--name]
                      [--no-wait]
                      [--remove-all-env-vars]
                      [--remove-env-vars]
                      [--replace-env-vars]
                      [--resource-group]
                      [--revision-suffix]
                      [--scale-rule-auth]
                      [--scale-rule-http-concurrency]
                      [--scale-rule-metadata]
                      [--scale-rule-name]
                      [--scale-rule-type]
                      [--secret-volume-mount]
                      [--set-env-vars]
                      [--subscription]
                      [--tags]
                      [--unbind]
                      [--workload-profile-name]
                      [--yaml]
```

Examples

Update a container app's container image.

Azure CLI

```
az containerapp update -n MyContainerapp -g MyResourceGroup \
                      --image myregistry.azurecr.io/my-app:v2.0
```

Update a container app's resource requirements and scale limits.

Azure CLI

```
az containerapp update -n MyContainerapp -g MyResourceGroup \
                      --cpu 0.5 --memory 1.0Gi \
                      --min-replicas 4 --max-replicas 8
```

Update a container app with an http scale rule

Azure CLI

```
az containerapp update -n myapp -g mygroup \
--scale-rule-name my-http-rule \
--scale-rule-http-concurrency 50
```

Update a container app with a custom scale rule

Azure CLI

```
az containerapp update -n myapp -g mygroup \
--scale-rule-name my-custom-rule \
--scale-rule-type my-custom-type \
--scale-rule-metadata key=value key2=value2 \
--scale-rule-auth triggerparam=secretpref triggerparam=secretpref
```

Optional Parameters

--args

A list of container startup command argument(s). Space-separated values e.g. "-c" "mycommand". Empty string to clear existing values.

--bind

Space separated list of services(bindings) to be connected to this app. e.g.
SVC_NAME1[:BIND_NAME1] SVC_NAME2[:BIND_NAME2]...

--command

A list of supported commands on the container that will executed during startup.
Space-separated values e.g. "/bin/queue" "mycommand". Empty string to clear
existing values.

--container-name

Name of the container.

--cpu

Required CPU in cores from 0.25 - 2.0, e.g. 0.5.

--ids

One or more resource IDs (space-delimited). It should be a complete resource ID containing all information of 'Resource Id' arguments. You should provide either --ids or other 'Resource Id' arguments.

--image -i

Container image, e.g. publisher/image-name:tag.

--max-replicas

The maximum number of replicas.

--memory

Required memory from 0.5 - 4.0 ending with "Gi", e.g. 1.0Gi.

--min-replicas

The minimum number of replicas.

--name -n

The name of the Containerapp. A name must consist of lower case alphanumeric characters or '-', start with a letter, end with an alphanumeric character, cannot have '--', and must be less than 32 characters.

--no-wait

Do not wait for the long-running operation to finish.

default value: False

--remove-all-env-vars

Remove all environment variable(s) from container..

default value: False

--remove-env-vars

Remove environment variable(s) from container. Space-separated environment variable names.

--replace-env-vars

Replace environment variable(s) in container. Other existing environment variables are removed. Space-separated values in 'key=value' format. If stored as a secret, value must start with 'secretref:' followed by the secret name.

--resource-group -g

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

--revision-suffix

User friendly suffix that is appended to the revision name.

--scale-rule-auth --sra

Scale rule auth parameters. Auth parameters must be in format "`= = ...`".

--scale-rule-http-concurrency --scale-rule-tcp-concurrency --srhc --src

The maximum number of concurrent requests before scale out. Only supported for http and tcp scale rules.

--scale-rule-metadata --srm

Scale rule metadata. Metadata must be in format "`= = ...`".

--scale-rule-name --srn

The name of the scale rule.

--scale-rule-type --srt

The type of the scale rule. Default: http. For more information please visit <https://learn.microsoft.com/azure/container-apps/scale-app#scale-triggers>.

--secret-volume-mount

Path to mount all secrets e.g. mnt/secrets.

--set-env-vars

Add or update environment variable(s) in container. Existing environment variables are not modified. Space-separated values in 'key=value' format. If stored as a secret,

value must start with 'secretref:' followed by the secret name.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--tags

Space-separated tags: key[=value] [key[=value] ...]. Use "" to clear existing tags.

--unbind

Space separated list of services(bindings) to be removed from this app. e.g.
BIND_NAME1...

--workload-profile-name -w

The friendly name for the workload profile.

--yaml

Path to a .yaml file with the configuration of a container app. All other parameters will be ignored. For an example, see <https://docs.microsoft.com/azure/container-apps/azure-resource-manager-api-spec#examples>.

▽ Global Parameters

--debug

Increase logging verbosity to show all debug logs.

--help -h

Show this help message and exit.

--only-show-errors

Only show errors, suppressing warnings.

--output -o

Output format.

--query

JMESPath query string. See <http://jmespath.org/> for more information and examples.

--subscription

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

--verbose

Increase logging verbosity. Use `--debug` for full debug logs.

Az.App

Reference

Microsoft Azure PowerShell: App cmdlets

Container Apps

Disable-AzContainerAppRevision	Deactivates a revision for a Container App
Enable-AzContainerAppRevision	Activates a revision for a Container App
Get-AzContainerApp	Get the properties of a Container App.
Get-AzContainerAppAuthConfig	Get a AuthConfig of a Container App.
Get-AzContainerAppManagedEnv	Get the properties of a Managed Environment used to host container apps.
Get-AzContainerAppManagedEnvCert	Get the specified Certificate.
Get-AzContainerAppManagedEnvDapr	Get a dapr component.
Get-AzContainerAppManagedEnvDaprSecret	List secrets for a dapr component
Get-AzContainerAppManagedEnvStorage	Get storage for a managedEnvironment.
Get-AzContainerAppRevision	Get a revision of a Container App.
Get-AzContainerAppSecret	List secrets for a container app
New-AzContainerApp	Create or update a Container App.
New-AzContainerAppAuthConfig	Create or update the AuthConfig for a Container App.
New-AzContainerAppCustomDomainObject	Create an in-memory object for CustomDomain.
New-AzContainerAppDaprMetadataObject	Create an in-memory object for DaprMetadata.
New-AzContainerAppEnvironmentVarObject	Create an in-memory object for EnvironmentVar.
New-AzContainerAppIdentityProviderObject	Create an in-memory object for IdentityProviders.
New-AzContainerAppManagedEnv	Creates or updates a Managed Environment used to

	host container apps.
New-AzContainerAppManagedEnvCert	Create or Update a Certificate.
New-AzContainerAppManagedEnvDapr	Creates or updates a Dapr Component in a Managed Environment.
New-AzContainerAppManagedEnvStorage	Create or update storage for a managedEnvironment.
New-AzContainerAppProbeHeaderObject	Create an in-memory object for ContainerAppProbeHttpGetHttpHeadersItem.
New-AzContainerAppProbeObject	Create an in-memory object for ContainerAppProbe.
New-AzContainerAppRegistryCredentialObject	Create an in-memory object for RegistryCredentials.
New-AzContainerAppScaleRuleAuthObject	Create an in-memory object for ScaleRuleAuth.
New-AzContainerAppScaleRuleObject	Create an in-memory object for ScaleRule.
New-AzContainerAppSecretObject	Create an in-memory object for Secret.
New-AzContainerAppTemplateObject	Create an in-memory object for Container.
New-AzContainerAppTrafficWeightObject	Create an in-memory object for TrafficWeight.
New-AzContainerAppVolumeMountObject	Create an in-memory object for VolumeMount.
New-AzContainerAppVolumeObject	Create an in-memory object for Volume.
Remove-AzContainerApp	Delete a Container App.
Remove-AzContainerAppAuthConfig	Delete a Container App AuthConfig.
Remove-AzContainerAppManagedEnv	Delete a Managed Environment if it does not have any container apps.
Remove-AzContainerAppManagedEnvCert	Deletes the specified Certificate.
Remove-AzContainerAppManagedEnvDapr	Delete a Dapr Component from a Managed Environment.
Remove-AzContainerAppManagedEnvStorage	Delete storage for a managedEnvironment.
Restart-AzContainerAppRevision	Restarts a revision for a Container App

Update-AzContainerApp	Patches a Container App using JSON Merge Patch
Update-AzContainerAppManagedEnvCert	Patches a certificate. Currently only patching of tags is supported

Azure Policy built-in definitions for Azure Container Apps

Article • 06/21/2023

This page is an index of [Azure Policy](#) built-in policy definitions for Azure Container Apps. For additional Azure Policy built-ins for other services, see [Azure Policy built-in definitions](#).

The name of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Version** column to view the source on the [Azure Policy GitHub repo](#).

Policy definitions

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
Authentication should be enabled on Container Apps	Container Apps Authentication is a feature that can prevent anonymous HTTP requests from reaching the Container App, or authenticate those that have tokens before they reach the Container App	AuditIfNotExists, Disabled	1.0.1
Container App environments should use network injection	Container Apps environments should use virtual network injection to: 1.Isolate Container Apps from the public internet 2.Enable network integration with resources on-premises or in other Azure virtual networks 3.Achieve more granular control over network traffic flowing to and from the environment.	Audit, Disabled, Deny	1.0.2
Container App should configure with volume mount	Enforce the use of volume mounts for Container Apps to ensure availability of persistent storage capacity.	Audit, Deny, Disabled	1.0.1
Container Apps environment should disable public network access	Disable public network access to improve security by exposing the Container Apps environment through an internal load balancer. This removes the need for a public IP address and prevents internet access to all Container Apps within the environment.	Audit, Deny, Disabled	1.0.1

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
Container Apps should disable external network access ↗	Disable external network access to your Container Apps by enforcing internal-only ingress. This will ensure inbound communication for Container Apps is limited to callers within the Container Apps environment.	Audit, Deny, Disabled	1.0.1 ↗
Container Apps should only be accessible over HTTPS ↗	Use of HTTPS ensures server/service authentication and protects data in transit from network layer eavesdropping attacks. Disabling 'allowInsecure' will result in the automatic redirection of requests from HTTP to HTTPS connections for container apps.	Audit, Deny, Disabled	1.0.1 ↗
Managed Identity should be enabled for Container Apps ↗	Enforcing managed identity ensures Container Apps can securely authenticate to any resource that supports Azure AD authentication	Audit, Deny, Disabled	1.0.1 ↗

Next steps

- See the built-ins on the [Azure Policy GitHub repo](#) ↗ .
- Review the [Azure Policy definition structure](#).
- Review [Understanding policy effects](#).

Azure Container Apps

Article • 06/02/2022

Azure Container Apps allows you to run containerized applications without worrying about orchestration or infrastructure. For a more detailed overview, see the [Azure Container Apps product page](#).

REST Operation Groups

Operation Group	Description
Certificates	Provides operations for managing TLS certificates.
Container Apps	Provides operations for managing container apps.
Container Apps Auth Configs	Provides operations for managing authentication.
Container Apps Revision Replicas	Provides operations for managing replicas.
Container Apps Revisions	Provides operations for managing revisions.
Container Apps Source Controls	Provides operations for managing source control configurations.
Dapr Components	Provides operations for managing Dapr components.
Managed Environments	Provides operations for managing Container Apps environments.
Managed Environments Storages	Provides operations for managing storages.
Namespaces	Provides operations for checking resource name availability.
Operations	Lists all available REST APIs for Azure Container Apps.

Azure.ResourceManager.AppContainers Namespace

Reference

Classes

AppContainersExtensions	A class to add extension methods to Azure.ResourceManager.AppContainers.
ContainerAppAuthConfigCollection	A class representing a collection of ContainerAppAuthConfigResource and their operations. Each ContainerAppAuthConfigResource in the collection will belong to the same instance of ContainerAppResource . To get a ContainerAppAuthConfigCollection instance call the GetContainerAppAuthConfigs method from an instance of ContainerAppResource .
ContainerAppAuthConfigData	A class representing the ContainerAppAuthConfig data model. Configuration settings for the Azure ContainerApp Service Authentication / Authorization feature.
ContainerAppAuthConfigResource	A Class representing a ContainerAppAuthConfig along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppAuthConfigResource from an instance of ArmClient using the GetContainerAppAuthConfigResource method. Otherwise you can get one from its parent resource ContainerAppResource using the GetContainerAppAuthConfig method.
ContainerAppCertificateData	A class representing the ContainerAppCertificate data model. Certificate used for Custom Domain bindings of Container Apps in a Managed Environment
ContainerAppCollection	A class representing a collection of ContainerAppResource and their operations. Each ContainerAppResource in the collection will belong to the same instance of ResourceGroupResource . To get a ContainerAppCollection instance call the GetContainerApps method from an instance of ResourceGroupResource .
ContainerAppConnectedEnvironmentCertificateCollection	A class representing a collection of ContainerAppConnectedEnvironmentCertificateResource and their operations. Each ContainerAppConnectedEnvironmentCertificateResource in the collection will belong to the same instance of ContainerAppConnectedEnvironmentResource . To get a ContainerAppConnectedEnvironmentCertificateCollection instance call the GetContainerAppConnectedEnvironmentCertificates method from an instance of ContainerAppConnectedEnvironmentResource .

ContainerAppConnectedEnvironmentCertificateResource	A Class representing a ContainerAppConnectedEnvironmentCertificate along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppConnectedEnvironmentCertificateResource from an instance of ArmClient using the GetContainerAppConnectedEnvironmentCertificateResource method. Otherwise you can get one from its parent resource ContainerAppConnectedEnvironmentResource using the GetContainerAppConnectedEnvironmentCertificate method.
ContainerAppConnectedEnvironmentCollection	A class representing a collection of ContainerAppConnectedEnvironmentResource and their operations. Each ContainerAppConnectedEnvironmentResource in the collection will belong to the same instance of ResourceGroupResource . To get a ContainerAppConnectedEnvironmentCollection instance call the GetContainerAppConnectedEnvironments method from an instance of ResourceGroupResource .
ContainerAppConnectedEnvironmentDaprComponentCollection	A class representing a collection of ContainerAppConnectedEnvironmentDaprComponentResource and their operations. Each ContainerAppConnectedEnvironmentDaprComponentResource in the collection will belong to the same instance of ContainerAppConnectedEnvironmentResource . To get a ContainerAppConnectedEnvironmentDaprComponentCollection instance call the GetContainerAppConnectedEnvironmentDaprComponents method from an instance of ContainerAppConnectedEnvironmentResource .
ContainerAppConnectedEnvironmentDaprComponentResource	A Class representing a ContainerAppConnectedEnvironmentDaprComponent along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppConnectedEnvironmentDaprComponentResource from an instance of ArmClient using the GetContainerAppConnectedEnvironmentDaprComponentResource method. Otherwise you can get one from its parent resource ContainerAppConnectedEnvironmentResource using the GetContainerAppConnectedEnvironmentDaprComponent method.
ContainerAppConnectedEnvironmentData	A class representing the ContainerAppConnectedEnvironment data model. An environment for Kubernetes cluster specialized for web workloads by Azure App Service
ContainerAppConnectedEnvironmentResource	A Class representing a ContainerAppConnectedEnvironment along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppConnectedEnvironmentResource from an instance of ArmClient using the GetContainerAppConnectedEnvironmentResource method. Otherwise you can get one from its parent resource

	<p>ResourceGroupResource using the GetContainerAppConnectedEnvironment method.</p>
ContainerApp Connected EnvironmentStorage Collection	<p>A class representing a collection of ContainerAppConnectedEnvironmentStorageResource and their operations. Each ContainerAppConnectedEnvironmentStorageResource in the collection will belong to the same instance of ContainerAppConnectedEnvironmentResource. To get a ContainerAppConnectedEnvironmentStorageCollection instance call the GetContainerAppConnectedEnvironmentStorages method from an instance of ContainerAppConnectedEnvironmentResource.</p>
ContainerApp Connected EnvironmentStorage Data	<p>A class representing the ContainerAppConnectedEnvironmentStorage data model. Storage resource for connectedEnvironment.</p>
ContainerApp Connected EnvironmentStorage Resource	<p>A Class representing a ContainerAppConnectedEnvironmentStorage along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppConnectedEnvironmentStorageResource from an instance of ArmClient using the GetContainerAppConnectedEnvironmentStorageResource method. Otherwise you can get one from its parent resource ContainerAppConnectedEnvironmentResource using the GetContainerAppConnectedEnvironmentStorage method.</p>
ContainerAppDapr ComponentData	<p>A class representing the ContainerAppDaprComponent data model. Dapr Component.</p>
ContainerAppData	<p>A class representing the ContainerApp data model.</p>
ContainerApp DetectorCollection	<p>A class representing a collection of ContainerAppDetectorResource and their operations. Each ContainerAppDetectorResource in the collection will belong to the same instance of ContainerAppResource. To get a ContainerAppDetectorCollection instance call the GetContainerAppDetectors method from an instance of ContainerAppResource.</p>
ContainerApp DetectorProperty Resource	<p>A Class representing a ContainerAppDetectorProperty along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppDetectorPropertyResource from an instance of ArmClient using the GetContainerAppDetectorPropertyResource method. Otherwise you can get one from its parent resource ContainerAppResource using the GetContainerAppDetectorProperty method.</p>
ContainerApp DetectorProperty RevisionCollection	<p>A class representing a collection of ContainerAppDetectorPropertyRevisionResource and their operations. Each ContainerAppDetectorPropertyRevisionResource in the collection will belong to the same instance of ContainerAppResource. To get a</p>

	<p>ContainerAppDetectorPropertyRevisionCollection instance call the GetContainerAppDetectorPropertyRevisions method from an instance of ContainerAppResource.</p>
ContainerAppDetectorPropertyRevisionResource	<p>A Class representing a ContainerAppDetectorPropertyRevision along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppDetectorPropertyRevisionResource from an instance of ArmClient using the GetContainerAppDetectorPropertyRevisionResource method. Otherwise you can get one from its parent resource ContainerAppResource using the GetContainerAppDetectorPropertyRevision method.</p>
ContainerAppDetectorResource	<p>A Class representing a ContainerAppDetector along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppDetectorResource from an instance of ArmClient using the GetContainerAppDetectorResource method. Otherwise you can get one from its parent resource ContainerAppResource using the GetContainerAppDetector method.</p>
ContainerAppDiagnosticData	<p>A class representing the ContainerAppDiagnostic data model. Diagnostics data for a resource.</p>
ContainerAppManagedEnvironmentCertificateCollection	<p>A class representing a collection of ContainerAppManagedEnvironmentCertificateResource and their operations. Each ContainerAppManagedEnvironmentCertificateResource in the collection will belong to the same instance of ContainerAppManagedEnvironmentResource. To get a ContainerAppManagedEnvironmentCertificateCollection instance call the GetContainerAppManagedEnvironmentCertificates method from an instance of ContainerAppManagedEnvironmentResource.</p>
ContainerAppManagedEnvironmentCertificateResource	<p>A Class representing a ContainerAppManagedEnvironmentCertificate along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppManagedEnvironmentCertificateResource from an instance of ArmClient using the GetContainerAppManagedEnvironmentCertificateResource method. Otherwise you can get one from its parent resource ContainerAppManagedEnvironmentResource using the GetContainerAppManagedEnvironmentCertificate method.</p>
ContainerAppManagedEnvironmentCollection	<p>A class representing a collection of ContainerAppManagedEnvironmentResource and their operations. Each ContainerAppManagedEnvironmentResource in the collection will belong to the same instance of ResourceGroupResource. To get a ContainerAppManagedEnvironmentCollection instance call the GetContainerAppManagedEnvironments method from an instance of ResourceGroupResource.</p>
ContainerApp	<p>A class representing a collection of</p>

Managed EnvironmentDapr Component Collection	<p>ContainerAppManagedEnvironmentDaprComponentResource and their operations. Each ContainerAppManagedEnvironmentDaprComponentResource in the collection will belong to the same instance of ContainerAppManagedEnvironmentResource. To get a ContainerAppManagedEnvironmentDaprComponentCollection instance call the <code>GetContainerAppManagedEnvironmentDaprComponents</code> method from an instance of ContainerAppManagedEnvironmentResource.</p>
ContainerApp Managed EnvironmentDapr Component Resource	<p>A Class representing a ContainerAppManagedEnvironmentDaprComponent along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppManagedEnvironmentDaprComponentResource from an instance of ArmClient using the <code>GetContainerAppManagedEnvironmentDaprComponentResource</code> method. Otherwise you can get one from its parent resource ContainerAppManagedEnvironmentResource using the <code>GetContainerAppManagedEnvironmentDaprComponent</code> method.</p>
ContainerApp Managed Environment Data	<p>A class representing the ContainerAppManagedEnvironmentData data model.</p>
ContainerApp Managed Environment Detector Collection	<p>A class representing a collection of ContainerAppManagedEnvironmentDetectorResource and their operations. Each ContainerAppManagedEnvironmentDetectorResource in the collection will belong to the same instance of ContainerAppManagedEnvironmentResource. To get a ContainerAppManagedEnvironmentDetectorCollection instance call the <code>GetContainerAppManagedEnvironmentDetectors</code> method from an instance of ContainerAppManagedEnvironmentResource.</p>
ContainerApp Managed Environment Detector Resource	<p>A Class representing a ContainerAppManagedEnvironmentDetector along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppManagedEnvironmentDetectorResource from an instance of ArmClient using the <code>GetContainerAppManagedEnvironmentDetectorResource</code> method. Otherwise you can get one from its parent resource ContainerAppManagedEnvironmentResource using the <code>GetContainerAppManagedEnvironmentDetector</code> method.</p>
ContainerApp Managed Environment Detector Resource Property Resource	<p>A Class representing a ContainerAppManagedEnvironmentDetectorResourceProperty along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppManagedEnvironmentDetectorResourcePropertyResource from an instance of ArmClient using the <code>GetContainerAppManagedEnvironmentDetectorResourcePropertyResource</code></p>

	<p>method. Otherwise you can get one from its parent resource ContainerAppManagedEnvironmentResource using the <code>GetContainerAppManagedEnvironmentDetectorResourceProperty</code> method.</p>
ContainerAppManagedEnvironmentResource	<p>A Class representing a ContainerAppManagedEnvironment along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppManagedEnvironmentResource from an instance of ArmClient using the <code>GetContainerAppManagedEnvironmentResource</code> method. Otherwise you can get one from its parent resource ResourceGroupResource using the <code>GetContainerAppManagedEnvironment</code> method.</p>
ContainerAppManagedEnvironmentStorageCollection	<p>A class representing a collection of ContainerAppManagedEnvironmentStorageResource and their operations. Each ContainerAppManagedEnvironmentStorageResource in the collection will belong to the same instance of ContainerAppManagedEnvironmentResource. To get a ContainerAppManagedEnvironmentStorageCollection instance call the <code>GetContainerAppManagedEnvironmentStorages</code> method from an instance of ContainerAppManagedEnvironmentResource.</p>
ContainerAppManagedEnvironmentStorageData	<p>A class representing the ContainerAppManagedEnvironmentStorage data model. Storage resource for managedEnvironment.</p>
ContainerAppManagedEnvironmentStorageResource	<p>A Class representing a ContainerAppManagedEnvironmentStorage along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppManagedEnvironmentStorageResource from an instance of ArmClient using the <code>GetContainerAppManagedEnvironmentStorageResource</code> method. Otherwise you can get one from its parent resource ContainerAppManagedEnvironmentResource using the <code>GetContainerAppManagedEnvironmentStorage</code> method.</p>
ContainerAppReplicaCollection	<p>A class representing a collection of ContainerAppReplicaResource and their operations. Each ContainerAppReplicaResource in the collection will belong to the same instance of ContainerAppRevisionResource. To get a ContainerAppReplicaCollection instance call the <code>GetContainerAppReplicas</code> method from an instance of ContainerAppRevisionResource.</p>
ContainerAppReplicaData	<p>A class representing the ContainerAppReplica data model. Container App Revision Replica.</p>
ContainerAppReplicaResource	<p>A Class representing a ContainerAppReplica along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppReplicaResource from an instance of ArmClient using the <code>GetContainerAppReplicaResource</code> method. Otherwise</p>

you can get one from its parent resource [ContainerAppRevisionResource](#) using the `GetContainerAppReplica` method.

ContainerApp Resource	A Class representing a ContainerApp along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppResource from an instance of ArmClient using the <code>GetContainerAppResource</code> method. Otherwise you can get one from its parent resource ResourceGroupResource using the <code>GetContainerApp</code> method.
ContainerApp RevisionCollection	A class representing a collection of ContainerAppRevisionResource and their operations. Each ContainerAppRevisionResource in the collection will belong to the same instance of ContainerAppResource . To get a ContainerAppRevisionCollection instance call the <code>GetContainerAppRevisions</code> method from an instance of ContainerAppResource .
ContainerApp RevisionData	A class representing the ContainerAppRevision data model. Container App Revision.
ContainerApp RevisionResource	A Class representing a ContainerAppRevision along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppRevisionResource from an instance of ArmClient using the <code>GetContainerAppRevisionResource</code> method. Otherwise you can get one from its parent resource ContainerAppResource using the <code>GetContainerAppRevision</code> method.
ContainerApp SourceControl Collection	A class representing a collection of ContainerAppSourceControlResource and their operations. Each ContainerAppSourceControlResource in the collection will belong to the same instance of ContainerAppResource . To get a ContainerAppSourceControlCollection instance call the <code>GetContainerAppSourceControls</code> method from an instance of ContainerAppResource .
ContainerApp SourceControlData	A class representing the ContainerAppSourceControl data model. Container App SourceControl.
ContainerApp SourceControl Resource	A Class representing a ContainerAppSourceControl along with the instance operations that can be performed on it. If you have a ResourceIdentifier you can construct a ContainerAppSourceControlResource from an instance of ArmClient using the <code>GetContainerAppSourceControlResource</code> method. Otherwise you can get one from its parent resource ContainerAppResource using the <code>GetContainerAppSourceControl</code> method.

Azure Resource Manager ContainerAppsApi client library for Java - version 1.0.0-beta.5

Article • 05/16/2023

Azure Resource Manager ContainerAppsApi client library for Java.

This package contains Microsoft Azure SDK for ContainerAppsApi Management SDK. Package tag package-preview-2022-11. For documentation on how to use this package, please see [Azure Management Libraries for Java](#).

We'd love to hear your feedback

We're always working on improving our products and the way we communicate with our users. So we'd love to learn what's working and how we can do better.

If you haven't already, please take a few minutes to [complete this short survey](#) we have put together.

Thank you in advance for your collaboration. We really appreciate your time!

Documentation

Various documentation is available to help you get started

- [API reference documentation](#)

Getting started

Prerequisites

- [Java Development Kit \(JDK\)](#) with version 8 or above
- [Azure Subscription](#)

Adding the package to your product

XML

```
<dependency>
    <groupId>com.azure.resourcemanager</groupId>
    <artifactId>azure-resourcemanager-appcontainers</artifactId>
    <version>1.0.0-beta.5</version>
</dependency>
```

Include the recommended packages

Azure Management Libraries require a `TokenCredential` implementation for authentication and an `HttpClient` implementation for HTTP client.

[Azure Identity](#) and [Azure Core Netty HTTP](#) packages provide the default implementation.

Authentication

By default, Azure Active Directory token authentication depends on correct configuration of the following environment variables.

- `AZURE_CLIENT_ID` for Azure client ID.
- `AZURE_TENANT_ID` for Azure tenant ID.
- `AZURE_CLIENT_SECRET` or `AZURE_CLIENT_CERTIFICATE_PATH` for client secret or client certificate.

In addition, Azure subscription ID can be configured via `AZURE_SUBSCRIPTION_ID` environment variable.

With above configuration, `azure` client can be authenticated using the following code:

Java

```
AzureProfile profile = new AzureProfile(AzureEnvironment.AZURE);
TokenCredential credential = new DefaultAzureCredentialBuilder()
    .authorityHost(profile.getEnvironment().getActiveDirectoryEndpoint())
    .build();
ContainerAppsApiManager manager = ContainerAppsApiManager
    .authenticate(credential, profile);
```

The sample code assumes global Azure. Please change `AzureEnvironment.AZURE` variable if otherwise.

See [Authentication](#) for more options.

Key concepts

See [API design](#) for general introduction on design and key concepts on Azure Management Libraries.

Examples

[Code snippets and samples](#)

Troubleshooting

Next steps

Contributing

For details on contributing to this repository, see the [contributing guide](#).

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit <https://cla.microsoft.com>.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., label, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repositories using our CLA.

This project has adopted the [Microsoft Open Source Code of Conduct](#). For more information see the [Code of Conduct FAQ](#) or contact opencode@microsoft.com with any additional questions or comments.

Azure ContainerApps API client library for JavaScript - version 1.1.2

Article • 10/11/2022

This package contains an isomorphic SDK (runs both in Node.js and in browsers) for Azure ContainerApps API client.

[Source code ↗](#) | [Package \(NPM\) ↗](#) | [API reference documentation](#) | [Samples ↗](#)

Getting started

Currently supported environments

- [LTS versions of Node.js ↗](#)
- Latest versions of Safari, Chrome, Edge and Firefox.

See our [support policy ↗](#) for more details.

Prerequisites

- An [Azure subscription ↗](#).

Install the `@azure/arm-appcontainers` package

Install the Azure ContainerApps API client library for JavaScript with `npm`:

Bash

```
npm install @azure/arm-appcontainers
```

Create and authenticate a `ContainerAppsAPIClient`

To create a client object to access the Azure ContainerApps API API, you will need the `endpoint` of your Azure ContainerApps API resource and a `credential`. The Azure ContainerApps API client can use Azure Active Directory credentials to authenticate. You can find the endpoint for your Azure ContainerApps API resource in the [Azure Portal ↗](#).

You can authenticate with Azure Active Directory using a credential from the [@azure/identity ↗](#) library or [an existing AAD Token ↗](#).

To use the [DefaultAzureCredential](#) provider shown below, or other credential providers provided with the Azure SDK, please install the `@azure/identity` package:

Bash

```
npm install @azure/identity
```

You will also need to **register a new AAD application and grant access to Azure ContainerApps API** by assigning the suitable role to your service principal (note: roles such as "Owner" will not grant the necessary permissions). Set the values of the client ID, tenant ID, and client secret of the AAD application as environment variables:

`AZURE_CLIENT_ID`, `AZURE_TENANT_ID`, `AZURE_CLIENT_SECRET`.

For more information about how to create an Azure AD Application check out [this guide](#).

JavaScript

```
const { ContainerAppsAPIClient } = require("@azure/arm-appcontainers");
const { DefaultAzureCredential } = require("@azure/identity");
// For client-side applications running in the browser, use
InteractiveBrowserCredential instead of DefaultAzureCredential. See
https://aka.ms/azsdk/js/identity/examples for more details.

const subscriptionId = "00000000-0000-0000-0000-000000000000";
const client = new ContainerAppsAPIClient(new DefaultAzureCredential(),
subscriptionId);

// For client-side applications running in the browser, use this code
instead:
// const credential = new InteractiveBrowserCredential({
//   tenantId: "<YOUR_TENANT_ID>",
//   clientId: "<YOUR_CLIENT_ID>"
// });
// const client = new ContainerAppsAPIClient(credential, subscriptionId);
```

JavaScript Bundle

To use this client library in the browser, first you need to use a bundler. For details on how to do this, please refer to our [bundling documentation](#).

Key concepts

ContainerAppsAPIClient

`ContainerAppsAPIClient` is the primary interface for developers using the Azure ContainerApps API client library. Explore the methods on this client object to understand the different features of the Azure ContainerApps API service that you can access.

Troubleshooting

Logging

Enabling logging may help uncover useful information about failures. In order to see a log of HTTP requests and responses, set the `AZURE_LOG_LEVEL` environment variable to `info`. Alternatively, logging can be enabled at runtime by calling `setLogLevel` in the `@azure/logger`:

JavaScript

```
const { setLogLevel } = require("@azure/logger");
setLogLevel("info");
```

For more detailed instructions on how to enable logs, you can look at the [@azure/logger package docs](#).

Next steps

Please take a look at the [samples](#) directory for detailed examples on how to use this library.

Contributing

If you'd like to contribute to this library, please read the [contributing guide](#) to learn more about how to build and test the code.

Related projects

- [Microsoft Azure SDK for JavaScript](#)

appcontainers Package

Reference

Packages

[aio](#)

[models](#)

[operations](#)

Classes

[ContainerAppsAPIClient](#)

ContainerAppsAPIClient.

Azure Container Apps samples

Article • 07/28/2022

Refer to the following samples to learn how to use Azure Container Apps in different contexts and paired with different technologies.

Name	Description
A/B Testing your ASP.NET Core apps using Azure Container Apps ↗	Shows how to use Azure App Configuration, ASP.NET Core Feature Flags, and Azure Container Apps revisions together to gradually release features or perform A/B tests.
gRPC with ASP.NET Core on Azure Container Apps ↗	This repository contains a simple scenario built to demonstrate how ASP.NET Core 6.0 can be used to build a cloud-native application hosted in Azure Container Apps that uses gRPC request/response transmission from Worker microservices. The gRPC service simultaneously streams sensor data to a Blazor server frontend, so you can watch the data be charted in real-time.
Deploy an Orleans Cluster to Container Apps ↗	An end-to-end sample and tutorial for getting a Microsoft Orleans cluster running on Azure Container Apps. Worker microservices rapidly transmit data to a back-end Orleans cluster for monitoring and storage, emulating thousands of physical devices in the field.
Deploy a shopping cart Orleans app to Container Apps ↗	An end-to-end example shopping cart app built in ASP.NET Core Blazor Server with Orleans deployed to Azure Container Apps.
ASP.NET Core front-end with two back-end APIs on Azure Container Apps ↗	This sample demonstrates ASP.NET Core 6.0 can be used to build a cloud-native application hosted in Azure Container Apps.
ASP.NET Core front-end with two back-end APIs on Azure Container Apps (with Dapr) ↗	Demonstrates how ASP.NET Core 6.0 is used to build a cloud-native application hosted in Azure Container Apps using Dapr.

Billing in Azure Container Apps

Article • 06/01/2023

Billing in Azure Container apps is based on your [plan type](#).

Plan type	Description
Consumption	Serverless environment where you're only billed for the resources your apps use when they're running.
Consumption + Dedicated workload profiles plan structure	A fully managed environment that supports both Consumption-based apps and Dedicated workload profiles that offer customized compute options for your apps. You're billed for each node in each workload profile .

Charges apply to resources allocated to each running replica. |

Consumption plan

Azure Container Apps consumption plan billing consists of two types of charges:

- **Resource consumption:** The amount of resources allocated to your container app on a per-second basis, billed in vCPU-seconds and GiB-seconds.
- **HTTP requests:** The number of HTTP requests your container app receives.

The following resources are free during each calendar month, per subscription:

- The first 180,000 vCPU-seconds
- The first 360,000 GiB-seconds
- The first 2 million HTTP requests

This article describes how to calculate the cost of running your container app. For pricing details in your account's currency, see [Azure Container Apps Pricing ↗](#).

Note

If you use Container Apps with [your own virtual network](#) or your apps utilize other Azure resources, additional charges may apply.

Resource consumption charges

Azure Container Apps runs replicas of your application based on the [scaling rules and replica count limits](#) you configure for each revision. You're charged for the amount of resources allocated to each replica while it's running.

There are 2 meters for resource consumption:

- **vCPU-seconds:** The number of vCPU cores allocated to your container app on a per-second basis.
- **GiB-seconds:** The amount of memory allocated to your container app on a per-second basis.

The first 180,000 vCPU-seconds and 360,000 GiB-seconds in each subscription per calendar month are free.

The rate you pay for resource consumption depends on the state of your container app's revisions and replicas. By default, replicas are charged at an *active* rate. However, in certain conditions, a replica can enter an *idle* state. While in an *idle* state, resources are billed at a reduced rate.

No replicas are running

When a revision is scaled to zero replicas, no resource consumption charges are incurred.

Minimum number of replicas are running

Idle usage charges may apply when a revision is running under a specific set of circumstances. To be eligible for idle charges, a revision must be:

- Configured with a [minimum replica count](#) greater than zero
- Scaled to the minimum replica count

Usage charges are calculated individually for each replica. A replica is considered idle when *all* of the following conditions are true:

- The replica is running in a revision that is currently eligible for idle charges.
- All of the containers in the replica have started and are running.
- The replica isn't processing any HTTP requests.
- The replica is using less than 0.01 vCPU cores.
- The replica is receiving less than 1,000 bytes per second of network traffic.

When a replica is idle, resource consumption charges are calculated at the reduced idle rates. When a replica isn't idle, the active rates apply.

More than the minimum number of replicas are running

When a revision is scaled above the [minimum replica count](#), all of its running replicas are charged for resource consumption at the active rate.

Request charges

In addition to resource consumption, Azure Container Apps also charges based on the number of HTTP requests received by your container app. Only requests that come from outside a Container Apps environment are billable.

- The first 2 million requests in each subscription per calendar month are free.
- [Health probe](#) requests are not billable.

Consumption + Dedicated workload profiles plan structure (preview)

Azure Container Apps Consumption + Dedicated plan structure consists of two plans within a single environment, each with their own billing model.

The billing for apps running in the Consumption plan within the Consumption + Dedicated plan structure is the same as the Consumption plan.

The billing for apps running in the Dedicated plan within the Consumption + Dedicated plan structure is as follows:

- **Dedicated workload profiles:** You're billed on a per-second basis for vCPU-seconds and GiB-seconds resources in all the workload profile instances in use. As profiles scale out, extra costs apply for the extra instances; as profiles scale in, billing is reduced.
- **Dedicated plan management:** You're billed a fixed cost for the Dedicated management plan when using Dedicated workload profiles. This cost is the same regardless of how many Dedicated workload profiles in use.

For instance, you are not billed any charges for Dedicated unless you use a Dedicated workload profile in your environment.

General terms

For pricing details in your account's currency, see [Azure Container Apps Pricing](#).

For best results, maximize the use of your allocated resources by calculating the needs of your container apps. Often you can run multiple apps on a single instance of a workload profile.

Quotas for Azure Container Apps

Article • 05/15/2023

The following quotas are on a per subscription basis for Azure Container Apps.

To request an increase in quota amounts for your container app, learn [how to request a limit increase](#) and [submit a support ticket](#).

The *Is Configurable* column in the following tables denotes a feature maximum may be increased through a [support request](#). For more information, see [how to request a limit increase](#).

Feature	Scope	Default	Is Configurable	Remarks
Environments	Region	Up to 15	Yes	Limit up to 15 environments per subscription, per region. For example, if you deploy to three regions you can get up to 45 environments for a single subscription.
Container Apps	Environment	Unlimited	n/a	
Revisions	Container app	100	No	
Replicas	Revision	300	Yes	

Consumption plan

Feature	Scope	Default	Is Configurable	Remarks
Cores	Replica	2	No	Maximum number of cores available to a revision replica.
Cores	Environment	100	Yes	Maximum number of cores an environment can accommodate. Calculated by the sum of cores requested by each active replica of all revisions in an environment.

Consumption + Dedicated plan structure

Consumption workload profile

Feature	Scope	Default	Is Configurable	Remarks
Cores	Replica	4	No	Maximum number of cores available to a revision replica.
Cores	Environment	100	Yes	Maximum number of cores the Consumption workload profile in a Consumption + Dedicated plan structure environment can accommodate. Calculated by the sum of cores requested by each active replica of all revisions in an environment.

Dedicated workload profiles

Feature	Scope	Default	Is Configurable	Remarks
Cores	Replica	Up to maximum cores a workload profile supports	No	Maximum number of cores available to a revision replica.
Cores	Environment	100	Yes	Maximum number of cores all Dedicated workload profiles in a Consumption + Dedicated plan structure environment can accommodate. Calculated by the sum of cores available in each node of all workload profile in a Consumption + Dedicated plan structure environment.

For more information regarding quotas, see the [Quotas roadmap](#) in the Azure Container Apps GitHub repository.

Note

[Free trial](#) and [Azure for Students](#) subscriptions are limited to one environment per subscription globally and ten (10) cores per environment.

Considerations

- If an environment runs out of allowed cores:
 - Provisioning times out with a failure
 - The app may be restricted from scaling out
- If you encounter unexpected capacity limits, open a support ticket

Azure Container Apps frequently asked questions

FAQ

This article lists commonly asked questions about Azure Container Apps together with related answers.

APIs

Does Azure Container Apps provide direct access to the underlying Kubernetes API?

No, there is no access to the Kubernetes API.

Can I import my Azure Container Apps API from the context of API Management?

Yes.

Billing

How is Azure Container Apps billed?

Refer to the [billing](#) page for details.

Configuration

Can I set up GitHub Actions to automatically build and deploy my code to Azure Container Apps?

Yes. Using Azure CLI, run `az containerapp github-action -h` to see the options. Using Azure portal, navigate to the "Continuous deployment" blade under your container app.

Data management

Where does Azure Container Apps store customer data?

Azure Container Apps does not move or store customer data out of the region it's deployed in.

Quotas

How can I request a quota increase?

To request quota increase, [submit a support ticket](#) with the following fields:

Property	Value
Issue Type	Technical
Subscription	Your subscription
Service	My Services
Service Type	Container Apps
Resource	maxoutquotas
Summary	An explanation for your request, including: <ul style="list-style-type: none">• What quotas you want increased• How much of an increase you would like• Business reasons for the increase
Problem Type	Configuration and Management
Problem Subtype	Scaling

Please keep in mind the following when it comes to quota increase requests:

- **Scaling apps vs environments:** There are a number of different quotas available to increase. Use these descriptions to help identify your needs:
 - **Increase apps and cores per environment:** Allows you to run more apps within an environment and/or more intensive apps. Recommended if your workloads can deploy within the same network and security boundaries.
 - **Increasing environments:** Recommended if your workloads need network or security boundaries. Note: Additional business context may be required if your request involves increasing environment-level quotas.
- **Regions:** Approvals for increase requests vary based on compute capacity available in Azure regions.
- **Specific compute requirements:** The platform supports 4GB per container app. Memory limits overrides are evaluated on a per-case basis.
- **Business reasoning for scaling:** You may be eligible for a quota increase request if the platform limits are blocking your workload demands. Scale limits overrides are evaluated on a per-case basis.

Dapr

What Dapr features and APIs are available in Azure Container Apps?

Each Dapr capability undergoes thorough evaluation to ensure it positively impacts customers running microservices in the Azure Container Apps environment, while providing the best possible experience.

Are alpha Dapr APIs and components supported or available in Azure Container Apps?

Azure Container Apps offers developers the flexibility to experiment with the latest Dapr alpha APIs and features on a self-service, opt-in basis. Although the availability of these alpha APIs and components is not guaranteed, you can stay ahead of the curve and explore cutting-edge technologies as they become available. While these alpha APIs and components are provided "as is" and "as available," their continuous evolution towards stable status ensures that developers can always be at the forefront of innovation.

What is the Dapr version release cadence in Azure Container Apps?

Dapr's typical release timeline is up to six weeks after [the Dapr OSS release](#). The latest Dapr version is made available in Azure Container Apps after rigorous testing. Rolling out to all regions can take roughly up to two weeks.

How can I request a Dapr feature enhancement for Azure Container Apps?

You can submit a feature request via the [Azure Container Apps GitHub repository](#). Make sure to include "Dapr" in the feature request title.