

Introduction to Azure for developers

Learn how you can build applications using Azure.

Overview



[Azure for developers](#)

[Key Azure services for developers](#)

[Host applications on Azure](#)

[Connect apps to Azure services](#)

[Create Azure resources](#)

[Key concepts for building Azure apps](#)

[Azure billing](#)

Key developer scenarios



[Develop apps that use Azure AI services](#)

[Passwordless connections for Azure Services](#)

Developer centers



[.NET on Azure](#)

[Java on Azure](#)

[JavaScript on Azure](#)

[Python on Azure](#)

[Go on Azure](#)

Azure for developers overview

Article • 10/18/2022

Azure is a cloud platform designed to simplify the process of building modern applications. Whether you choose to host your applications entirely in Azure or extend your on-premises applications with Azure services, Azure helps you create applications that are scalable, reliable, and maintainable.

Azure supports the most popular programming languages in use today, including Python, JavaScript, Java, .NET and Go. With a comprehensive SDK library and extensive support in tools you already use like VS Code, Visual Studio, IntelliJ, and Eclipse, Azure is designed to take advantage of skills you already have and make you productive right away.

Application development scenarios on Azure

You can incorporate Azure into your application in different ways depending on your needs.

<https://www.microsoft.com/en-us/videoplayer/embed/RE50LmJ?postJs||Msg=true>

- **Application hosting on Azure** - Azure can host your entire application stack from web applications and APIs to databases to storage services. Azure supports a variety of hosting models from fully managed services to containers to virtual machines. When using fully managed Azure services, your applications can take advantage of the scalability, high-availability, and security built in to Azure.
- **Consuming cloud services from existing on-premises applications** - Existing on-premises apps can incorporate Azure services to extend their capabilities. For example, an application could use Azure Blob Storage to store files in the cloud, Azure Key Vault to securely store application secrets, or [Azure Cognitive Search](#) to add full-text search capability. These services are fully managed by Azure and can be easily added to your existing apps without changing your current application architecture or deployment model.
- **Container based architectures** - Azure provides a variety of container based services to support your app modernization journey. Whether you need a private registry for your container images, are containerizing an existing app for ease of deployment, deploying microservices based applications, or managing containers at scale, Azure has solutions that support your needs.

- **Modern serverless architectures** - Azure Functions simplify building solutions to handle event-driven workflows, whether responding to HTTP requests, handling file uploads in Blob storage, or processing events in a queue. You write only the code necessary to handle your event without worrying about servers or framework code. Further, you can take advantage of over 250 connectors to other Azure and third-party services to tackle your toughest integration problems.

Key Azure services for developers

Article • 11/22/2023

This article introduces some of the key Azure services that are used most frequently as a developer. For a comprehensive list of all Azure services, see the [Azure documentation hub page](#).

App hosting and compute

Service	Description
 Azure App Service	Host .NET, Java, Node.js, and Python web applications and APIs in a fully managed Azure service. You only need to deploy your code to Azure. Azure takes care of all the infrastructure management like high availability, load balancing, and autoscaling.
 Azure Static Web Apps	Host static web apps built using frameworks like Gatsby, Hugo, or VuePress, or modern web apps built using Angular, React, Svelte, or Vue. Static web apps automatically build and deploy based off of code changes and feature API integration with Azure Functions.
 Azure Functions	A serverless compute platform for creating small, discrete segments of code that can be triggered from a variety of different events. Common applications include building serverless APIs or orchestrating event-drive architectures.
 Azure Container Instances	Run Docker containers on-demand in a managed, serverless Azure environment. Azure Container Instances is a solution for any scenario that can operate in isolated containers, without orchestration.
 Azure Kubernetes Services	Quickly deploy a production ready Kubernetes cluster to the cloud and offload the operational overhead to Azure. Azure handles critical tasks, like health monitoring and maintenance. You only need to manage and maintain the agent nodes.
 Azure Spring Apps	Host Spring Boot microservice applications in Azure, no code changes required. Azure Spring Apps provides monitoring, configuration management, service discovery, CI/CD integration and more.
 Azure Virtual Machines	Host your app using virtual machines in Azure when you need more control over your computing environment. Azure VMs offer a flexible, scalable computing environment for both Linux and Windows virtual machines.

Azure AI services

Azure AI services help you create intelligent applications with pre-built and customizable APIs and models. Example applications include natural language processing for conversations, search, monitoring, translation, speech, vision, and decision-making.

Service	Description
 Azure OpenAI	Use powerful language models including the GPT-3, Codex and Embeddings model series for content generation, summarization, semantic search, and natural language to code translation.
 Azure AI Speech	Transcribe audible speech into readable, searchable text or convert text to lifelike speech for more natural interfaces.
 Azure AI Language	Use natural language processing (NLP) to identify key phrases and conduct sentiment analysis from text.
 Azure AI Translator	Translate more than 100 languages and dialects.
 Azure AI Vision	Analyze content in images and video.
 Azure AI Search	Information retrieval at scale for traditional and conversational search applications, with security and options for AI enrichment and vectorization.
 Azure AI Document Intelligence	Document extraction service that understands your forms allowing you to quickly extract text and structure from documents.

Data

Service	Description
 Azure SQL	A family of SQL Server database engine products in the cloud.
 Azure SQL Database	A fully managed, cloud-based version of SQL Server.
 Azure Cosmos DB	A fully managed, cloud-based NoSQL database. Azure Cosmos DB features multiple APIs, including APIs compatible MongoDB , Cassandra and Gremlin .
 Azure Database for PostgreSQL	A fully managed, cloud-based PostgreSQL database service based on PostgreSQL Community Edition.

Service	Description
	Azure Database for MySQL A fully managed, cloud-based MySQL database service based in the MySQL Community Edition.
	Azure Database for MariaDB A fully managed, cloud-based MariaDB database service based on the MariaDB community edition.
	Azure Cache for Redis A secure data cache and messaging broker that provides high throughput and low-latency access to data for applications.

Storage

[Azure Storage](#) products offer secure and scalable cloud and hybrid data storage services. Offerings include services for hybrid storage solutions, and services to transfer, share, and back up data.

Service	Description
	Azure Blob Storage Azure Blob Storage allows your applications to store and retrieve files in the cloud. Azure Storage is highly scalable to store massive amounts of data and data is stored redundantly to ensure high availability.
	Azure Data Lake Storage Azure Data Lake Storage is designed to support big data analytics by providing scalable, cost-effective storage for structured, semi-structured or unstructured data.

Messaging

These are some of the most popular services that manage sending, receiving, and routing of messages from and to apps.

Service	Description
	Azure Service Bus A fully managed enterprise message broker supporting both point to point and publish-subscribe integrations. It's ideal for building decoupled applications, queue-based load leveling, or facilitating communication between microservices.
	Azure Event Hubs Azure Event Hubs is a managed service that can ingest and process massive data streams from websites, apps, or devices.
	A simple and reliable queue that can handle large workloads.

Identity and security

Service	Description
 Microsoft Entra ID	Manage user identities and control access to your apps, data, and resources.
 Azure Key Vault	Store and access application secrets like connection strings and API keys in an encrypted vault with restricted access to make sure your secrets and your application aren't compromised.
 App Configuration	A fast and scalable service to centrally manage application settings and feature flags.

Management

Service	Description
 Azure Monitor	A comprehensive monitoring solution for collecting, analyzing, and responding to monitoring data from your cloud and on-premises environments.
 Application Insights	This feature of Azure Monitor provides Application Performance Management (APM) for enhancing the performance, reliability, and quality of your live web applications.

Hosting applications on Azure

Article • 10/25/2023

Azure provides a variety of different ways to host your application depending on your needs. This article suggests services to match requirements. It isn't prescriptive. You can mix and match services to meet your needs. Most production environments use a combination of services to meet their business and organizational needs.

[https://www.microsoft.com/en-us/videoplayer/embed/RE50vLy?postJsIMsg=true ↗](https://www.microsoft.com/en-us/videoplayer/embed/RE50vLy?postJsIMsg=true)

Simplicity and control

Azure hosting services are provided with two considerations:

- **Simplicity versus control**
 - Simple hosting platforms require less configuration and management but provide less control over the underlying infrastructure.
 - More complex hosting platforms require more configuration and management but provide more control over the underlying infrastructure.
- **Cloud-native versus Azure-native**
 - Cloud-native can be thought of as cloud-portable using open-source workloads such as containers and open-source technologies such as Dapr. The applications you build can be deployed to any cloud provider.
 - Azure-native is specific to Azure with an investment in Azure-specific tools and technologies to manage that infrastructure. While these services include container workloads, they also include code-first, low-code, and infrastructure tooling specific to Azure with an emphasis on connecting and integration between Azure services.

Simplified hosting

Simplified hosting solutions are fully managed by Azure. You're responsible for the functionality such as code and environment configuration. Azure manages the underlying runtime and infrastructure including updates and patches. Simplified hosting is the Azure-native approach.

- [Logic Apps](#): Create and run automated workflows with little to no code.
- [Power Automate](#): Use when you need to automate business processes and workflows.
- [Azure Static Web Apps](#): Deploy generated static web apps such as Blazor and React.

- [Azure Functions Apps](#): serverless code or container hosting.

Balanced hosting

Balanced hosting solutions balance the need for simplicity with the need for control. You're responsible for the functionality such as code and environment configuration. Azure manages the underlying runtime and infrastructure including updates and patches. You can also bring your own container to the service. Balanced hosting is both Azure-native and Cloud-native.

- [Azure App Service](#): Full-service web hosting including language runtimes, containers, and automation workloads.
- [Azure Container Apps](#): Serverless container hosting.
- [Azure Spring Apps](#): Migrate Spring Boot applications to the Azure cloud.

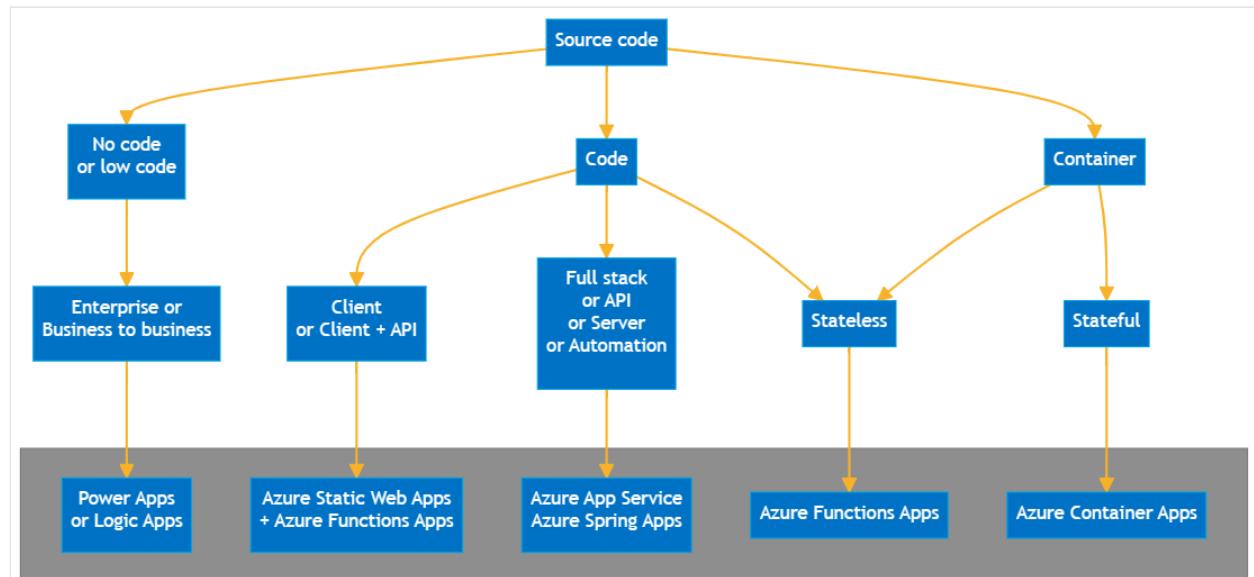
Controlled hosting

Controlled hosting solutions give you full control over the underlying infrastructure. You're responsible for updates and patches as well as your code, assets, and environment configuration. Controlled hosting is the cloud-native approach.

- [Azure Virtual Machines](#): Full control of VM.
- [Azure Kubernetes Service](#): Full control of Kubernetes cluster.

Source-code hosting

For developers new to Azure who want to start **new development**, use the following chart to find the suggested hosting solution.



No code or low code

Azure supports no-code solutions are part of its Azure-Cloud approach.

- [Logic Apps](#): Use a visual designer with prebuilt operations to develop a workflow for your enterprise and business-to-business scenarios.
- [Power Automate](#) such as [Power apps](#): Use when you need to automate business processes and workflows within the Microsoft 365 organization.

Code vs container

Low-code hosting solutions are designed to allow you to bring your code functionality without having to manage the application infrastructure.

- [Azure Static Web Apps](#): deploy generated static web apps.
- [Azure Functions](#): deploy code functions in supported languages without having to manage the application infrastructure.

Code-first hosting solutions are designed to host code. You can deploy your code directly to the hosting solution.

- [Azure App Service](#): full-service web hosting.
- [Azure Spring Apps](#): Spring Boot applications.

Container-first hosting solutions are designed to host containers. The service provides container-specific configuration options and features. You're responsible for the compute used inside the container. The services which host containers move from managed control to full responsibility so you only take on the amount of container management you want.

Kubernetes-centric orchestration hosting includes:

Service	Focus	Use
Azure Kubernetes Service	Cloud-native	Use for Kubernetes clusters with a declarative approach using configuration files and external artifacts.
Azure Service Fabric	Azure-native	Use an imperative approach to deploying microservices across clusters of machines. It provides a programming model that allows developers to write code that describes the desired state of the system, and the Service Fabric runtime takes care of making the system match that state.

Preconfigured container hosting means the orchestration options are preconfigured for you. Your ability to communicate between containers or container clusters might require an additional service such as [Dapr](#).

Service	Use
Azure App Service	full-service web hosting
Azure Spring Apps	Spring Boot applications
Azure Container Apps	serverless container hosting
Azure Container Instances	simple, single container hosting

Azure provides a container registry to store and manage your container images or you can use a third-party container registry.

Service	Use
Azure Container Registry	Use when you build and host your own container images, which can be triggered with source code commits and base image updates.

Serverless

Serverless hosting solutions are designed to run stateless code, which includes a consumption-based pricing tier that scales to zero when not used.

Service	Use
Azure Container Apps	Container hosting.
Azure Functions	Code or container hosting.

Microservices

Microservices hosting solutions are designed to run small, independent services that work together to make up a larger application. Microservices are typically deployed as containers.

Service	Use
Azure Container Apps	Use for serverless containerized microservices.
Azure Functions	Use for serverless code or containerized microservices.

Cloud edge

Cloud edge is a term to indicate if the Cloud service is located to benefit the user (client) or the application (server).

Client compute

Client compute is compute that runs on the client away from the Azure cloud. Client compute is typically used for client-side rendering and client-side processing such as browser-based or mobile applications.

Service	Use
Azure Static Web Apps	Use for static web apps that use client-side rendering such as React, Angular, Svelte, Vue, and Blazor.

Client availability

Service	Use
Azure Front Door	Use for all internet-facing applications to provide a global cached and secure network to your static and dynamic assets including DDoS protection, end-to-end TLS encryption, application firewalls, and geo-filtering.

Server compute

Server compute assets are files that are processed by the server before being served to the client. Dynamic assets are developed using back-end server compute, optionally integrated with other Azure services.

Service	Use
Azure App Service	Use this service for typical web hosting. This supports a wide set of functionality API endpoints, full-stack applications, and background tasks. This service comes with many programming language runtimes as well as the ability to provide your own stack, language, or workload from a container.
Azure Functions	Use this service to provide your own code in the supported languages for either HTTP endpoints or event-based triggers from Azure services.
Azure Spring Apps	Use to deploy Spring Boot applications without code changes.

Service	Use
Azure Container Apps	Use to host managed microservices and containerized applications on a serverless platform.
Azure Container Instances	Use this for simple container scenarios that don't need container orchestration.
Azure Kubernetes Service	Use this service when you need a Kubernetes cluster. The control plane to manage the cluster is created and provided for you at no extra cost.

Server Endpoint Management

Server Endpoint Management is the ability to manage your server endpoint and its compute through a gateway. This gateway provides functionality such as versioning, caching, transformation, API policies, and monitoring.

Service	Use
Azure API Management	Use this service when you productize your REST, OpenAPI, and GraphQL APIs with an API gateway including quotas and rate limits, authentication and authorization, transformation, and cached responses.
Azure Application Gateway	Use for regional load balancing (OSI layer 7). It can be used to route traffic based on URL path or host headers, and it supports SSL offloading, cookie-based session affinity, and Web Application Firewall (WAF) capabilities.
Azure Front Door	Use for global load balancing (OSI layer 7) to provide a global cached and secure network to your static and dynamic assets including DDoS protection, end-to-end TLS encryption, application firewalls, and geo-filtering.
Azure Traffic Manager	Use for distributing traffic by DNS (OSI layer 7) to your public facing applications across the global Azure regions. Traffic Manager uses DNS to direct client requests to the appropriate service endpoint based on a traffic-routing method. It supports various traffic-routing methods such as priority, performance, and geographic routing. It is ideal for managing traffic across multiple regions or data centers.

Automated compute

Automated compute is automated by an event such as a timed schedule or another Azure service and is typically used for background processing, batch processing, or long-running processes.

Service	Use
Power Automate	Use when you need to automate business processes and workflows.
Azure Functions	Use when you need to run code based on a timed schedule or in response to events in other Azure services.
Container services (Azure Container Instances , Azure Kubernetes Service , Azure Container Apps)	Use for standard automatable workloads
Azure Batch	Use when you need high-performance automation.

Hybrid cloud

Hybrid cloud is a computing environment that connects a company's on-premises private cloud services and third-party public cloud into a single, flexible infrastructure for running the organization's applications and workloads.

Service	Use
Azure Arc	Use when need to manage your entire environment, both cloud and on-premises resources including security, governance, inventory, and management.

If you don't need to maintain your own infrastructure, you can use Azure Stack HCI to run virtual machines on-premises.

High performance computing

High-performance computing (HPC) is the use of parallel processing for running advanced application programs efficiently, reliably and quickly. The term applies especially to systems that function above a teraflop or 10^{12} floating-point operations per second.

Service	Use
Azure Batch	Azure Batch creates and manages a pool of compute nodes (virtual machines), installs the applications you want to run, and schedules jobs to run on the nodes. Developers can use Batch as a platform service to build SaaS applications or client apps where large-scale execution is required.

Service	Use
Azure BareMetal Instances	Use when you need to run in a nonvirtualized environment with root-level access to the operating system, storage and network.
Azure Quantum workspace	Use when you need to develop and experiment with quantum algorithms.
Microsoft Genomics	Use for ISO-certified, HIPAA-compliant genomic processing.

Learn more about [High-performance computing on Azure](#).

Event-based compute

Event-based compute is compute that is triggered by an event such as a timed schedule or another Azure service. Event-based compute is typically used for background processing, batch processing, or long-running processes.

Service	Use
Power Virtual Agents	Use when you need to create chatbots with a no-code interface.
Azure Functions	Use when you need to run code based on a timed schedule or in response to events in other Azure services.
Azure Service Bus Messaging	Use when you need to decouple applications and services.

CI/CD compute

CI/CD compute is compute that is used to build and deploy your application.

Service	Description
Azure DevOps	Use Azure DevOps for tight integration with the Azure cloud including authentication and authorization to the hosted agents, which build and deploy your application.
GitHub Actions	Use GitHub Actions to build and deploy your GitHub repository applications. Use the Azure CLI to securely access Azure within the action.
Azure Virtual Machines	If you use another CI/CD system, you can use Azure Virtual Machines to host your CI/CD system.

Java resources

- [Java hosting options](#)
- [Java migration to Azure](#)

Additional resources

- [Azure Architecture Center: Choose an Azure compute service](#)

Develop AI apps using Azure AI services

Build applications with generative AI capabilities on Azure.

Fundamentals of generative AI

OVERVIEW

[Introduction to generative AI](#)

[Concepts and considerations](#)

[Augment LLMs with RAG And Fine-tuning](#)

[Advanced Retrieval-Augmented Generation](#)

AI app templates

OVERVIEW

[AI app templates overview](#)

GET STARTED

[Chat with your data using Python](#)

[Chat with your data using JavaScript](#)

[Chat with your data using Java](#)

[Chat with your data using .NET](#)

Resources by programming language

OVERVIEW

[Python](#)

[JavaScript](#)

[Java](#)

[C# and .NET](#)

Connect your app to Azure Services

Article • 10/18/2022

Azure offers a variety of services that applications can take advantage of regardless of whether they are hosted in Azure or on-premises. For example you could:

- Use Azure Blob Storage to store and retrieve files in the cloud.
- Add full text searching capability to your application using Azure Cognitive Search.
- Use Azure Service Bus to handle messaging between different components of a microservices architecture.
- Use Text Analytics to identify and redact sensitive data in a document.

Azure services offer the benefit that they are fully managed by Azure.

Accessing Azure Services from Application Code

There are two ways to access Azure service from your application code.

- **Azure SDK** - Available for .NET, Java, JavaScript, Python and Go.
- **Azure REST API** - Available from all languages.

When possible, it is recommended to use the Azure SDK to access Azure services from application code. Advantages of using the Azure SDK include:

- **Accessing Azure services is just like using any other library.** You import the appropriate SDK package into your application, create a client object, and then call methods on the client object to communicate with your Azure resource.
- **Simplifies the process of authenticating your application to Azure.** When creating an SDK client object, you include the right credentials and the SDK takes care of authenticating your calls to Azure
- **Simplified programming model.** Internally, the Azure SDK calls the Azure REST API. However, the Azure SDK has built in error handling, retry logic, and result pagination making programming against the SDK simpler than calling the REST API directly.

Azure SDK

The Azure SDK allows programmatic access to Azure services from .NET, Java, JavaScript, Python, and Go applications. Applications install the necessary packages from their

respective package manager and then call methods to programmatically access Azure resources.

<https://www.microsoft.com/en-us/videoplayer/embed/RE50C7t?postJslIMsg=true>

More information about the Azure SDK for each language can be found in each language's developer center.

Language	Overview	Package list
	.NET Azure SDK for .NET overview	Azure SDK for .NET package list
	Java Azure SDK for Java overview	Azure SDK for Java package list
	JavaScript Azure SDK for JavaScript overview	Azure SDK for JavaScript package list
	Python Azure SDK for Python overview	Azure SDK for Python package list
	Go Azure SDK for Go overview	Azure SDK for Go package list

Azure REST API

Programming languages not supported by the Azure SDK can make use of the Azure REST API. Details of how to call the Azure REST API and a full list of operations are available in the [Azure REST API overview](#).

[Azure REST API overview](#)

How do I create and manage resources in Azure?

Article • 10/18/2022

Azure provides a variety of tools to create and manage the Azure resources used by your application.

[https://www.microsoft.com/en-us/videoplayer/embed/RE50C5I?postJsIMsg=true ↗](https://www.microsoft.com/en-us/videoplayer/embed/RE50C5I?postJsIMsg=true)

Different tools are designed to support different use cases, and most Azure developers use a combination of different tools depending on the job they need to perform. For example, you might:

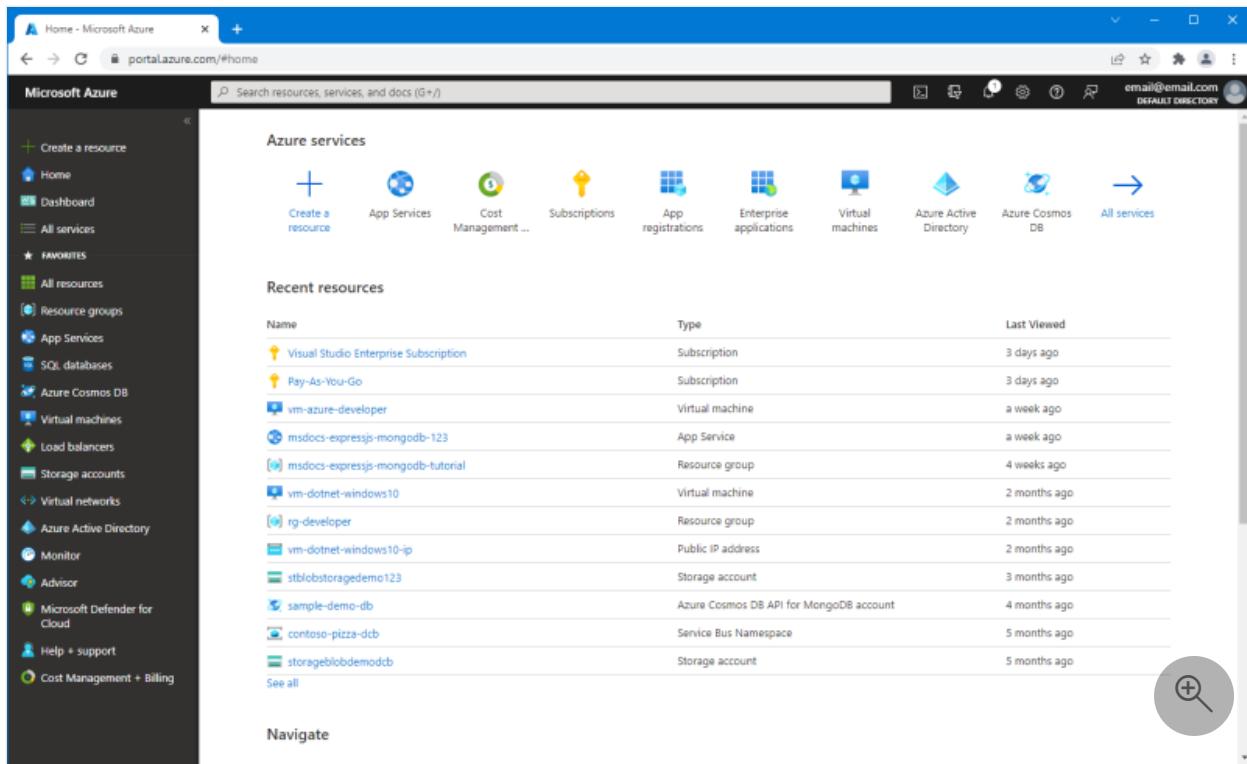
- **Use a GUI tool like the Azure portal or the Azure Tools extension for VS Code** when prototyping Azure resources for a new application. GUI tools guide you through the process of creating new services and let you review and select the options for a service using drop-down menus and other graphical elements.
- **Write a script using the Azure CLI or Azure PowerShell** to automate a common task. For example, you might create a script that creates a basic dev environment for a new web application consisting of an Azure App Service, a database, and blob storage. Writing a script ensures the resources are created the same way each time and is faster to run than clicking through a UI.
- **Use Infrastructure as Code (IaC) tools to declaratively deploy and manage Azure resources.** Tools like Terraform, Ansible, or Bicep allow you to codify the Azure resources needed for a solution in declarative syntax, ensuring the consistent deployment of Azure resources across environments and preventing environmental drift.

Azure portal

The [Azure portal](#) ↗ is a web-based interface designed for managing Azure resources.

The Azure portal features:

- An easy to use, web-based UI for creating and managing Azure resources
- The ability to create configurable dashboards
- Access to subscription settings and billing information

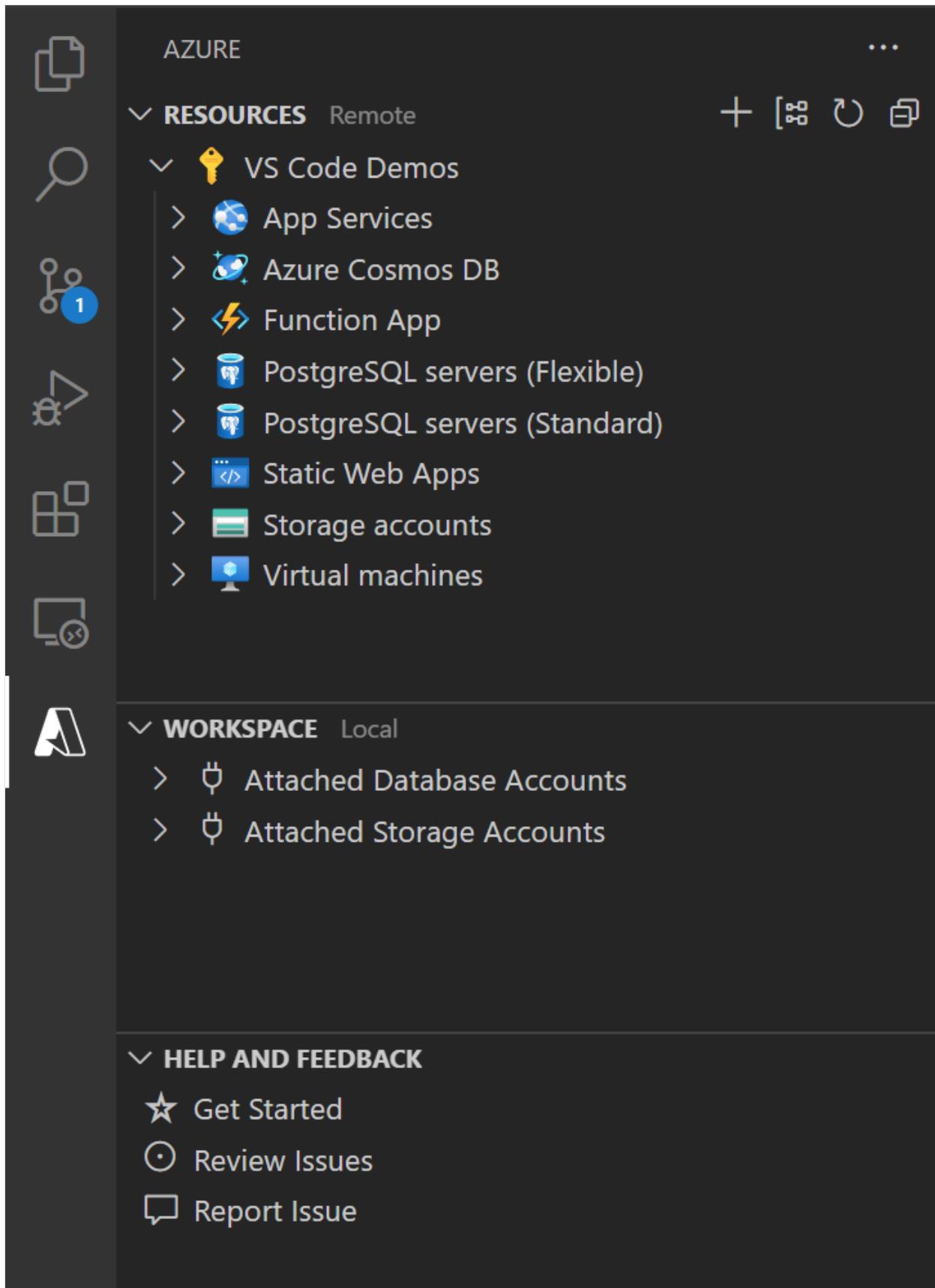


VS Code Azure Tools Extension Pack

Developers using [Visual Studio Code](#) can manage Azure resources right from VS Code using the [Azure Tools Extension Pack](#) for VS Code. Using the Azure Tools Extension Pack can:

- Create, manage, and deploy code to web sites using Azure App Service.
- Create, browse, and query Azure databases
- Create, debug, and deploy Azure Functions directly from VS Code
- Deploy containerized applications from VS Code

[Download Azure Tools extension pack](#)



Command line tools

Command line tools offer the benefits of efficiency, repeatability, and the ability to script recurring tasks. Azure provides two different command line tools to choose from. The

Azure CLI and Azure PowerShell are functionally equivalent. You only need to select and use the tool that best fits your individual workflow.

Azure CLI

The [Azure CLI](#) is a cross-platform command line tool that runs on Windows, Linux and macOS. The Azure CLI:

- Features a concise, efficient syntax for managing Azure resource.
- Outputs results as JSON (by default). Results can also be formatted as YAML, an ASCII table or tab-separated values with no keys.
- Provides the ability to query and shape output through the use of [JMESPath queries](#).

Azure CLI commands are easily incorporated into popular scripting languages like [Bash](#) giving you the ability to script common tasks.

```
Azure CLI

LOCATION='eastus'
RESOURCE_GROUP_NAME='msdocs-expressjs-mongodb-tutorial'

WEB_APP_NAME='msdocs-expressjs-mongodb-123'
APP_SERVICE_PLAN_NAME='msdocs-expressjs-mongodb-plan-123'
RUNTIME='NODE|14-lts'

# Create a resource group
az group create \
    --location $LOCATION \
    --name $RESOURCE_GROUP_NAME

# Create an app service plan
az appservice plan create \
    --name $APP_SERVICE_PLAN_NAME \
    --resource-group $RESOURCE_GROUP_NAME \
    --sku B1 \
    --is-linux

# Create the web app in the app service
az webapp create \
    --name $WEB_APP_NAME \
    --runtime $RUNTIME \
    --plan $APP_SERVICE_PLAN_NAME \
    --resource-group $RESOURCE_GROUP_NAME
```

Azure PowerShell

Azure PowerShell is a set of cmdlets for managing Azure resources directly from PowerShell. Azure PowerShell is installed as a PowerShell module and works with PowerShell 7.0.6 LTS and PowerShell 7.1.3 or higher on all platforms including Windows, macOS, and Linux. It's also compatible with Windows PowerShell 5.1.

Azure PowerShell is tightly integrated with the PowerShell language. Commands follow a verb-noun format and data is returned as PowerShell objects. If you are already familiar with PowerShell scripting, Azure PowerShell is a natural choice.

Azure PowerShell

```
$location = 'eastus'  
$resourceGroupName = 'msdocs-blob-storage-demo-azps'  
$storageAccountName = 'stblobstoragedemo999'  
  
# Create a resource group  
New-AzResourceGroup  
    -Location $location  
    -Name $resourceGroupName  
  
# Create the storage account  
New-AzStorageAccount  
    -Name $storageAccountName  
    -ResourceGroupName $resourceGroupName  
    -Location $location  
    -SkuName Standard_LRS
```

For more information on choosing between Azure CLI and Azure PowerShell, see the article [Choose the right command-line tool](#).

Infrastructure as Code tools

Infrastructure as Code is the process of managing and provisioning resources through declarative configuration files. Infrastructure as code tools use a declarative end state specification to guarantee a set of resources are created and configured the same way each time. Further, most infrastructure as code tools monitor resources to make sure they remain configured in the desired state.

For infrastructure deployments that are automated, repeated, and reliable, Azure supports a variety of Infrastructure as Code tools.

Bicep

Bicep is a domain-specific language (DSL) that uses declarative syntax to deploy Azure resources. It provides concise syntax, reliable type safety, and support for code reuse.

Bicep

```
param location string = resourceGroup().location
param storageAccountName string =
'toylaunch${uniqueString(resourceGroup().id)}'

resource storageAccount 'Microsoft.Storage/storageAccounts@2021-06-01' = {
  name: storageAccountName
  location: location
  sku: {
    name: 'Standard_LRS'
  }
  kind: 'StorageV2'
  properties: {
    accessTier: 'Hot'
  }
}
```

Terraform

Hashicorp Terraform is an open-source tool for provisioning and managing cloud infrastructure. It codifies infrastructure in configuration files that describe the topology of cloud resources. The Terraform CLI provides a simple mechanism to deploy and version configuration files to Azure.

Terraform

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "main" {
  name      = "${var.prefix}-resources"
  location = var.location
}

resource "azurerm_app_service_plan" "main" {
  name          = "${var.prefix}-asp"
  location      = azurerm_resource_group.main.location
  resource_group_name = azurerm_resource_group.main.name
  kind          = "Linux"
  reserved      = true

  sku {
    tier = "Standard"
    size = "S1"
  }
}

resource "azurerm_app_service" "main" {
```

```

name          = "${var.prefix}-appservice"
location      = azurerm_resource_group.main.location
resource_group_name = azurerm_resource_group.main.name
app_service_plan_id = azurerm_app_service_plan.main.id

site_config {
    linux_fx_version = "NODE|10.14"
}
}

```

Ansible

[Ansible](#) is an open-source product that automates cloud provisioning, configuration management, and application deployments. Using Ansible you can provision virtual machines, containers, and network and complete cloud infrastructures. Also, Ansible allows you to automate the deployment and configuration of resources in your environment.

yml

```

- hosts: localhost
  connection: local
  vars:
    resource_group: myResourceGroup
    webapp_name: myfirstWebApp
    plan_name: myAppServicePlan
    location: eastus
  tasks:
    - name: Create a resource group
      azure_rm_resourcegroup:
        name: "{{ resource_group }}"
        location: "{{ location }}"
    - name: Create App Service on Linux with Java Runtime
      azure_rm_webapp:
        resource_group: "{{ resource_group }}"
        name: "{{ webapp_name }}"
        plan:
          resource_group: "{{ resource_group }}"
          name: "{{ plan_name }}"
          is_linux: true
          sku: S1
          number_of_workers: 1
        frameworks:
          - name: "java"
            version: "8"
            settings:
              java_container: tomcat
              java_container_version: 8.5

```

Azure SDK and REST APIs

Azure resources can also be created programmatically from code. This allows you to write applications that dynamically provision Azure resources in response to user requests. The Azure SDK provides resource management packages in .NET, Go, Java, JavaScript and Python that allow Azure resources to be created and managed directly in code. Alternatively, the Azure REST API allows Azure resources to be managed through HTTP requests to a RESTful endpoint.

[Using the Azure SDK for .NET](#)

[Using the Azure SDK for Go](#)

[Using the Azure SDK for Java](#)

[Using the Azure SDK for JavaScript](#)

[Using the Azure SDK for Python](#)

[Using the Azure REST APIs](#)

Key concepts for building Azure apps

Article • 10/18/2022

Before you get too far in designing your application to run on Azure, chances are you'll need to do a little planning ahead of time. As you get started, there are some basic Azure concepts that you need to understand to make the best decisions for your scenario. Considerations include:

Azure regions

A region is a set of datacenters deployed within a latency-defined perimeter and connected through a dedicated regional low-latency network. Azure gives you the flexibility to deploy applications where you need to, including across multiple regions to deliver cross-region resiliency when necessary.

Typically, you want all of the resources for a solution to be in the same region to minimize latency between different components of your application. This means if your solution consists of an Azure App Service, a database, and Azure Blob storage, all of these resources should be created in the same Azure region.

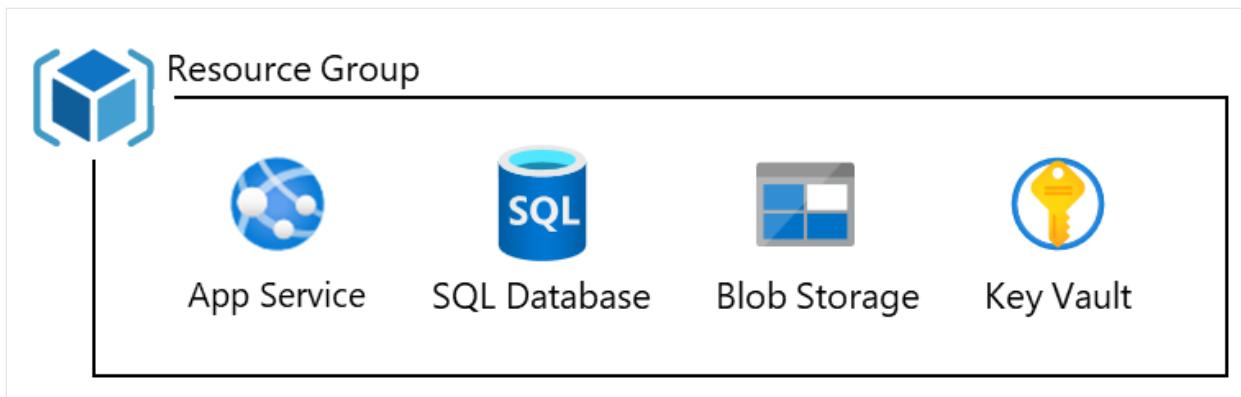
Not every Azure service is available in every region. The [Products available by region](#) page can help you find a region where the Azure services needed by your app are available.

<https://www.microsoft.com/en-us/videoplayer/embed/RE50C5F?postJs||Msg=true>

Azure resource group

A Resource Group in Azure is a logical container to group Azure Resources together. Every Azure resource must belong to one and only one resource group.

Resource groups are most often used to group together all of the Azure resources needed for a solution in Azure. For example, say you've a web application deployed to Azure App Service that uses a SQL database, Azure Storage, and also Azure Key Vault. It's common practice to put all of the Azure resources needed for this solution into a single resource group.

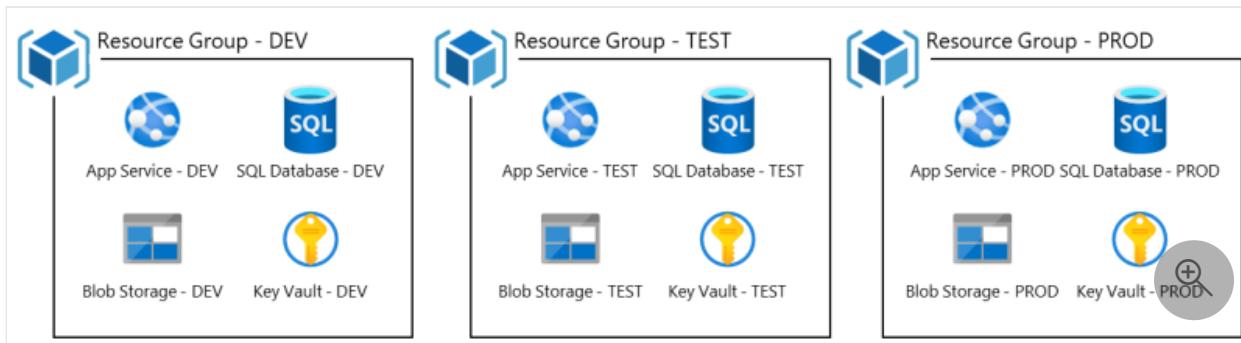


This makes it easier to tell what resources are needed for the application to run and what resources are related to each other. As such, the first step in creating resources for an app in Azure is usually creating the resource group that will serve as a container for the app's resources.

[https://www.microsoft.com/en-us/videoplayer/embed/RE50C5E?postJslIMsg=true ↗](https://www.microsoft.com/en-us/videoplayer/embed/RE50C5E?postJslIMsg=true)

Environments

If you've developed on-premises, you are familiar with promoting your code through dev, test, and production environments. In Azure, to create separate environments you would create a separate set of Azure resources for each environment you need.



Since it's important that each environment be an exact copy, it's recommended to either [script the creation of resources](#) needed for an environment or use [Infrastructure as Code \(IaC\) tools](#) to declaratively specify the configuration of each environment. This makes sure that the environment creation process is repeatable and also give you the ability to spin up new environments on demand, for example for performance or security testing of your application.

[https://www.microsoft.com/en-us/videoplayer/embed/RE50C5M?postJslIMsg=true ↗](https://www.microsoft.com/en-us/videoplayer/embed/RE50C5M?postJslIMsg=true)

DevOps Support

Whether it's publishing your apps to Azure with continuous integration or provisioning resources for a new environment, Azure integrates with most of the popular DevOps

tools. You can work with the tools that you already have and maximize your existing experience with support for tools like:

- [GitHub Actions](#)
- [Azure DevOps](#)
- [Octopus Deploy ↗](#)
- [Jenkins](#)
- [Terraform](#)
- [Ansible](#)
- [Chef ↗](#)

How am I billed?

Article • 10/18/2022

When creating applications that use Azure, you need to understand the factors that influence the cost of the solutions you create. You will also want to understand how you can estimate the cost of a solution, how you're billed, and how you can monitor the costs incurred in your Azure subscriptions.

What is an Azure Account?

Your Azure account is what allows you to sign in to Azure. You may have an Azure account through the organization you work for or the school you attend. You may also create an individual Azure account for personal use linked to your Microsoft account. If you're looking to learn about and experiment with Azure, you can [create an Azure account for free](#).

[Create a free Azure account](#)

If you're using an Azure account from your workplace or school, your organization's Azure administrators has likely assigned different groups and roles to your account that govern what you can and cannot do in Azure. If you can't create a certain type of resource, check with your Azure administrator on the permissions assigned to your account.

What is an Azure subscription?

Billing for Azure resources is done on a per-subscription basis. An Azure subscription therefore defines a set of Azure resources that will be invoiced together.

Organizations often create multiple Azure subscriptions for billing and management purposes. For example, an organization may choose to create one subscription for each department in the organization such that each department pays for their own Azure resources. *When creating Azure resources, it's important to pay attention to what subscription you're creating the resources in because the owner of that subscription will pay for those resources.*

If you have an individual Azure account tied to your Microsoft account, it's also possible to have multiple subscriptions. For example, a user might have both a Visual Studio Enterprise subscription that provides monthly Azure credits and a Pay-as-you-go subscription which bills to their credit card. In this scenario, you again want to be sure

and choose the right subscription when creating Azure resources to avoid an unexpected bill for Azure services.

<https://www.microsoft.com/en-us/videoplayer/embed/RE50ydl?postJs||Msg=true>

What factors influence the cost of a service on Azure?

There are several factors that can influence the cost of a given service in Azure.

- **Compute power** - Compute power refers to the amount of CPU and memory assigned to a resource. The more compute power allocated to a resource, the higher the cost will be. Many Azure services include the ability to elastically scale, allowing you to ramp up compute power when demand is high but scale back and save money when demand is low.
- **Storage amount** - Most storage services are billed based on the amount of data you want to store.
- **Storage hardware** - Some storage services provide options on the type of hardware your data will be stored on. Depending on the type of data you're storing, you may want a more long-term storage option with slower read and write speeds, or you may be willing to pay for low latency read and writes for highly transactional operations.
- **Bandwidth** - Most services bill ingress and egress separately. Ingress is the amount of bandwidth required to handle incoming requests. Egress is the amount of bandwidth required to handle outgoing data that satisfies those requests.
- **Per use** - Some services bill based on the number of times the service is used or a count of the number of requests that are handled or the number of some entity (such as Azure Active Directory user accounts) that have been configured.
- **Per service** - Some services simply charge a straight monthly fee.
- **Region** - Sometimes, services have different prices depending on the region (data center) where it's hosted.

Azure Pricing Calculator

Most Azure solutions involve multiple Azure services, making it challenging to determine the cost of a solution upfront. For this reason, Azure provides the [Azure Pricing Calculator](#) to help estimate how much a solution will cost.

[Azure Pricing Calculator](#)

Where can I find our current spend in Azure?

The Azure portal provides an easy to navigate and visual presentation of all the services your organization utilized during a particular month. You can view by service, by resource group, and so on.

To access billing information in the Azure portal, [sign in to the Azure portal](#) and follow these steps.

Instructions	Screenshot
<p>To view billing information for your Azure account:</p> <ol style="list-style-type: none">1. In the search box at the top of the page, type <i>Billing*</i>.2. Select the <i>Cost Management + Billing</i> item in the dialog.	
<p>You will be taken to the Cost Management + Billing Overview page. On this page you can:</p> <ol style="list-style-type: none">1. Use the left-hand menu to review <i>Invoices</i> and <i>Payment methods</i> for your subscriptions.2. View a list of your subscriptions and their current charges. Selecting a subscription from the table will take you to detailed cost information about that subscription.	
<p>The details page for each subscription allows you to:</p> <ol style="list-style-type: none">1. Perform <i>Cost analysis</i> and set up <i>Cost alerts</i> on the subscription.2. View detailed costs by resource in the subscription.	

You can also access the Cost Management + Billing overview page directly.

Azure Cost Management in the Azure Portal

Cost information can also be accessed programmatically to create a customized and easily accessible view into your cloud spend for management via the Billing API.

- [Azure Billing libraries for .NET](#)
- [Azure Billing libraries for Python](#)
- [Azure Resource Manager Billing client library for Java - Version 1.0.0-beta.1](#)
- [All other programming languages - RESTful API](#)
- [Azure consumption API overview](#)

What tools are available to monitor and analyze my cloud spend?

Two services are available to set up and manage your cloud costs.

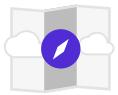
- The first is **cost alerts** which allows you to set spending thresholds and receive notifications as your bill nears those thresholds.
- The second is **Azure Cost Management** which helps you plan for and control your costs, providing cost analysis, budgets, recommendations, and allows you to export cost management data for analysis in Excel or your own custom reporting.

Learn more about cost alerts and **Azure Cost Management**:

- [Use cost alerts to monitor usage and spending](#)
- [What is Azure Cost Management + Billing?](#)
- [How to optimize your cloud investment with Azure Cost Management](#)

Azure for .NET developers

Learn to use the Azure SDK for .NET. Browse API reference, sample code, tutorials, quickstarts, conceptual articles and more. Know that .NET ❤️ Azure.



OVERVIEW

[Introduction to Azure and .NET](#)



QUICKSTART

[Create an ASP.NET Core web app in Azure](#)



QUICKSTART

[Build a serverless function](#)



TUTORIAL

[ASP.NET Core and Docker](#)



DEPLOY

[Deploy a .NET app with Azure DevOps](#)



TUTORIAL

[Authentication end-to-end in App Service](#)



TRAINING

[Secure custom APIs with Microsoft Identity](#)



GET STARTED

[Azure SDK for .NET](#)

Featured content

Learn to develop .NET apps leveraging a variety of Azure services.

Create web apps

- [App Service overview](#)
- [Azure Functions overview](#)
- [Host a web app with Azure App Service](#)
- [Develop, test, and deploy an Azure Function with](#)

Create cloud native apps

- [Build cloud native apps using .NET Aspire](#)
- [Build your first .NET Aspire app](#)

Create intelligent apps with AI

- [AI for .NET overview](#)
- [Build a chat app](#)
- [Generate images](#)

Visual Studio

- 🔗 Publish and manage your APIs with Azure API Management
- 🔗 ASP.NET Core web app with App Service and Azure SQL Database
- 🔗 Managed identity with ASP.NET and Azure SQL Database
- 🔗 Web API with CORS in Azure App Service

- 🚀 Run and debug a microservice in Kubernetes
- 📅 Create and deploy a cloud-native ASP.NET Core microservice
- 🕒 Deploy and debug multiple containers in AKS
- 🔧 Dynamic configuration and feature flags using Azure App Config
- 📅 Deploy a .NET Core app to Azure Container Registry

- ☰ Understand prompt engineering

- ☰ Understand generative AI and LLMs
- ☰ Implement RAG using vector search
- ☰ Get started with the .NET enterprise chat sample
- ☰ Learning resources and samples

Create mobile apps

- 🔗 Consume REST services from Azure in Xamarin apps
- 🔗 Create a Xamarin.Forms app with .NET SDK and Azure Cosmos DB's API for MongoDB
- 🔗 Azure Blob storage client library with Xamarin.Forms
- 🔗 Send push notifications to Xamarin.Forms apps using ASP.NET Core and Azure Notification Hubs
- ☰ Add authentication and manage user identities in your mobile apps

Work with storage & data

- ☰ Choose the right data storage option
- 🚀 Use .NET to query an Azure SQL Database or Azure SQL Managed Instance
- 🚀 Use .NET to query Azure PostgreSQL
- ☰ Use the Repository Pattern with Azure Cosmos DB ↗
- 🕒 Connect to and query an Azure Database for PostgreSQL database
- 📅 Persist and retrieve relational data with Entity Framework Core
- 📅 Build a .NET Core app with Azure Cosmos DB in Visual Studio Code
- 📅 Store application data with Azure Blob storage

Authentication and security

- 🔗 Microsoft identity platform (Azure AD) overview
- 🔗 Secure your application by using OpenID Connect and Azure AD
- 🔗 Secure custom APIs with Microsoft Identity
- 🔗 Secure an ASP.NET Core web app with the ASP.NET Identity framework
- 🔗 Add sign-in to Microsoft to an ASP.NET web app
- 🔗 End-to-end authentication in App Service
- 🔗 Use Azure Key Vault with ASP.NET Core
- 🔗 Integrate Azure AD B2C with a web API
- 🔗 Azure Identity client library for .NET

Messaging on Azure

- 🔗 Storing messages with Azure queues
- 🔗 Inter-application messaging with Azure Service Bus
- 🔗 Streaming big data with Event Hubs

Diagnostics and monitoring

- 🚀 Azure Monitor Application Insights quickstart
- 📅 Capture and view page load times in your Azure web app

Azure SDK for .NET

- ☰ Packages
- ☰ Authentication for apps
- ☰ Logging
- ☰ SDK example app
- ☰ Tools checklist
- ☰ API reference

- Building event-based applications with Event Grid
- Use Azure Queue Storage
- Use Azure Service Bus queues
- Ingest data in real-time through Azure Event Hubs
- Route custom events with Event Grid

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Capture Application Insights telemetry with .NET Core ILogger
- Application Insights for Worker Service applications
- Troubleshoot an app in Azure App Service using Visual Studio

.NET and Azure community resources

.NET

- [.NET documentation](#)
- [ASP.NET documentation](#)

Webcasts and shows

- [Azure Friday ↗](#)
- [The Cloud Native Show](#)
- [On .NET](#)
- [.NET Community Standup ↗](#)
- [On .NET Live ↗](#)

Open Source

- [Azure SDK for .NET ↗](#)
- [Microsoft Identity Web ↗](#)
- [.NET Platform ↗](#)

Are you interested in contributing to the .NET docs? For more information, see our [contributor guide](#).

Azure for Java developer documentation

Get started developing apps for the cloud with these tutorials and tools for Java developers.

Get started with Java on Azure

-  [Code, deploy, and scale Java your way](#)
-  [Code using the Java tools you know and love](#)
-  [Deploy with confidence and ease](#)
-  [Scale with security, monitoring, automation](#)
-  [Choose the right Azure services](#)

[See more >](#)

Azure AI for Java

-  [Develop using Azure AI services](#)
-  [Enterprise chat using RAG](#)
-  [Scale chat using RAG with Azure Container Apps](#)

[See more >](#)

Java learning resources

-  [Overview](#)
-  [Java on Azure samples](#)
-  [Get Java help from Microsoft](#)

[See more >](#)

Azure for Java quickstarts

-  [Deploy a Java SE web app to Linux](#)
-  [Create a serverless function](#)
-  [Deploy Spring Cloud microservices](#)

[See more >](#)

Tools, IDEs, and supported JDKs

-  [Java support](#)
-  [Java JDK installation](#)
-  [Java Docker images for Azure](#)

[See more >](#)

Migrate to Azure

-  [Spring to Azure Spring Apps](#)
-  [Tomcat to Azure App Service](#)
-  [WebLogic to Azure Virtual Machines](#)

[See more >](#)

Azure App Service

- Create a Java app
- Configure Java
- Deploy a Spring app with MySQL

[See App Service documentation >](#)

Azure Spring Apps

- What is Azure Spring Apps?
- Launch your first app
- Enterprise tier

[See more >](#)

Secure apps using the Microsoft identity platform

- Overview
- Secure Spring Boot apps
- Secure Tomcat apps

[See more >](#)

Azure SDK for Java

- Libraries, drivers, and Spring modules
- Azure development using Maven
- Introducing Azure SDK for Java

[See more >](#)

Spring on Azure integration

- What is Spring Cloud Azure?
- Spring Data for Azure Cosmos DB
- Deploy a Spring Boot app

[See more >](#)

Containerization

- Overview
- Establish a baseline
- Containerize for Kubernetes

[See more >](#)

Azure Functions

- Create an Azure Function
- Create a Spring Cloud Function
- Developer guide

[See Azure Functions documentation >](#)

Monitoring Java apps

- Get started with Application Insights
- Get started with ELK
- Monitor Spring apps

[See Azure Monitor documentation >](#)

Securing Java apps

- Enable end-user authentication
- Microsoft Authentication Library
- Manage app secrets

[See Active Directory documentation >](#)

Java EE, Jakarta EE, and MicroProfile

- Oracle WebLogic Server on Azure VMs
- Deploy a Java EE app to AKS
- Deploy a MicroProfile App to Azure App Service

[See Java EE documentation >](#)

Tools



Azure Toolkit for IntelliJ



Visual Studio Code ↗



Azure Toolkit for Eclipse



Eclipse Microprofile



Maven ↗



Gradle ↗



Azure CLI



Jenkins on Azure

Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates.

Azure for JavaScript & Node.js developers

Explore the power of JavaScript on Azure through Quickstarts, How-To Guides, codes samples and more.

New to Azure?

GET STARTED

[What is Azure?](#)

[Azure Fundamentals](#)

[Install Node.js](#)

[Set up development environment](#)

[Authenticate to Azure SDK](#)

[Authenticate your web app users](#)

Deploy and host apps

TUTORIAL

[Select a hosting service](#)

[Client: JamStack upload image to Storage](#)

[Client: JamStack + auth](#)

[Migrate to serverless](#)

[Serverless: Getting started](#)

[Server: Deploy Express.js](#)

[Server from VM: Express.js with NGINX](#)

AI

QUICKSTART

[Enterprise chat with RAG](#)

[Evaluate chat app](#)

[Scale Azure OpenAI with Azure Container Apps](#)

[Scale Azure OpenAI with Azure API Management](#)

[Add search to website](#)

End to end development



[Contoso Solution](#)

[Developer tools](#)

[Sample ↗](#)

Azure SDK client library tutorials



[Static web app: Analyze image with Computer Vision](#)

[Static web app: Upload file to Storage Blob](#)

[Express.js: Add Application Insights logging](#)

Web + Data



[Storage on Azure](#)

[Full stack serverless with Mongoose](#)

[Serverless API + DB](#)

[Express.js + MongoDB \(training\)](#)

[Express.js + MongoDB \(docs\)](#)

Use Azure client libraries (SDK)

GET STARTED

[Use the Azure SDKs for JS/TS](#)

[Azure SDK Blog ↗](#)

[SDK latest version ↗](#)

[SDK Reference Documentation](#)

[SDK Source code for JS ↗](#)

[Samples browser](#)

Developer Guides

GET STARTED

[Azure Cosmos DB MongoDB](#)

[Azure Cosmos DB no-SQL](#)

[Key Vault Keys](#)

[Key Vault Secrets](#)

[Azure Storage Dev Guide](#)

Developer tools

GET STARTED

[Visual Studio Code \(IDE\) ↗](#)

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

[Azure Developer CLI](#)

[Azure Static Web Apps CLI ↗](#)

[Azure Functions core tools CLI ↗](#)

[Windows Terminal ↗](#)

Windows Subsystem for Linux (WSL)

Azure for Python Developers

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

Azure libraries (SDK)

GET STARTED

[Get started](#)

[Set up your local dev environment](#)

[Get to know the Azure libraries](#)

[Learn library usage patterns](#)

[Authenticate with Azure services](#)

Web apps

TUTORIAL

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

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

[Deploy Web App with PostgreSQL](#)

[Deploy Web App with Managed Identity](#)

[Deploy using GitHub Actions](#)

AI

QUICKSTART

[Develop using Azure AI services](#)

[Python enterprise RAG chat sample](#)

Containers



[Python containers overview](#)

[Deploy to App Service](#)

[Deploy to Container Apps](#)

[Deploy a Kubernetes cluster](#)

Data and storage



[SQL databases](#)

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

[Big data and analytics](#)

Machine learning



[Create an ML experiment](#)

[Train a prediction model](#)

[Create ML pipelines](#)

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

[Serverless, Cloud ETL](#)

Serverless functions



[Deploy using Visual Studio Code](#)

[Deploy using the command line](#)

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

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

Developer tools

GET STARTED

[Visual Studio Code \(IDE\) ↗](#)

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

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

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

Azure for Go developers

Learn to use the Azure SDK for Go, browse API references, sample code, tutorials, quickstarts, conceptual articles, and more.

Get started

OVERVIEW

[Take your first steps with Go](#)

[What is the Azure SDK for Go?](#)

DOWNLOAD

[Install the Azure SDK for Go ↗](#)

Data

QUICKSTART

[Use Blob Storage with Go](#)

[Connect to an Azure Database for PostgreSQL](#)

[Connect to an Azure Database for MySQL](#)

[Query an Azure SQL database](#)

Virtual Machines

QUICKSTART

[Authenticate with a managed identity](#)

Serverless

QUICKSTART

Containers

QUICKSTART

[Build and containerize a Go app ↗](#)

[Azure Container Apps](#)

Open source

REFERENCE

[Dapr \(Distributed Application Runtime\) ↗](#)

[KEDA \(Kubernetes Event Driven Autoscaler\) ↗](#)

[KEDA HTTP Add-on ↗](#)

Passwordless connections for Azure services

Article • 06/02/2023

ⓘ Note

Passwordless connections is a language-agnostic feature spanning multiple Azure services. Although the current documentation focuses on a few languages and services, we're currently in the process of producing additional documentation for other languages and services.

This article describes the security challenges with passwords and introduces passwordless connections for Azure services.

Security challenges with passwords and secrets

Passwords and secret keys should be used with caution, and developers must never place them in an unsecure location. Many apps connect to backend database, cache, messaging, and eventing services using usernames, passwords, and access keys. If exposed, these credentials could be used to gain unauthorized access to sensitive information such as a sales catalog that you built for an upcoming campaign, or customer data that must be private.

Embedding passwords in an application itself presents a huge security risk for many reasons, including discovery through a code repository. Many developers externalize such passwords using environment variables so that applications can load them from different environments. However, this only shifts the risk from the code itself to an execution environment. Anyone who gains access to the environment can steal passwords, which in turn, increases your data exfiltration risk.

The following code example demonstrates how to connect to Azure Storage using a storage account key. Many developers gravitate towards this solution because it feels familiar to options they've worked with in the past, even though it isn't an ideal solution. If your application currently uses access keys, consider migrating to passwordless connections.

C#

```
// Connection using secret access keys
BlobServiceClient blobServiceClient = new(
```

```
new Uri("https://<storage-account-name>.blob.core.windows.net"),
new StorageSharedKeyCredential("<storage-account-name>", "<your-access-key>"));
```

Developers must be diligent to never expose these types of keys or secrets in an unsecure location. Many companies have strict security requirements to connect to Azure services without exposing passwords to developers, operators, or anyone else. They often use a vault to store and load passwords into applications, and further reduce the risk by adding password-rotation requirements and procedures. This approach, in turn, increases the operational complexity and, at times, leads to application connection outages.

Passwordless connections and Zero Trust

You can now use passwordless connections in your apps to connect to Azure-based services without any need to rotate passwords. In some cases, all you need is configuration—no new code is required. Zero Trust uses the principle of "never trust, always verify, and credential-free". This means securing all communications by trusting machines or users only after verifying identity and prior to granting them access to backend services.

The recommended authentication option for secure, passwordless connections is to use managed identities and Azure role-based access control (RBAC) in combination. With this approach, you don't have to manually track and manage many different secrets for managed identities because these tasks are securely handled internally by Azure.

You can configure passwordless connections to Azure services using Service Connector or you can configure them manually. Service Connector enables managed identities in app hosting services like Azure Spring Apps, Azure App Service, and Azure Container Apps. Service Connector also configures backend services with passwordless connections using managed identities and Azure RBAC, and hydrates applications with necessary connection information.

If you inspect the running environment of an application configured for passwordless connections, you can see the full connection string. The connection string carries, for example, a database server address, a database name, and an instruction to delegate authentication to an Azure authentication plugin, but it doesn't contain any passwords or secrets.

The following video illustrates passwordless connections from apps to Azure services, using Java applications as an example. Similar coverage for other languages is forthcoming.

<https://www.youtube-nocookie.com/embed/X6nR3AjlwJw>

Introducing DefaultAzureCredential

Passwordless connections to Azure services through Azure AD and Role Based Access control (RBAC) can be implemented using `DefaultAzureCredential` from the Azure Identity client libraries.

ⓘ Important

Some languages must implement `DefaultAzureCredential` explicitly in their code, while others utilize `DefaultAzureCredential` internally through underlying plugins or drivers.

`DefaultAzureCredential` supports multiple authentication methods and automatically determines which should be used at runtime. This approach enables your app to use different authentication methods in different environments (local dev vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` searches for credentials varies between languages:

- .NET
- C++ ↗
- Go ↗
- Java
- JavaScript
- Python

For example, when working locally with .NET, `DefaultAzureCredential` will generally authenticate using the account the developer used to sign-in to Visual Studio, Azure CLI, or Azure PowerShell. When the app is deployed to Azure, `DefaultAzureCredential` will automatically discover and use the [managed identity](#) of the associated hosting service, such as Azure App Service. No code changes are required for this transition.

ⓘ Note

A managed identity provides a security identity to represent an app or service. The identity is managed by the Azure platform and doesn't require you to provision or

rotate secrets. You can read more about managed identities in the [overview](#) documentation.

The following code example demonstrates how to connect to Service Bus using passwordless connections. Other documentation describes how to migrate to this setup for a specific service in more detail. A .NET app can pass an instance of `DefaultAzureCredential` into the constructor of a service client class. `DefaultAzureCredential` will automatically discover the credentials that are available in that environment.

C#

```
ServiceBusClient serviceBusClient = new(
    new Uri("https://<your-service-bus-namespace>.blob.core.windows.net"),
    new DefaultAzureCredential());
```

See also

For a more detailed explanation of passwordless connections, see the developer guide [Configure passwordless connections between multiple Azure apps and services](#).

Configure passwordless connections between multiple Azure apps and services

Article • 02/09/2024

Applications often require secure connections between multiple Azure services simultaneously. For example, an enterprise Azure App Service instance might connect to several different storage accounts, an Azure SQL database instance, a service bus, and more.

[Managed identities](#) are the recommended authentication option for secure, passwordless connections between Azure resources. Developers do not have to manually track and manage many different secrets for managed identities, since most of these tasks are handled internally by Azure. This tutorial explores how to manage connections between multiple services using managed identities and the Azure Identity client library.

Compare the types of managed identities

Azure provides the following types of managed identities:

- **System-assigned managed identities** are directly tied to a single Azure resource. When you enable a system-assigned managed identity on a service, Azure will create a linked identity and handle administrative tasks for that identity internally. When the Azure resource is deleted, the identity is also deleted.
- **User-assigned managed identities** are independent identities that are created by an administrator and can be associated with one or more Azure resources. The lifecycle of the identity is independent of those resources.

You can read more about best practices and when to use system-assigned identities versus user-assigned identities in the [identities best practice recommendations](#).

Explore DefaultAzureCredential

Managed identities are generally implemented in your application code through a class called `DefaultAzureCredential` from the `Azure.Identity` client library.

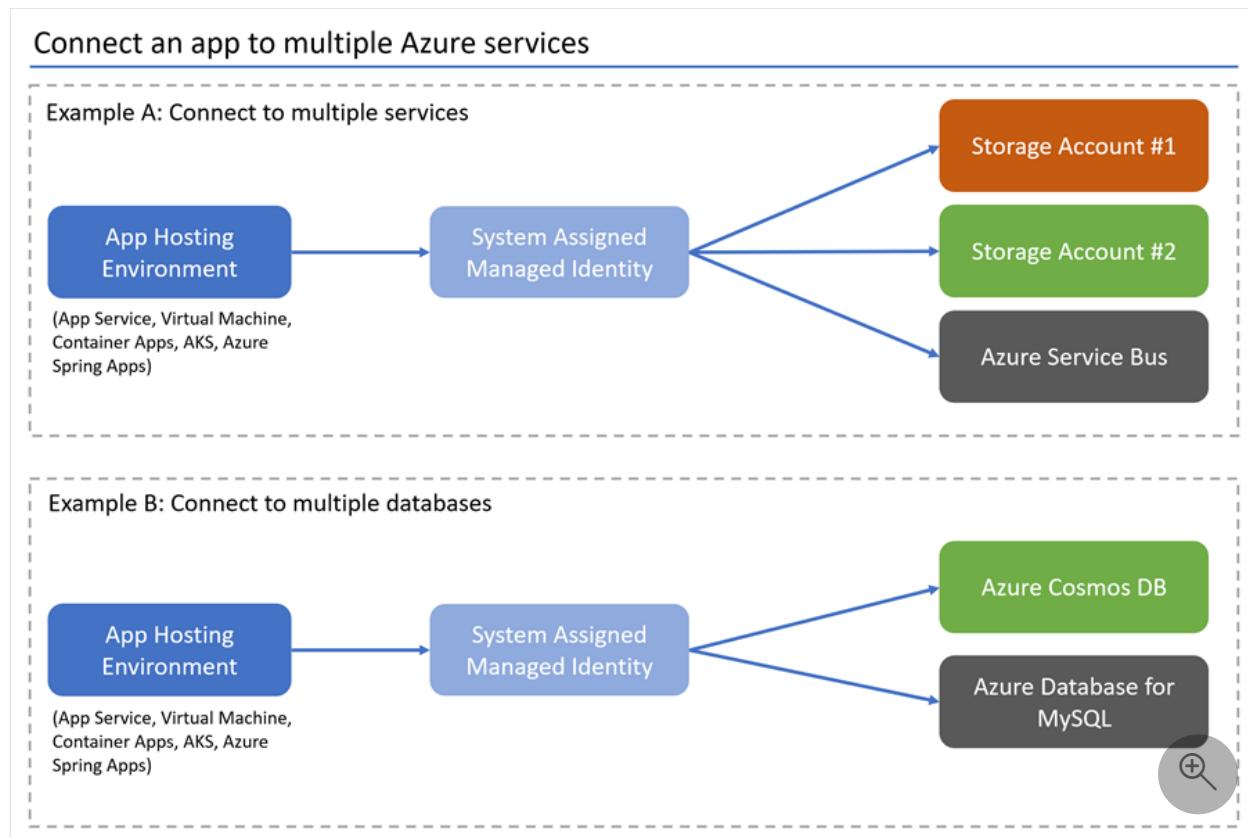
`DefaultAzureCredential` supports multiple authentication methods and automatically

determines which should be used at runtime. You can read more about this approach in the [DefaultAzureCredential overview](#).

Connect an Azure hosted app to multiple Azure services

You have been tasked with connecting an existing app to multiple Azure services and databases using passwordless connections. The application is an ASP.NET Core Web API hosted on Azure App Service, though the steps below apply to other Azure hosting environments as well, such as Azure Spring Apps, Virtual Machines, Container Apps and AKS.

This tutorial applies to the following architectures, though it can be adapted to many other scenarios as well through minimal configuration changes.



The following steps demonstrate how to configure an app to use a system-assigned managed identity and your local development account to connect to multiple Azure Services.

Create a system-assigned managed identity

1. In the Azure portal, navigate to the hosted application that you would like to connect to other services.

2. On the service overview page, select **Identity**.
3. Toggle the **Status** setting to **On** to enable a system assigned managed identity for the service.

The screenshot shows the 'Identity' blade for an Azure App Service named 'msdocs-web-app-123'. The left sidebar lists navigation options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Deployment (Quickstart, Deployment slots, Deployment Center), Settings (Configuration, Authentication, Application Insights), and Identity. The 'Identity' option is highlighted with a red box. The main content area shows the current configuration: 'System assigned' is selected under 'Managed identities', and the 'Status' switch is set to 'On'. Other fields include 'Object (principal) ID' (65634e65-f3eb-4fe7-aff3-fd31d035084b) and a 'Permissions' section for 'Azure role assignments'. A note at the bottom states: 'This resource is registered with Azure Active Directory. The managed identity can be configured to allow access to other resources.'

Assign roles to the managed identity for each connected service

1. Navigate to the overview page of the storage account you would like to grant access your identity access to.
2. Select **Access Control (IAM)** from the storage account navigation.
3. Choose **+ Add** and then **Add role assignment**.

The screenshot shows the Azure Storage account 'identitymigrationstorage' under the 'Access Control (IAM)' section. The left sidebar has a red box around the 'Access Control (IAM)' option. The main area has a red box around the 'Add role assignment' button at the top.

4. In the **Role** search box, search for *Storage Blob Data Contributor*, which grants permissions to perform read and write operations on blob data. You can assign whatever role is appropriate for your use case. Select the *Storage Blob Data Contributor* from the list and choose **Next**.
5. On the **Add role assignment** screen, for the **Assign access to** option, select **Managed identity**. Then choose **+Select members**.
6. In the flyout, search for the managed identity you created by entering the name of your app service. Select the system assigned identity, and then choose **Select** to close the flyout menu.

The screenshot shows the 'Add role assignment' wizard. Step 1 (highlighted with a red box) shows the 'Assign access to' dropdown with 'Managed identity' selected. Step 2 (highlighted with a red box) shows the 'Members' button. Step 3 (highlighted with a red box) shows the 'Select members' flyout with the search bar containing 'msdocs-web'. Step 4 (highlighted with a red box) shows the 'Selected members' list with 'msdocs-web-app-123'. Step 5 (highlighted with a red box) shows the 'Select' button. Step 6 (highlighted with a red box) shows the 'Next' button.

7. Select **Next** a couple times until you're able to select **Review + assign** to finish the role assignment.
8. Repeat this process for the other services you would like to connect to.

Local development considerations

You can also enable access to Azure resources for local development by assigning roles to a user account the same way you assigned roles to your managed identity.

1. After assigning the **Storage Blob Data Contributor** role to your managed identity, under **Assign access to**, this time select **User, group or service principal**. Choose **+ Select members** to open the flyout menu again.
2. Search for the *user@domain* account or Microsoft Entra security group you would like to grant access to by email address or name, and then select it. This should be the same account you use to sign-in to your local development tooling with, such as Visual Studio or the Azure CLI.

ⓘ Note

You can also assign these roles to a Microsoft Entra security group if you are working on a team with multiple developers. You can then place any developer inside that group who needs access to develop the app locally.

Implement the application code

C#

Inside of your project, add a reference to the `Azure.Identity` NuGet package. This library contains all of the necessary entities to implement `DefaultAzureCredential`. You can also add any other Azure libraries that are relevant to your app. For this example, the `Azure.Storage.Blobs` and `Azure.KeyVault.Keys` packages are added in order to connect to Blob Storage and Key Vault.

.NET CLI

```
dotnet add package Azure.Identity  
dotnet add package Azure.Storage.Blobs  
dotnet add package Azure.KeyVault.Keys
```

At the top of your `Program.cs` file, add the following using statements:

C#

```
using Azure.Identity;  
using Azure.Storage.Blobs;
```

```
using Azure.Security.KeyVault.Keys;
```

In the `Program.cs` file of your project code, create instances of the necessary services your app will connect to. The following examples connect to Blob Storage and service bus using the corresponding SDK classes.

C#

```
var blobServiceClient = new BlobServiceClient(  
    new Uri("https://<your-storage-account>.blob.core.windows.net"),  
    new DefaultAzureCredential(credOptions));  
  
var serviceBusClient = new ServiceBusClient("<your-namespace>", new  
DefaultAzureCredential());  
var sender = serviceBusClient.CreateSender("producttracking");
```

When this application code runs locally, `DefaultAzureCredential` will search a credential chain for the first available credentials. If the `Managed_Identity_Client_ID` is null locally, it will automatically use the credentials from your local Azure CLI or Visual Studio sign-in. You can read more about this process in the [Azure Identity library overview](#).

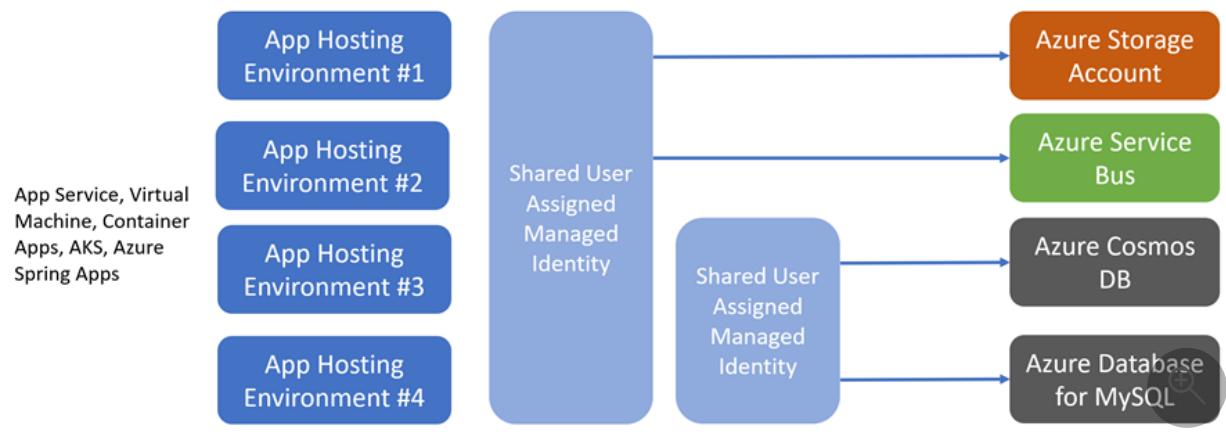
When the application is deployed to Azure, `DefaultAzureCredential` will automatically retrieve the `Managed_Identity_Client_ID` variable from the app service environment. That value becomes available when a managed identity is associated with your app.

This overall process ensures that your app can run securely locally and in Azure without the need for any code changes.

Connect multiple apps using multiple managed identities

Although the apps in the previous example all shared the same service access requirements, real environments are often more nuanced. Consider a scenario where multiple apps all connect to the same storage accounts, but two of the apps also access different services or databases.

Connect multiple apps to Azure services using shared user-assigned identities



To configure this setup in your code, make sure your application registers separate services to connect to each storage account or database. Make sure to pull in the correct managed identity client IDs for each service when configuring `DefaultAzureCredential`. The following code example configures the following service connections:

- Two connections to separate storage accounts using a shared user-assigned managed identity
- A connection to Azure Cosmos DB and Azure SQL services using a second shared user-assigned managed identity

C#

```
C#  
  
// Get the first user-assigned managed identity ID to connect to shared storage  
const clientIdStorage =  
Environment.GetEnvironmentVariable("Managed_Identity_Client_ID_Storage")  
;  
  
// First blob storage client that using a managed identity  
BlobServiceClient blobServiceClient = new BlobServiceClient(  
    new Uri("https://<receipt-storage-account>.blob.core.windows.net"),  
    new DefaultAzureCredential()  
{  
    ManagedIdentityClientId = clientIdStorage  
});  
  
// Second blob storage client that using a managed identity  
BlobServiceClient blobServiceClient2 = new BlobServiceClient(  
    new Uri("https://<contract-storage-account>.blob.core.windows.net"),  
    new DefaultAzureCredential()  
{  
    ManagedIdentityClientId = clientIdStorage  
});
```

```

});;

// Get the second user-assigned managed identity ID to connect to shared
// databases
var clientIDdatabases =
Environment.GetEnvironmentVariable("Managed_Identity_Client_ID_Databases"
");

// Create an Azure Cosmos DB client
CosmosClient client = new CosmosClient(
    accountEndpoint:
Environment.GetEnvironmentVariable("COSMOS_ENDPOINT",
EnvironmentVariableTarget.Process),
    new DefaultAzureCredential()
{
    ManagedIdentityClientId = clientIDdatabases
});

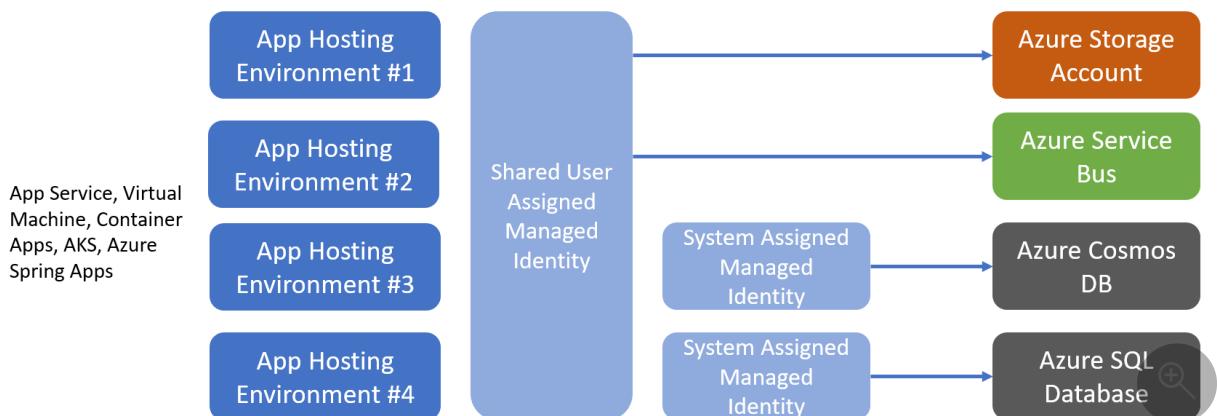
// Open a connection to Azure SQL using a managed identity
string ConnectionString1 = @"Server=<azure-sql-
hostname>.database.windows.net; User Id=ObjectIdOfManagedIdentity;
Authentication=Active Directory Default; Database=<database-name>";

using (SqlConnection conn = new SqlConnection(ConnectionString1))
{
    conn.Open();
}

```

You can also associate a user-assigned managed identity as well as a system-assigned managed identity to a resource simultaneously. This can be useful in scenarios where all of the apps require access to the same shared services, but one of the apps also has a very specific dependency on an additional service. Using a system-assigned identity also ensures that the identity tied to that specific app is deleted when the app is deleted, which can help keep your environment clean.

Connect multiple apps to Azure services using user-assigned and system-assigned identities



These types of scenarios are explored in more depth in the [identities best practice recommendations](#).

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections. You can read the following resources to explore the concepts discussed in this article in more depth:

- [Authorize access to blobs using Microsoft Entra ID](#)
- To learn more about .NET Core, see [Get started with .NET in 10 minutes ↗](#).

Managed identity best practice recommendations

Article • 10/23/2023

Managed identities for Azure resources is a feature of Microsoft Entra ID. Each of the [Azure services that support managed identities for Azure resources](#) are subject to their own timeline. Make sure you review the [availability](#) status of managed identities for your resource and [known issues](#) before you begin.

Choosing system or user-assigned managed identities

User-assigned managed identities are more efficient in a broader range of scenarios than system-assigned managed identities. See the table below for some scenarios and the recommendations for user-assigned or system-assigned.

User-assigned identities can be used by multiple resources, and their life cycles are decoupled from the resources' life cycles with which they're associated. [Read which resources support managed identities](#).

This life cycle allows you to separate your resource creation and identity administration responsibilities. User-assigned identities and their role assignments can be configured in advance of the resources that require them. Users who create the resources only require the access to assign a user-assigned identity, without the need to create new identities or role assignments.

As system-assigned identities are created and deleted along with the resource, role assignments can't be created in advance. This sequence can cause failures while deploying infrastructure if the user creating the resource doesn't also have access to create role assignments.

If your infrastructure requires that multiple resources require access to the same resources, a single user-assigned identity can be assigned to them. Administration overhead will be reduced, as there are fewer distinct identities and role assignments to manage.

If you require that each resource has its own identity, or have resources that require a unique set of permissions and want the identity to be deleted as the resource is deleted, then you should use a system-assigned identity.

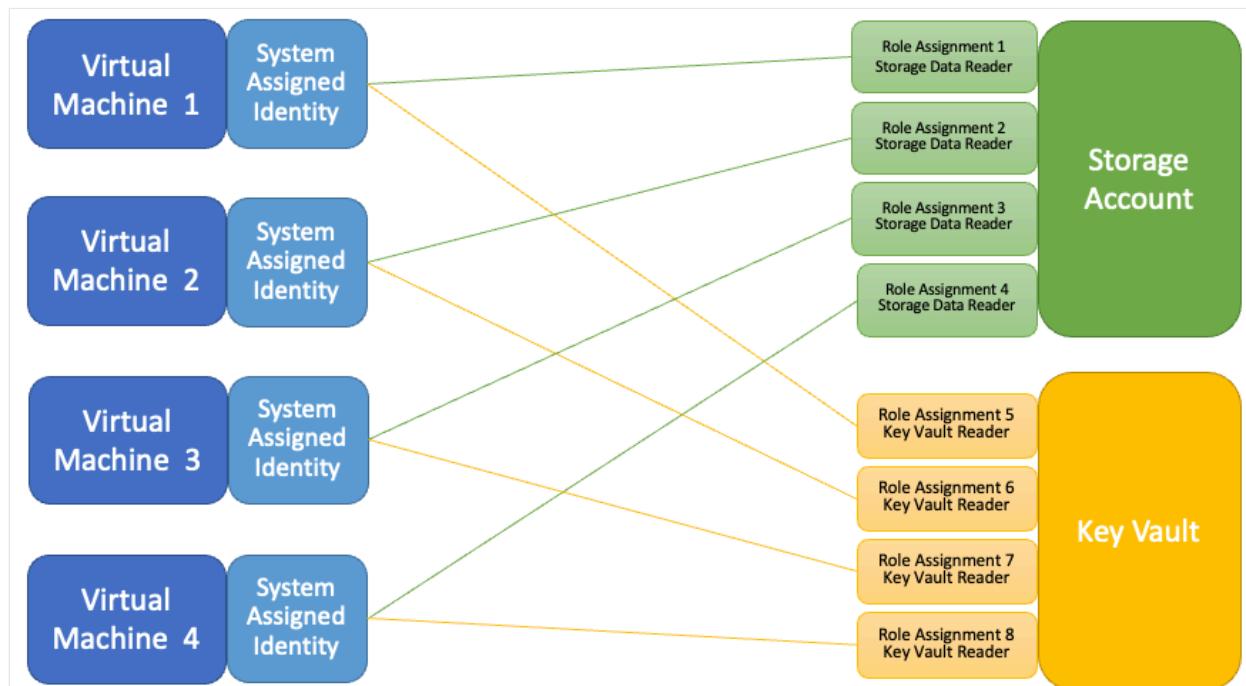
Scenario	Recommendation	Notes
Rapid creation of resources (for example, ephemeral computing) with managed identities	User-assigned identity	<p>If you attempt to create multiple managed identities in a short space of time – for example, deploying multiple virtual machines each with their own system-assigned identity - you may exceed the rate limit for Microsoft Entra object creations, and the request will fail with an HTTP 429 error.</p> <p>If resources are being created or deleted rapidly, you may also exceed the limit on the number of resources in Microsoft Entra ID if using system-assigned identities. While a deleted system-assigned identity is no longer accessible by any resource, it will count towards your limit until fully purged after 30 days.</p> <p>Deploying the resources associated with a single user-assigned identity will require the creation of only one Service Principal in Microsoft Entra ID, avoiding the rate limit. Using a single identity that is created in advance will also reduce the risk of replication delays that could occur if multiple resources are created each with their own identity.</p> <p>Read more about the Azure subscription service limits.</p>
Replicated resources/applications	User-assigned identity	<p>Resources that carry out the same task – for example, duplicated web servers or identical functionality running in an app service and in an application on a virtual machine – typically require the same permissions.</p> <p>By using the same user-assigned identity, fewer role assignments are required which reduces the management overhead. The resources don't have to be of the same type.</p>
Compliance	User-assigned identity	<p>If your organization requires that all identity creation must go through an approval process, using a single user-assigned identity across multiple resources will</p>

Scenario	Recommendation	Notes
		require fewer approvals than system-assigned Identities, which are created as new resources are created.
Access required before a resource is deployed	User-assigned identity	Some resources may require access to certain Azure resources as part of their deployment. In this case, a system-assigned identity may not be created in time so a pre-existing user-assigned identity should be used.
Audit Logging	System-assigned identity	If you need to log which specific resource carried out an action, rather than which identity, use a system-assigned identity.
Permissions Lifecycle Management	System-assigned identity	If you require that the permissions for a resource be removed along with the resource, use a system-assigned identity.

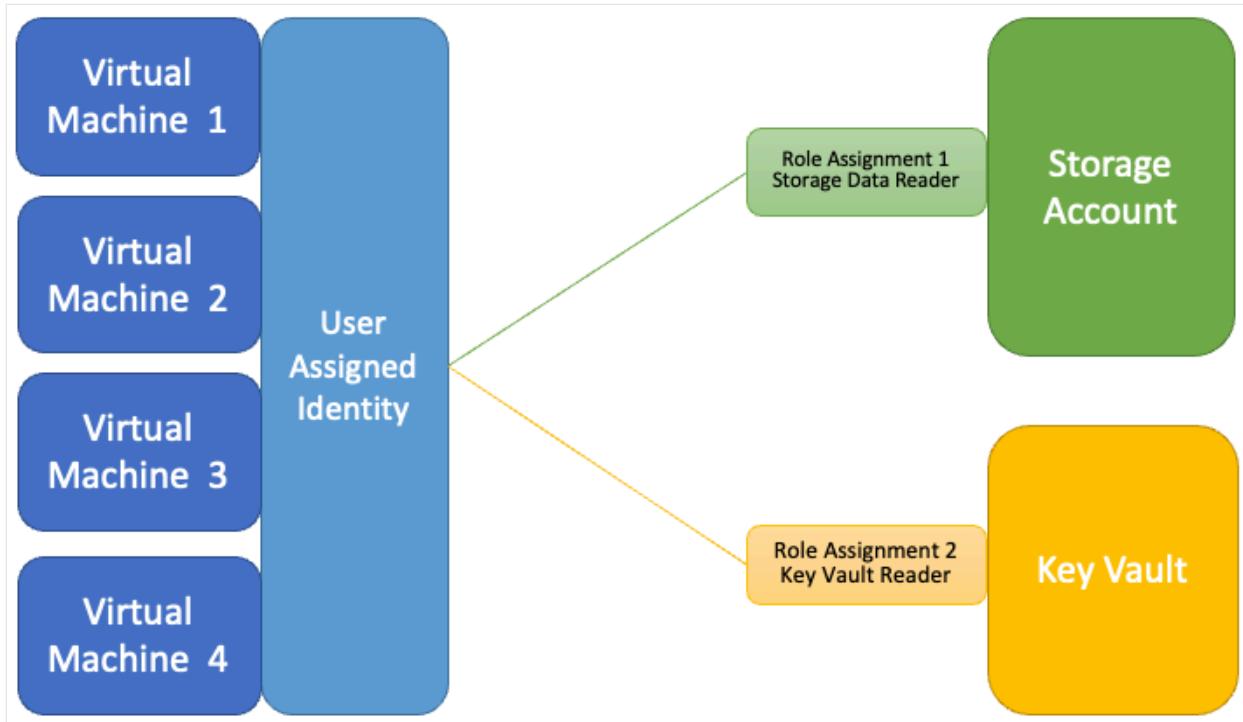
Using user-assigned identities to reduce administration

The diagrams demonstrate the difference between system-assigned and user-assigned identities, when used to allow several virtual machines to access two storage accounts.

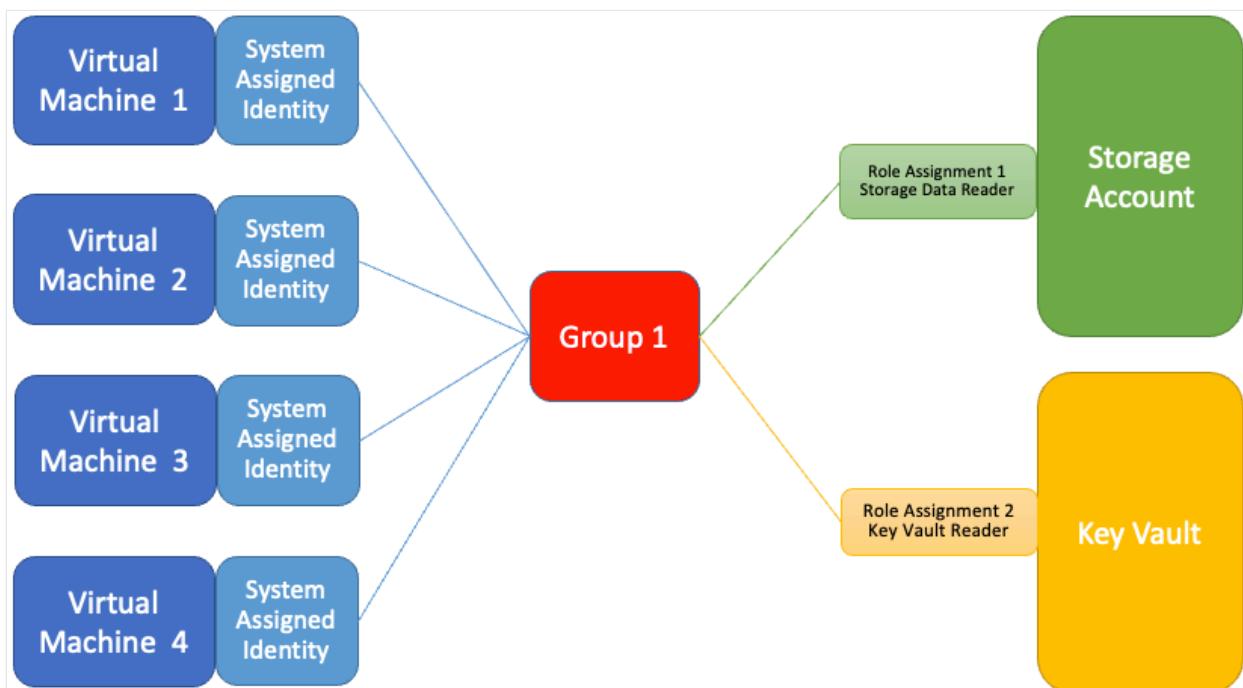
The diagram shows four virtual machines with system-assigned identities. Each virtual machine has the same role assignments that grants them access to two storage accounts.



When a user-assigned identity is associated with the four virtual machines, only two role assignments are required, compared to eight with system-assigned identities. If the virtual machines' identity requires more role assignments, they'll be granted to all the resources associated with this identity.



Security groups can also be used to reduce the number of role assignments that are required. This diagram shows four virtual machines with system-assigned identities, which have been added to a security group, with the role assignments added to the group instead of the system-assigned identities. While the result is similar, this configuration doesn't offer the same Resource Manager template capabilities as user-assigned identities.

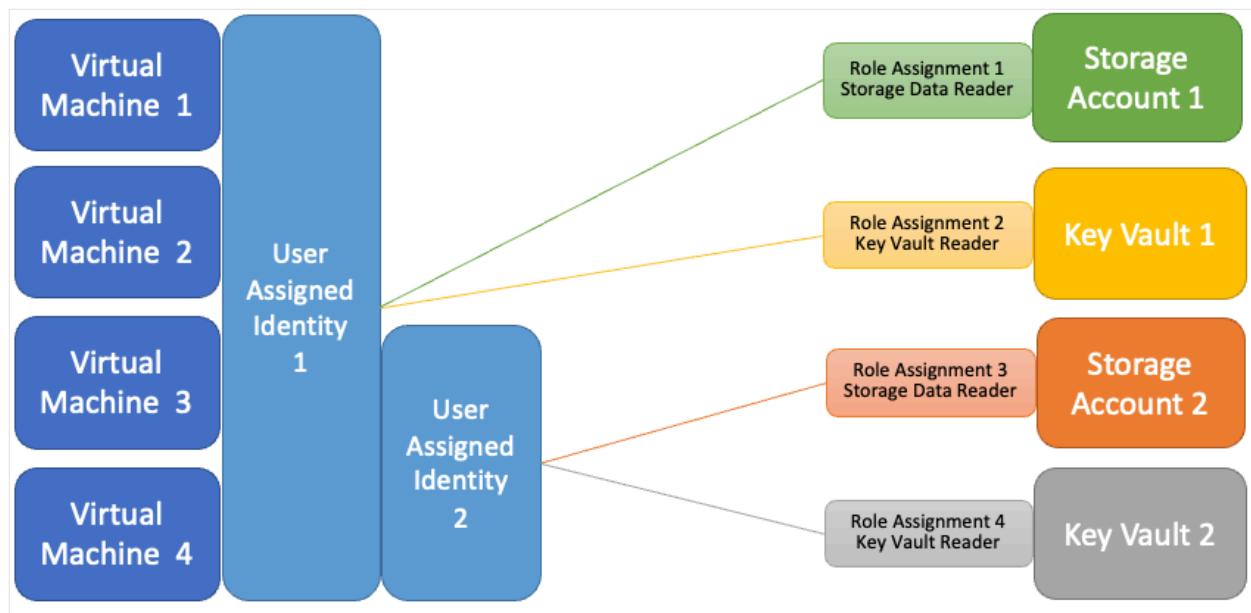


Multiple managed identities

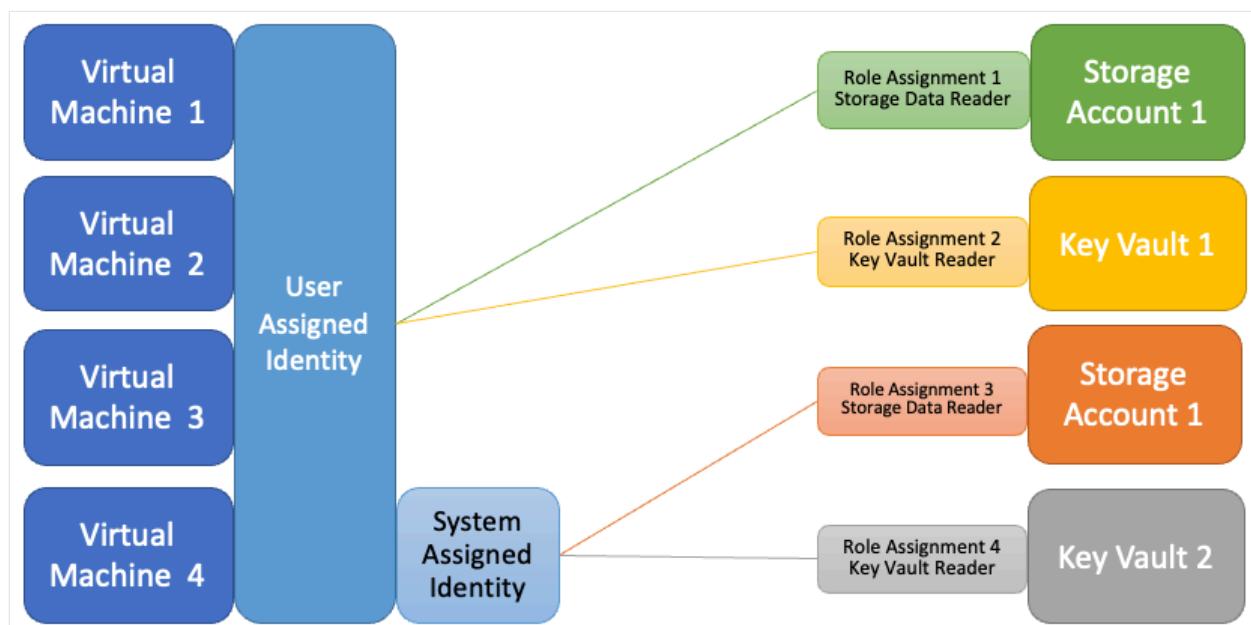
Resources that support managed identities can have both a system-assigned identity and one or more user-assigned identities.

This model provides the flexibility to both use a shared user-assigned identity and apply granular permissions when needed.

In the example below, "Virtual Machine 3" and "Virtual Machine 4" can access both storage accounts and key vaults, depending on which user-assigned identity they use while authenticating.



In the example below, "Virtual Machine 4" has both a user-assigned identity, giving it access to both storage accounts and key vaults, depending on which identity is used while authenticating. The role assignments for the system-assigned identity are specific to that virtual machine.



Limits

View the limits for [managed identities](#) and for [custom roles and role assignments](#).

Follow the principle of least privilege when granting access

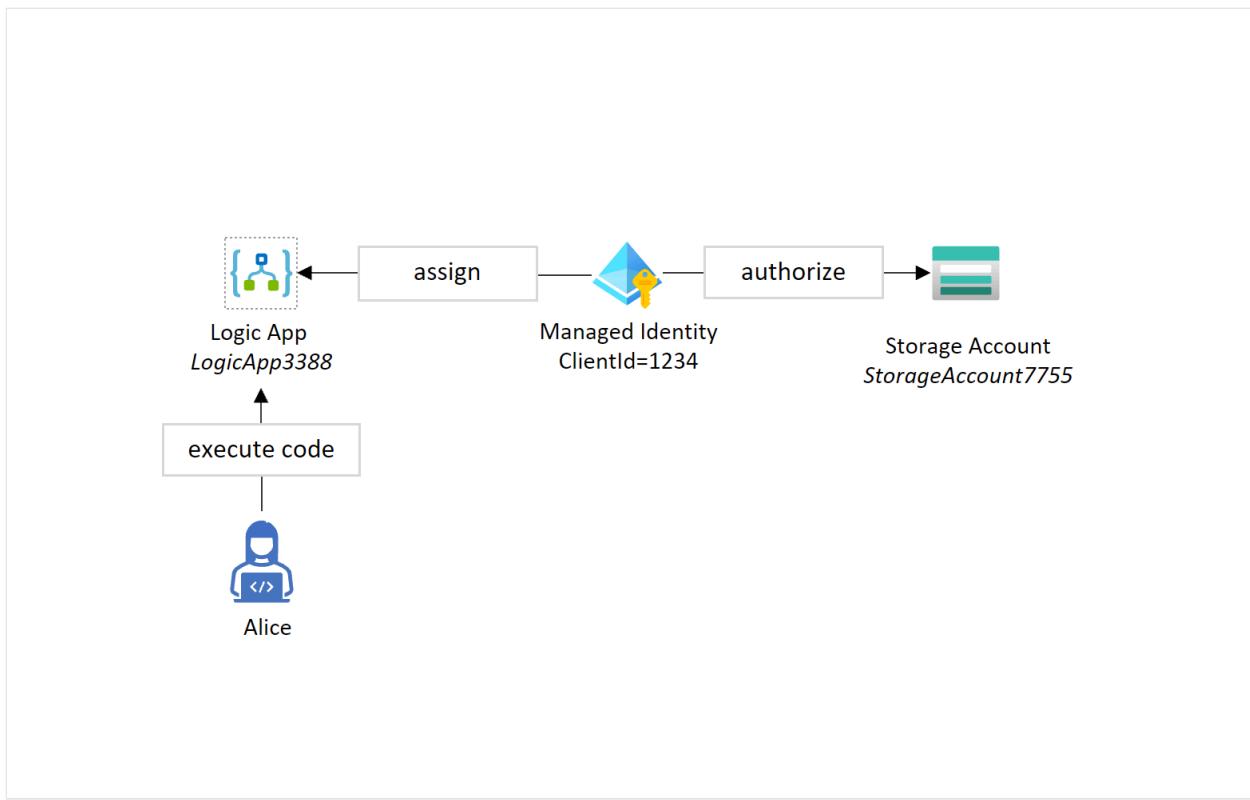
When granting any identity, including a managed identity, permissions to access services, always grant the least permissions needed to perform the desired actions. For example, if a managed identity is used to read data from a storage account, there is no need to allow that identity permissions to also write data to the storage account. Granting extra permissions, for example, making the managed identity a contributor on an Azure subscription when it's not needed, increases the security blast radius associated with the identity. One must always minimize the security blast radius so that compromising that identity causes minimum damage.

Consider the effect of assigning managed identities to Azure resources and/or granting assign permissions to a user

It is important to note that when an Azure resource, such as an Azure Logic App, an Azure function, or a Virtual Machine, etc. is assigned a managed identity, all the permissions granted to the managed identity are now available to the Azure resource. This is important because if a user has access to install or execute code on this resource, then the user has access to all the identities assigned/associated to the Azure resource. The purpose of managed identity is to give code running on an Azure resource access to other resources, without developers needing to handle or put credentials directly into code to get that access.

For example, if a managed Identity (ClientId = 1234) has been granted read/write access to **StorageAccount7755** and has been assigned to **LogicApp3388**, then Alice, who does not have direct access to the storage account but has permission to execute code within **LogicApp3388** can also read/write data to/from **StorageAccount7755** by executing the code that uses the managed identity.

Similarly, if Alice has permissions to assign the managed identity herself, she can assign it to a different Azure resource and have access to all the permissions available to the managed identity.



In general, when granting a user administrative access to a resource that can execute code (such as a Logic App) and has a managed identity, consider if the role being assigned to the user can install or run code on the resource, and if yes only assign that role if the user really needs it.

Maintenance

System-assigned identities are automatically deleted when the resource is deleted, while the lifecycle of a user-assigned identity is independent of any resources with which it's associated.

You'll need to manually delete a user-assigned identity when it's no longer required, even if no resources are associated with it.

Role assignments aren't automatically deleted when either system-assigned or user-assigned managed identities are deleted. These role assignments should be manually deleted so the limit of role assignments per subscription isn't exceeded.

Role assignments that are associated with deleted managed identities will be displayed with "Identity not found" when viewed in the portal. [Read more](#).

Storage Blob Data Reader		
<input type="checkbox"/>		Identity not found. ⓘ Unable to find identity.
		Unknown

Role assignments which are no longer associated with a user or service principal will appear with an `ObjectType` value of `Unknown`. In order to remove them, you can pipe several Azure PowerShell commands together to first get all the role assignments, filter to only those with an `ObjectType` value of `Unknown` and then remove those role assignments from Azure.

```
Azure PowerShell
```

```
Get-AzRoleAssignment | Where-Object {$_ ObjectType -eq "Unknown"} | Remove-AzRoleAssignment
```

Limitation of using managed identities for authorization

Using Microsoft Entra ID **groups** for granting access to services is a great way to simplify the authorization process. The idea is simple – grant permissions to a group and add identities to the group so that they inherit the same permissions. This is a well-established pattern from various on-premises systems and works well when the identities represent users. Another option to control authorization in Microsoft Entra ID is by using [App Roles](#), which allows you to declare **roles** that are specific to an app (rather than groups, which are a global concept in the directory). You can then [assign app roles to managed identities](#) (as well as users or groups).

In both cases, for non-human identities such as Microsoft Entra Applications and Managed identities, the exact mechanism of how this authorization information is presented to the application is not ideally suited today. Today's implementation with Microsoft Entra ID and Azure Role Based Access Control (Azure RBAC) uses access tokens issued by Microsoft Entra ID for authentication of each identity. If the identity is added to a group or role, this is expressed as claims in the access token issued by Microsoft Entra ID. Azure RBAC uses these claims to further evaluate the authorization rules for allowing or denying access.

Given that the identity's groups and roles are claims in the access token, any authorization changes do not take effect until the token is refreshed. For a human user that's typically not a problem, because a user can acquire a new access token by logging out and in again (or waiting for the token lifetime to expire, which is 1 hour by default). Managed identity tokens on the other hand are cached by the underlying Azure infrastructure for performance and resiliency purposes: the back-end services for managed identities maintain a cache per resource URI for around 24 hours. This means that it can take several hours for changes to a managed identity's group or role membership to take effect. Today, it is not possible to force a managed identity's token

to be refreshed before its expiry. If you change a managed identity's group or role membership to add or remove permissions, you may therefore need to wait several hours for the Azure resource using the identity to have the correct access.

If this delay is not acceptable for your requirements, consider alternatives to using groups or roles in the token. To ensure that changes to permissions for managed identities take effect quickly, we recommend that you group Azure resources using a [user-assigned managed identity](#) with permissions applied directly to the identity, instead of adding to or removing managed identities from a Microsoft Entra group that has permissions. A user-assigned managed identity can be used like a group because it can be assigned to one or more Azure resources to use it. The assignment operation can be controlled using the [Managed identity contributor](#) and [Managed identity operator](#) role.

Migrate a .NET application to use passwordless connections with Azure SQL Database

Article • 09/29/2023

Applies to:  Azure SQL Database

Application requests to Azure SQL Database must be authenticated. Although there are multiple options for authenticating to Azure SQL Database, you should prioritize passwordless connections in your applications when possible. Traditional authentication methods that use passwords or secret keys create security risks and complications. Visit the [passwordless connections for Azure services](#) hub to learn more about the advantages of moving to passwordless connections. The following tutorial explains how to migrate an existing application to connect to Azure SQL Database to use passwordless connections instead of a username and password solution.

Configure the Azure SQL Database

Passwordless connections use Microsoft Entra authentication to connect to Azure services, including Azure SQL Database. Microsoft Entra authentication, you can manage identities in a central location to simplify permission management. Learn more about configuring Microsoft Entra authentication for your Azure SQL Database:

- [Microsoft Entra authentication overview](#)
- [Configure Microsoft Entra auth](#)

For this migration guide, ensure you have a Microsoft Entra admin assigned to your Azure SQL Database.

1. Navigate to the **Microsoft Entra** page of your logical server.
2. Select **Set admin** to open the **Microsoft Entra ID** flyout menu.
3. In the **Microsoft Entra ID** flyout menu, search for the user you want to assign as admin.
4. Select the user and choose **Select**.

The screenshot shows the Microsoft Entra admin center interface. At the top, there's a search bar, 'Set admin', 'Remove admin', and 'Save' buttons. On the left, a sidebar lists 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', and 'Settings'. Under 'Settings', 'Microsoft Entra ID' is selected and highlighted with a red box. Other options include 'SQL databases' and 'SQL elastic pools'. On the right, the main content area is titled 'Microsoft Entra authentication only'. It explains that only Microsoft Entra ID will be used for authentication, and SQL authentication will be disabled. A checkbox for 'Support only Microsoft Entra authentication for this server' is checked. A magnifying glass icon is in the bottom right corner.

Configure your local development environment

Passwordless connections can be configured to work for both local and Azure hosted environments. In this section, you'll apply configurations to allow individual users to authenticate to Azure SQL Database for local development.

Sign-in to Azure

For local development, make sure you're signed-in with the same Azure AD account you want to use to access Azure SQL Database. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

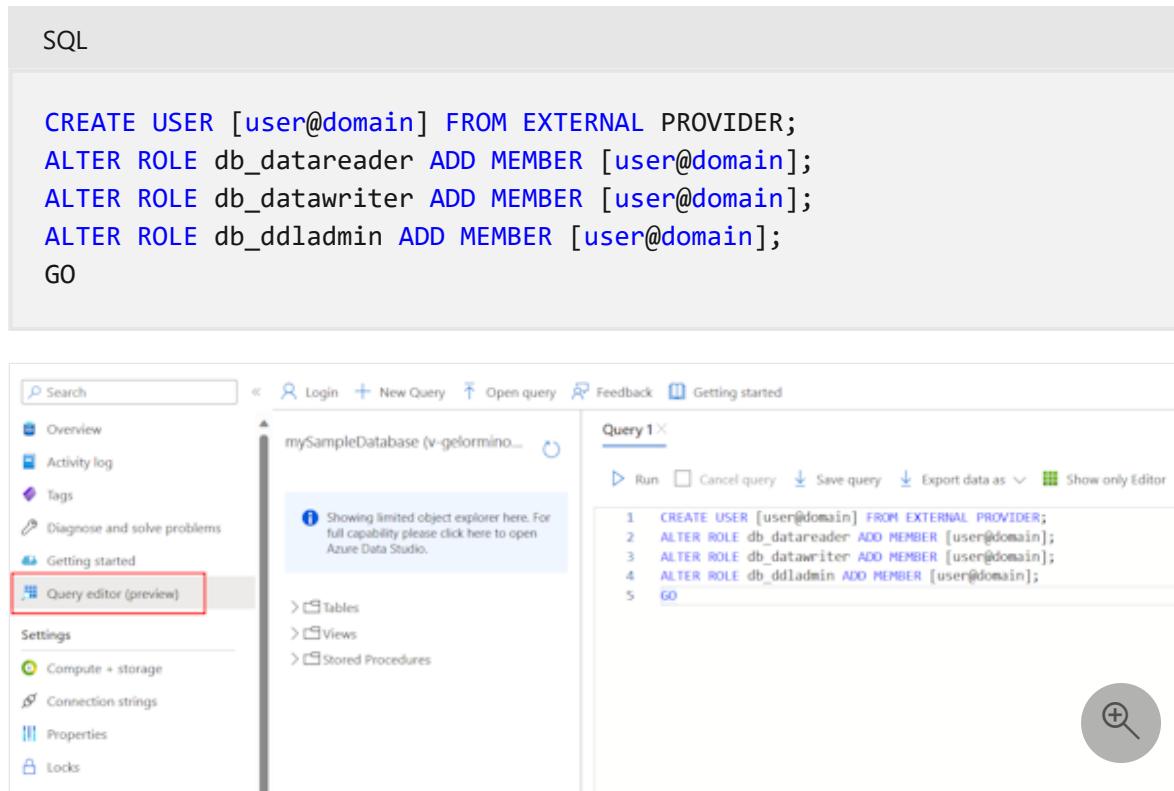
The screenshot shows the Azure CLI interface. A blue header bar says 'Azure CLI'. Below it, a grey bar contains the text 'Sign-in to Azure through the Azure CLI using the following command:'. A light grey terminal window shows the command 'az login' in blue text. The background of the entire interface is white.

Create a database user and assign roles

Create a user in Azure SQL Database. The user should correspond to the Azure account you used to sign-in locally via development tools like Visual Studio or IntelliJ.

1. In the [Azure portal](#), browse to your SQL database and select **Query editor (preview)**.
2. Select **Continue as <your-username>** on the right side of the screen to sign into the database using your account.

3. On the query editor view, run the following T-SQL commands:



```
CREATE USER [user@domain] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [user@domain];
ALTER ROLE db_datawriter ADD MEMBER [user@domain];
ALTER ROLE db_ddladmin ADD MEMBER [user@domain];
GO
```

Running these commands assigns the [SQL DB Contributor](#) role to the account specified. This role allows the identity to read, write, and modify the data and schema of your database. For more information about the roles assigned, see [Fixed-database roles](#).

Update the local connection configuration

Existing application code that connects to Azure SQL Database using the `Microsoft.Data.SqlClient` library or Entity Framework Core will continue to work with passwordless connections. However, you must update your database connection string to use the passwordless format. For example, the following code works with both SQL authentication and passwordless connections:

```
C#  
  
string connectionString =
app.Configuration.GetConnectionString("AZURE_SQL_CONNECTIONSTRING")!;  
  
using var conn = new SqlConnection(connectionString);
conn.Open();  
  
var command = new SqlCommand("SELECT * FROM Persons", conn);
using SqlDataReader reader = command.ExecuteReader();
```

To update the referenced connection string (`AZURE_SQL_CONNECTIONSTRING`) to use the passwordless connection string format:

1. Locate your connection string. For local development with .NET applications, this is usually stored in one of the following locations:
 - The `appsettings.json` configuration file for your project.
 - The `launchsettings.json` configuration file for Visual Studio projects.
 - Local system or container environment variables.
2. Replace the connection string value with the following passwordless format.

Update the `<database-server-name>` and `<database-name>` placeholders with your own values:

JSON

```
Server=tcp:<database-server-name>.database.windows.net,1433;Initial  
Catalog=<database-name>;  
Encrypt=True;TrustServerCertificate=False;Connection  
Timeout=30;Authentication="Active Directory Default";
```

Test the app

Run your app locally and verify that the connections to Azure SQL Database are working as expected. Keep in mind that it may take several minutes for changes to Azure users and roles to propagate through your Azure environment. Your application is now configured to run locally without developers having to manage secrets in the application itself.

Configure the Azure hosting environment

Once your app is configured to use passwordless connections locally, the same code can authenticate to Azure SQL Database after it's deployed to Azure. The sections that follow explain how to configure a deployed application to connect to Azure SQL Database using a [managed identity](#). Managed identities provide an automatically managed identity in Microsoft Entra ID ([formerly Azure Active Directory](#)) for applications to use when connecting to resources that support Microsoft Entra authentication. Learn more about managed identities:

- [Passwordless overview](#)
- [Managed identity best practices](#)

Create the managed identity

Create a user-assigned managed identity using the Azure portal or the Azure CLI. Your application uses the identity to authenticate to other services.

Azure portal

1. At the top of the Azure portal, search for *Managed identities*. Select the **Managed Identities** result.
2. Select **+ Create** at the top of the **Managed Identities** overview page.
3. On the **Basics** tab, enter the following values:
 - **Subscription:** Select your desired subscription.
 - **Resource group:** Select your desired resource group.
 - **Region:** Select a region near your location.
 - **Name:** Enter a recognizable name for your identity, such as *MigrationIdentity*.
4. Select **Review + create** at the bottom of the page.
5. When the validation checks finish, select **Create**. Azure creates a new user-assigned identity.

After the resource is created, select **Go to resource** to view the details of the managed identity.

Create User Assigned Managed Identity

Basics Tags Review + create

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Test Subscription

Resource group * ⓘ passwordlesstesting
Create new

Instance details

Region * ⓘ South Central US

Name * ⓘ MigrationIdentity

Review + create < Previous Next : Tags > 

Associate the managed identity with your web app

Configure your web app to use the user-assigned managed identity you created.

Azure portal

Complete the following steps in the Azure portal to associate the user-assigned managed identity with your app. These same steps apply to the following Azure services:

- Azure Spring Apps
- Azure Container Apps
- Azure virtual machines
- Azure Kubernetes Service
- Navigate to the overview page of your web app.

1. Select **Identity** from the left navigation.

2. On the **Identity** page, switch to the **User assigned** tab.
3. Select **+ Add** to open the **Add user assigned managed identity** flyout.
4. Select the subscription you used previously to create the identity.
5. Search for the **MigrationIdentity** by name and select it from the search results.
6. Select **Add** to associate the identity with your app.

The screenshot shows the Azure portal's Identity blade. The 'User assigned' tab is selected. A red box highlights the '+ Add' button. The 'Selected identities' section shows 'passwordlessuser' selected. A search bar and a magnifying glass icon are also visible.

Create a database user for the identity and assign roles

Create a SQL database user that maps back to the user-assigned managed identity. Assign the necessary SQL roles to the user to allow your app to read, write, and modify the data and schema of your database.

1. In the Azure portal, browse to your SQL database and select **Query editor (preview)**.
2. Select **Continue as <username>** on the right side of the screen to sign into the database using your account.
3. On the query editor view, run the following T-SQL commands:

SQL

```
CREATE USER [user-assigned-identity-name] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [user-assigned-identity-name];
ALTER ROLE db_datawriter ADD MEMBER [user-assigned-identity-name];
ALTER ROLE db_ddladmin ADD MEMBER [user-assigned-identity-name];
GO
```

```
1 CREATE USER [user-assigned-identity-name] FROM EXTERNAL PROVIDER;
2 ALTER ROLE db_datareader ADD MEMBER [user-assigned-identity-name];
3 ALTER ROLE db_datawriter ADD MEMBER [user-assigned-identity-name];
4 ALTER ROLE db_ddladmin ADD MEMBER [user-assigned-identity-name];
5 GO
```

Running these commands assigns the [SQL DB Contributor](#) role to the user-assigned managed identity. This role allows the identity to read, write, and modify the data and schema of your database.

ⓘ Important

Use caution when assigning database user roles in enterprise production environments. In those scenarios, the app shouldn't perform all operations using a single, elevated identity. Try to implement the principle of least privilege by configuring multiple identities with specific permissions for specific tasks.

You can read more about configuring database roles and security on the following resources:

- [Tutorial: Secure a database in Azure SQL Database](#)
- [Authorize database access to SQL Database](#)

Update the connection string

Update your Azure app configuration to use the passwordless connection string format. Connection strings are generally stored as environment variables in your app hosting environment. The following instructions focus on App Service, but other Azure hosting services provide similar configurations.

1. Navigate to the configuration page of your App Service instance and locate the Azure SQL Database connection string.
2. Select the edit icon and update the connection string value to match following format. Change the <database-server-name> and <database-name> placeholders with the values of your own service.

JSON

```
Server=tcp:<database-server-name>.database.windows.net,1433;Initial  
Catalog=<database-name>;  
Encrypt=True;TrustServerCertificate=False;Connection  
Timeout=30;Authentication="Active Directory Default";
```

3. Save your changes and restart the application if it does not do so automatically.

Test the application

Test your app to make sure everything is still working. It may take a few minutes for all of the changes to propagate through your Azure environment.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Passwordless overview](#)
- [Managed identity best practices](#)
- [Tutorial: Secure a database in Azure SQL Database](#)
- [Authorize database access to SQL Database](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Migrate a Java application to use passwordless connections with Azure SQL Database

Article • 10/19/2023

This article explains how to migrate from traditional authentication methods to more secure, passwordless connections with Azure SQL Database.

Application requests to Azure SQL Database must be authenticated. Azure SQL Database provides several different ways for apps to connect securely. One of the ways is to use passwords. However, you should prioritize passwordless connections in your applications when possible.

Compare authentication options

When the application authenticates with Azure SQL Database, it provides a username and password pair to connect to the database. Depending on where the identities are stored, there are two types of authentication: Microsoft Entra authentication and Azure SQL Database authentication.

Microsoft Entra authentication

Microsoft Entra authentication is a mechanism for connecting to Azure SQL Database using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

Using Microsoft Entra ID for authentication provides the following benefits:

- Authentication of users across Azure Services in a uniform way.
- Management of password policies and password rotation in a single place.
- Multiple forms of authentication supported by Microsoft Entra ID, which can eliminate the need to store passwords.
- Customers can manage database permissions using external (Microsoft Entra ID) groups.
- Microsoft Entra authentication uses Azure SQL database users to authenticate identities at the database level.
- Support of token-based authentication for applications connecting to Azure SQL Database.

Azure SQL Database authentication

You can create accounts in Azure SQL Database. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `sys.database_principals` table. Because these passwords are stored in Azure SQL Database, you need to manage the rotation of the passwords by yourself.

Although it's possible to connect to Azure SQL Database with passwords, you should use them with caution. You must be diligent to never expose the passwords in an unsecure location. Anyone who gains access to the passwords is able to authenticate. For example, there's a risk that a malicious user can access the application if a connection string is accidentally checked into source control, sent through an unsecure email, pasted into the wrong chat, or viewed by someone who shouldn't have permission. Instead, consider updating your application to use passwordless connections.

Introducing passwordless connections

With a passwordless connection, you can connect to Azure services without storing any credentials in the application code, its configuration files, or in environment variables.

Many Azure services support passwordless connections, for example via Azure Managed Identity. These techniques provide robust security features that you can implement using `DefaultAzureCredential` from the Azure Identity client libraries. In this tutorial, you'll learn how to update an existing application to use `DefaultAzureCredential` instead of alternatives such as connection strings.

`DefaultAzureCredential` supports multiple authentication methods and automatically determines which should be used at runtime. This approach enables your app to use different authentication methods in different environments (local dev vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` searches for credentials can be found in the [Azure Identity library overview](#). For example, when working locally, `DefaultAzureCredential` will generally authenticate using the account the developer used to sign in to Visual Studio. When the app is deployed to Azure, `DefaultAzureCredential` will automatically switch to use a [managed identity](#). No code changes are required for this transition.

To ensure that connections are passwordless, you must take into consideration both local development and the production environment. If a connection string is required in either place, then the application isn't passwordless.

In your local development environment, you can authenticate with Azure CLI, Azure PowerShell, Visual Studio, or Azure plugins for Visual Studio Code or IntelliJ. In this case, you can use that credential in your application instead of configuring properties.

When you deploy applications to an Azure hosting environment, such as a virtual machine, you can assign managed identity in that environment. Then, you won't need to provide credentials to connect to Azure services.

 **Note**

A managed identity provides a security identity to represent an app or service. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. You can read more about managed identities in the [overview](#) documentation.

 **Note**

Since the JDBC driver for Azure SQL Database doesn't support passwordless connections from local environments yet, this article will focus only on applications deployed to Azure hosting environments and how to migrate them to use passwordless connections.

Migrate an existing application to use passwordless connections

The following steps explain how to migrate an existing application to use passwordless connections instead of a password-based solution.

0) Prepare the working environment

First, use the following command to set up some environment variables.

Bash

```
export AZ_RESOURCE_GROUP=<YOUR_RESOURCE_GROUP>
export AZ_DATABASE_SERVER_NAME=<YOUR_DATABASE_SERVER_NAME>
export AZ_DATABASE_NAME=demo
export CURRENT_USERNAME=$(az ad signed-in-user show --query
userPrincipalName --output tsv)
```

```
export CURRENT_USER_OBJECTID=$(az ad signed-in-user show --query id --output tsv)
```

Replace the placeholders with the following values, which are used throughout this article:

- <YOUR_RESOURCE_GROUP>: The name of the resource group your resources are in.
- <YOUR_DATABASE_SERVER_NAME>: The name of your Azure SQL Database server. It should be unique across Azure.

1) Configure Azure SQL Database

1.1) Enable Microsoft Entra ID-based authentication

To use Microsoft Entra ID access with Azure SQL Database, you should set the Microsoft Entra admin user first. Only a Microsoft Entra Admin user can create/enable users for Microsoft Entra ID-based authentication.

If you're using Azure CLI, run the following command to make sure it has sufficient permission:

Bash

```
az login --scope https://graph.microsoft.com/.default
```

Then, run following command to set the Microsoft Entra admin:

Azure CLI

```
az sql server ad-admin create \
--resource-group $AZ_RESOURCE_GROUP \
--server $AZ_DATABASE_SERVER_NAME \
--display-name $CURRENT_USERNAME \
--object-id $CURRENT_USER_OBJECTID
```

This command will set the Microsoft Entra admin to the current signed-in user.

ⓘ Note

You can only create one Microsoft Entra admin per Azure SQL Database server. Selection of another one will overwrite the existing Microsoft Entra admin configured for the server.

2) Migrate the app code to use passwordless connections

Next, use the following steps to update your code to use passwordless connections. Although conceptually similar, each language uses different implementation details.

Java

1. Inside your project, add the following reference to the `azure-identity` package. This library contains all of the entities necessary to implement passwordless connections.

XML

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-identity</artifactId>
    <version>1.5.4</version>
</dependency>
```

2. Enable the Microsoft Entra managed identity authentication in the JDBC URL. Identify the locations in your code that currently create a `java.sql.Connection` to connect to Azure SQL Database. Update your code to match the following example:

Java

```
String url =
"jdbc:sqlserver://$AZ_DATABASE_SERVER_NAME.database.windows.net:143
3;databaseName=$AZ_DATABASE_NAME;authentication=ActiveDirectoryMSI;
"
Connection con = DriverManager.getConnection(url);
```

3. Replace the two `$AZ_DATABASE_SERVER_NAME` variables and one `$AZ_DATABASE_NAME` variable with the values that you configured at the beginning of this article.
4. Remove the `user` and `password` from the JDBC URL.

3) Configure the Azure hosting environment

After your application is configured to use passwordless connections, the same code can authenticate to Azure services after it's deployed to Azure. For example, an application

deployed to an Azure App Service instance that has a managed identity assigned can connect to Azure Storage.

In this section, you'll execute two steps to enable your application to run in an Azure hosting environment in a passwordless way:

- Assign the managed identity for your Azure hosting environment.
- Assign roles to the managed identity.

ⓘ Note

Azure also provides **Service Connector**, which can help you connect your hosting service with SQL server. With Service Connector to configure your hosting environment, you can omit the step of assigning roles to your managed identity because Service Connector will do it for you. The following section describes how to configure your Azure hosting environment in two ways: one via Service Connector and the other by configuring each hosting environment directly.

ⓘ Important

Service Connector's commands require [Azure CLI](#) 2.41.0 or higher.

Assign the managed identity using the Azure portal

The following steps show you how to assign a system-assigned managed identity for various web hosting services. The managed identity can securely connect to other Azure services using the app configurations you set up previously.

App Service

1. On the main overview page of your Azure App Service instance, select **Identity** from the navigation pane.
2. On the **System assigned** tab, make sure to set the **Status** field to **on**. A system assigned identity is managed by Azure internally and handles administrative tasks for you. The details and IDs of the identity are never exposed in your code.

The screenshot shows the Azure portal interface for managing the identity of an App Service. The left sidebar lists various settings like Overview, Activity log, and Deployment. The main panel shows the 'Identity' section, which is highlighted with a red box. The 'System assigned' tab is selected, and its status is 'On'. The 'Object (principal) ID' is displayed as 27a5a14c-9a0f-4f50-a82b-f022f74a8764. A note at the bottom states: 'This resource is registered with Azure Active Directory. The managed identity can be configured to allow access to'.

You can also assign managed identity on an Azure hosting environment using the Azure CLI.

App Service

You can assign a managed identity to an Azure App Service instance with the [az webapp identity assign](#) command, as shown in the following example:

Azure CLI

```
export AZ_MI_OBJECT_ID=$(az webapp identity assign \
    --resource-group $AZ_RESOURCE_GROUP \
    --name <service-instance-name> \
    --query principalId \
    --output tsv)
```

Assign roles to the managed identity

Next, grant permissions to the managed identity you created to access your SQL database.

If you connected your services using Service Connector, the previous step's commands already assigned the role, so you can skip this step.

Test the app

After making these code changes, you can build and redeploy the application. Then, browse to your hosted application in the browser. Your app should be able to connect to the Azure SQL database successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Authorize access to blob data with managed identities for Azure resources](#).
- [Authorize access to blobs using Microsoft Entra ID](#)

Migrate a Node.js application to use passwordless connections with Azure SQL Database

Article • 09/29/2023

Applies to:  [Azure SQL Database](#)

Application requests to Azure SQL Database must be authenticated. Although there are multiple options for authenticating to Azure SQL Database, you should prioritize passwordless connections in your applications when possible. Traditional authentication methods that use passwords or secret keys create security risks and complications. Visit the [passwordless connections for Azure services](#) hub to learn more about the advantages of moving to passwordless connections.

The following tutorial explains how to migrate an existing Node.js application to connect to Azure SQL Database to use passwordless connections instead of a username and password solution.

Configure the Azure SQL Database

Passwordless connections use Microsoft Entra authentication to connect to Azure services, including Azure SQL Database. Microsoft Entra authentication, you can manage identities in a central location to simplify permission management. Learn more about configuring Microsoft Entra authentication for your Azure SQL Database:

- [Microsoft Entra authentication overview](#)
- [Configure Microsoft Entra auth](#)

For this migration guide, ensure you have a Microsoft Entra admin assigned to your Azure SQL Database.

1. Navigate to the **Microsoft Entra** page of your logical server.
2. Select **Set admin** to open the **Microsoft Entra ID** flyout menu.
3. In the **Microsoft Entra ID** flyout menu, search for the user you want to assign as admin.
4. Select the user and choose **Select**.

The screenshot shows the Microsoft Entra admin center interface. At the top, there's a search bar, 'Set admin', 'Remove admin', and 'Save' buttons. On the left, a sidebar lists 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', and 'Settings'. Under 'Settings', 'Microsoft Entra ID' is selected and highlighted with a red box. Other options include 'SQL databases' and 'SQL elastic pools'. On the right, there's a large circular button with a magnifying glass icon.

Configure your local development environment

Passwordless connections can be configured to work for both local and Azure-hosted environments. In this section, you apply configurations to allow individual users to authenticate to Azure SQL Database for local development.

Sign-in to Azure

For local development, make sure you're signed-in with the same Azure AD account you want to use to access Azure SQL Database. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

The screenshot shows the Azure CLI interface. A sidebar on the left has 'Azure CLI' selected. The main area contains the text 'Sign-in to Azure through the Azure CLI using the following command:' followed by a code editor window showing the command 'az login'.

Create a database user and assign roles

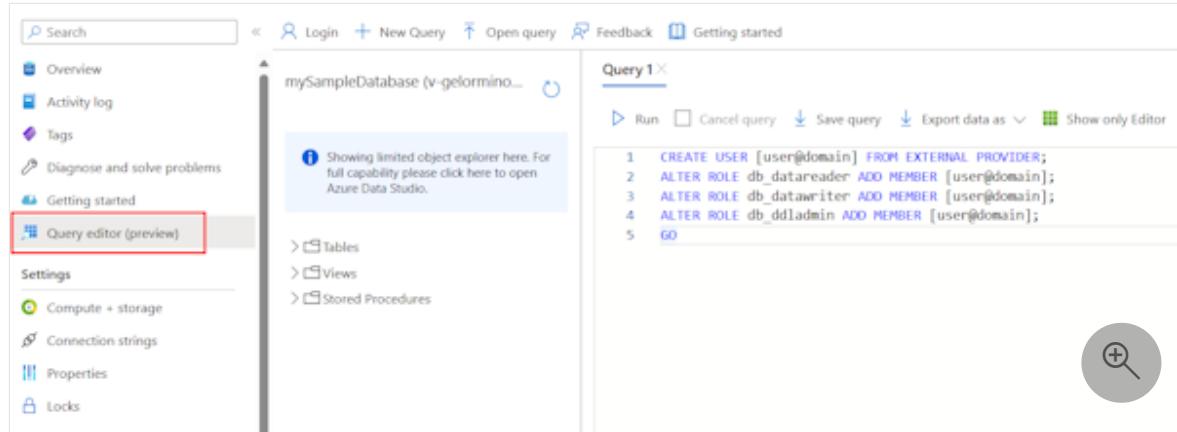
Create a user in Azure SQL Database. The user should correspond to the Azure account you used to sign-in locally in the [Sign-in to Azure](#) section.

1. In the [Azure portal](#), browse to your SQL database and select **Query editor (preview)**.
2. Select **Continue as <your-username>** on the right side of the screen to sign into the database using your account.

3. On the query editor view, run the following T-SQL commands:

```
SQL

CREATE USER [user@domain] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [user@domain];
ALTER ROLE db_datawriter ADD MEMBER [user@domain];
ALTER ROLE db_ddladmin ADD MEMBER [user@domain];
GO
```



Running these commands assigns the [SQL DB Contributor](#) role to the account specified. This role allows the identity to read, write, and modify the data and schema of your database. For more information about the roles assigned, see [Fixed-database roles](#).

Update the local connection configuration

1. Create environment settings for your application.

```
ini

AZURE_SQL_SERVER=<YOURSERVERNAME>.database.windows.net
AZURE_SQL_DATABASE=<YOURDATABASENAME>
AZURE_SQL_PORT=1433
```

2. Existing application code that connects to Azure SQL Database using the [Node.js SQL Driver - tedious](#) continues to work with passwordless connections with minor changes. To use a [user-assigned managed identity](#), pass the `authentication.type` and `options.clientId` properties.

```
Node.js

import sql from 'mssql';

// Environment settings - no user or password
```

```
const server = process.env.AZURE_SQL_SERVER;
const database = process.env.AZURE_SQL_DATABASE;
const port = parseInt(process.env.AZURE_SQL_PORT);

// Passwordless configuration
const config = {
    server,
    port,
    database,
    authentication: {
        type: 'azure-active-directory-default',
    },
    options: {
        encrypt: true,
        clientId: process.env.AZURE_CLIENT_ID // <----- user-assigned
managed identity
    }
};

// Existing application code
export default class Database {
    config = {};
    poolconnection = null;
    connected = false;

    constructor(config) {
        this.config = config;
        console.log(`Database: config: ${JSON.stringify(config)}`);
    }

    async connect() {
        try {
            console.log(`Database connecting...${this.connected}`);
            if (this.connected === false) {
                this.poolconnection = await sql.connect(this.config);
                this.connected = true;
                console.log('Database connection successful');
            } else {
                console.log('Database already connected');
            }
        } catch (error) {
            console.error(`Error connecting to database:
${JSON.stringify(error)}`);
        }
    }

    async disconnect() {
        try {
            this.poolconnection.close();
            console.log('Database connection closed');
        } catch (error) {
            console.error(`Error closing database connection:
${error}`);
        }
    }
}
```

```
async executeQuery(query) {
    await this.connect();
    const request = this.poolconnection.request();
    const result = await request.query(query);

    return result.rowsAffected[0];
}

const databaseClient = new Database(config);
const result = await databaseClient.executeQuery(`select * from mytable
where id = 10`);
```

The `AZURE_CLIENT_ID` environment variable is created later in this tutorial.

Test the app

Run your app locally and verify that the connections to Azure SQL Database are working as expected. Keep in mind that it may take several minutes for changes to Azure users and roles to propagate through your Azure environment. Your application is now configured to run locally without developers having to manage secrets in the application itself.

Configure the Azure hosting environment

Once your app is configured to use passwordless connections locally, the same code can authenticate to Azure SQL Database after it's deployed to Azure. The sections that follow explain how to configure a deployed application to connect to Azure SQL Database using a [managed identity](#). Managed identities provide an automatically managed identity in Microsoft Entra ID ([formerly Azure Active Directory](#)) for applications to use when connecting to resources that support Microsoft Entra authentication. Learn more about managed identities:

- [Passwordless overview](#)
- [Managed identity best practices](#)
- [Managed identities in Microsoft Entra for Azure SQL](#)

Create the managed identity

Create a user-assigned managed identity using the Azure portal or the Azure CLI. Your application uses the identity to authenticate to other services.

1. At the top of the Azure portal, search for *Managed identities*. Select the **Managed Identities** result.
2. Select **+ Create** at the top of the **Managed Identities** overview page.
3. On the **Basics** tab, enter the following values:
 - **Subscription:** Select your desired subscription.
 - **Resource group:** Select your desired resource group.
 - **Region:** Select a region near your location.
 - **Name:** Enter a recognizable name for your identity, such as *MigrationIdentity*.
4. Select **Review + create** at the bottom of the page.
5. When the validation checks finish, select **Create**. Azure creates a new user-assigned identity.

After the resource is created, select **Go to resource** to view the details of the managed identity.

Home > Managed Identities >

Create User Assigned Managed Identity ...

[Basics](#) [Tags](#) [Review + create](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

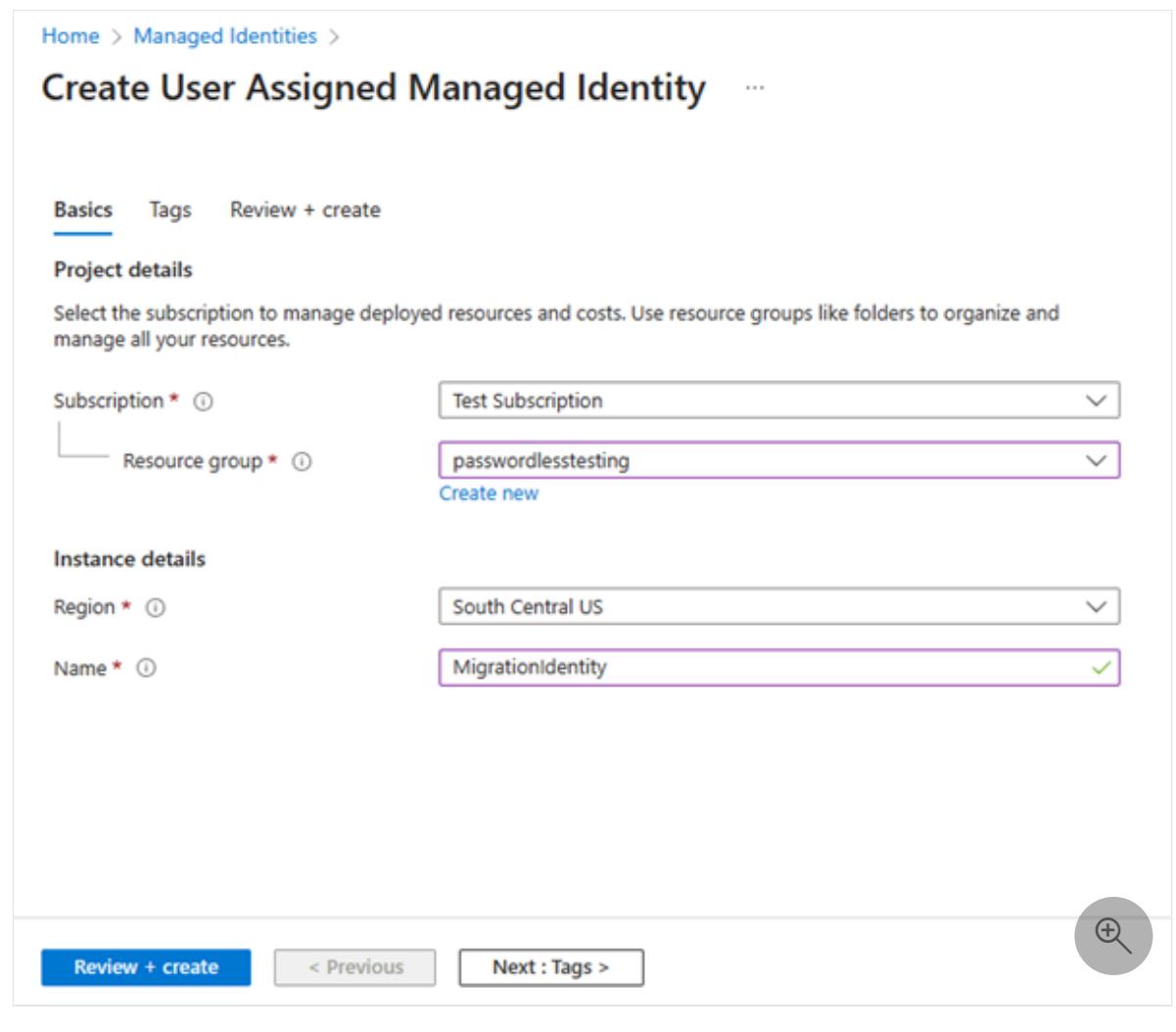
Resource group * ⓘ [Create new](#)

Instance details

Region * ⓘ

Name * ⓘ 

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



Associate the managed identity with your web app

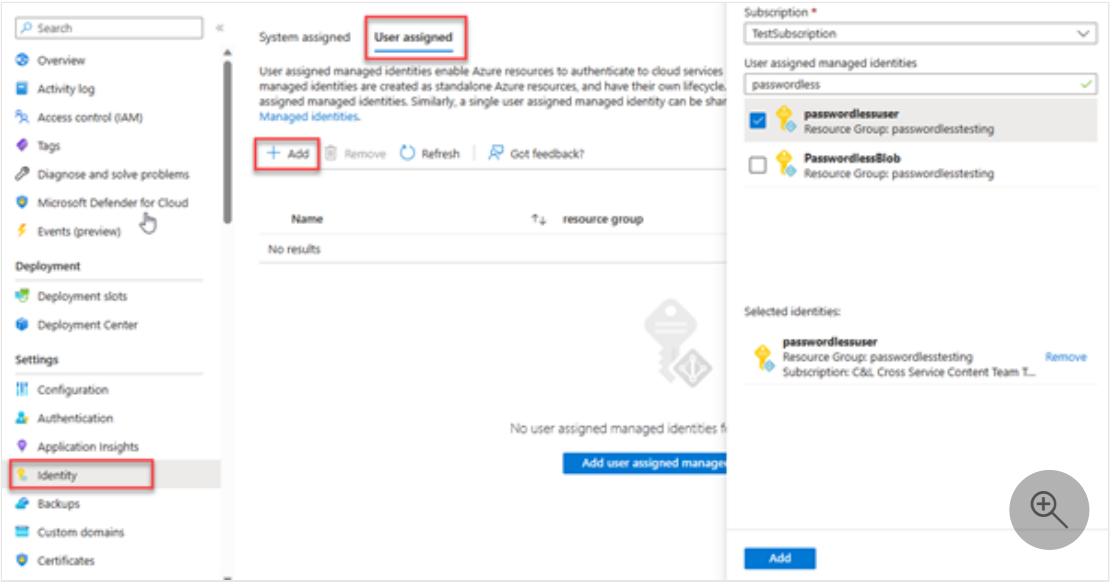
Configure your web app to use the user-assigned managed identity you created.

Azure portal

Complete the following steps in the Azure portal to associate the user-assigned managed identity with your app. These same steps apply to the following Azure services:

- Azure Spring Apps
- Azure Container Apps
- Azure virtual machines
- Azure Kubernetes Service
- Navigate to the overview page of your web app.

1. Select **Identity** from the left navigation.
2. On the **Identity** page, switch to the **User assigned** tab.
3. Select **+ Add** to open the **Add user assigned managed identity** flyout.
4. Select the subscription you used previously to create the identity.
5. Search for the **MigrationIdentity** by name and select it from the search results.
6. Select **Add** to associate the identity with your app.



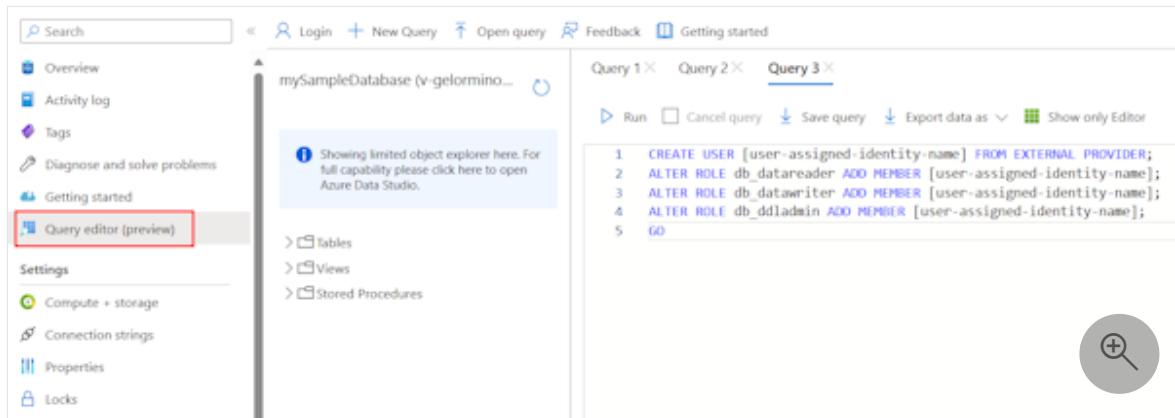
Create a database user for the identity and assign roles

Create a SQL database user that maps back to the user-assigned managed identity. Assign the necessary SQL roles to the user to allow your app to read, write, and modify the data and schema of your database.

1. In the Azure portal, browse to your SQL database and select **Query editor (preview)**.
2. Select **Continue as <username>** on the right side of the screen to sign into the database using your account.
3. On the query editor view, run the following T-SQL commands:

```
SQL

CREATE USER [user-assigned-identity-name] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [user-assigned-identity-name];
ALTER ROLE db_datawriter ADD MEMBER [user-assigned-identity-name];
ALTER ROLE db_ddladmin ADD MEMBER [user-assigned-identity-name];
GO
```



Running these commands assigns the [SQL DB Contributor](#) role to the user-assigned managed identity. This role allows the identity to read, write, and modify the data and schema of your database.

ⓘ Important

Use caution when assigning database user roles in enterprise production environments. In those scenarios, the app shouldn't perform all operations using a single, elevated identity. Try to implement the principle of least privilege by configuring multiple identities with specific permissions for specific tasks.

You can read more about configuring database roles and security on the following resources:

- [Tutorial: Secure a database in Azure SQL Database](#)
- [Authorize database access to SQL Database](#)

Create an app setting for the managed identity client ID

To use the **user-assigned** managed identity, create an `AZURE_CLIENT_ID` environment variable and set it equal to the client ID of the managed identity. You can set this variable in the **Configuration** section of your app in the Azure portal. You can find the client ID in the **Overview** section of the managed identity resource in the Azure portal.

Save your changes and restart the application if it doesn't do so automatically.

If you need to use a **system-assigned** managed identity, omit the `options.clientId` property. You still need to pass the `authentication.type` property.

Node.js

```
const config = {
  server,
  port,
  database,
  authentication: {
    type: 'azure-active-directory-default'
  },
  options: {
    encrypt: true
  }
};
```

Test the application

Test your app to make sure everything is still working. It may take a few minutes for all of the changes to propagate through your Azure environment.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- Passwordless overview
 - Managed identity best practices
 - Tutorial: Secure a database in Azure SQL Database
 - Authorize database access to SQL Database
-

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback 

Migrate a Python application to use passwordless connections with Azure SQL Database

Article • 10/11/2023

Applies to:  Azure SQL Database

Application requests to Azure SQL Database must be authenticated. Although there are multiple options for authenticating to Azure SQL Database, you should prioritize passwordless connections in your applications when possible. Traditional authentication methods that use passwords or secret keys create security risks and complications. Visit the [passwordless connections for Azure services](#) hub to learn more about the advantages of moving to passwordless connections. The following tutorial explains how to migrate an existing Python application to connect to Azure SQL Database to use passwordless connections instead of a username and password solution.

Configure the Azure SQL Database

Passwordless connections use Microsoft Entra authentication to connect to Azure services, including Azure SQL Database. Microsoft Entra authentication, you can manage identities in a central location to simplify permission management. Learn more about configuring Microsoft Entra authentication for your Azure SQL Database:

- [Microsoft Entra authentication overview](#)
- [Configure Microsoft Entra auth](#)

For this migration guide, ensure you have a Microsoft Entra admin assigned to your Azure SQL Database.

1. Navigate to the **Microsoft Entra** page of your logical server.
2. Select **Set admin** to open the **Microsoft Entra ID** flyout menu.
3. In the **Microsoft Entra ID** flyout menu, search for the user you want to assign as admin.
4. Select the user and choose **Select**.

The screenshot shows the Microsoft Entra admin center interface. At the top, there's a search bar, 'Set admin', 'Remove admin', and 'Save' buttons. On the left, a sidebar lists 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', and 'Settings'. Under 'Settings', 'Microsoft Entra ID' is selected and highlighted with a red box. Other options include 'SQL databases' and 'SQL elastic pools'. On the right, the main content area is titled 'Microsoft Entra authentication only', explaining that only Microsoft Entra ID will be used for authentication. It includes a checkbox for 'Support only Microsoft Entra authentication for this server' which is checked. A magnifying glass icon is in the bottom right corner.

Configure your local development environment

Passwordless connections can be configured to work for both local and Azure hosted environments. In this section, you apply configurations to allow individual users to authenticate to Azure SQL Database for local development.

Sign-in to Azure

For local development, make sure you're signed-in with the same Azure AD account you want to use to access Azure SQL Database. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

The screenshot shows the Azure CLI interface. The title bar says 'Azure CLI'. Below it, a command-line window displays the 'az login' command. The command is highlighted in blue, indicating it is being run or has been typed.

Create a database user and assign roles

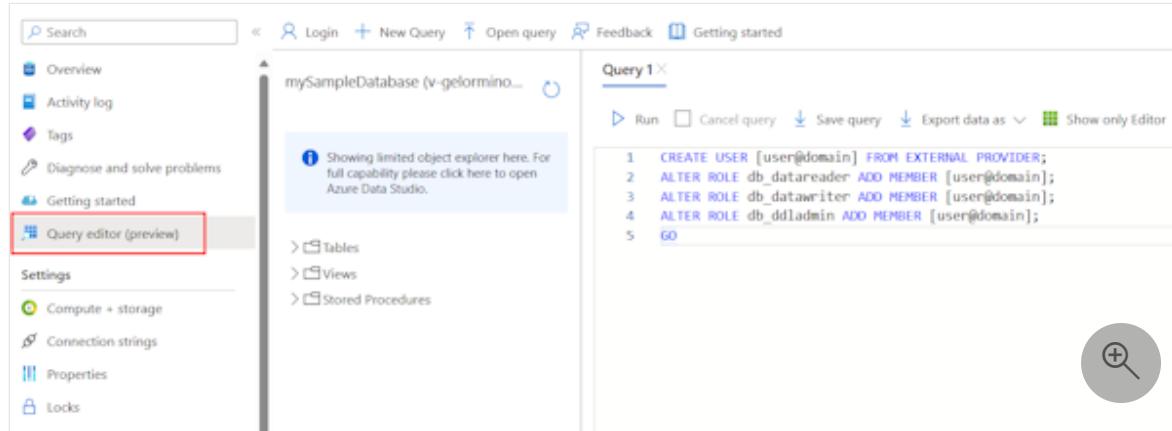
Create a user in Azure SQL Database. The user should correspond to the Azure account you used to sign-in locally in the [Sign-in to Azure](#) section.

1. In the [Azure portal](#), browse to your SQL database and select **Query editor (preview)**.
2. Select **Continue as <your-username>** on the right side of the screen to sign into the database using your account.

3. On the query editor view, run the following T-SQL commands:

```
SQL

CREATE USER [user@domain] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [user@domain];
ALTER ROLE db_datawriter ADD MEMBER [user@domain];
ALTER ROLE db_ddladmin ADD MEMBER [user@domain];
GO
```



Running these commands assigns the [SQL DB Contributor](#) role to the account specified. This role allows the identity to read, write, and modify the data and schema of your database. For more information about the roles assigned, see [Fixed-database roles](#).

Update the local connection configuration

Existing application code that connects to Azure SQL Database using the [Python SQL Driver - pyodbc](#) continues to work with passwordless connections with minor changes. For example, the following code works with both SQL authentication and passwordless connections when running locally and when deployed to Azure App Service.

```
Python

import os
import pyodbc, struct
from azure.identity import DefaultAzureCredential

connection_string = os.environ["AZURE_SQL_CONNECTIONSTRING"]

def get_all():
    with get_conn() as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM Persons")
        # Do something with the data
    return
```

```
def get_conn():
    credential =
    DefaultAzureCredential(exclude_interactive_browser_credential=False)
    token_bytes =
    credential.get_token("https://database.windows.net/.default").token.encode("UTF-16-LE")
    token_struct = struct.pack(f'<I{len(token_bytes)}s', len(token_bytes),
    token_bytes)
    SQL_COPT_SS_ACCESS_TOKEN = 1256 # This connection option is defined by
    microsoft in msodbcsql.h
    conn = pyodbc.connect(connection_string, attrs_before=
    {SQL_COPT_SS_ACCESS_TOKEN: token_struct})
    return conn
```

💡 Tip

In this example code, the App Service environment variable `WEBSITE_HOSTNAME` is used to determine what environment the code is running in. For other deployment scenarios, you can use other environment variables to determine the environment.

To update the referenced connection string (`AZURE_SQL_CONNECTIONSTRING`) for local development, use the passwordless connection string format:

```
Driver={ODBC Driver 18 for SQL Server};Server=tcp:<database-server-
name>.database.windows.net,1433;Database=<database-
name>;Encrypt=yes;TrustServerCertificate=no;Connection Timeout=30
```

Test the app

Run your app locally and verify that the connections to Azure SQL Database are working as expected. Keep in mind that it may take several minutes for changes to Azure users and roles to propagate through your Azure environment. Your application is now configured to run locally without developers having to manage secrets in the application itself.

Configure the Azure hosting environment

Once your app is configured to use passwordless connections locally, the same code can authenticate to Azure SQL Database after it's deployed to Azure. The sections that follow explain how to configure a deployed application to connect to Azure SQL

Database using a [managed identity](#). Managed identities provide an automatically managed identity in Microsoft Entra ID ([formerly Azure Active Directory](#)) for applications to use when connecting to resources that support Microsoft Entra authentication. Learn more about managed identities:

- [Passwordless overview](#)
- [Managed identity best practices](#)

Create the managed identity

Create a user-assigned managed identity using the Azure portal or the Azure CLI. Your application uses the identity to authenticate to other services.

Azure portal

1. At the top of the Azure portal, search for *Managed identities*. Select the **Managed Identities** result.
2. Select **+ Create** at the top of the **Managed Identities** overview page.
3. On the **Basics** tab, enter the following values:
 - **Subscription**: Select your desired subscription.
 - **Resource group**: Select your desired resource group.
 - **Region**: Select a region near your location.
 - **Name**: Enter a recognizable name for your identity, such as *MigrationIdentity*.
4. Select **Review + create** at the bottom of the page.
5. When the validation checks finish, select **Create**. Azure creates a new user-assigned identity.

After the resource is created, select **Go to resource** to view the details of the managed identity.

Create User Assigned Managed Identity

Basics Tags Review + create

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Test Subscription

Resource group * ⓘ passwordlesstesting
Create new

Instance details

Region * ⓘ South Central US

Name * ⓘ MigrationIdentity

Review + create < Previous Next : Tags > 

Associate the managed identity with your web app

Configure your web app to use the user-assigned managed identity you created.

Azure portal

Complete the following steps in the Azure portal to associate the user-assigned managed identity with your app. These same steps apply to the following Azure services:

- Azure Spring Apps
- Azure Container Apps
- Azure virtual machines
- Azure Kubernetes Service
- Navigate to the overview page of your web app.

1. Select **Identity** from the left navigation.

2. On the **Identity** page, switch to the **User assigned** tab.
3. Select **+ Add** to open the **Add user assigned managed identity** flyout.
4. Select the subscription you used previously to create the identity.
5. Search for the **MigrationIdentity** by name and select it from the search results.
6. Select **Add** to associate the identity with your app.

The screenshot shows the Azure portal's Identity blade. The 'User assigned' tab is selected. A red box highlights the '+ Add' button. The 'Selected identities' section shows 'passwordlessuser' selected. A search bar and a magnifying glass icon are also visible.

Create a database user for the identity and assign roles

Create a SQL database user that maps back to the user-assigned managed identity. Assign the necessary SQL roles to the user to allow your app to read, write, and modify the data and schema of your database.

1. In the Azure portal, browse to your SQL database and select **Query editor (preview)**.
2. Select **Continue as <username>** on the right side of the screen to sign into the database using your account.
3. On the query editor view, run the following T-SQL commands:

SQL

```
CREATE USER [user-assigned-identity-name] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [user-assigned-identity-name];
ALTER ROLE db_datawriter ADD MEMBER [user-assigned-identity-name];
ALTER ROLE db_ddladmin ADD MEMBER [user-assigned-identity-name];
GO
```

```
1 CREATE USER [user-assigned-identity-name] FROM EXTERNAL PROVIDER;
2 ALTER ROLE db_datareader ADD MEMBER [user-assigned-identity-name];
3 ALTER ROLE db_datawriter ADD MEMBER [user-assigned-identity-name];
4 ALTER ROLE db_ddladmin ADD MEMBER [user-assigned-identity-name];
5 GO
```

Running these commands assigns the [SQL DB Contributor](#) role to the user-assigned managed identity. This role allows the identity to read, write, and modify the data and schema of your database.

ⓘ Important

Use caution when assigning database user roles in enterprise production environments. In those scenarios, the app shouldn't perform all operations using a single, elevated identity. Try to implement the principle of least privilege by configuring multiple identities with specific permissions for specific tasks.

You can read more about configuring database roles and security on the following resources:

- [Tutorial: Secure a database in Azure SQL Database](#)
- [Authorize database access to SQL Database](#)

Update the connection string

Update your Azure app configuration to use the passwordless connection string format. The format should be the same used in your local environment.

Connection strings can be stored as environment variables in your app hosting environment. The following instructions focus on App Service, but other Azure hosting services provide similar configurations.

```
Driver={ODBC Driver 18 for SQL Server};Server=tcp:<database-server-name>.database.windows.net,1433;Database=<database-name>;Encrypt=yes;TrustServerCertificate=no;Connection Timeout=30
```

<database-server-name> is the name of your Azure SQL Database server and <database-name> is the name of your Azure SQL Database.

Create an app setting for the managed identity client ID

To use the user-assigned managed identity, create an AZURE_CLIENT_ID environment variable and set it equal to the client ID of the managed identity. You can set this variable in the **Configuration** section of your app in the Azure portal. You can find the client ID in the **Overview** section of the managed identity resource in the Azure portal.

Save your changes and restart the application if it doesn't do so automatically.

Note

The example connection code shown in this migration guide uses the [DefaultAzureCredential](#) class when deployed. Specifically, it uses the `DefaultAzureCredential` without passing the user-assigned managed identity client ID to the constructor. In this scenario, the fallback is to check for the `AZURE_CLIENT_ID` environment variable. If the `AZURE_CLIENT_ID` environment variable doesn't exist, a system-assigned managed identity will be used if configured.

If you pass the managed identity client ID in the `DefaultAzureCredential` constructor, the connection code can still be used locally and deployed because the authentication process falls back to interactive authentication in the local scenario. For more information, see the [Azure Identity client library for Python](#).

Test the application

Test your app to make sure everything is still working. It may take a few minutes for all of the changes to propagate through your Azure environment.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Passwordless overview](#)
- [Managed identity best practices](#)

- [Tutorial: Secure a database in Azure SQL Database](#)
 - [Authorize database access to SQL Database](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Migrate an application to use passwordless connections with Azure Cosmos DB for NoSQL

Article • 06/01/2023

Application requests to Azure Cosmos DB for NoSQL must be authenticated. Although there are multiple options for authenticating to Azure Cosmos DB, you should prioritize passwordless connections in your applications when possible. Traditional authentication methods that use connection strings with passwords or secret keys create security risks and complications. Visit the [passwordless connections for Azure services](#) hub to learn more about the advantages of moving to passwordless connections.

The following tutorial explains how to migrate an existing application to connect to Azure Cosmos DB for NoSQL using passwordless connections instead of a key-based solution.

Configure roles and users for local development authentication

When developing locally with passwordless authentication, make sure the user account that connects to Cosmos DB is assigned a role with the correct permissions to perform data operations. Currently, Azure Cosmos DB for NoSQL doesn't include built-in roles for data operations, but you can create your own using the Azure CLI or PowerShell.

Roles consist of a collection of permissions or actions that a user is allowed to perform, such as read, write, and delete. You can read more about [configuring role-based access control \(RBAC\)](#) in the Cosmos DB security configuration documentation.

Create the custom role

1. Create a role using the `az role definition create` command. Pass in the Cosmos DB account name and resource group, followed by a body of JSON that defines the custom role. The following example creates a role named `PasswordlessReadWrite` with permissions to read and write items in Cosmos DB containers. The role is also scoped to the account level using `/`.

Azure CLI

```
az cosmosdb sql role definition create \
    --account-name <cosmosdb-account-name> \
    --resource-group <resource-group-name> \
    --body '{
        "RoleName": "PasswordlessReadWrite",
        "Type": "CustomRole",
        "AssignableScopes": ["/"],
        "Permissions": [
            "DataActions": [
                "Microsoft.DocumentDB/databaseAccounts/readMetadata",
                "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/*"
            ],
            "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/*"
        ]
    }'
```

2. When the command completes, copy the ID value from the `name` field and paste it somewhere for later use.
3. Assign the role you created to the user account or service principal that will connect to Cosmos DB. During local development, this will generally be your own account that's logged into a development tool like Visual Studio or the Azure CLI. Retrieve the details of your account using the `az ad user` command.

Azure CLI

```
az ad user show --id "<your-email-address>"
```

4. Copy the value of the `id` property out of the results and paste it somewhere for later use.
5. Assign the custom role you created to your user account using the `az cosmosdb sql role assignment create` command and the IDs you copied previously.

Azure CLI

```
az cosmosdb sql role assignment create \
    --account-name <cosmosdb-account-name> \
    --resource-group <resource-group-name> \
    --scope "/" \
    --principal-id <your-user-id> \
    --role-definition-id <your-custom-role-id>
```

Sign-in to Azure locally

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

Azure CLI

```
az login
```

Migrate the app code to use passwordless connections

.NET

1. To use `DefaultAzureCredential` in a .NET application, install the `Azure.Identity` package:

.NET CLI

```
dotnet add package Azure.Identity
```

2. At the top of your file, add the following code:

C#

```
using Azure.Identity;
```

3. Identify the locations in your code that create a `CosmosClient` object to connect to Azure Cosmos DB. Update your code to match the following example.

C#

```
DefaultAzureCredential credential = new();  
  
using CosmosClient client = new(
```

```
    accountEndpoint:  
    Environment.GetEnvironmentVariable("COSMOS_ENDPOINT"),  
    tokenCredential: credential  
);
```

Run the app locally

After making these code changes, run your application locally. The new configuration should pick up your local credentials, such as the Azure CLI, Visual Studio, or IntelliJ. The roles you assigned to your local dev user in Azure allows your app to connect to the Azure service locally.

Configure the Azure hosting environment

Once your application is configured to use passwordless connections and runs locally, the same code can authenticate to Azure services after it's deployed to Azure. The sections that follow explain how to configure a deployed application to connect to Azure Cosmos DB using a managed identity.

Create the managed identity

You can create a user-assigned managed identity using the Azure portal or the Azure CLI. Your application uses the identity to authenticate to other services.

Azure portal

1. At the top of the Azure portal, search for *Managed identities*. Select the **Managed Identities** result.
2. Select **+ Create** at the top of the **Managed Identities** overview page.
3. On the **Basics** tab, enter the following values:
 - **Subscription:** Select your desired subscription.
 - **Resource Group:** Select your desired resource group.
 - **Region:** Select a region near your location.
 - **Name:** Enter a recognizable name for your identity, such as *MigrationIdentity*.
4. Select **Review + create** at the bottom of the page.
5. When the validation checks finish, select **Create**. Azure creates a new user-assigned identity.

After the resource is created, select **Go to resource** to view the details of the managed identity.

The screenshot shows the 'Create User Assigned Managed Identity' wizard. At the top, there's a breadcrumb navigation: Home > Managed Identities >. The main title is 'Create User Assigned Managed Identity' with a three-dot ellipsis to its right. Below the title, there are three tabs: 'Basics' (which is selected), 'Tags', and 'Review + create'. The 'Project details' section asks to select a subscription and resource group. The 'Subscription' dropdown is set to 'Test Subscription'. The 'Resource group' dropdown is set to 'passwordlesstesting', with a 'Create new' link below it. The 'Instance details' section includes 'Region' (set to 'South Central US') and 'Name' (set to 'MigrationIdentity'). At the bottom of the wizard, there are buttons for 'Review + create', '< Previous', 'Next : Tags >', and a magnifying glass icon for search.

Associate the managed identity with your web app

You need to configure your web app to use the managed identity you created. Assign the identity to your app using either the Azure portal or the Azure CLI.

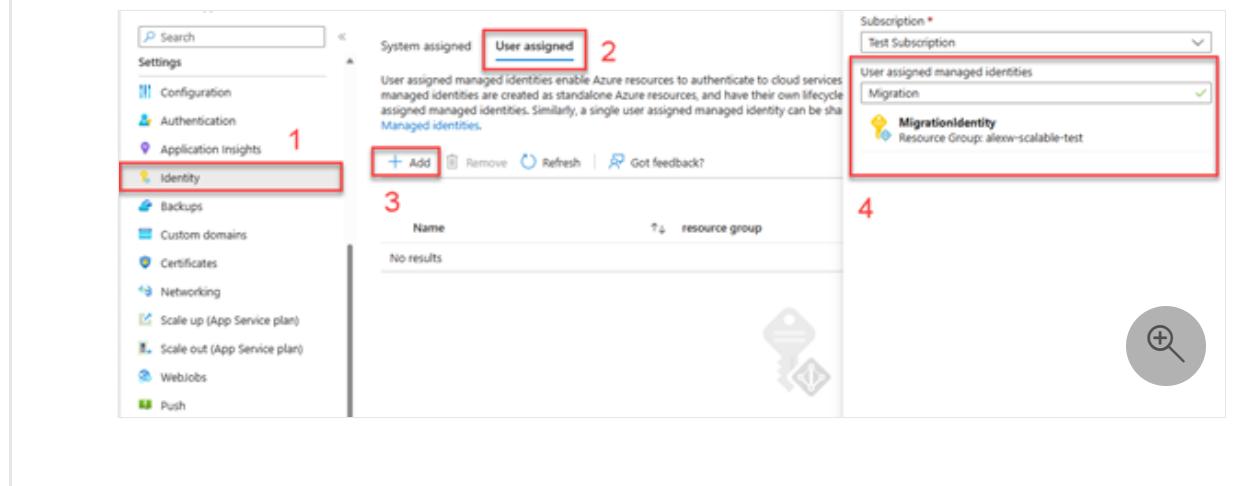
Azure portal

Complete the following steps in the Azure portal to associate an identity with your app. These same steps apply to the following Azure services:

- Azure Spring Apps
- Azure Container Apps
- Azure virtual machines
- Azure Kubernetes Service

1. Navigate to the overview page of your web app.

2. Select **Identity** from the left navigation.
3. On the **Identity** page, switch to the **User assigned** tab.
4. Select **+ Add** to open the **Add user assigned managed identity** flyout.
5. Select the subscription you used previously to create the identity.
6. Search for the **MigrationIdentity** by name and select it from the search results.
7. Select **Add** to associate the identity with your app.



Assign roles to the managed identity

Grant permissions to the managed identity by assigning it the custom role you created, just like you did with your local development user.

To assign a role at the resource level using the Azure CLI, you first must retrieve the resource ID using the [az cosmosdb show](#) command. You can filter the output properties using the `--query` parameter.

```
Azure CLI
az cosmosdb show \
--resource-group '<resource-group-name>' \
--name '<cosmosdb-name>' \
--query id
```

Copy the output ID from the preceding command. You can then assign roles using the [az role assignment create](#) command of the Azure CLI.

```
Azure CLI
az role assignment create \
--assignee "<your-managed-identity-name>" \
```

```
--role "PasswordlessReadWrite" \
--scope "<cosmosdb-resource-id>"
```

Update the application code

You need to configure your application code to look for the specific managed identity you created when it's deployed to Azure. In some scenarios, explicitly setting the managed identity for the app also prevents other environment identities from accidentally being detected and used automatically.

1. On the managed identity overview page, copy the client ID value to your clipboard.
2. Apply the following language-specific changes:

.NET

Create a `DefaultAzureCredentialOptions` object and pass it to `DefaultAzureCredential`. Set the `ManagedIdentityClientId` property to the client ID.

C#

```
DefaultAzureCredential credential = new(
    new DefaultAzureCredentialOptions
    {
        ManagedIdentityClientId = managedIdentityClientId
    });

```

3. Redeploy your code to Azure after making this change in order for the configuration updates to be applied.

Test the app

After deploying the updated code, browse to your hosted application in the browser. Your app should be able to connect to Cosmos DB successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Authorize access to blobs using Microsoft Entra ID](#)
- To learn more about .NET, see [Get started with .NET in 10 minutes](#).

Migrate an application to use passwordless connections with Azure Event Hubs

Article • 06/12/2023

Application requests to Azure services must be authenticated using configurations such as account access keys or passwordless connections. However, you should prioritize passwordless connections in your applications when possible. Traditional authentication methods that use passwords or secret keys create security risks and complications. Visit the [passwordless connections for Azure services](#) hub to learn more about the advantages of moving to passwordless connections.

The following tutorial explains how to migrate an existing application to connect using passwordless connections. These same migration steps should apply whether you're using access keys, connection strings, or another secrets-based approach.

Configure your local development environment

Passwordless connections can be configured to work for both local and Azure-hosted environments. In this section, you'll apply configurations to allow individual users to authenticate to Azure Event Hubs for local development.

Assign user roles

When developing locally, make sure that the user account that is accessing Azure Event Hubs has the correct permissions. You'll need the **Azure Event Hubs Data Receiver** and **Azure Event Hubs Data Sender** roles to read and write message data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the **Azure Event Hubs Data Sender** and **Azure Event Hubs Data Receiver** roles to your user account. These role grants read and write access to event hub messages.

Azure portal

1. In the Azure portal, locate your event hub using the main search bar or left navigation.
2. On the event hub overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the Azure portal's Access control (IAM) interface for an Event Hubs Namespace. The left sidebar lists various management options like Overview, Activity log, and Access control (IAM). The Access control (IAM) option is selected and highlighted with a red box. The main content area has tabs for Check access, Role assignments, Roles, Deny assignments, and Classic administrators. Under the 'My access' section, there's a 'View my access' button. Below it, the 'Check access' section allows reviewing access levels for users, groups, service principals, or managed identities. Two large callout boxes are present: one for 'Grant access to this resource' (with a 'Learn more' link and an 'Add role assignment' button highlighted with a red box) and another for 'View access to this resource' (with a 'View' button).

5. Use the search box to filter the results to the desired role. For this example, search for *Azure Event Hubs Data Sender* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

9. Repeat these steps for the Azure Event Hubs Data Receiver role to allow the account to send and receive messages.

ⓘ Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Sign-in to Azure locally

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

```
Azure CLI
```

```
az login
```

Update the application code to use passwordless connections

The Azure Identity client library, for each of the following ecosystems, provides a `DefaultAzureCredential` class that handles passwordless authentication to Azure:

- .NET
- C++ ↗
- Go ↗
- Java
- Node.js
- Python

`DefaultAzureCredential` supports multiple authentication methods. The method to use is determined at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code. See the preceding links for the order and locations in which `DefaultAzureCredential` looks for credentials.

.NET

1. To use `DefaultAzureCredential` in a .NET application, install the `Azure.Identity` package:

.NET CLI

```
dotnet add package Azure.Identity
```

2. At the top of your file, add the following code:

C#

```
using Azure.Identity;
```

3. Identify the locations in your code that create an `EventHubProducerClient` or `EventProcessorClient` object to connect to Azure Event Hubs. Update your code to match the following example:

C#

```
DefaultAzureCredential credential = new();
var eventHubNamespace =
"https://{namespace}.servicebus.windows.net";

// Event Hubs producer
EventHubProducerClient producerClient = new(
    eventHubNamespace,
    eventHubName,
    credential);

// Event Hubs processor
EventProcessorClient processorClient = new(
    storageClient,
    EventHubConsumerClient.DefaultConsumerGroupName,
    eventHubNamespace,
```

```
eventHubName,  
credential);
```

4. Make sure to update the event hubs namespace in the URI of your `EventHubProducerClient` or `EventProcessorClient` objects. You can find the namespace name on the overview page of the Azure portal.

The screenshot shows the Azure portal interface for an 'Event Hubs Namespace'. The top navigation bar includes a search bar, a star icon, and three dots. Below the bar, there are buttons for '+ Event Hub', 'Delete', 'Refresh', and 'Feedback'. The main content area has a title 'passwordlessevents' with a subtitle 'Event Hubs Namespace'. A red box highlights the title. On the left, a sidebar lists 'Overview', 'Activity log', 'Access control (IAM)', and 'Tags'. The 'Overview' tab is selected. On the right, under the 'Essentials' section, it shows 'Resource group (move) : passwordlesstesting', 'Status : Active', and 'Location : South Central US'.

Run the app locally

After making these code changes, run your application locally. The new configuration should pick up your local credentials, such as the Azure CLI, Visual Studio, or IntelliJ. The roles you assigned to your user in Azure allows your app to connect to the Azure service locally.

Configure the Azure hosting environment

Once your application is configured to use passwordless connections and runs locally, the same code can authenticate to Azure services after it's deployed to Azure. The sections that follow explain how to configure a deployed application to connect to Azure Event Hubs using a [managed identity](#). Managed identities provide an automatically managed identity in Microsoft Entra ID for applications to use when connecting to resources that support Microsoft Entra authentication. Learn more about managed identities:

- [Passwordless Overview](#)
- [Managed identity best practices](#)

Create the managed identity

You can create a user-assigned managed identity using the Azure portal or the Azure CLI. Your application uses the identity to authenticate to other services.

1. At the top of the Azure portal, search for *Managed identities*. Select the **Managed Identities** result.
2. Select **+ Create** at the top of the **Managed Identities** overview page.
3. On the **Basics** tab, enter the following values:
 - **Subscription:** Select your desired subscription.
 - **Resource Group:** Select your desired resource group.
 - **Region:** Select a region near your location.
 - **Name:** Enter a recognizable name for your identity, such as *MigrationIdentity*.
4. Select **Review + create** at the bottom of the page.
5. When the validation checks finish, select **Create**. Azure creates a new user-assigned identity.

After the resource is created, select **Go to resource** to view the details of the managed identity.

Home > Managed Identities >

Create User Assigned Managed Identity ...

[Basics](#) [Tags](#) [Review + create](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

Resource group * ⓘ [Create new](#)

Instance details

Region * ⓘ

Name * ⓘ 

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

Associate the managed identity with your web app

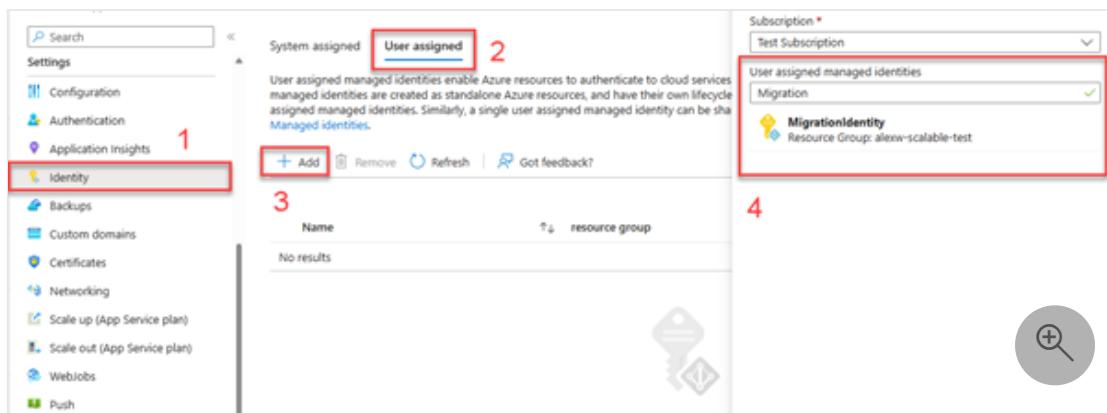
You need to configure your web app to use the managed identity you created. Assign the identity to your app using either the Azure portal or the Azure CLI.

Azure portal

Complete the following steps in the Azure portal to associate an identity with your app. These same steps apply to the following Azure services:

- Azure Spring Apps
- Azure Container Apps
- Azure virtual machines
- Azure Kubernetes Service

1. Navigate to the overview page of your web app.
2. Select **Identity** from the left navigation.
3. On the **Identity** page, switch to the **User assigned** tab.
4. Select **+ Add** to open the **Add user assigned managed identity** flyout.
5. Select the subscription you used previously to create the identity.
6. Search for the **MigrationIdentity** by name and select it from the search results.
7. Select **Add** to associate the identity with your app.



Assign roles to the managed identity

Next, you need to grant permissions to the managed identity you created to access your event hub. Grant permissions by assigning a role to the managed identity, just like you did with your local development user.

1. Navigate to your event hub overview page and select **Access Control (IAM)** from the left navigation.

2. Choose **Add role assignment**

The screenshot shows the Azure portal's Access Control (IAM) interface for a storage account. The left sidebar has a red box around the 'Access Control (IAM)' option. The main area has a red box around the 'Add role assignment' button in the 'Grant access to this resource' section.

3. In the **Role** search box, search for *Azure Event Hub Data Sender*, which is a common role used to manage data operations for queues. You can assign whatever role is appropriate for your use case. Select the *Azure Event Hub Data Sender* from the list and choose **Next**.
4. On the **Add role assignment** screen, for the **Assign access to** option, select **Managed identity**. Then choose **+Select members**.
5. In the flyout, search for the managed identity you created by name and select it from the results. Choose **Select** to close the flyout menu.

The screenshot shows the 'Add role assignment' screen and its 'Select members' flyout. Step 1 highlights the 'Managed identity' option under 'Assign access to'. Step 2 highlights the '+Select members' button. Step 3 highlights the search bar in the flyout with 'msdocs-web'. Step 4 highlights the 'Selected members' list with 'msdocs-web-app-123'. Step 5 highlights the 'Select' button in the flyout. Step 6 highlights the 'Next' button at the bottom of the main screen.

6. Select **Next** a couple times until you're able to select **Review + assign** to finish the role assignment.
7. Repeat these steps for the **Azure Event Hub Data Receiver** role.

Update the application code

You need to configure your application code to look for the specific managed identity you created when it's deployed to Azure. In some scenarios, explicitly setting the managed identity for the app also prevents other environment identities from accidentally being detected and used automatically.

1. On the managed identity overview page, copy the client ID value to your clipboard.
2. Apply the following language-specific changes:

.NET

Create a `DefaultAzureCredentialOptions` object and pass it to `DefaultAzureCredential`. Set the `ManagedIdentityClientId` property to the client ID.

C#

```
DefaultAzureCredential credential = new(
    new DefaultAzureCredentialOptions
    {
        ManagedIdentityClientId = managedIdentityClientId
    });

```

3. Redeploy your code to Azure after making this change in order for the configuration updates to be applied.

Test the app

After deploying the updated code, browse to your hosted application in the browser. Your app should be able to connect to the event hub successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Passwordless connections for Azure services](#)
- To learn more about .NET, see [Get started with .NET in 10 minutes](#).

Migrate an application to use passwordless connections with Azure Event Hubs for Kafka

Article • 05/30/2023

This article explains how to migrate from traditional authentication methods to more secure, passwordless connections with Azure Event Hubs for Kafka.

Application requests to Azure Event Hubs for Kafka must be authenticated. Azure Event Hubs for Kafka provides different ways for apps to connect securely. One of the ways is to use a connection string. However, you should prioritize passwordless connections in your applications when possible.

Passwordless connections are supported since Spring Cloud Azure 4.3.0. This article is a migration guide for removing credentials from Spring Cloud Stream Kafka applications.

Compare authentication options

When the application authenticates with Azure Event Hubs for Kafka, it provides an authorized entity to connect the Event Hubs namespace. Apache Kafka protocols provide multiple Simple Authentication and Security Layer (SASL) mechanisms for authentication. According to the SASL mechanisms, there are two authentication options that you can use to authorize access to your secure resources: Microsoft Entra authentication and Shared Access Signature (SAS) authentication.

Microsoft Entra authentication

Microsoft Entra authentication is a mechanism for connecting to Azure Event Hubs for Kafka using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage service principal identities and other Microsoft services in a central location, which simplifies permission management.

Using Microsoft Entra ID for authentication provides the following benefits:

- Authentication of users across Azure services in a uniform way.
- Management of password policies and password rotation in a single place.
- Multiple forms of authentication supported by Microsoft Entra ID, which can eliminate the need to store passwords.

- Customers can manage Event Hubs permissions using external (Microsoft Entra ID) groups.
- Support for token-based authentication for applications connecting to Azure Event Hubs for Kafka.

SAS authentication

Event Hubs also provides Shared Access Signatures (SAS) for delegated access to Event Hubs for Kafka resources.

Although it's possible to connect to Azure Event Hubs for Kafka with SAS, it should be used with caution. You must be diligent to never expose the connection strings in an unsecure location. Anyone who gains access to the connection strings is able to authenticate. For example, there's a risk that a malicious user can access the application if a connection string is accidentally checked into source control, sent through an unsecure email, pasted into the wrong chat, or viewed by someone who shouldn't have permission. Instead, authorizing access using the OAuth 2.0 token-based mechanism provides superior security and ease of use over SAS. Consider updating your application to use passwordless connections.

Introducing passwordless connections

With a passwordless connection, you can connect to Azure services without storing any credentials in the application code, its configuration files, or in environment variables.

Many Azure services support passwordless connections, for example via Azure Managed Identity. These techniques provide robust security features that you can implement using `DefaultAzureCredential` from the Azure Identity client libraries. In this tutorial, you'll learn how to update an existing application to use `DefaultAzureCredential` instead of alternatives such as connection strings.

`DefaultAzureCredential` supports multiple authentication methods and automatically determines which should be used at runtime. This approach enables your app to use different authentication methods in different environments (local dev vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` searches for credentials can be found in the [Azure Identity library overview](#). For example, when working locally, `DefaultAzureCredential` will generally authenticate using the account the developer used to sign in to Visual Studio. When the app is deployed to Azure,

`DefaultAzureCredential` will automatically switch to use a [managed identity](#). No code changes are required for this transition.

To ensure that connections are passwordless, you must take into consideration both local development and the production environment. If a connection string is required in either place, then the application isn't passwordless.

In your local development environment, you can authenticate with Azure CLI, Azure PowerShell, Visual Studio, or Azure plugins for Visual Studio Code or IntelliJ. In this case, you can use that credential in your application instead of configuring properties.

When you deploy applications to an Azure hosting environment, such as a virtual machine, you can assign managed identity in that environment. Then, you won't need to provide credentials to connect to Azure services.

 **Note**

A managed identity provides a security identity to represent an app or service. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. You can read more about managed identities in the [overview](#) documentation.

Migrate an existing application to use passwordless connections

The following steps explain how to migrate an existing application to use passwordless connections instead of a SAS solution.

0) Prepare the working environment for local development authentication

First, use the following command to set up some environment variables.

Bash

```
export AZ_RESOURCE_GROUP=<YOUR_RESOURCE_GROUP>
export AZ_EVENTHUBS_NAMESPACE_NAME=<YOUR_EVENTHUBS_NAMESPACE_NAME>
export AZ_EVENTHUB_NAME=<YOUR_EVENTHUB_NAME>
```

Replace the placeholders with the following values, which are used throughout this article:

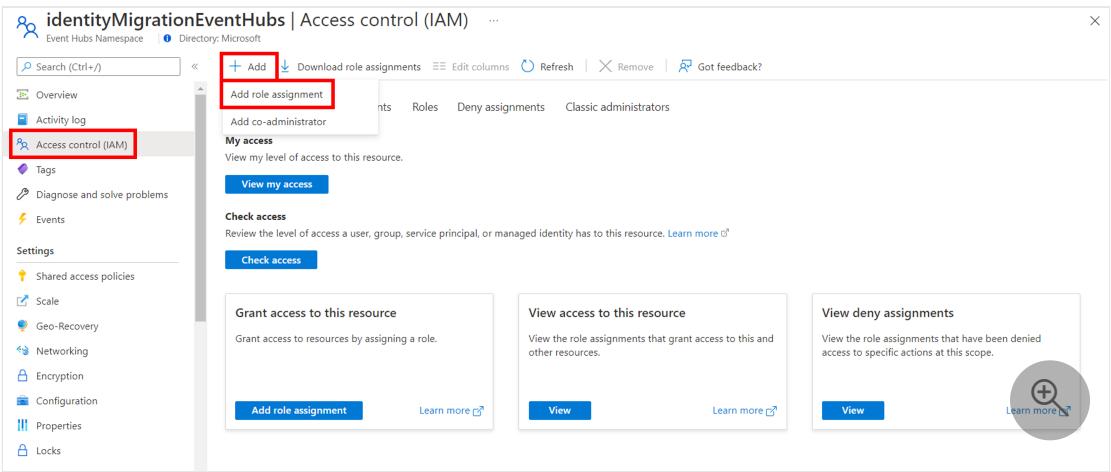
- <YOUR_RESOURCE_GROUP>: The name of the resource group you'll use.
- <YOUR_EVENTHUBS_NAMESPACE_NAME>: The name of the Azure Event Hubs namespace you'll use.
- <YOUR_EVENTHUB_NAME> : The name of the event hub you'll use.

1) Grant permission for Azure Event Hubs

If you want to run this sample locally with Microsoft Entra authentication, be sure your user account has authenticated via Azure Toolkit for IntelliJ, Visual Studio Code Azure Account plugin, or Azure CLI. Also, be sure the account has been granted sufficient permissions.

Azure portal

1. In the Azure portal, locate your Event Hubs namespace using the main search bar or left navigation.
2. On the Event Hubs overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



5. Use the search box to filter the results to the desired role. For this example, search for *Azure Event Hubs Data Sender* and *Azure Event Hubs Data Receiver* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **Select members**.

7. In the dialog, search for your Microsoft Entra username (usually your `user@domain` email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

For more information about granting access roles, see [Authorize access to Event Hubs resources using Microsoft Entra ID](#).

2) Sign in and migrate the app code to use passwordless connections

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to on your Event Hubs. You can authenticate via the Azure CLI, Visual Studio, Azure PowerShell, or other tools such as IntelliJ.

Azure CLI

Sign in to Azure through the Azure CLI by using the following command:

```
Azure CLI
```

```
az login
```

Next, use the following steps to update your Spring Kafka application to use passwordless connections. Although conceptually similar, each framework uses different implementation details.

Java

1. Inside your project, open the `pom.xml` file and add the following reference:

```
XML
```

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-identity</artifactId>
    <version>1.6.0</version>
</dependency>
```

2. After migration, implement [AuthenticateCallbackHandler](#) and [OAuthBearerToken](#) in your project for OAuth2 authentication, as shown in the following example.

Java

```
public class KafkaOAuth2AuthenticateCallbackHandler implements AuthenticateCallbackHandler {

    private static final Duration ACCESS_TOKEN_REQUEST_BLOCK_TIME =
Duration.ofSeconds(30);
    private static final String TOKEN_AUDIENCE_FORMAT =
"%s://%s/.default";

    private Function<TokenCredential, Mono<OAuthBearerTokenImp>>
resolveToken;
    private final TokenCredential credential = new
DefaultAzureCredentialBuilder().build();

    @Override
    public void configure(Map<String, ?> configs, String mechanism,
List<AppConfigurationEntry> jaasConfigEntries) {
        TokenRequestContext request =
buildTokenRequestContext(configs);
        this.resolveToken = tokenCredential ->
tokenCredential.getToken(request).map(OAuthBearerTokenImp::new);
    }

    private TokenRequestContext buildTokenRequestContext(Map<String,
?> configs) {
        URI uri = buildEventHubsServerUri(configs);
        String tokenAudience = buildTokenAudience(uri);

        TokenRequestContext request = new TokenRequestContext();
        request.addScopes(tokenAudience);
        return request;
    }

    @SuppressWarnings("unchecked")
    private URI buildEventHubsServerUri(Map<String, ?> configs) {
        String bootstrapServer =
Arrays.asList(configs.get(BOOTSTRAP_SERVERS_CONFIG)).get(0).toString();
        bootstrapServer = bootstrapServer.replaceAll("\\\\[\\\\]", "");
        URI uri = URI.create("https://" + bootstrapServer);
        return uri;
    }

    private String buildTokenAudience(URI uri) {
        return String.format(TOKEN_AUDIENCE_FORMAT, uri.getScheme(),
uri.getHost());
    }
}
```

```

@Override
public void handle(Callback[] callbacks) throws
UnsupportedCallbackException {
    for (Callback callback : callbacks) {
        if (callback instanceof OAuthBearerTokenCallback) {
            OAuthBearerTokenCallback oauthCallback =
(OAuthBearerTokenCallback) callback;
            this.resolveToken
                .apply(credential)
                .doOnNext(oauthCallback::token)
                .doOnError(throwable ->
oauthCallback.error("invalid_grant", throwable.getMessage(), null))
                .block(ACCESS_TOKEN_REQUEST_BLOCK_TIME);
        } else {
            throw new UnsupportedCallbackException(callback);
        }
    }
}

@Override
public void close() {
    // NOOP
}
}

```

Java

```

public class OAuthBearerTokenImp implements OAuthBearerToken {
    private final AccessToken accessToken;
    private final JWTClaimsSet claims;

    public OAuthBearerTokenImp(AccessToken accessToken) {
        this.accessToken = accessToken;
        try {
            claims =
JWTParser.parse(accessToken.getToken()).getJWTClaimsSet();
        } catch (ParseException exception) {
            throw new SaslAuthenticationException("Unable to parse
the access token", exception);
        }
    }

    @Override
    public String value() {
        return accessToken.getToken();
    }

    @Override
    public Long startTimeMs() {
        return claims.getIssueTime().getTime();
    }

    @Override

```

```

        public long lifetimeMs() {
            return claims.getExpirationTime().getTime();
        }

        @Override
        public Set<String> scope() {
            // Referring to https://docs.microsoft.com/azure/active-
            // directory/develop/access-tokens#payload-claims, the scp
            // claim is a String which is presented as a space
            // separated list.
            return Optional.ofNullable(claims.getClaim("scp"))
                .map(s -> Arrays.stream((String) s)
                    .split(" "))
                .collect(Collectors.toSet())
                .orElse(null);
        }

        @Override
        public String principalName() {
            return (String) claims.getClaim("upn");
        }

        public boolean isExpired() {
            return accessToken.isExpired();
        }
    }
}

```

3. When you create your Kafka producer or consumer, add the configuration needed to support the [SASL/OAUTHBEARER](#) mechanism. The following examples show what your code should look like before and after migration. In both examples, replace the <eventhubs-namespace> placeholder with the name of your Event Hubs namespace.

Before migration, your code should look like the following example:

Java

```

Properties properties = new Properties();
properties.put(CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG, "<eventhubs-namespace>.servicebus.windows.net:9093");
properties.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
"SASL_SSL");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
properties.put(SaslConfigs.SASL_MECHANISM, "PLAIN");
properties.put(SaslConfigs.SASL_JAAS_CONFIG,
String.format("org.apache.kafka.common.security.plain.PlainLoginMod
ule required username=\"$ConnectionString\" password=\"%s\";",

```

```
connectionString));
return new KafkaProducer<>(properties);
```

After migration, your code should look like the following example. In this example, replace the `<path-to-your-KafkaOAuth2AuthenticateCallbackHandler>` placeholder with the full class name for your implemented `KafkaOAuth2AuthenticateCallbackHandler`.

Java

```
Properties properties = new Properties();
properties.put(CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG, "<eventhubs-namespace>.servicebus.windows.net:9093");
properties.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
"SASL_SSL");
properties.put(SaslConfigs.SASL_MECHANISM, "OAUTHBEARER");
properties.put(SaslConfigs.SASL_JAAS_CONFIG,
"org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required");
properties.put(SaslConfigs.SASL_LOGIN_CALLBACK_HANDLER_CLASS, "<path-to-your-KafkaOAuth2AuthenticateCallbackHandler>");
return new KafkaProducer<>(properties);
```

Run the app locally

After making these code changes, run your application locally. The new configuration should pick up your local credentials, assuming you're logged into a compatible IDE or command line tool, such as the Azure CLI, Visual Studio, or IntelliJ. The roles you assigned to your local dev user in Azure will allow your app to connect to the Azure service locally.

3) Configure the Azure hosting environment

After your application is configured to use passwordless connections and it runs locally, the same code can authenticate to Azure services after it's deployed to Azure. For example, an application deployed to an Azure Spring Apps instance that has a managed identity assigned can connect to Azure Event Hubs for Kafka.

In this section, you'll execute two steps to enable your application to run in an Azure hosting environment in a passwordless way:

- Assign the managed identity for your Azure hosting environment.
- Assign roles to the managed identity.

Note

Azure also provides **Service Connector**, which can help you connect your hosting service with Event Hubs. With Service Connector to configure your hosting environment, you can omit the step of assigning roles to your managed identity because Service Connector will do it for you. The following section describes how to configure your Azure hosting environment in two ways: one via Service Connector and the other by configuring each hosting environment directly.

Important

Service Connector's commands require [Azure CLI](#) 2.41.0 or higher.

Assign the managed identity for your Azure hosting environment

The following steps show you how to assign a system-assigned managed identity for various web hosting services. The managed identity can securely connect to other Azure Services using the app configurations you set up previously.

App Service

1. On the main overview page of your Azure App Service instance, select **Identity** from the navigation pane.
2. On the **System assigned** tab, make sure to set the **Status** field to **on**. A system assigned identity is managed by Azure internally and handles administrative tasks for you. The details and IDs of the identity are never exposed in your code.

The screenshot shows the Azure portal interface for managing the identity of an App Service. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Deployment (Quickstart, Deployment slots, Deployment Center), Settings (Configuration, Authentication, Application Insights), and Identity. The 'Identity' section is highlighted with a red box. The main content area shows the 'System assigned' tab selected (also highlighted with a red box). It includes a status switch (set to 'On'), an object ID (27a5a14c-9a0f-4f50-a82b-f022f74a8764), and a permissions section for 'Azure role assignments'. A note at the bottom indicates the resource is registered with Azure Active Directory.

You can also assign managed identity on an Azure hosting environment by using the Azure CLI.

App Service

You can assign a managed identity to an Azure App Service instance with the [az webapp identity assign](#) command, as shown in the following example.

Azure CLI

```
export AZURE_MANAGED_IDENTITY_ID=$(az webapp identity assign \
--resource-group $AZ_RESOURCE_GROUP \
--name <app-service-name> \
--query principalId \
--output tsv)
```

Assign roles to the managed identity

Next, grant permissions to the managed identity you created to access your Event Hubs namespace. You can grant permissions by assigning a role to the managed identity, just like you did with your local development user.

If you connected your services using the Service Connector, you don't need to complete this step. The following necessary configurations were handled for you:

- If you selected a managed identity when you created the connection, a system-assigned managed identity was created for your app and assigned the *Azure Event Hubs Data Sender* and *Azure Event Hubs Data Receiver* roles on the Event Hubs.
- If you chose to use a connection string, the connection string was added as an app environment variable.

Test the app

After making these code changes, browse to your hosted application in the browser. Your app should be able to connect to the Azure Event Hubs for Kafka successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Authorize access to blob data with managed identities for Azure resources](#)
- [Authorize access to blobs using Microsoft Entra ID](#)

Migrate an application to use passwordless connections with Azure Database for MySQL

Article • 10/19/2023

This article explains how to migrate from traditional authentication methods to more secure, passwordless connections with Azure Database for MySQL.

Application requests to Azure Database for MySQL must be authenticated. Azure Database for MySQL provides several different ways for apps to connect securely. One of the ways is to use passwords. However, you should prioritize passwordless connections in your applications when possible.

Compare authentication options

When the application authenticates with Azure Database for MySQL, it provides a username and password pair to connect to the database. Depending on where the identities are stored, there are two types of authentication: Microsoft Entra authentication and MySQL authentication.

Microsoft Entra authentication

Microsoft Entra authentication is a mechanism for connecting to Azure Database for MySQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

Using Microsoft Entra ID for authentication provides the following benefits:

- Authentication of users across Azure Services in a uniform way.
- Management of password policies and password rotation in a single place.
- Multiple forms of authentication supported by Microsoft Entra ID, which can eliminate the need to store passwords.
- Customers can manage database permissions using external (Microsoft Entra ID) groups.
- Microsoft Entra authentication uses MySQL database users to authenticate identities at the database level.
- Support of token-based authentication for applications connecting to Azure Database for MySQL.

MySQL authentication

You can create accounts in MySQL. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `user` table. Because these passwords are stored in MySQL, you need to manage the rotation of the passwords by yourself.

Although it's possible to connect to Azure Database for MySQL with passwords, you should use them with caution. You must be diligent to never expose the passwords in an unsecure location. Anyone who gains access to the passwords is able to authenticate. For example, there's a risk that a malicious user can access the application if a connection string is accidentally checked into source control, sent through an unsecure email, pasted into the wrong chat, or viewed by someone who shouldn't have permission. Instead, consider updating your application to use passwordless connections.

Introducing passwordless connections

With a passwordless connection, you can connect to Azure services without storing any credentials in the application code, its configuration files, or in environment variables.

Many Azure services support passwordless connections, for example via Azure Managed Identity. These techniques provide robust security features that you can implement using `DefaultAzureCredential` from the Azure Identity client libraries. In this tutorial, you'll learn how to update an existing application to use `DefaultAzureCredential` instead of alternatives such as connection strings.

`DefaultAzureCredential` supports multiple authentication methods and automatically determines which should be used at runtime. This approach enables your app to use different authentication methods in different environments (local dev vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` searches for credentials can be found in the [Azure Identity library overview](#). For example, when working locally, `DefaultAzureCredential` will generally authenticate using the account the developer used to sign in to Visual Studio. When the app is deployed to Azure, `DefaultAzureCredential` will automatically switch to use a [managed identity](#). No code changes are required for this transition.

To ensure that connections are passwordless, you must take into consideration both local development and the production environment. If a connection string is required in either place, then the application isn't passwordless.

In your local development environment, you can authenticate with Azure CLI, Azure PowerShell, Visual Studio, or Azure plugins for Visual Studio Code or IntelliJ. In this case, you can use that credential in your application instead of configuring properties.

When you deploy applications to an Azure hosting environment, such as a virtual machine, you can assign managed identity in that environment. Then, you won't need to provide credentials to connect to Azure services.

ⓘ Note

A managed identity provides a security identity to represent an app or service. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. You can read more about managed identities in the [overview](#) documentation.

Migrate an existing application to use passwordless connections

The following steps explain how to migrate an existing application to use passwordless connections instead of a password-based solution.

0) Prepare the working environment

First, use the following command to set up some environment variables.

Bash

```
export AZ_RESOURCE_GROUP=<YOUR_RESOURCE_GROUP>
export AZ_DATABASE_SERVER_NAME=<YOUR_DATABASE_SERVER_NAME>
export AZ_DATABASE_NAME=demo
export AZ_SQL_AD_NON_ADMIN_USERNAME=
<YOUR_AZURE_AD_NON_ADMIN_USER_DISPLAY_NAME>
export AZ_SQL_AD_MI_USERNAME=<YOUR_AZURE_AD_MI_DISPLAY_NAME>
export AZ_USER_IDENTITY_NAME=<YOUR_USER_ASSIGNED_MANAGED_IDENTITY_NAME>
export CURRENT_USERNAME=$(az ad signed-in-user show --query
userPrincipalName --output tsv)
export CURRENT_USER_OBJECTID=$(az ad signed-in-user show --query id --output
tsv)
```

Replace the placeholders with the following values, which are used throughout this article:

- <YOUR_RESOURCE_GROUP>: The name of the resource group your resources are in.

- <YOUR_DATABASE_SERVER_NAME>: The name of your MySQL server, which should be unique across Azure.
- <YOUR_AZURE_AD_NON_ADMIN_USER_DISPLAY_NAME>: The display name of your Microsoft Entra non-admin user. Make sure the name is a valid user in your Microsoft Entra tenant.
- <YOUR_AZURE_AD_MI_DISPLAY_NAME>: The display name of Microsoft Entra user for your managed identity. Make sure the name is a valid user in your Microsoft Entra tenant.
- <YOUR_USER_ASSIGNED_MANAGED_IDENTITY_NAME>: The name of your user-assigned managed identity server, which should be unique across Azure.

1) Configure Azure Database for MySQL

1.1) Enable Microsoft Entra ID-based authentication

To use Microsoft Entra ID access with Azure Database for MySQL, you should set the Microsoft Entra admin user first. Only a Microsoft Entra Admin user can create/enable users for Microsoft Entra ID-based authentication.

If you're using Azure CLI, run the following command to make sure it has sufficient permission:

Bash

```
az login --scope https://graph.microsoft.com/.default
```

Run the following command to the create user identity for assigning:

Azure CLI

```
az identity create \
--resource-group $AZ_RESOURCE_GROUP \
--name $AZ_USER_IDENTITY_NAME
```

ⓘ Important

After creating the user-assigned identity, ask your *Global Administrator* or *Privileged Role Administrator* to grant the following permissions for this identity: `User.Read.All`, `GroupMember.Read.All`, and `Application.Read.ALL`. For more information, see the [Permissions](#) section of [Active Directory authentication](#).

Run the following command to assign the identity to the MySQL server for creating the Microsoft Entra admin:

```
Azure CLI

az mysql flexible-server identity assign \
--resource-group $AZ_RESOURCE_GROUP \
--server-name $AZ_DATABASE_SERVER_NAME \
--identity $AZ_USER_IDENTITY_NAME
```

Then, run following command to set the Microsoft Entra admin:

```
Azure CLI

az mysql flexible-server ad-admin create \
--resource-group $AZ_RESOURCE_GROUP \
--server-name $AZ_DATABASE_SERVER_NAME \
--display-name $CURRENT_USERNAME \
--object-id $CURRENT_USER_OBJECTID \
--identity $AZ_USER_IDENTITY_NAME
```

This command will set the Microsoft Entra admin to the current signed-in user.

 **Note**

You can only create one Microsoft Entra admin per MySQL server. Selection of another one will overwrite the existing Microsoft Entra admin configured for the server.

2) Configure Azure Database for MySQL for local development

2.1) Configure a firewall rule for local IP

Azure Database for MySQL instances are secured by default. They have a firewall that doesn't allow any incoming connection.

You can skip this step if you're using Bash because the `flexible-server create` command already detected your local IP address and set it on MySQL server.

If you're connecting to your MySQL server from Windows Subsystem for Linux (WSL) on a Windows computer, you need to add the WSL host ID to your firewall. Obtain the IP address of your host machine by running the following command in WSL:

```
Bash
```

```
cat /etc/resolv.conf
```

Copy the IP address following the term `nameserver`, then use the following command to set an environment variable for the WSL IP address:

```
Bash
```

```
export AZ_WSL_IP_ADDRESS=<the-copied-IP-address>
```

Then, use the following command to open the server's firewall to your WSL-based app:

```
Azure CLI
```

```
az mysql server firewall-rule create \
--resource-group $AZ_RESOURCE_GROUP \
--name $AZ_DATABASE_SERVER_NAME-database-allow-local-ip-wsl \
--server $AZ_DATABASE_SERVER_NAME \
--start-ip-address $AZ_WSL_IP_ADDRESS \
--end-ip-address $AZ_WSL_IP_ADDRESS \
--output tsv
```

2.2) Create a MySQL non-admin user and grant permission

Next, create a non-admin Microsoft Entra user and grant all permissions on the `$AZ_DATABASE_NAME` database to it. You can change the database name `$AZ_DATABASE_NAME` to fit your needs.

Create a SQL script called `create_ad_user.sql` for creating a non-admin user. Add the following contents and save it locally:

```
Bash
```

```
export AZ_MYSQL_AD_NON_ADMIN_USERID=$(az ad signed-in-user show --query id --output tsv)

cat << EOF > create_ad_user.sql
SET aad_auth_validate_oids_in_tenant = OFF;
CREATE AADUSER '$AZ_MYSQL_AD_NON_ADMIN_USERNAME' IDENTIFIED BY
'$AZ_MYSQL_AD_NON_ADMIN_USERID';
GRANT ALL PRIVILEGES ON $AZ_DATABASE_NAME.* TO
'$AZ_MYSQL_AD_NON_ADMIN_USERNAME'@'%';
FLUSH privileges;
EOF
```

Then, use the following command to run the SQL script to create the Microsoft Entra non-admin user:

Bash

```
mysql -h $AZ_DATABASE_SERVER_NAME.mysql.database.azure.com --user  
$CURRENT_USERNAME --enable-cleartext-plugin --password=$(az account get-  
access-token --resource-type oss-rdbms --output tsv --query accessToken) <  
create_ad_user.sql
```

Now use the following command to remove the temporary SQL script file:

Bash

```
rm create_ad_user.sql
```

ⓘ Note

You can read more detailed information about creating MySQL users in [Create users in Azure Database for MySQL](#).

3) Sign in and migrate the app code to use passwordless connections

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to on your MySQL. You can authenticate via the Azure CLI, Visual Studio, Azure PowerShell, or other tools such as IntelliJ.

Azure CLI

Sign in to Azure through the Azure CLI by using the following command:

Azure CLI

```
az login
```

Next, use the following steps to update your code to use passwordless connections. Although conceptually similar, each language uses different implementation details.

Java

1. Inside your project, add the following reference to the `azure-identity-extensions` package. This library contains all of the entities necessary to implement passwordless connections.

XML

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-identity-extensions</artifactId>
    <version>1.0.0</version>
</dependency>
```

2. Enable the Azure MySQL authentication plugin in the JDBC URL. Identify the locations in your code that currently create a `java.sql.Connection` to connect to Azure Database for MySQL. Update `url` and `user` in your `application.properties` file to match the following values:

properties

```
url=jdbc:mysql://$AZ_DATABASE_SERVER_NAME.mysql.database.azure.com:3306/$AZ_DATABASE_NAME?
serverTimezone=UTC&sslMode=REQUIRED&defaultAuthenticationPlugin=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin&authenticationPlugins=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin
user=$AZ_MYSQL_AD_NON_ADMIN_USERNAME
```

ⓘ Note

If you're using the `MysqlConnectionPoolDataSource` class as the datasource in your application, be sure to remove

```
defaultAuthenticationPlugin=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin
```

 from the URL.

properties

```
url=jdbc:mysql://$AZ_DATABASE_SERVER_NAME.mysql.database.azure.com:3306/$AZ_DATABASE_NAME?
serverTimezone=UTC&sslMode=REQUIRED&authenticationPlugins=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin
user=$AZ_MYSQL_AD_NON_ADMIN_USERNAME
```

3. Replace the one `$AZ_DATABASE_SERVER_NAME` variable, one `$AZ_DATABASE_NAME` variable and one `$AZ_MYSQL_AD_NON_ADMIN_USERNAME` variable with the values that you configured at the beginning of this article.

4. Remove the `password` from the JDBC URL.

Run the app locally

After making these code changes, run your application locally. The new configuration should pick up your local credentials if you're signed in to a compatible IDE or command line tool, such as the Azure CLI, Visual Studio, or IntelliJ. The roles you assigned to your local dev user in Azure will allow your app to connect to the Azure service locally.

4) Configure the Azure hosting environment

After your application is configured to use passwordless connections and it runs locally, the same code can authenticate to Azure services after it's deployed to Azure. For example, an application deployed to an Azure App Service instance that has a managed identity assigned can connect to Azure Storage.

In this section, you'll execute two steps to enable your application to run in an Azure hosting environment in a passwordless way:

- Assign the managed identity for your Azure hosting environment.
- Assign roles to the managed identity.

ⓘ Note

Azure also provides **Service Connector**, which can help you connect your hosting service with PostgreSQL. With Service Connector to configure your hosting environment, you can omit the step of assigning roles to your managed identity because Service Connector will do it for you. The following section describes how to configure your Azure hosting environment in two ways: one via Service Connector and the other by configuring each hosting environment directly.

ⓘ Important

Service Connector's commands require **Azure CLI** 2.41.0 or higher.

Assign the managed identity using the Azure portal

The following steps show you how to assign a system-assigned managed identity for various web hosting services. The managed identity can securely connect to other Azure Services using the app configurations you set up previously.

App Service

1. On the main overview page of your Azure App Service instance, select **Identity** from the navigation pane.

2. On the **System assigned** tab, make sure to set the **Status** field to **on**. A system assigned identity is managed by Azure internally and handles administrative tasks for you. The details and IDs of the identity are never exposed in your code.

You can also assign managed identity on an Azure hosting environment by using the Azure CLI.

App Service

You can assign a managed identity to an Azure App Service instance with the [az webapp identity assign](#) command, as shown in the following example:

```
Azure CLI

export AZ_MI_OBJECT_ID=$(az webapp identity assign \
    --resource-group $AZ_RESOURCE_GROUP \
    --name <service-instance-name> \
    --query principalId \
    --output tsv)
```

Assign roles to the managed identity

Next, grant permissions to the managed identity you assigned to access your MySQL instance.

These steps will create a Microsoft Entra user for the managed identity and grant all permissions for the database `$AZ_DATABASE_NAME` to it. You can change the database name `$AZ_DATABASE_NAME` to fit your needs.

First, create a SQL script called `create_ad_user.sql` for creating a non-admin user. Add the following contents and save it locally:

```
Bash

export AZ_MYSQL_AD_MI_USERID=$(az ad sp show --id $AZ_MI_OBJECT_ID --query
appId --output tsv)

cat << EOF > create_ad_user.sql
SET aad_auth_validate_oids_in_tenant = OFF;
CREATE AADUSER '$AZ_MYSQL_AD_MI_USERNAME' IDENTIFIED BY
'$AZ_MYSQL_AD_MI_USERID';
GRANT ALL PRIVILEGES ON $AZ_DATABASE_NAME.* TO
'$AZ_MYSQL_AD_MI_USERNAME'@'%';
FLUSH privileges;
EOF
```

Then, use the following command to run the SQL script to create the Microsoft Entra non-admin user:

```
Bash

mysql -h $AZ_DATABASE_SERVER_NAME.mysql.database.azure.com --user
$CURRENT_USERNAME --enable-cleartext-plugin --password=$(az account get-
```

```
access-token --resource-type oss-rdbms --output tsv --query accessToken) <
create_ad_user.sql
```

Now use the following command to remove the temporary SQL script file:

Bash

```
rm create_ad_user.sql
```

Test the app

Before deploying the app to the hosting environment, you need to make one more change to the code because the application is going to connect to MySQL using the user created for the managed identity.

Java

Update your code to use the user created for the managed identity:

Java

```
properties.put("user", "$AZ_MYSQL_AD_MI_USERNAME");
```

After making these code changes, you can build and redeploy the application. Then, browse to your hosted application in the browser. Your app should be able to connect to the MySQL database successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Authorize access to blob data with managed identities for Azure resources](#).
- [Authorize access to blobs using Microsoft Entra ID](#)

Migrate an application to use passwordless connections with Azure Database for PostgreSQL

Article • 10/19/2023

This article explains how to migrate from traditional authentication methods to more secure, passwordless connections with Azure Database for PostgreSQL.

Application requests to Azure Database for PostgreSQL must be authenticated. Azure Database for PostgreSQL provides several different ways for apps to connect securely. One of the ways is to use passwords. However, you should prioritize passwordless connections in your applications when possible.

Compare authentication options

When the application authenticates with Azure Database for PostgreSQL, it provides a username and password pair to connect the database. Depending on where the identities are stored, there are two types of authentication: Microsoft Entra authentication and PostgreSQL authentication.

Microsoft Entra authentication

Microsoft Entra authentication is a mechanism for connecting to Azure Database for PostgreSQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

Using Microsoft Entra ID for authentication provides the following benefits:

- Authentication of users across Azure Services in a uniform way.
- Management of password policies and password rotation in a single place.
- Multiple forms of authentication supported by Microsoft Entra ID, which can eliminate the need to store passwords.
- Customers can manage database permissions using external (Microsoft Entra ID) groups.
- Microsoft Entra authentication uses PostgreSQL database users to authenticate identities at the database level.
- Support of token-based authentication for applications connecting to Azure Database for PostgreSQL.

PostgreSQL authentication

You can create accounts in PostgreSQL. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `user` table. Because these passwords are stored in PostgreSQL, you need to manage the rotation of the passwords by yourself.

Although it's possible to connect to Azure Database for PostgreSQL with passwords, you should use them with caution. You must be diligent to never expose the passwords in an unsecure location. Anyone who gains access to the passwords is able to authenticate. For example, there's a risk that a malicious user can access the application if a connection string is accidentally checked into source control, sent through an unsecure email, pasted into the wrong chat, or viewed by someone who shouldn't have permission. Instead, consider updating your application to use passwordless connections.

Introducing passwordless connections

With a passwordless connection, you can connect to Azure services without storing any credentials in the application code, its configuration files, or in environment variables.

Many Azure services support passwordless connections, for example via Azure Managed Identity. These techniques provide robust security features that you can implement using `DefaultAzureCredential` from the Azure Identity client libraries. In this tutorial, you'll learn how to update an existing application to use `DefaultAzureCredential` instead of alternatives such as connection strings.

`DefaultAzureCredential` supports multiple authentication methods and automatically determines which should be used at runtime. This approach enables your app to use different authentication methods in different environments (local dev vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` searches for credentials can be found in the [Azure Identity library overview](#). For example, when working locally, `DefaultAzureCredential` will generally authenticate using the account the developer used to sign in to Visual Studio. When the app is deployed to Azure, `DefaultAzureCredential` will automatically switch to use a [managed identity](#). No code changes are required for this transition.

To ensure that connections are passwordless, you must take into consideration both local development and the production environment. If a connection string is required in either place, then the application isn't passwordless.

In your local development environment, you can authenticate with Azure CLI, Azure PowerShell, Visual Studio, or Azure plugins for Visual Studio Code or IntelliJ. In this case, you can use that credential in your application instead of configuring properties.

When you deploy applications to an Azure hosting environment, such as a virtual machine, you can assign managed identity in that environment. Then, you won't need to provide credentials to connect to Azure services.

ⓘ Note

A managed identity provides a security identity to represent an app or service. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. You can read more about managed identities in the [overview](#) documentation.

Migrate an existing application to use passwordless connections

The following steps explain how to migrate an existing application to use passwordless connections instead of a password-based solution.

0) Prepare the working environment

First, use the following command to set up some environment variables.

Bash

```
export AZ_RESOURCE_GROUP=<YOUR_RESOURCE_GROUP>
export AZ_DATABASE_SERVER_NAME=<YOUR_DATABASE_SERVER_NAME>
export AZ_DATABASE_NAME=demo
export AZ_POSTGRESQL_AD_NON_ADMIN_USERNAME=
<YOUR_AZURE_AD_NON_ADMIN_USER_DISPLAY_NAME>
export AZ_LOCAL_IP_ADDRESS=<YOUR_LOCAL_IP_ADDRESS>
export CURRENT_USERNAME=$(az ad signed-in-user show --query
userPrincipalName --output tsv)
```

Replace the placeholders with the following values, which are used throughout this article:

- <YOUR_RESOURCE_GROUP>: The name of the resource group your resources are in.
- <YOUR_DATABASE_SERVER_NAME>: The name of your PostgreSQL server. It should be unique across Azure.

- <YOUR_AZURE_AD_NON_ADMIN_USER_DISPLAY_NAME>: The display name of your Microsoft Entra non-admin user. Make sure the name is a valid user in your Microsoft Entra tenant.
- <YOUR_LOCAL_IP_ADDRESS>: The IP address of your local computer, from which you'll run your Spring Boot application. One convenient way to find it is to open whatismyip.akamai.com.

1) Configure Azure Database for PostgreSQL

1.1) Enable Microsoft Entra ID-based authentication

To use Microsoft Entra ID access with Azure Database for PostgreSQL, you should set the Microsoft Entra admin user first. Only a Microsoft Entra Admin user can create/enable users for Microsoft Entra ID-based authentication.

To set up a Microsoft Entra administrator after creating the server, follow the steps in [Manage Microsoft Entra roles in Azure Database for PostgreSQL - Flexible Server](#).

ⓘ Note

PostgreSQL Flexible Server can create multiple Microsoft Entra administrators.

2) Configure Azure Database for PostgreSQL for local development

2.1) Configure a firewall rule for local IP

Azure Database for PostgreSQL instances are secured by default. They have a firewall that doesn't allow any incoming connection. To be able to use your database, you need to add a firewall rule that will allow the local IP address to access the database server.

Because you configured your local IP address at the beginning of this article, you can open the server's firewall by running the following command:

Azure CLI

```
az postgres flexible-server firewall-rule create \
    --resource-group $AZ_RESOURCE_GROUP \
    --name $AZ_DATABASE_SERVER_NAME \
    --rule-name $AZ_DATABASE_SERVER_NAME-database-allow-local-ip \
    --start-ip-address $AZ_LOCAL_IP_ADDRESS \
```

```
--end-ip-address $AZ_LOCAL_IP_ADDRESS \
--output tsv
```

If you're connecting to your PostgreSQL server from Windows Subsystem for Linux (WSL) on a Windows computer, you need to add the WSL host ID to your firewall.

Obtain the IP address of your host machine by running the following command in WSL:

Bash

```
cat /etc/resolv.conf
```

Copy the IP address following the term `nameserver`, then use the following command to set an environment variable for the WSL IP Address:

Bash

```
export AZ_WSL_IP_ADDRESS=<the-copied-IP-address>
```

Then, use the following command to open the server's firewall to your WSL-based app:

Azure CLI

```
az postgres flexible-server firewall-rule create \
--resource-group $AZ_RESOURCE_GROUP \
--name $AZ_DATABASE_SERVER_NAME \
--rule-name $AZ_DATABASE_SERVER_NAME-database-allow-local-ip \
--start-ip-address $AZ_WSL_IP_ADDRESS \
--end-ip-address $AZ_WSL_IP_ADDRESS \
--output tsv
```

2.2) Create a PostgreSQL non-admin user and grant permission

Next, create a non-admin Microsoft Entra user and grant all permissions on the `$AZ_DATABASE_NAME` database to it. You can change the database name `$AZ_DATABASE_NAME` to fit your needs.

Create a SQL script called `create_ad_user_local.sql` for creating a non-admin user. Add the following contents and save it locally:

Bash

```
cat << EOF > create_ad_user_local.sql
select * from
pgadauth_create_principal('$AZ_POSTGRESQL_AD_NON_ADMIN_USERNAME', false,
```

```
false);  
EOF
```

Then, use the following command to run the SQL script to create the Microsoft Entra non-admin user:

Bash

```
psql "host=$AZ_DATABASE_SERVER_NAME.postgres.database.azure.com  
user=$CURRENT_USERNAME dbname=postgres port=5432 password=$(az account get-  
access-token --resource-type oss-rdbms --output tsv --query accessToken)  
sslmode=require" < create_ad_user_local.sql
```

Now use the following command to remove the temporary SQL script file:

Bash

```
rm create_ad_user_local.sql
```

Note

You can read more detailed information about creating PostgreSQL users in [Create users in Azure Database for PostgreSQL](#).

3) Sign in and migrate the app code to use passwordless connections

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to on your PostgreSQL. You can authenticate via the Azure CLI, Visual Studio, Azure PowerShell, or other tools such as IntelliJ.

Azure CLI

Sign in to Azure through the Azure CLI by using the following command:

Azure CLI

```
az login
```

Next, use the following steps to update your code to use passwordless connections. Although conceptually similar, each language uses different implementation details.

1. Inside your project, add the following reference to the `azure-identity-extensions` package. This library contains all of the entities necessary to implement passwordless connections.

XML

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-identity-extensions</artifactId>
    <version>1.0.0</version>
</dependency>
```

2. Enable the Azure PostgreSQL authentication plugin in JDBC URL. Identify the locations in your code that currently create a `java.sql.Connection` to connect to Azure Database for PostgreSQL. Update `url` and `user` in your `application.properties` file to match the following values:

properties

```
url=jdbc:postgresql://$AZ_DATABASE_SERVER_NAME.postgres.database.azure.com:5432/$AZ_DATABASE_NAME?
sslmode=require&authenticationPluginClassName=com.azure.identity.extensions.jdbc.postgresql.AzurePostgresqlAuthenticationPlugin
user=$AZ_POSTGRESQL_AD_NON_ADMIN_USERNAME
```

3. Replace the `$AZ_POSTGRESQL_AD_NON_ADMIN_USERNAME` and the two `$AZ_DATABASE_SERVER_NAME` variables with the value that you configured at the beginning of this article.

Run the app locally

After making these code changes, run your application locally. The new configuration should pick up your local credentials if you're signed in to a compatible IDE or command line tool, such as the Azure CLI, Visual Studio, or IntelliJ. The roles you assigned to your local dev user in Azure will allow your app to connect to the Azure service locally.

4) Configure the Azure hosting environment

After your application is configured to use passwordless connections and it runs locally, the same code can authenticate to Azure services after it's deployed to Azure. For example, an application deployed to an Azure App Service instance that has a managed identity assigned can connect to Azure Storage.

In this section, you'll execute two steps to enable your application to run in an Azure hosting environment in a passwordless way:

- Assign the managed identity for your Azure hosting environment.
- Assign roles to the managed identity.

Note

Azure also provides **Service Connector**, which can help you connect your hosting service with PostgreSQL. With Service Connector to configure your hosting environment, you can omit the step of assigning roles to your managed identity because Service Connector will do it for you. The following section describes how to configure your Azure hosting environment in two ways: one via Service Connector and the other by configuring each hosting environment directly.

Important

Service Connector's commands require [Azure CLI](#) 2.41.0 or higher.

Assign the managed identity using the Azure portal

The following steps show you how to assign a system-assigned managed identity for various web hosting services. The managed identity can securely connect to other Azure Services using the app configurations you set up previously.

App Service

1. On the main overview page of your Azure App Service instance, select **Identity** from the navigation pane.
2. On the **System assigned** tab, make sure to set the **Status** field to **on**. A system assigned identity is managed by Azure internally and handles administrative tasks for you. The details and IDs of the identity are never exposed in your code.

The screenshot shows the Azure portal interface for managing the identity of an App Service. The left sidebar lists various settings like Overview, Activity log, and Deployment. The main panel is titled 'Identity' and shows two tabs: 'System assigned' (which is selected and highlighted with a red box) and 'User assigned'. Below the tabs, it says 'A system assigned managed identity is restricted to one per resource and is tied to the lifecycle of this resource. Have to store any credentials in code. Learn more about Managed identities.' There are buttons for Save, Discard, Refresh, and Got feedback? A status switch is set to 'On' (also highlighted with a red box). The 'Object (principal) ID' is listed as '27a5a14c-9a0f-4f50-a82b-f022f74a8764'. A note at the bottom states: 'This resource is registered with Azure Active Directory. The managed identity can be configured to allow access to'.

You can also assign managed identity on an Azure hosting environment by using the Azure CLI.

App Service

You can assign a managed identity to an Azure App Service instance with the [az webapp identity assign](#) command, as shown in the following example:

Azure CLI

```
export AZ_MI_OBJECT_ID=$(az webapp identity assign \
    --resource-group $AZ_RESOURCE_GROUP \
    --name <service-instance-name> \
    --query principalId \
    --output tsv)
```

Assign roles to the managed identity

Next, grant permissions to the managed identity you assigned to access your PostgreSQL instance.

Service Connector

If you connected your services using Service Connector, the previous step's commands already assigned the role, so you can skip this step.

Test the app

Before deploying the app to the hosting environment, you need to make one more change to the code because the application is going to connect to PostgreSQL using the user created for the managed identity.

Java

Update your code to use the user created for the managed identity:

ⓘ Note

If you used the Service Connector command, skip this step.

Java

```
properties.put("user", "$AZ_POSTGRESQL_AD_MI_USERNAME");
```

After making these code changes, you can build and redeploy the application. Then, browse to your hosted application in the browser. Your app should be able to connect to the PostgreSQL database successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Authorize access to blob data with managed identities for Azure resources](#).
- [Authorize access to blobs using Microsoft Entra ID](#)

Migrate an application to use passwordless connections with Azure Service Bus

Article • 06/12/2023

Application requests to Azure Service Bus must be authenticated using either account access keys or passwordless connections. However, you should prioritize passwordless connections in your applications when possible. This tutorial explores how to migrate from traditional authentication methods to more secure, passwordless connections.

Security risks associated with access keys

The following code example demonstrates how to connect to Azure Service Bus using a connection string that includes an access key. When you create a Service Bus, Azure generates these keys and connection strings automatically. Many developers gravitate towards this solution because it feels familiar to options they've worked with in the past. If your application currently uses connection strings, consider migrating to passwordless connections using the steps described in this document.

.NET

C#

```
await using ServiceBusClient client = new("<CONNECTION-STRING>");
```

Connection strings should be used with caution. Developers must be diligent to never expose the keys in an unsecure location. Anyone who gains access to the key is able to authenticate. For example, if an account key is accidentally checked into source control, sent through an unsecure email, pasted into the wrong chat, or viewed by someone who shouldn't have permission, there's risk of a malicious user accessing the application. Instead, consider updating your application to use passwordless connections.

Migrate to passwordless connections

Many Azure services support passwordless connections through Microsoft Entra ID and Role Based Access control (RBAC). These techniques provide robust security features

and can be implemented using `DefaultAzureCredential` from the Azure Identity client libraries.

ⓘ Important

Some languages must implement `DefaultAzureCredential` explicitly in their code, while others utilize `DefaultAzureCredential` internally through underlying plugins or drivers.

`DefaultAzureCredential` supports multiple authentication methods and automatically determines which should be used at runtime. This approach enables your app to use different authentication methods in different environments (local dev vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` searches for credentials can be found in the [Azure Identity library overview](#) and varies between languages. For example, when working locally with .NET, `DefaultAzureCredential` will generally authenticate using the account the developer used to sign-in to Visual Studio, Azure CLI, or Azure PowerShell. When the app is deployed to Azure, `DefaultAzureCredential` will automatically discover and use the [managed identity](#) of the associated hosting service, such as Azure App Service. No code changes are required for this transition.

ⓘ Note

A managed identity provides a security identity to represent an app or service. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. You can read more about managed identities in the [overview](#) documentation.

The following code example demonstrates how to connect to Service Bus using passwordless connections. The next section describes how to migrate to this setup for a specific service in more detail.

A .NET application can pass an instance of `DefaultAzureCredential` into the constructor of a service client class. `DefaultAzureCredential` will automatically discover the credentials that are available in that environment.

C#

```
ServiceBusClient serviceBusClient = new(  
    new Uri($"https://{{serviceBusNamespace}}.blob.core.windows.net"),
```

```
new DefaultAzureCredential());
```

Steps to migrate an app to use passwordless authentication

The following steps explain how to migrate an existing application to use passwordless connections instead of a key-based solution. You'll first configure a local development environment, and then apply those concepts to an Azure app hosting environment. These same migration steps should apply whether you're using access keys directly, or through connection strings.

Configure roles and users for local development authentication

When developing locally, make sure that the user account that is accessing Service Bus has the correct permissions. In this example you'll use the **Azure Service Bus Data Owner** role to send and receive data, though more granular roles are also available. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account scoped to a specific Service Bus namespace, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Azure Service Bus Data Owner** role to your user account, which allows you to send and receive data.

Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. In the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the Service Bus overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'Access control (IAM)' page for a Service Bus namespace named 'spsbusns11102'. The left sidebar has 'Access control (IAM)' selected. The top navigation bar includes a search bar, a '+ Add' button (highlighted with a red box and yellow circle 1), a 'Download role assignments' button (highlighted with a yellow circle 2), and a 'Get feedback?' link. A tooltip for the 'Add role assignment' button (highlighted with a yellow circle 3) states: 'To read this customer directory, so some options are disabled. If you require access to these options, add co-administrator.' Below the top bar are buttons for 'Edit columns', 'Refresh', 'Remove', and 'Check access'. The 'Role assignments' tab is selected. The main area contains sections for 'My access', 'Check access', 'Grant access to this resource', and 'View access to this resource'. The 'Add role assignment' button is visible in the 'Grant access to this resource' section.

5. Use the search box to filter the results to the desired role. For this example, search for *Azure Service Bus Data Owner* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Sign-in and migrate the app code to use passwordless connections

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to for the Service Bus namespace. You can authenticate via the Azure CLI, Visual Studio, Azure PowerShell, or other tools such as IntelliJ.

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

```
Azure CLI
```

```
az login
```

Next, update your code to use passwordless connections.

.NET

1. To use `DefaultAzureCredential` in a .NET application, install the `Azure.Identity` package:

```
.NET CLI
```

```
dotnet add package Azure.Identity
```

2. At the top of your file, add the following code:

```
C#
```

```
using Azure.Identity;
```

3. Identify the code that creates a `ServiceBusClient` object to connect to Azure Service Bus. Update your code to match the following example:

```
C#
```

```
var serviceBusNamespace =
"https://{namespace}.servicebus.windows.net";
ServiceBusClient client = new(
    serviceBusNamespace,
    new DefaultAzureCredential());
```

Run the app locally

After making these code changes, run your application locally. The new configuration should pick up your local credentials, such as the Azure CLI, Visual Studio, or IntelliJ. The roles you assigned to your local dev user in Azure will allow your app to connect to the Azure service locally.

Configure the Azure hosting environment

Once your application is configured to use passwordless connections and runs locally, the same code can authenticate to Azure services after it's deployed to Azure. For example, an application deployed to an Azure App Service instance that has a managed identity enabled can connect to Azure Service Bus.

Create the managed identity using the Azure portal

The following steps demonstrate how to create a system-assigned managed identity for various web hosting services. The managed identity can securely connect to other Azure Services using the app configurations you set up previously.

Service Connector

Some app hosting environments support Service Connector, which helps you connect Azure compute services to other backing services. Service Connector automatically configures network settings and connection information. You can learn more about Service Connector and which scenarios are supported on the [overview page](#).

The following compute services are currently supported:

- Azure App Service
- Azure Spring Cloud
- Azure Container Apps (preview)

For this migration guide you'll use App Service, but the steps are similar on Azure Spring Apps and Azure Container Apps.

 **Note**

Azure Spring Apps currently only supports Service Connector using connection strings.

1. On the main overview page of your App Service, select **Service Connector** from the left navigation.
2. Select **+ Create** from the top menu and the **Create connection** panel will open. Enter the following values:
 - **Service type:** Choose **Service bus**.
 - **Subscription:** Select the subscription you would like to use.
 - **Connection Name:** Enter a name for your connection, such as *connector_appservice_servicebus*.
 - **Client type:** Leave the default value selected or choose the specific client you'd like to use.

Select **Next: Authentication**.

3. Make sure **System assigned managed identity (Recommended)** is selected, and then choose **Next: Networking**.
4. Leave the default values selected, and then choose **Next: Review + Create**.
5. After Azure validates your settings, select **Create**.

The Service Connector will automatically create a system-assigned managed identity for the app service. The connector will also assign the managed identity a **Azure Service Bus Data Owner** role for the service bus you selected.

Alternatively, you can also enable managed identity on an Azure hosting environment using the Azure CLI.

 **Service Connector**

You can use Service Connector to create a connection between an Azure compute hosting environment and a target service using the Azure CLI. The CLI automatically handles creating a managed identity and assigns the proper role, as explained in the [portal instructions](#).

If you're using an Azure App Service, use the `az webapp connection` command:

Azure CLI

```
az webapp connection create servicebus \
--resource-group <resource-group-name> \
--name <webapp-name> \
--target-resource-group <target-resource-group-name> \
--namespace <target-service-bus-namespace> \
--system-identity
```

If you're using Azure Spring Apps, use the `az spring connection` command:

Azure CLI

```
az spring connection create servicebus \
--resource-group <resource-group-name> \
--service <service-instance-name> \
--app <app-name> \
--deployment <deployment-name> \
--target-resource-group <target-resource-group> \
--namespace <target-service-bus-namespace> \
--system-identity
```

If you're using Azure Container Apps, use the `az containerapp connection` command:

Azure CLI

```
az containerapp connection create servicebus \
--resource-group <resource-group-name> \
--name <webapp-name> \
--target-resource-group <target-resource-group-name> \
--namespace <target-service-bus-namespace> \
--system-identity
```

Assign roles to the managed identity

Next, you need to grant permissions to the managed identity you created to access your Service Bus. You can do this by assigning a role to the managed identity, just like you did with your local development user.

Service Connector

If you connected your services using the Service Connector you don't need to complete this step. The necessary configurations were handled for you:

- If you selected a managed identity while creating the connection, a system-assigned managed identity was created for your app and assigned the **Azure Service Bus Data Owner** role on the Service Bus.
- If you selected connection string, the connection string was added as an app environment variable.

Test the app

After making these code changes, browse to your hosted application in the browser. Your app should be able to connect to the Service Bus successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

Migrate an application to use passwordless connections with Azure Blob Storage

Article • 05/10/2023

Application requests to Azure services must be authenticated using configurations such as account access keys or passwordless connections. However, you should prioritize passwordless connections in your applications when possible. Traditional authentication methods that use passwords or secret keys create security risks and complications. Visit the [passwordless connections for Azure services](#) hub to learn more about the advantages of moving to passwordless connections.

The following tutorial explains how to migrate an existing application to connect using passwordless connections. These same migration steps should apply whether you're using access keys, connection strings, or another secrets-based approach.

Configure roles and users for local development authentication

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

 **Important**

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'Access Control (IAM)' page for a storage account named 'identitymigrationstorage'. The left sidebar has 'Access Control (IAM)' selected. The top navigation bar includes a search bar, 'Add' (highlighted with a red box), 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. Below the navigation is a table with columns: 'Add', 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. A sub-menu for 'Add' shows 'Add role assignment' (highlighted with a red box) and 'Add co-administrator'. The main content area has sections for 'My access', 'Check access', and 'Grant access to this resource'. The 'Grant access to this resource' section contains a 'Add role assignment' button (highlighted with a red box) and a 'Learn more' link. The 'View deny assignments' section also has a 'View' button and a 'Learn more' link.

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.

8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Sign-in and migrate the app code to use passwordless connections

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

```
Azure CLI
```

```
az login
```

Next, update your code to use passwordless connections.

.NET

1. To use `DefaultAzureCredential` in a .NET application, install the `Azure.Identity` package:

```
.NET CLI
```

```
dotnet add package Azure.Identity
```

2. At the top of your file, add the following code:

```
C#
```

```
using Azure.Identity;
```

3. Identify the locations in your code that create a `BlobServiceClient` to connect to Azure Blob Storage. Update your code to match the following example:

C#

```
DefaultAzureCredential credential = new();

BlobServiceClient blobServiceClient = new(
    new Uri($"https://{{storageAccountName}}.blob.core.windows.net"),
    credential);
```

4. Make sure to update the storage account name in the URI of your `BlobServiceClient`. You can find the storage account name on the overview page of the Azure portal.

The screenshot shows the Azure Storage Account Overview page for a storage account named "identitymigrationstorage". The account type is listed as "Storage account". The left sidebar includes links for Overview, Activity log, Tags, Diagnose and solve problems, Access Control (IAM), Data migration, Events, and Storage browser. The main content area displays the following details under the "Essentials" section:

Setting	Value
Resource group (move)	: alexw-identity-revamp
Location	: East US
Primary/Secondary Location	: Primary: East US, Secondary: West US
Subscription (move)	: C&L Cross Service Content Team Testing
Subscription ID	:
Disk state	: Primary: Available, Secondary: Available
Tags (edit)	: Click here to add tags

Run the app locally

After making these code changes, run your application locally. The new configuration should pick up your local credentials, such as the Azure CLI, Visual Studio, or IntelliJ. The roles you assigned to your local dev user in Azure allows your app to connect to the Azure service locally.

Configure the Azure hosting environment

Once your application is configured to use passwordless connections and runs locally, the same code can authenticate to Azure services after it's deployed to Azure. The sections that follow explain how to configure a deployed application to connect to Azure Blob Storage using a managed identity.

Create the managed identity

You can create a user-assigned managed identity using the Azure portal or the Azure CLI. Your application uses the identity to authenticate to other services.

Azure portal

1. At the top of the Azure portal, search for *Managed identities*. Select the **Managed Identities** result.
2. Select **+ Create** at the top of the **Managed Identities** overview page.
3. On the **Basics** tab, enter the following values:
 - **Subscription:** Select your desired subscription.
 - **Resource Group:** Select your desired resource group.
 - **Region:** Select a region near your location.
 - **Name:** Enter a recognizable name for your identity, such as *MigrationIdentity*.
4. Select **Review + create** at the bottom of the page.
5. When the validation checks finish, select **Create**. Azure creates a new user-assigned identity.

After the resource is created, select **Go to resource** to view the details of the managed identity.

Create User Assigned Managed Identity

Basics Tags Review + create

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Test Subscription

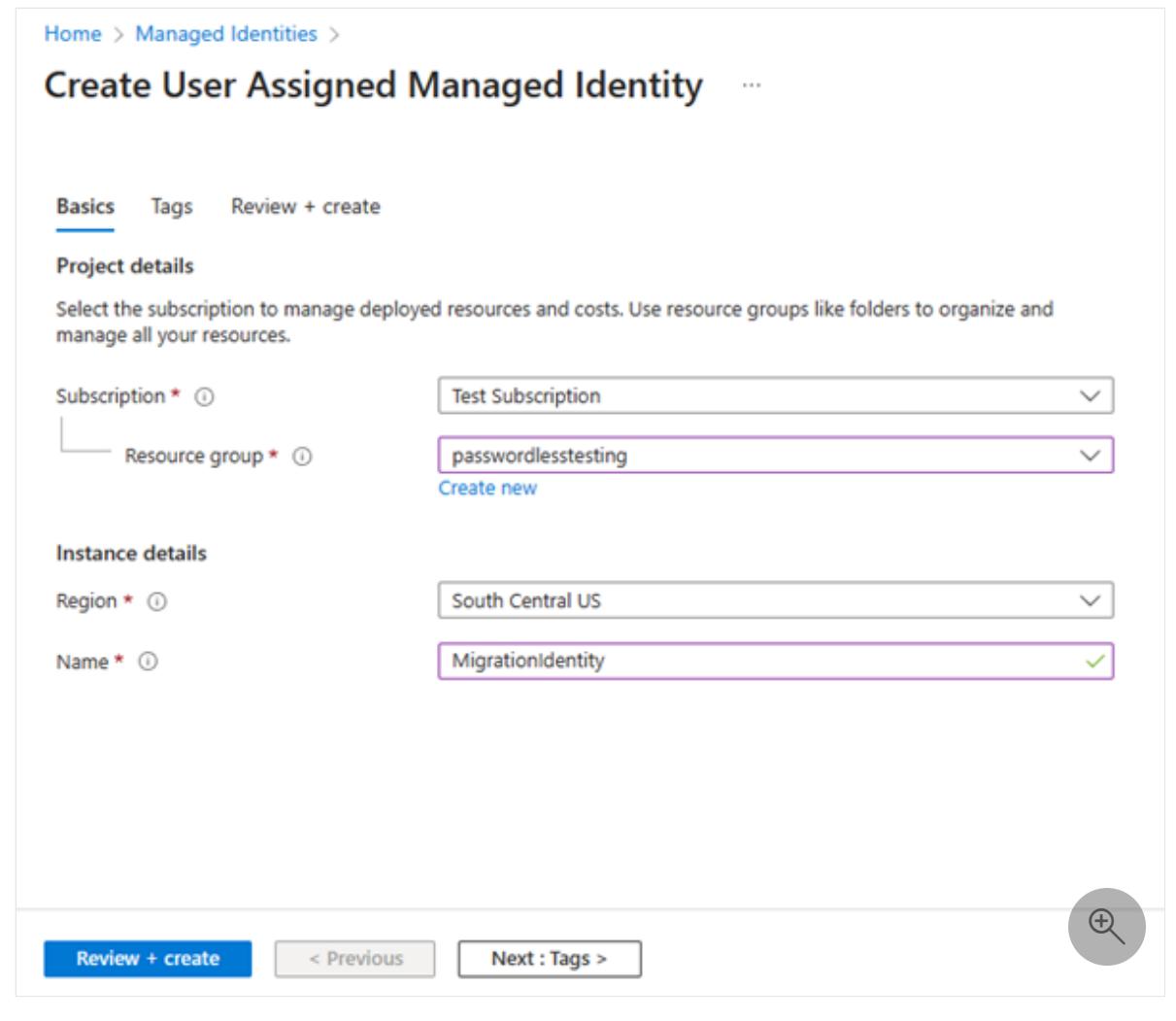
Resource group * ⓘ passwordlesstesting
Create new

Instance details

Region * ⓘ South Central US

Name * ⓘ MigrationIdentity

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



Associate the managed identity with your web app

You need to configure your web app to use the managed identity you created. Assign the identity to your app using either the Azure portal or the Azure CLI.

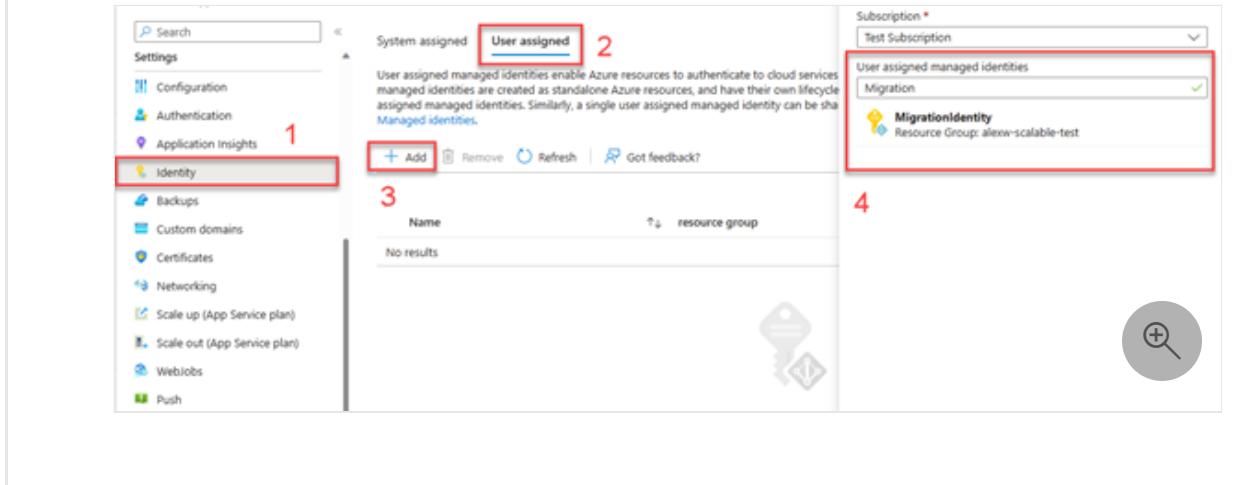
Azure portal

Complete the following steps in the Azure portal to associate an identity with your app. These same steps apply to the following Azure services:

- Azure Spring Apps
- Azure Container Apps
- Azure virtual machines
- Azure Kubernetes Service

1. Navigate to the overview page of your web app.
2. Select **Identity** from the left navigation.
3. On the **Identity** page, switch to the **User assigned** tab.

4. Select **+ Add** to open the **Add user assigned managed identity** flyout.
5. Select the subscription you used previously to create the identity.
6. Search for the **MigrationIdentity** by name and select it from the search results.
7. Select **Add** to associate the identity with your app.



Assign roles to the managed identity

Next, you need to grant permissions to the managed identity you created to access your storage account. Grant permissions by assigning a role to the managed identity, just like you did with your local development user.

Azure portal

1. Navigate to your storage account overview page and select **Access Control (IAM)** from the left navigation.
2. Choose **Add role assignment**

The screenshot shows the Azure Storage Access Control (IAM) interface. On the left, there's a sidebar with various navigation options like Overview, Activity log, Tags, Diagnose and solve problems, and Access Control (IAM), which is currently selected and highlighted with a red box. The main area has tabs for Check access, Role assignments, Roles, Deny assignments, and Classic administrators. Under Check access, there's a 'My access' section with a 'View my access' button. Below it is a 'Check access' section where you can search by name or email address. To the right, there's a 'Grant access to this resource' panel with an 'Add role assignment' button highlighted with a red box. There's also a 'View deny assignments' section with a 'View' button and a 'Learn more' link.

3. In the **Role** search box, search for *Storage Blob Data Contributor*, which is a common role used to manage data operations for blobs. You can assign whatever role is appropriate for your use case. Select the *Storage Blob Data Contributor* from the list and choose **Next**.
4. On the **Add role assignment** screen, for the **Assign access to** option, select **Managed identity**. Then choose **+Select members**.
5. In the flyout, search for the managed identity you created by name and select it from the results. Choose **Select** to close the flyout menu.

The screenshot shows the 'Add role assignment' screen and its associated flyout. The main screen has tabs for Role, Members*, Conditions (optional), Review + assign, and the Members tab is selected. A 'Selected role' dropdown shows 'Storage Blob Data Contributor'. The 'Assign access to' section has a radio button for 'Managed identity' selected. Below it is a 'Members' section with a '+ Select members' button. Step 1 points to the 'Assign access to' section, and step 2 points to the '+ Select members' button. At the bottom, there are 'Review + assign', 'Previous', 'Next', and '6' buttons. The flyout window is titled 'Select members' and contains a search bar with 'msdocs-web' typed in. Step 3 points to the search bar. Below it is a 'Selected members:' list containing 'msdocs-web-app-123' with a 'Remove' button. Step 4 points to this list. At the bottom of the flyout are 'Select' and 'Close' buttons, with step 5 pointing to the 'Select' button. Step 6 points to the 'Next' button at the bottom of the main screen.

6. Select **Next** a couple times until you're able to select **Review + assign** to finish the role assignment.

Update the application code

You need to configure your application code to look for the specific managed identity you created when it's deployed to Azure. In some scenarios, explicitly setting the managed identity for the app also prevents other environment identities from accidentally being detected and used automatically.

1. On the managed identity overview page, copy the client ID value to your clipboard.
2. Apply the following language-specific changes:

.NET

Create a `DefaultAzureCredentialOptions` object and pass it to `DefaultAzureCredential`. Set the `ManagedIdentityClientId` property to the client ID.

C#

```
DefaultAzureCredential credential = new(
    new DefaultAzureCredentialOptions
    {
        ManagedIdentityClientId = managedIdentityClientId
    });

```

3. Redeploy your code to Azure after making this change in order for the configuration updates to be applied.

Test the app

After deploying the updated code, browse to your hosted application in the browser. Your app should be able to connect to the storage account successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- Authorize access to blobs using Microsoft Entra ID
- To learn more about .NET Core, see [Get started with .NET in 10 minutes](#).

Migrate an application to use passwordless connections with Azure Queue Storage

Article • 05/10/2023

Application requests to Azure services must be authenticated using configurations such as account access keys or passwordless connections. However, you should prioritize passwordless connections in your applications when possible. Traditional authentication methods that use passwords or secret keys create security risks and complications. Visit the [passwordless connections for Azure services](#) hub to learn more about the advantages of moving to passwordless connections.

The following tutorial explains how to migrate an existing application to connect using passwordless connections. These same migration steps should apply whether you're using access keys, connection strings, or another secrets-based approach.

Configure your local development environment

Passwordless connections can be configured to work for both local and Azure-hosted environments. In this section, you'll apply configurations to allow individual users to authenticate to Azure Queue Storage for local development.

Assign user roles

When developing locally, make sure that the user account that is accessing Queue Storage has the correct permissions. You'll need the **Storage Queue Data Contributor** role to read and write queue data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the **Storage Queue Data Contributor** role to your user account. This role grants read and write access to queue data in your storage account.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'Access Control (IAM)' page for a storage account named 'identitymigrationstorage'. The left sidebar has a red box around the 'Access Control (IAM)' item. The top navigation bar has a red box around the '+ Add' button, which is currently highlighted. A dropdown menu is open, showing 'Add role assignment' as the selected option. The main content area includes sections for 'My access', 'Check access', and 'Grant access to this resource' (which also has a red box around its 'Add role assignment' button).

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Queue Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

ⓘ Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Sign-in to Azure locally

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

```
Azure CLI
```

```
az login
```

Update the application code to use passwordless connections

The Azure Identity client library, for each of the following ecosystems, provides a `DefaultAzureCredential` class that handles passwordless authentication to Azure:

- [.NET](#)
- [Go ↗](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

`DefaultAzureCredential` supports multiple authentication methods. The method to use is determined at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code. See the preceding links for the order and locations in which `DefaultAzureCredential` looks for credentials.

.NET

1. To use `DefaultAzureCredential` in a .NET application, install the `Azure.Identity` package:

```
.NET CLI
```

```
dotnet add package Azure.Identity
```

2. At the top of your file, add the following code:

```
C#
```

```
using Azure.Identity;
```

3. Identify the locations in your code that create a `QueueClient` object to connect to Azure Queue Storage. Update your code to match the following example:

```
C#
```

```
DefaultAzureCredential credential = new();

QueueClient queueClient = new(
    new Uri($"https://{{storageAccountName}}.queue.core.windows.net/{{queueName}}"),
    new DefaultAzureCredential());
```

4. Make sure to update the storage account name in the URI of your `QueueClient` object. You can find the storage account name on the overview page of the Azure portal.

The screenshot shows the Azure Storage account 'identitymigrationstorage' in the 'Overview' tab. The account is a 'Storage account'. Key details shown include:

- Resource group ([move](#)): alexw-identity-revamp
- Location: East US
- Primary/Secondary Location: Primary: East US, Secondary: West US
- Subscription ([move](#)): C&L Cross Service Content Team Testing
- Subscription ID: [redacted]
- Disk state: Primary: Available, Secondary: Available
- Tags ([edit](#)): Click here to add tags

A search bar at the top left and a navigation menu on the left side are also visible.

Run the app locally

After making these code changes, run your application locally. The new configuration should pick up your local credentials, such as the Azure CLI, Visual Studio, or IntelliJ. The roles you assigned to your user in Azure allows your app to connect to the Azure service locally.

Configure the Azure hosting environment

Once your application is configured to use passwordless connections and runs locally, the same code can authenticate to Azure services after it's deployed to Azure. The sections that follow explain how to configure a deployed application to connect to Azure Queue Storage using a [managed identity](#). Managed identities provide an automatically managed identity in Microsoft Entra ID for applications to use when connecting to resources that support Microsoft Entra authentication. Learn more about managed identities:

- [Passwordless Overview](#)
- [Managed identity best practices](#)

Create the managed identity

You can create a user-assigned managed identity using the Azure portal or the Azure CLI. Your application uses the identity to authenticate to other services.

Azure portal

1. At the top of the Azure portal, search for *Managed identities*. Select the **Managed Identities** result.
2. Select **+ Create** at the top of the **Managed Identities** overview page.
3. On the **Basics** tab, enter the following values:
 - **Subscription:** Select your desired subscription.
 - **Resource Group:** Select your desired resource group.
 - **Region:** Select a region near your location.
 - **Name:** Enter a recognizable name for your identity, such as *MigrationIdentity*.
4. Select **Review + create** at the bottom of the page.
5. When the validation checks finish, select **Create**. Azure creates a new user-assigned identity.

After the resource is created, select **Go to resource** to view the details of the managed identity.

The screenshot shows the 'Create User Assigned Managed Identity' page in the Azure portal. The 'Basics' tab is selected. The 'Project details' section includes fields for 'Subscription' (set to 'Test Subscription') and 'Resource group' (set to 'passwordlesstesting', with a 'Create new' option). The 'Instance details' section includes fields for 'Region' (set to 'South Central US') and 'Name' (set to 'MigrationIdentity'). At the bottom, there are buttons for 'Review + create' (highlighted in blue), '< Previous', 'Next : Tags >', and a magnifying glass icon.

Associate the managed identity with your web app

You need to configure your web app to use the managed identity you created. Assign the identity to your app using either the Azure portal or the Azure CLI.

Azure portal

Complete the following steps in the Azure portal to associate an identity with your app. These same steps apply to the following Azure services:

- Azure Spring Apps
- Azure Container Apps
- Azure virtual machines
- Azure Kubernetes Service

1. Navigate to the overview page of your web app.
2. Select **Identity** from the left navigation.
3. On the **Identity** page, switch to the **User assigned** tab.
4. Select **+ Add** to open the **Add user assigned managed identity** flyout.
5. Select the subscription you used previously to create the identity.
6. Search for the **MigrationIdentity** by name and select it from the search results.
7. Select **Add** to associate the identity with your app.

The screenshot shows the Azure portal's Identity blade. Step 1 highlights the 'Identity' menu item in the left sidebar. Step 2 highlights the 'User assigned' tab in the top navigation. Step 3 highlights the '+ Add' button. Step 4 highlights the search results for 'MigrationIdentity', which is selected. A magnifying glass icon in the bottom right corner indicates a search function.

Assign roles to the managed identity

Next, you need to grant permissions to the managed identity you created to access your storage account. Grant permissions by assigning a role to the managed identity, just like you did with your local development user.

1. Navigate to your storage account overview page and select **Access Control (IAM)** from the left navigation.

2. Choose **Add role assignment**

The screenshot shows the Azure Storage Access Control (IAM) interface. On the left, there's a sidebar with various options like Overview, Activity log, Tags, and Access Control (IAM). The Access Control (IAM) option is highlighted with a red box. The main area has tabs for Check access, Role assignments, Roles, Deny assignments, and Classic administrators. Under Check access, there's a section for 'My access' with a 'View my access' button. Below it is a 'Check access' section where 'User, group, or service principal' is selected. To the right, there are three boxes: 'Grant access to this resource' with an 'Add role assignment' button (highlighted with a red box), 'View deny assignments' with a 'View' button, and a 'Learn more' link. At the bottom right of the main area is a 'Learn more' link with a magnifying glass icon.

3. In the **Role** search box, search for *Storage Queue Data Contributor*, which is a common role used to manage data operations for queues. You can assign whatever role is appropriate for your use case. Select the *Storage Queue Data Contributor* from the list and choose **Next**.
4. On the **Add role assignment** screen, for the **Assign access to** option, select **Managed identity**. Then choose **+Select members**.
5. In the flyout, search for the managed identity you created by name and select it from the results. Choose **Select** to close the flyout menu.

The screenshot shows the 'Add role assignment' dialog and a separate 'Select members' flyout window. In the main dialog, the 'Members' tab is selected. The 'Selected role' is 'Storage Blob Data Contributor'. The 'Assign access to' dropdown is set to 'Managed identity' (highlighted with a red box, labeled 1). Below it, the 'Members' field contains '+ Select members' (highlighted with a red box, labeled 2). At the bottom, the 'Next' button is highlighted with a red box (labeled 6). In the 'Select members' flyout, the search bar shows 'msdocs-web' (highlighted with a red box, labeled 3). The results list shows 'msdocs-web-app-123' under 'Selected members' (highlighted with a red box, labeled 4). The 'Select' button at the bottom right of the flyout is highlighted with a red box (labeled 5).

6. Select **Next** a couple times until you're able to select **Review + assign** to finish the role assignment.

Update the application code

You need to configure your application code to look for the specific managed identity you created when it's deployed to Azure. In some scenarios, explicitly setting the managed identity for the app also prevents other environment identities from accidentally being detected and used automatically.

1. On the managed identity overview page, copy the client ID value to your clipboard.
2. Apply the following language-specific changes:

.NET

Create a `DefaultAzureCredentialOptions` object and pass it to `DefaultAzureCredential`. Set the `ManagedIdentityClientId` property to the client ID.

C#

```
DefaultAzureCredential credential = new(
    new DefaultAzureCredentialOptions
    {
        ManagedIdentityClientId = managedIdentityClientId
    });

```

3. Redeploy your code to Azure after making this change in order for the configuration updates to be applied.

Test the app

After deploying the updated code, browse to your hosted application in the browser. Your app should be able to connect to the storage account successfully. Keep in mind that it may take several minutes for the role assignments to propagate through your Azure environment. Your application is now configured to run both locally and in a production environment without the developers having to manage secrets in the application itself.

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections.

You can read the following resources to explore the concepts discussed in this article in more depth:

- [Authorize access to blobs using Microsoft Entra ID](#)
- To learn more about .NET, see [Get started with .NET in 10 minutes](#).

Tutorial: Create a passwordless connection to a database service via Service Connector

Article • 10/31/2023

Passwordless connections use managed identities to access Azure services. With this approach, you don't have to manually track and manage secrets for managed identities. These tasks are securely handled internally by Azure.

Service Connector enables managed identities in app hosting services like Azure Spring Apps, Azure App Service, and Azure Container Apps. Service Connector also configures database services, such as Azure Database for PostgreSQL, Azure Database for MySQL, and Azure SQL Database, to accept managed identities.

In this tutorial, you use the Azure CLI to complete the following tasks:

- ✓ Check your initial environment with the Azure CLI.
- ✓ Create a passwordless connection with Service Connector.
- ✓ Use the environment variables or configurations generated by Service Connector to access a database service.

Prerequisites

- [Azure CLI](#) version 2.48.1 or higher.
- An Azure account with an active subscription. [Create an Azure account for free](#).
- An app deployed to [Azure App Service](#) in a [region supported by Service Connector](#).

Set up your environment

Account

Sign in with the Azure CLI via `az login`. If you're using Azure Cloud Shell or are already logged in, confirm your authenticated account with `az account show`.

Install the Service Connector passwordless extension

Install the Service Connector passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

Create a passwordless connection

Next, we use Azure App Service as an example to create a connection using managed identity.

If you use:

- Azure Spring Apps: use `az spring connection create` instead. For more examples, see [Connect Azure Spring Apps to the Azure database](#).
- Azure Container Apps: use `az containerapp connection create` instead. For more examples, see [Create and connect a PostgreSQL database with identity connectivity](#).

ⓘ Note

If you use the Azure portal, go to the **Service Connector** blade of **Azure App Service**, **Azure Spring Apps**, or **Azure Container Apps**, and select **Create** to create a connection. The Azure portal will automatically compose the command for you and trigger the command execution on Cloud Shell.

The following Azure CLI command uses a `--client-type` parameter. Run the `az webapp connection create postgres-flexible -h` to get the supported client types, and choose the one that matches your application.

User-assigned managed identity

Azure CLI

```
az webapp connection create postgres-flexible \
--resource-group $RESOURCE_GROUP \
--name $APPSERVICE_NAME \
--target-resource-group $RESOURCE_GROUP \
--server $POSTGRESQL_HOST \
--database $DATABASE_NAME \
--user-identity client-id=XX subs-id=XX \
--client-type java
```

This Service Connector command completes the following tasks in the background:

- Enable system-assigned managed identity, or assign a user identity for the app `$APPSERVICE_NAME` hosted by Azure App Service/Azure Spring Apps/Azure Container Apps.
- Set the Microsoft Entra admin to the current signed-in user.
- Add a database user for the system-assigned managed identity, user-assigned managed identity, or service principal. Grant all privileges of the database `$DATABASE_NAME` to this user. The username can be found in the connection string in preceding command output.
- Set configurations named `AZURE_MYSQL_CONNECTIONSTRING`, `AZURE_POSTGRESQL_CONNECTIONSTRING`, or `AZURE_SQL_CONNECTIONSTRING` to the Azure resource based on the database type.
 - For App Service, the configurations are set in the **App Settings** blade.
 - For Spring Apps, the configurations are set when the application is launched.
 - For Container Apps, the configurations are set to the environment variables. You can get all configurations and their values in the **Service Connector** blade in the Azure portal.

Connect to a database with Microsoft Entra authentication

After creating the connection, you can use the connection string in your application to connect to the database with Microsoft Entra authentication. For example, you can use the following solutions to connect to the database with Microsoft Entra authentication.

.NET

For .NET, there's not a plugin or library to support passwordless connections. You can get an access token for the managed identity or service principal using client library like [Azure.Identity](#). Then you can use the access token as the password to connect to the database. When using the code below, uncomment the part of the code snippet for the authentication type you want to use.

C#

```
using Azure.Identity;
using Azure.Core;
using Npgsql;

// Uncomment the following lines according to the authentication type.
// For system-assigned identity.
```

```

// var sqlServerTokenProvider = new DefaultAzureCredential();

// For user-assigned identity.
// var sqlServerTokenProvider = new DefaultAzureCredential(
//     new DefaultAzureCredentialOptions
//     {
//         ManagedIdentityClientId =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTID");
//     }
// );

// For service principal.
// var tenantId =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_TENANTID");
// var clientId =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTID");
// var clientSecret =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTSECRET");
// var sqlServerTokenProvider = new ClientSecretCredential(tenantId,
clientId, clientSecret);

// Acquire the access token.
AccessToken accessToken = await sqlServerTokenProvider.GetTokenAsync(
    new TokenRequestContext(scopes: new string[]
{
    "https://osrdbms-aad.database.windows.net/.default"
}));


// Combine the token with the connection string from the environment
variables provided by Service Connector.
string connectionString =
$"{
{Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CONNECTIONSTRING")
};Password={accessToken.Token}";


// Establish the connection.
using (var connection = new NpgsqlConnection(connectionString))
{
    Console.WriteLine("Opening connection using access token...");
    connection.Open();
}

```

Next, if you have created tables and sequences in PostgreSQL flexible server before using Service Connector, you need to connect as the owner and grant permission to `<aad-username>` created by Service Connector. The username from the connection string or configuration set by Service Connector should look like `aad_<connection name>`. If you use the Azure portal, select the expand button next to the `Service Type` column and get the value. If you use Azure CLI, check `configurations` in the CLI command output.

Then, execute the query to grant permission

Azure CLI

```
az extension add --name rdbms-connect

az postgres flexible-server execute -n <postgres-name> -u <owner-username> -
p "<owner-password>" -d <database-name> --querytext "GRANT ALL PRIVILEGES ON
ALL TABLES IN SCHEMA public TO \"<aad-username>\";GRANT ALL PRIVILEGES ON
ALL SEQUENCES IN SCHEMA public TO \"<aad username>\";"
```

The `<owner-username>` and `<owner-password>` is the owner of the existing table that can grant permissions to others. `<aad-username>` is the user created by Service Connector. Replace them with the actual value.

Validate the result with the command:

Azure CLI

```
az postgres flexible-server execute -n <postgres-name> -u <owner-username> -
p "<owner-password>" -d <database-name> --querytext "SELECT
distinct(table_name) FROM information_schema.table_privileges WHERE
grantee='<aad-username>' AND table_schema='public';" --output table
```

Deploy the application to an Azure hosting service

Finally, deploy your application to an Azure hosting service. That source service can use a managed identity to connect to the target database on Azure.

App Service

For Azure App Service, you can deploy the application code via the `az webapp deploy` command. For more information, see [Quickstart: Deploy an ASP.NET web app](#).

Then you can check the log or call the application to see if it can connect to the Azure database successfully.

Troubleshooting

Permission

If you encounter any permission-related errors, confirm the Azure CLI signed-in user with the command `az account show`. Make sure you log in with the correct account. Next, confirm that you have the following permissions that might be required to create a passwordless connection with Service Connector.

Permission	Operation
<code>Microsoft.DBforPostgreSQL/flexibleServers/read</code>	Required to get information of database server
<code>Microsoft.DBforPostgreSQL/flexibleServers/write</code>	Required to enable Microsoft Entra authentication for database server
<code>Microsoft.DBforPostgreSQL/flexibleServers/firewallRules/write</code>	Required to create firewall rule in case the local IP address is blocked
<code>Microsoft.DBforPostgreSQL/flexibleServers/firewallRules/delete</code>	Required to revert the firewall rule created by Service Connector to avoid security issue
<code>Microsoft.DBforPostgreSQL/flexibleServers/administrators/read</code>	Required to check if Azure CLI login user is a database server Microsoft Entra administrator
<code>Microsoft.DBforPostgreSQL/flexibleServers/administrators/write</code>	Required to add Azure CLI login user as database server Microsoft Entra administrator

In some cases, the permissions aren't required. For example, if the Azure CLI-authenticated user is already an Active Directory Administrator on SQL server, you don't need to have the `Microsoft.Sql/servers/administrators/write` permission.

Microsoft Entra ID

If you get an error `ERROR: AADSTS53003: Your device is required to be managed to access this resource.`, ask your IT department for help with joining this device to Microsoft Entra ID. For more information, see [Microsoft Entra joined devices](#).

Service Connector needs to access Microsoft Entra ID to get information of your account and managed identity of hosting service. You can use the following command to check

if your device can access Microsoft Entra ID:

Azure CLI

```
az ad signed-in-user show
```

If you don't log in interactively, you might also get the error and [Interactive authentication is needed](#). To resolve the error, log in with the `az login` command.

Network connectivity

If your database server is in Virtual Network, ensure your environment that runs the Azure CLI command can access the server in the Virtual Network.

Next steps

For more information about Service Connector and passwordless connections, see the following resources:

[Service Connector documentation](#)

[Passwordless connections for Azure services](#)

Integrate Azure SQL Database with Service Connector

Article • 02/02/2024

This page shows supported authentication methods and clients, and shows sample code you can use to connect compute services to Azure SQL Database using Service Connector. You might still be able to connect to Azure SQL Database using other methods. This page also shows default environment variable names and values you get when you create the service connection.

Supported compute services

Service Connector can be used to connect the following compute services to Azure SQL Database:

- Azure App Service
- Azure Functions
- Azure Container Apps
- Azure Spring Apps

Supported authentication types and clients

The table below shows which combinations of authentication methods and clients are supported for connecting your compute service to Azure SQL Database using Service Connector. A “Yes” indicates that the combination is supported, while a “No” indicates that it is not supported.

[+] Expand table

Client type	System-assigned managed identity	User-assigned managed identity	Secret/connection string	Service principal
.NET	Yes	Yes	Yes	Yes
Go	No	No	Yes	No
Java	Yes	Yes	Yes	Yes
Java - Spring Boot	Yes	Yes	Yes	Yes

Client type	System-assigned managed identity	User-assigned managed identity	Secret/connection string	Service principal
Node.js	Yes	Yes	Yes	Yes
PHP	No	No	Yes	No
Python	Yes	Yes	Yes	Yes
Python - Django	No	No	Yes	No
Ruby	No	No	Yes	No
None	Yes	Yes	Yes	Yes

This table indicates that the Secret/connection string method is supported for all client types. The System-assigned managed identity, User-assigned managed identity, and Service principal methods are supported for .NET, Java, Java - Spring Boot, Node.js, Python, and None client types. These methods are not supported for Go, PHP, Django, and Ruby client types.

 **Note**

System-assigned managed identity, User-assigned managed identity and Service principal are only supported on Azure CLI.

Default environment variable names or application properties and sample code

Use the connection details below to connect compute services to Azure SQL Database. For each example below, replace the placeholder texts <sql-server>, <sql-database>, <sql-username>, and <sql-password> with your own server name, database name, user ID and password. For more information about naming conventions, check the [Service Connector internals](#) article.

System-assigned Managed Identity

.NET

 Expand table

Default environment variable name	Description	Sample value
AZURE_SQL_CONNECTIONSTRING	Azure SQL Database connection string	Data Source=<sql-server>.database.windows.net,1433;Initial Catalog=<sql-database>;Authentication=ActiveDirectoryManagedIdentity

Sample code

Refer to the steps and code below to connect to Azure SQL Database using a system-assigned managed identity.

.NET

1. Install dependencies.

Bash

```
dotnet add package Microsoft.Data.SqlClient
```

2. Get the Azure SQL Database connection string from the environment variable added by Service Connector.

C#

```
using Microsoft.Data.SqlClient;

string connectionString =
    Environment.GetEnvironmentVariable("AZURE_SQL_CONNECTIONSTRING")!;

using var connection = new SqlConnection(connectionString);
connection.Open();
```

For more information, see [Using Active Directory Managed Identity authentication](#).

For more information, see [Homepage for client programming to Microsoft SQL Server](#).

User-assigned managed identity

[Expand table](#)

Default environment variable name	Description	Sample value
AZURE_SQL_CONNECTIONSTRING	Azure SQL Database connection string	Data Source=<sql-server>.database.windows.net,1433;Initial Catalog=<sql-database>;User ID=<identity-client-ID>;Authentication=ActiveDirectoryManagedIdentity

Sample code

Refer to the steps and code below to connect to Azure SQL Database using a user-assigned managed identity.

1. Install dependencies.

Bash

```
dotnet add package Microsoft.Data.SqlClient
```

2. Get the Azure SQL Database connection string from the environment variable added by Service Connector.

C#

```
using Microsoft.Data.SqlClient;

string connectionString =
    Environment.GetEnvironmentVariable("AZURE_SQL_CONNECTIONSTRING")!;

using var connection = new SqlConnection(connectionString);
connection.Open();
```

For more information, see [Using Active Directory Managed Identity authentication](#).

For more information, see [Homepage for client programming to Microsoft SQL Server](#).

Connection String

.NET

 Expand table

Default environment variable name	Description	Sample value
AZURE_SQL_CONNECTION_STRING	Azure SQL Database connection string	Data Source=<sql-server>.database.windows.net,1433;Initial Catalog=<sql-database>;Password=<sql-password>

Sample code

Refer to the steps and code below to connect to Azure SQL Database using a connection string.

.NET

1. Install dependencies.

Bash

```
dotnet add package Microsoft.Data.SqlClient
```

2. Get the Azure SQL Database connection string from the environment variable added by Service Connector.

C#

```
using Microsoft.Data.SqlClient;  
  
string connectionString =
```

```

Environment.GetEnvironmentVariable("AZURE_SQL_CONNECTIONSTRING")!;

using var connection = new SqlConnection(connectionString);
connection.Open();

```

For more information, see [Homepage for client programming to Microsoft SQL Server](#).

Service Principal

.NET

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

Default environment variable name	Description	Example value
AZURE_SQL_CLIENTID	Your client ID	<client-ID>
AZURE_SQL_CLIENTSECRET	Your client secret	<client-secret>
AZURE_SQL_TENANTID	Your tenant ID	<tenant-ID>
AZURE_SQL_CONNECTIONSTRING	Azure SQL Database connection string	Data Source=<sql-server>.database.windows.net,1433;Initial Catalog=<sql-database>;User ID=a30eedc-e75f-4301-b1a9-56e81e0ce99c;Password=asdfghwerty;Authentication=ActiveDirectoryServicePrincipal

Sample code

Refer to the steps and code below to connect to Azure SQL Database using a service principal.

.NET

1. Install dependencies.

Bash

```
dotnet add package Microsoft.Data.SqlClient
```

2. Get the Azure SQL Database connection string from the environment variable added by Service Connector.

C#

```
using Microsoft.Data.SqlClient;

string connectionString =
    Environment.GetEnvironmentVariable("AZURE_SQL_CONNECTIONSTRING")!;

using var connection = new SqlConnection(connectionString);
connection.Open();
```

For more information, see [Using Active Directory Managed Identity authentication](#).

For more information, see [Homepage for client programming to Microsoft SQL Server](#).

Next steps

Follow the tutorial listed below to learn more about Service Connector.

[Learn about Service Connector concepts](#)

Integrate Azure Database for MySQL with Service Connector

Article • 02/02/2024

This page shows supported authentication methods and clients, and shows sample code you can use to connect Azure Database for MySQL - Flexible Server to other cloud services using Service Connector. This page also shows default environment variable names and values (or Spring Boot configuration) you get when you create the service connection.

ⓘ Important

Azure Database for MySQL single server is on the retirement path. We strongly recommend that you upgrade to Azure Database for MySQL flexible server. For more information about migrating to Azure Database for MySQL flexible server, see [What's happening to Azure Database for MySQL Single Server?](#)

Supported compute services

Service Connector can be used to connect the following compute services to Azure Database for MySQL:

- Azure App Service
- Azure Functions
- Azure Container Apps
- Azure Spring Apps

Supported authentication types and client types

The table below shows which combinations of authentication methods and clients are supported for connecting your compute service to Azure Database for MySQL using Service Connector. A "Yes" indicates that the combination is supported, while a "No" indicates that it is not supported.

[[]] Expand table

Client type	System-assigned managed identity	User-assigned managed identity	Secret/connection string	Service principal
.NET	Yes	Yes	Yes	Yes
Go (go-sql-driver for mysql)	Yes	Yes	Yes	Yes
Java (JDBC)	Yes	Yes	Yes	Yes
Java - Spring Boot (JDBC)	Yes	Yes	Yes	Yes
Node.js (mysql)	Yes	Yes	Yes	Yes
Python (mysql-connector-python)	Yes	Yes	Yes	Yes
Python-Django	Yes	Yes	Yes	Yes
PHP (MySQLi)	Yes	Yes	Yes	Yes
Ruby (mysql2)	Yes	Yes	Yes	Yes
None	Yes	Yes	Yes	Yes

This table indicates that all combinations of client types and authentication methods in the table are supported. All client types can use any of the authentication methods to connect to Azure Database for MySQL using Service Connector.

Note

System-assigned managed identity, User-assigned managed identity and Service principal are only supported on Azure CLI.

Default environment variable names or application properties and sample code

Reference the connection details and sample code in following tables, according to your connection's authentication type and client type, to connect compute services to Azure Database for MySQL. For more information about naming conventions, check the [Service Connector internals](#) article.

System-assigned Managed Identity

.NET

[+] Expand table

Default environment variable name	Description	Example value
AZURE_MYSQL_CONNECTIONSTRING	ADO.NET MySQL connection string	Server=<MySQL-DB-name>.mysql.database.azure.com;Database=<MySQL-DB-name>;Port=3306;User Id=<MySQL-DBusername>;SSL Mode=Required;

Sample code

Refer to the steps and code below to connect to Azure Database for MySQL using a system-assigned managed identity.

.NET

For .NET, there's not a plugin or library to support passwordless connections. You can get an access token for the managed identity or service principal using client library like [Azure.Identity](#). Then you can use the access token as the password to connect to the database. When using the code below, uncomment the part of the code snippet for the authentication type you want to use.

C#

```
using Azure.Core;
using Azure.Identity;
using MySqlConnector;

// Uncomment the following lines according to the authentication type.
// For system-assigned managed identity.
// var credential = new DefaultAzureCredential();

// For user-assigned managed identity.
// var credential = new DefaultAzureCredential(
//     new DefaultAzureCredentialOptions
//     {
//         ManagedIdentityClientId =
// Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTID");
//     });

```

```

// For service principal.
// var tenantId =
Environment.GetEnvironmentVariable("AZURE_MYSQL_TENANTID");
// var clientId =
Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTID");
// var clientSecret =
Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTSECRET");
// var credential = new ClientSecretCredential(tenantId, clientId,
clientSecret);

var tokenRequestContext = new TokenRequestContext(
    new[] { "https://osrdbms-aad.database.windows.net/.default" });
AccessToken accessToken = await
credential.GetTokenAsync(tokenRequestContext);
// Open a connection to the MySQL server using the access token.
string connectionString =
$"{
{Environment.GetEnvironmentVariable("AZURE_MYSQL_CONNECTIONSTRING")};Pas
sword={accessToken.Token}";

using var connection = new MySqlConnection(connectionString);
Console.WriteLine("Opening connection using access token...");
await connection.OpenAsync();

// do something

```

For more code samples, see [Connect to Azure databases from App Service without secrets using a managed identity](#).

User-assigned Managed Identity

.NET

[] Expand table

Default environment variable name	Description	Example value
AZURE_MYSQL_CLIENTID	Your client ID	<identity-client-ID>
AZURE_MYSQL_CONNECTIONSTRING	ADO.NET MySQL connection string	Server=<MySQL-DB-name>.mysql.database.azure.com;Database=<MySQL-DB-name>;Port=3306;User Id=<MySQL-DBusername>;SSL Mode=Required;

Sample code

Refer to the steps and code below to connect to Azure Database for MySQL using a user-assigned managed identity.

.NET

For .NET, there's not a plugin or library to support passwordless connections. You can get an access token for the managed identity or service principal using client library like [Azure.Identity](#). Then you can use the access token as the password to connect to the database. When using the code below, uncomment the part of the code snippet for the authentication type you want to use.

C#

```
using Azure.Core;
using Azure.Identity;
using MySqlConnector;

// Uncomment the following lines according to the authentication type.
// For system-assigned managed identity.
// var credential = new DefaultAzureCredential();

// For user-assigned managed identity.
// var credential = new DefaultAzureCredential(
//     new DefaultAzureCredentialOptions
//     {
//         ManagedIdentityClientId =
// Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTID");
//     });

// For service principal.
// var tenantId =
Environment.GetEnvironmentVariable("AZURE_MYSQL_TENANTID");
// var clientId =
Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTID");
// var clientSecret =
Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTSECRET");
// var credential = new ClientSecretCredential(tenantId, clientId,
clientSecret);

var tokenRequestContext = new TokenRequestContext(
    new[] { "https://osssrdbms-aad.database.windows.net/.default" });
AccessToken accessToken = await
credential.GetTokenAsync(tokenRequestContext);
// Open a connection to the MySQL server using the access token.
string connectionString =
$"{
{Environment.GetEnvironmentVariable("AZURE_MYSQL_CONNECTIONSTRING")};Pas
sword={accessToken.Token}";
```

```
using var connection = new MySqlConnection(connectionString);
Console.WriteLine("Opening connection using access token...");
await connection.OpenAsync();

// do something
```

For more code samples, see [Connect to Azure databases from App Service without secrets using a managed identity](#).

Connection String

.NET

[+] [Expand table](#)

Default environment variable name	Description	Example value
AZURE_MYSQL_CONNECTIONSTRING	ADO.NET MySQL connection string	Server=<MySQL-DB-name>.mysql.database.azure.com;Database=<MySQL-DB-name>;Port=3306;User Id=<MySQL-DBusername>;Password=<MySQL-DBpassword>;SSL Mode=Required

Sample code

Refer to the steps and code below to connect to Azure Database for MySQL using a connection string.

.NET

1. Install dependencies. Follow the guidance to [install connector/.NET MySQL](#)
2. In code, get MySQL connection string from environment variables added by Service Connector service. To establish encrypted connection to MySQL server over SSL, refer to [these steps](#).

C#

```
using System;
using System.Data;
using MySql.Data.MySqlClient;
```

```

string connectionString =
Environment.GetEnvironmentVariable("AZURE_MYSQL_CONNECTIONSTRING");
using (MySqlConnection connection = new
MySqlConnection(connectionString))
{
    connection.Open();
}

```

Service Principal

.NET

 Expand table

Default environment variable name	Description	Example value
AZURE_MYSQL_CLIENTID	Your client ID	<client-ID>
AZURE_MYSQL_CLIENTSECRET	Your client secret	<client-secret>
AZURE_MYSQL_TENANTID	Your tenant ID	<tenant-ID>
AZURE_MYSQL_CONNECTIONSTRING	ADO.NET MySQL connection string	Server=<MySQL-DB-name>.mysql.database.azure.com;Database=<MySQL-DB-name>;Port=3306;User Id=<MySQL-DBusername>;SSL Mode=Required

Sample code

Refer to the steps and code below to connect to Azure Database for MySQL using a service principal.

.NET

For .NET, there's not a plugin or library to support passwordless connections. You can get an access token for the managed identity or service principal using client library like [Azure.Identity](#). Then you can use the access token as the password to

connect to the database. When using the code below, uncomment the part of the code snippet for the authentication type you want to use.

```
C#  
  
using Azure.Core;  
using Azure.Identity;  
using MySqlConnector;  
  
// Uncomment the following lines according to the authentication type.  
// For system-assigned managed identity.  
// var credential = new DefaultAzureCredential();  
  
// For user-assigned managed identity.  
// var credential = new DefaultAzureCredential(  
//     new DefaultAzureCredentialOptions  
//     {  
//         ManagedIdentityClientId =  
Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTID");  
//     });  
  
// For service principal.  
// var tenantId =  
Environment.GetEnvironmentVariable("AZURE_MYSQL_TENANTID");  
// var clientId =  
Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTID");  
// var clientSecret =  
Environment.GetEnvironmentVariable("AZURE_MYSQL_CLIENTSECRET");  
// var credential = new ClientSecretCredential(tenantId, clientId,  
clientSecret);  
  
var tokenRequestContext = new TokenRequestContext(  
    new[] { "https://osssrdbms-aad.database.windows.net/.default" });  
AccessToken accessToken = await  
credential.GetTokenAsync(tokenRequestContext);  
// Open a connection to the MySQL server using the access token.  
string connectionString =  
    $"  
{Environment.GetEnvironmentVariable("AZURE_MYSQL_CONNECTIONSTRING")};Pas  
sword={accessToken.Token}";  
  
using var connection = new MySqlConnection(connectionString);  
Console.WriteLine("Opening connection using access token...");  
await connection.OpenAsync();  
  
// do something
```

For more code samples, see [Connect to Azure databases from App Service without secrets using a managed identity](#).

Next steps

Follow the documentations to learn more about Service Connector.

[Learn about Service Connector concepts](#)

Integrate Azure Database for PostgreSQL with Service Connector

Article • 02/05/2024

This page shows supported authentication methods and clients, and shows sample code you can use to connect Azure Database for PostgreSQL to other cloud services using Service Connector. You might still be able to connect to Azure Database for PostgreSQL in other programming languages without using Service Connector. This page also shows default environment variable names and values (or Spring Boot configuration) you get when you create the service connection.

Supported compute services

Service Connector can be used to connect the following compute services to Azure Database for PostgreSQL:

- Azure App Service
- Azure Functions
- Azure App Configuration
- Azure Spring Apps

Supported authentication types and client types

The table below shows which combinations of authentication methods and clients are supported for connecting your compute service to Azure Database for PostgreSQL using Service Connector. A "Yes" indicates that the combination is supported, while a "No" indicates that it is not supported.

[+] Expand table

Client type	System-assigned managed identity	User-assigned managed identity	Secret/connection string	Service principal
.NET	Yes	Yes	Yes	Yes
Go (pg)	Yes	Yes	Yes	Yes
Java (JDBC)	Yes	Yes	Yes	Yes

Client type	System-assigned managed identity	User-assigned managed identity	Secret/connection string	Service principal
Java - Spring Boot (JDBC)	Yes	Yes	Yes	Yes
Node.js (pg)	Yes	Yes	Yes	Yes
PHP (native)	Yes	Yes	Yes	Yes
Python (psycopg2)	Yes	Yes	Yes	Yes
Python-Django	Yes	Yes	Yes	Yes
Ruby (ruby-pg)	Yes	Yes	Yes	Yes
None	Yes	Yes	Yes	Yes

This table indicates that all combinations of client types and authentication methods in the table are supported. All client types can use any of the authentication methods to connect to Azure Database for PostgreSQL using Service Connector.

ⓘ Note

System-assigned managed identity, User-assigned managed identity and Service principal are only supported on Azure CLI.

Default environment variable names or application properties and sample code

Reference the connection details and sample code in the following tables, according to your connection's authentication type and client type, to connect compute services to Azure Database for PostgreSQL. For more information about naming conventions, check the [Service Connector internals](#) article.

System-assigned Managed Identity

.NET

 Expand table

Default environment variable name	Description	Example value
AZURE_POSTGRESQL_CONNECTIONSTRING	.NET PostgreSQL connection string	Server=<PostgreSQL-server-name>.postgres.database.azure.com;Database=<database-name>;Port=5432;SslMode=Require;User Id=<username>;

Sample code

Refer to the steps and code below to connect to Azure Database for PostgreSQL using a system-assigned managed identity.

.NET

For .NET, there's not a plugin or library to support passwordless connections. You can get an access token for the managed identity or service principal using client library like [Azure.Identity](#). Then you can use the access token as the password to connect to the database. When using the code below, uncomment the part of the code snippet for the authentication type you want to use.

C#

```
using Azure.Identity;
using Azure.Core;
using Npgsql;

// Uncomment the following lines according to the authentication type.
// For system-assigned identity.
// var sqlServerTokenProvider = new DefaultAzureCredential();

// For user-assigned identity.
// var sqlServerTokenProvider = new DefaultAzureCredential(
//     new DefaultAzureCredentialOptions
//     {
//         ManagedIdentityClientId =
// Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTID");
//     }
// );

// For service principal.
// var tenantId =
// Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_TENANTID");
// var clientId =
// Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTID");
// var clientSecret =
// Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTSECRET");
```

```

// var sqlServerTokenProvider = new ClientSecretCredential(tenantId,
clientId, clientSecret);

// Acquire the access token.
AccessToken accessToken = await sqlServerTokenProvider.GetTokenAsync(
    new TokenRequestContext(scopes: new string[]
    {
        "https://osssrdbms-aad.database.windows.net/.default"
    }));
}

// Combine the token with the connection string from the environment
variables provided by Service Connector.
string connectionString =
$"{
{Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CONNECTIONSTRING")}
};Password={accessToken.Token}";

// Establish the connection.
using (var connection = new NpgsqlConnection(connectionString))
{
    Console.WriteLine("Opening connection using access token...");
    connection.Open();
}

```

Next, if you have created tables and sequences in PostgreSQL flexible server before using Service Connector, you need to connect as the owner and grant permission to `<aad-username>` created by Service Connector. The username from the connection string or configuration set by Service Connector should look like `aad_<connection name>`. If you use the Azure portal, select the expand button next to the `Service Type` column and get the value. If you use Azure CLI, check `configurations` in the CLI command output.

Then, execute the query to grant permission

Azure CLI

```

az extension add --name rdbms-connect

az postgres flexible-server execute -n <postgres-name> -u <owner-username> -
p "<owner-password>" -d <database-name> --querytext "GRANT ALL PRIVILEGES ON
ALL TABLES IN SCHEMA public TO \"<aad-username>\";GRANT ALL PRIVILEGES ON
ALL SEQUENCES IN SCHEMA public TO \"<aad username>\";"

```

The `<owner-username>` and `<owner-password>` is the owner of the existing table that can grant permissions to others. `<aad-username>` is the user created by Service Connector. Replace them with the actual value.

Validate the result with the command:

Azure CLI

```
az postgres flexible-server execute -n <postgres-name> -u <owner-username> -p "<owner-password>" -d <database-name> --querytext "SELECT distinct(table_name) FROM information_schema.table_privileges WHERE grantee='<aad-username>' AND table_schema='public';" --output table
```

User-assigned Managed Identity

.NET

[+] Expand table

Default environment variable name	Description	Example value
AZURE_POSTGRESQL_CLIENTID	Your client ID	<identity-client-ID>
AZURE_POSTGRESQL_CONNECTIONSTRING	.NET PostgreSQL connection string	Server=<PostgreSQL-server-name>.postgres.database.azure.com;Database=<database-name>;Port=5432;Ssl Mode=Require;User Id=<username>;

Sample code

Refer to the steps and code below to connect to Azure Database for PostgreSQL using a user-assigned managed identity.

.NET

For .NET, there's not a plugin or library to support passwordless connections. You can get an access token for the managed identity or service principal using client library like [Azure.Identity](#). Then you can use the access token as the password to connect to the database. When using the code below, uncomment the part of the code snippet for the authentication type you want to use.

C#

```
using Azure.Identity;
using Azure.Core;
using Npgsql;
```

```

// Uncomment the following lines according to the authentication type.
// For system-assigned identity.
// var sqlServerTokenProvider = new DefaultAzureCredential();

// For user-assigned identity.
// var sqlServerTokenProvider = new DefaultAzureCredential(
//     new DefaultAzureCredentialOptions
//     {
//         ManagedIdentityClientId =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTID");
//     }
// );

// For service principal.
// var tenantId =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_TENANTID");
// var clientId =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTID");
// var clientSecret =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTSECRET");
// var sqlServerTokenProvider = new ClientSecretCredential(tenantId,
clientId, clientSecret);

// Acquire the access token.
AccessToken accessToken = await sqlServerTokenProvider.GetTokenAsync(
    new TokenRequestContext(scopes: new string[]
{
    "https://osssrdbms-aad.database.windows.net/.default"
}));


// Combine the token with the connection string from the environment
variables provided by Service Connector.
string connectionString =
$"{
{Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CONNECTIONSTRING")
};Password={accessToken.Token}";


// Establish the connection.
using (var connection = new NpgsqlConnection(connectionString))
{
    Console.WriteLine("Opening connection using access token...");
    connection.Open();
}

```

Next, if you have created tables and sequences in PostgreSQL flexible server before using Service Connector, you need to connect as the owner and grant permission to `<aad-username>` created by Service Connector. The username from the connection string or configuration set by Service Connector should look like `aad_<connection name>`. If you use the Azure portal, select the expand button next to the `Service Type` column and get the value. If you use Azure CLI, check `configurations` in the CLI command output.

Then, execute the query to grant permission

Azure CLI

```
az extension add --name rdbms-connect

az postgres flexible-server execute -n <postgres-name> -u <owner-username> -
p "<owner-password>" -d <database-name> --querytext "GRANT ALL PRIVILEGES ON
ALL TABLES IN SCHEMA public TO \"<aad-username>\";GRANT ALL PRIVILEGES ON
ALL SEQUENCES IN SCHEMA public TO \"<aad username>\";"
```

The `<owner-username>` and `<owner-password>` is the owner of the existing table that can grant permissions to others. `<aad-username>` is the user created by Service Connector. Replace them with the actual value.

Validate the result with the command:

Azure CLI

```
az postgres flexible-server execute -n <postgres-name> -u <owner-username> -
p "<owner-password>" -d <database-name> --querytext "SELECT
distinct(table_name) FROM information_schema.table_privileges WHERE
grantee='<aad-username>' AND table_schema='public';" --output table
```

Connection String

.NET

 Expand table

Default environment variable name	Description	Example value
<code>AZURE_POSTGRESQL_CONNECTIONSTRING</code>	.NET PostgreSQL connection string	<code>Server=<PostgreSQL-server-name>.postgres.database.azure.com;Database=<database-name>;Port=5432;SslMode=Require;User Id=<username>;</code>

Sample code

Refer to the steps and code below to connect to Azure Database for PostgreSQL using a connection string.

1. Install dependencies. Follow the guidance to [install Npgsql](#)
2. In code, get the PostgreSQL connection string from environment variables added by Service Connector service. To set TSL configurations for PostgreSQL server, refer to [these steps](#).

C#

```
using System;
using Npgsql;

string connectionString =
Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CONNECTIONSTRING");
using (NpgsqlConnection connection = new
NpgsqlConnection(connectionString))
{
    connection.Open();
}
```

Service Principal

[+] Expand table

Default environment variable name	Description	Example value
AZURE_POSTGRESQL_CLIENTID	Your client ID	<client-ID>
AZURE_POSTGRESQL_CLIENTSECRET	Your client secret	<client-secret>
AZURE_POSTGRESQL_TENANTID	Your tenant ID	<tenant-ID>
AZURE_POSTGRESQL_CONNECTIONSTRING	.NET PostgreSQL connection string	Server=<PostgreSQL-server-name>.postgres.database.azure.com;Database=<database-name>;Port=5432;SslMode=Require;User Id=<username>;

Sample code

Refer to the steps and code below to connect to Azure Database for PostgreSQL using a service principal.

.NET

For .NET, there's not a plugin or library to support passwordless connections. You can get an access token for the managed identity or service principal using client library like [Azure.Identity](#). Then you can use the access token as the password to connect to the database. When using the code below, uncomment the part of the code snippet for the authentication type you want to use.

C#

```
using Azure.Identity;
using Azure.Core;
using Npgsql;

// Uncomment the following lines according to the authentication type.
// For system-assigned identity.
// var sqlServerTokenProvider = new DefaultAzureCredential();

// For user-assigned identity.
// var sqlServerTokenProvider = new DefaultAzureCredential(
//     new DefaultAzureCredentialOptions
//     {
//         ManagedIdentityClientId =
// Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTID");
//     }
// );

// For service principal.
// var tenantId =
// Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_TENANTID");
// var clientId =
// Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTID");
// var clientSecret =
// Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CLIENTSECRET");
// var sqlServerTokenProvider = new ClientSecretCredential(tenantId,
// clientId, clientSecret);

// Acquire the access token.
AccessToken accessToken = await sqlServerTokenProvider.GetTokenAsync(
    new TokenRequestContext(scopes: new string[]
    {
        "https://osssrdbms-aad.database.windows.net/.default"
    }));
    
// Combine the token with the connection string from the environment
variables provided by Service Connector.
```

```

string connectionString =
    $""
{Environment.GetEnvironmentVariable("AZURE_POSTGRESQL_CONNECTIONSTRING")
};Password={accessToken.Token}";

// Establish the connection.
using (var connection = new NpgsqlConnection(connectionString))
{
    Console.WriteLine("Opening connection using access token...");
    connection.Open();
}

```

Next, if you have created tables and sequences in PostgreSQL flexible server before using Service Connector, you need to connect as the owner and grant permission to `<aad-username>` created by Service Connector. The username from the connection string or configuration set by Service Connector should look like `aad_<connection name>`. If you use the Azure portal, select the expand button next to the `Service Type` column and get the value. If you use Azure CLI, check `configurations` in the CLI command output.

Then, execute the query to grant permission

Azure CLI

```

az extension add --name rdbms-connect

az postgres flexible-server execute -n <postgres-name> -u <owner-username> -
p "<owner-password>" -d <database-name> --querytext "GRANT ALL PRIVILEGES ON
ALL TABLES IN SCHEMA public TO \"<aad-username>\";GRANT ALL PRIVILEGES ON
ALL SEQUENCES IN SCHEMA public TO \"<aad username>\";"

```

The `<owner-username>` and `<owner-password>` is the owner of the existing table that can grant permissions to others. `<aad-username>` is the user created by Service Connector. Replace them with the actual value.

Validate the result with the command:

Azure CLI

```

az postgres flexible-server execute -n <postgres-name> -u <owner-username> -
p "<owner-password>" -d <database-name> --querytext "SELECT
distinct(table_name) FROM information_schema.table_privileges WHERE
grantee='<aad-username>' AND table_schema='public';" --output table

```

Next steps

Follow the tutorials listed below to learn more about Service Connector.

[Learn about Service Connector concepts](#)

Configure passwordless connections between multiple Azure apps and services

Article • 02/09/2024

Applications often require secure connections between multiple Azure services simultaneously. For example, an enterprise Azure App Service instance might connect to several different storage accounts, an Azure SQL database instance, a service bus, and more.

[Managed identities](#) are the recommended authentication option for secure, passwordless connections between Azure resources. Developers do not have to manually track and manage many different secrets for managed identities, since most of these tasks are handled internally by Azure. This tutorial explores how to manage connections between multiple services using managed identities and the Azure Identity client library.

Compare the types of managed identities

Azure provides the following types of managed identities:

- **System-assigned managed identities** are directly tied to a single Azure resource. When you enable a system-assigned managed identity on a service, Azure will create a linked identity and handle administrative tasks for that identity internally. When the Azure resource is deleted, the identity is also deleted.
- **User-assigned managed identities** are independent identities that are created by an administrator and can be associated with one or more Azure resources. The lifecycle of the identity is independent of those resources.

You can read more about best practices and when to use system-assigned identities versus user-assigned identities in the [identities best practice recommendations](#).

Explore DefaultAzureCredential

Managed identities are generally implemented in your application code through a class called `DefaultAzureCredential` from the `Azure.Identity` client library.

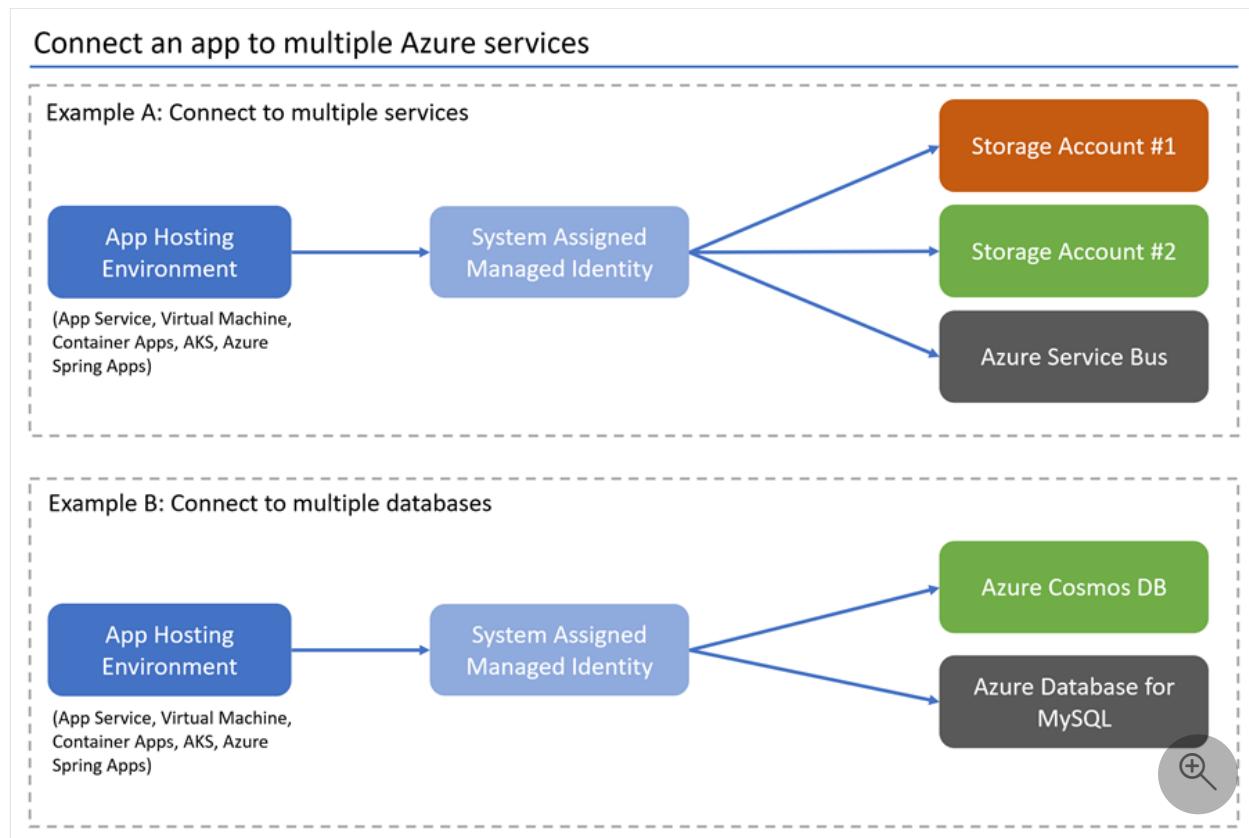
`DefaultAzureCredential` supports multiple authentication methods and automatically

determines which should be used at runtime. You can read more about this approach in the [DefaultAzureCredential overview](#).

Connect an Azure hosted app to multiple Azure services

You have been tasked with connecting an existing app to multiple Azure services and databases using passwordless connections. The application is an ASP.NET Core Web API hosted on Azure App Service, though the steps below apply to other Azure hosting environments as well, such as Azure Spring Apps, Virtual Machines, Container Apps and AKS.

This tutorial applies to the following architectures, though it can be adapted to many other scenarios as well through minimal configuration changes.



The following steps demonstrate how to configure an app to use a system-assigned managed identity and your local development account to connect to multiple Azure Services.

Create a system-assigned managed identity

1. In the Azure portal, navigate to the hosted application that you would like to connect to other services.

2. On the service overview page, select **Identity**.
3. Toggle the **Status** setting to **On** to enable a system assigned managed identity for the service.

The screenshot shows the 'Identity' blade for an Azure App Service named 'msdocs-web-app-123'. The left sidebar lists navigation options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Deployment (Quickstart, Deployment slots, Deployment Center), Settings (Configuration, Authentication, Application Insights), and Identity. The 'Identity' option is highlighted with a red box. The main content area shows the current configuration: 'System assigned' is selected under 'Managed identities', and the 'Status' switch is set to 'On'. Other fields include 'Object (principal) ID' (65634e65-f3eb-4fe7-aff3-fd31d035084b) and a 'Permissions' section for 'Azure role assignments'. A note at the bottom states: 'This resource is registered with Azure Active Directory. The managed identity can be configured to allow access to other resources.'

Assign roles to the managed identity for each connected service

1. Navigate to the overview page of the storage account you would like to grant access your identity access to.
2. Select **Access Control (IAM)** from the storage account navigation.
3. Choose **+ Add** and then **Add role assignment**.

The screenshot shows the Azure Storage account 'identitymigrationstorage' under the 'Access Control (IAM)' section. The left sidebar has a red box around the 'Access Control (IAM)' option. The main area has a red box around the 'Add role assignment' button at the top.

4. In the **Role** search box, search for *Storage Blob Data Contributor*, which grants permissions to perform read and write operations on blob data. You can assign whatever role is appropriate for your use case. Select the *Storage Blob Data Contributor* from the list and choose **Next**.
5. On the **Add role assignment** screen, for the **Assign access to** option, select **Managed identity**. Then choose **+Select members**.
6. In the flyout, search for the managed identity you created by entering the name of your app service. Select the system assigned identity, and then choose **Select** to close the flyout menu.

The screenshot shows the 'Add role assignment' wizard. Step 1 (highlighted with a red box) shows the 'Assign access to' dropdown with 'Managed identity' selected. Step 2 (highlighted with a red box) shows the 'Members' button. Step 3 (highlighted with a red box) shows the 'Select members' flyout with the search bar containing 'msdocs-web'. Step 4 (highlighted with a red box) shows the 'Selected members' list with 'msdocs-web-app-123'. Step 5 (highlighted with a red box) shows the 'Select' button. Step 6 (highlighted with a red box) shows the 'Next' button.

7. Select **Next** a couple times until you're able to select **Review + assign** to finish the role assignment.
8. Repeat this process for the other services you would like to connect to.

Local development considerations

You can also enable access to Azure resources for local development by assigning roles to a user account the same way you assigned roles to your managed identity.

1. After assigning the **Storage Blob Data Contributor** role to your managed identity, under **Assign access to**, this time select **User, group or service principal**. Choose **+ Select members** to open the flyout menu again.
2. Search for the *user@domain* account or Microsoft Entra security group you would like to grant access to by email address or name, and then select it. This should be the same account you use to sign-in to your local development tooling with, such as Visual Studio or the Azure CLI.

ⓘ Note

You can also assign these roles to a Microsoft Entra security group if you are working on a team with multiple developers. You can then place any developer inside that group who needs access to develop the app locally.

Implement the application code

C#

Inside of your project, add a reference to the `Azure.Identity` NuGet package. This library contains all of the necessary entities to implement `DefaultAzureCredential`. You can also add any other Azure libraries that are relevant to your app. For this example, the `Azure.Storage.Blobs` and `Azure.KeyVault.Keys` packages are added in order to connect to Blob Storage and Key Vault.

.NET CLI

```
dotnet add package Azure.Identity  
dotnet add package Azure.Storage.Blobs  
dotnet add package Azure.KeyVault.Keys
```

At the top of your `Program.cs` file, add the following using statements:

C#

```
using Azure.Identity;  
using Azure.Storage.Blobs;
```

```
using Azure.Security.KeyVault.Keys;
```

In the `Program.cs` file of your project code, create instances of the necessary services your app will connect to. The following examples connect to Blob Storage and service bus using the corresponding SDK classes.

C#

```
var blobServiceClient = new BlobServiceClient(  
    new Uri("https://<your-storage-account>.blob.core.windows.net"),  
    new DefaultAzureCredential(credOptions));  
  
var serviceBusClient = new ServiceBusClient("<your-namespace>", new  
DefaultAzureCredential());  
var sender = serviceBusClient.CreateSender("producttracking");
```

When this application code runs locally, `DefaultAzureCredential` will search a credential chain for the first available credentials. If the `Managed_Identity_Client_ID` is null locally, it will automatically use the credentials from your local Azure CLI or Visual Studio sign-in. You can read more about this process in the [Azure Identity library overview](#).

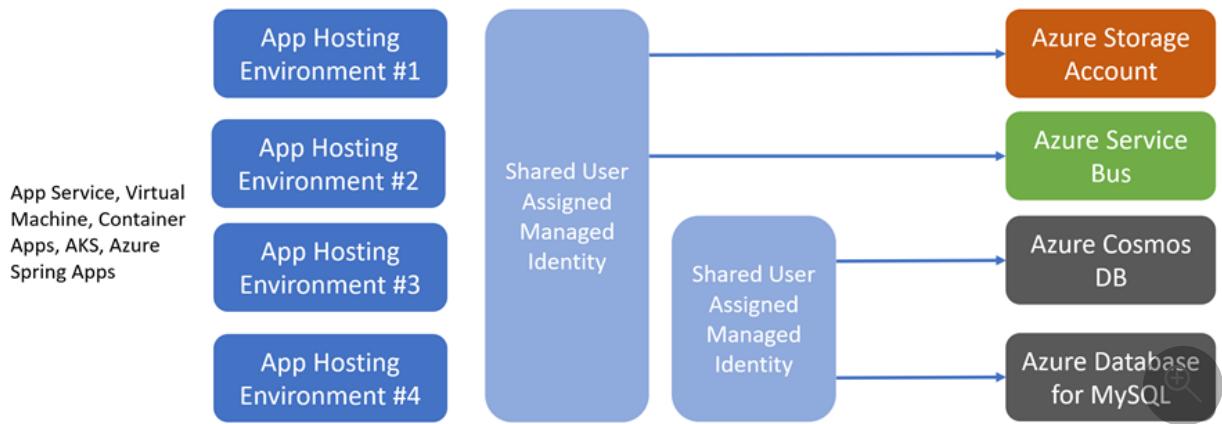
When the application is deployed to Azure, `DefaultAzureCredential` will automatically retrieve the `Managed_Identity_Client_ID` variable from the app service environment. That value becomes available when a managed identity is associated with your app.

This overall process ensures that your app can run securely locally and in Azure without the need for any code changes.

Connect multiple apps using multiple managed identities

Although the apps in the previous example all shared the same service access requirements, real environments are often more nuanced. Consider a scenario where multiple apps all connect to the same storage accounts, but two of the apps also access different services or databases.

Connect multiple apps to Azure services using shared user-assigned identities



To configure this setup in your code, make sure your application registers separate services to connect to each storage account or database. Make sure to pull in the correct managed identity client IDs for each service when configuring `DefaultAzureCredential`. The following code example configures the following service connections:

- Two connections to separate storage accounts using a shared user-assigned managed identity
- A connection to Azure Cosmos DB and Azure SQL services using a second shared user-assigned managed identity

C#

```
C#  
  
// Get the first user-assigned managed identity ID to connect to shared storage  
const clientIdStorage =  
Environment.GetEnvironmentVariable("Managed_Identity_Client_ID_Storage")  
;  
  
// First blob storage client that using a managed identity  
BlobServiceClient blobServiceClient = new BlobServiceClient(  
    new Uri("https://<receipt-storage-account>.blob.core.windows.net"),  
    new DefaultAzureCredential()  
{  
    ManagedIdentityClientId = clientIdStorage  
});  
  
// Second blob storage client that using a managed identity  
BlobServiceClient blobServiceClient2 = new BlobServiceClient(  
    new Uri("https://<contract-storage-account>.blob.core.windows.net"),  
    new DefaultAzureCredential()  
{  
    ManagedIdentityClientId = clientIdStorage  
});
```

```

});;

// Get the second user-assigned managed identity ID to connect to shared
// databases
var clientIDdatabases =
Environment.GetEnvironmentVariable("Managed_Identity_Client_ID_Databases"
");

// Create an Azure Cosmos DB client
CosmosClient client = new CosmosClient(
    accountEndpoint:
Environment.GetEnvironmentVariable("COSMOS_ENDPOINT",
EnvironmentVariableTarget.Process),
    new DefaultAzureCredential()
{
    ManagedIdentityClientId = clientIDdatabases
});

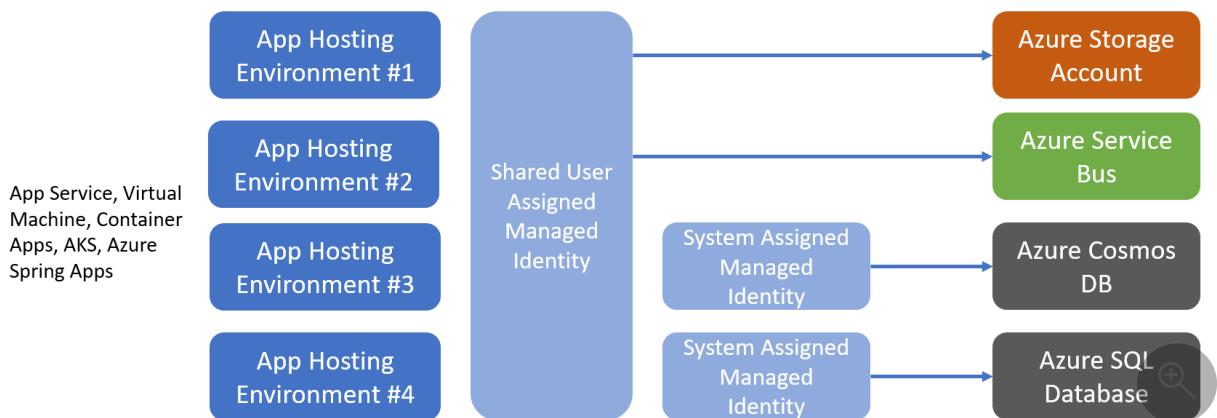
// Open a connection to Azure SQL using a managed identity
string ConnectionString1 = @"Server=<azure-sql-
hostname>.database.windows.net; User Id=ObjectIdOfManagedIdentity;
Authentication=Active Directory Default; Database=<database-name>";

using (SqlConnection conn = new SqlConnection(ConnectionString1))
{
    conn.Open();
}

```

You can also associate a user-assigned managed identity as well as a system-assigned managed identity to a resource simultaneously. This can be useful in scenarios where all of the apps require access to the same shared services, but one of the apps also has a very specific dependency on an additional service. Using a system-assigned identity also ensures that the identity tied to that specific app is deleted when the app is deleted, which can help keep your environment clean.

Connect multiple apps to Azure services using user-assigned and system-assigned identities



These types of scenarios are explored in more depth in the [identities best practice recommendations](#).

Next steps

In this tutorial, you learned how to migrate an application to passwordless connections. You can read the following resources to explore the concepts discussed in this article in more depth:

- [Authorize access to blobs using Microsoft Entra ID](#)
- To learn more about .NET Core, see [Get started with .NET in 10 minutes ↗](#).

Configure managed identities on Azure virtual machines (VMs)

Article • 05/24/2024

Managed identities for Azure resources is a feature of Microsoft Entra ID. Each of the [Azure services that support managed identities for Azure resources](#) are subject to their own timeline. Make sure you review the [availability](#) status of managed identities for your resource and [known issues](#) before you begin.

Managed identities for Azure resources provide Azure services with an automatically managed identity in Microsoft Entra ID. You can use this identity to authenticate to any service that supports Microsoft Entra authentication, without having credentials in your code.

In this article, you learn how to enable and disable system and user-assigned managed identities for an Azure Virtual Machine (VM), using the Azure portal.

Prerequisites

- If you're unfamiliar with managed identities for Azure resources, check out the [overview section](#).
- If you don't already have an Azure account, [sign up for a free account](#) before continuing.

System-assigned managed identity

In this section, you learn how to enable and disable the system-assigned managed identity for VM using the Azure portal.

Enable system-assigned managed identity during creation of a VM

To enable system-assigned managed identity on a VM during its creation, your account needs the [Virtual Machine Contributor](#) role assignment. No other Microsoft Entra directory role assignments are required.

- Under the **Management** tab in the **Identity** section, switch **Managed service identity** to **On**.

Create a virtual machine

Basics Disks Networking **Management** Guest config Tags Review + create

Configure monitoring and management options for your VM.

MONITORING

Boot diagnostics On Off

OS guest diagnostics On Off

* Diagnostics storage account

IDENTITY

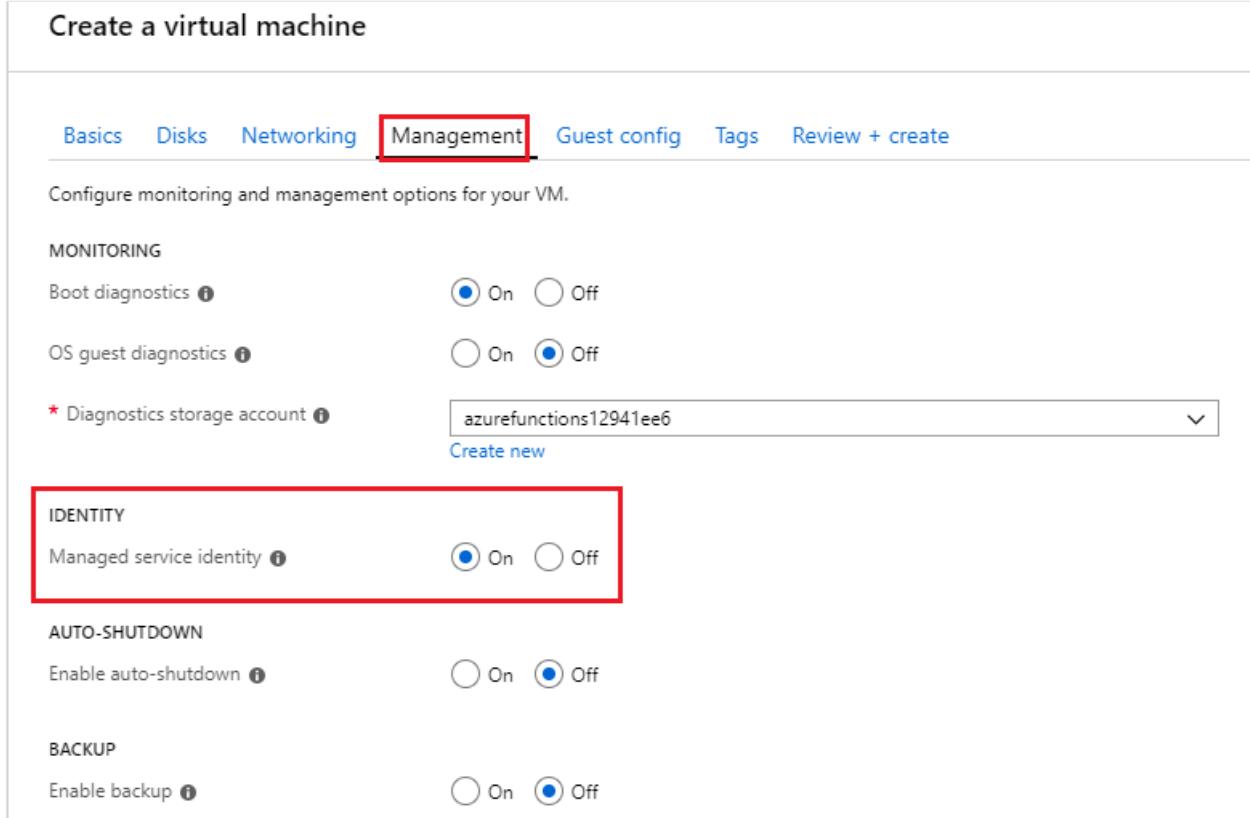
Managed service identity On Off

AUTO-SHUTDOWN

Enable auto-shutdown On Off

BACKUP

Enable backup On Off



Refer to the following Quickstarts to create a VM:

- Create a Windows virtual machine with the Azure portal
- Create a Linux virtual machine with the Azure portal

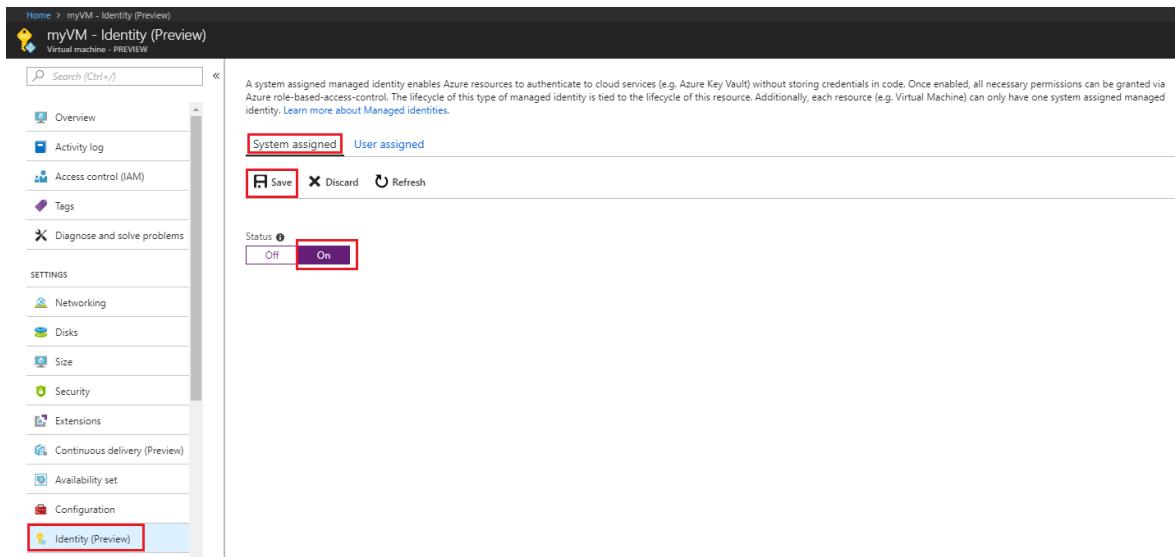
Enable system-assigned managed identity on an existing VM

Tip

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

To enable system-assigned managed identity on a VM that was originally provisioned without it, your account needs the [Virtual Machine Contributor](#) role assignment. No other Microsoft Entra directory role assignments are required.

1. Sign in to the [Azure portal](#) using an account associated with the Azure subscription that contains the VM.
2. Navigate to the desired Virtual Machine and select **Identity**.
3. Under **System assigned, Status**, select **On** and then click **Save**:

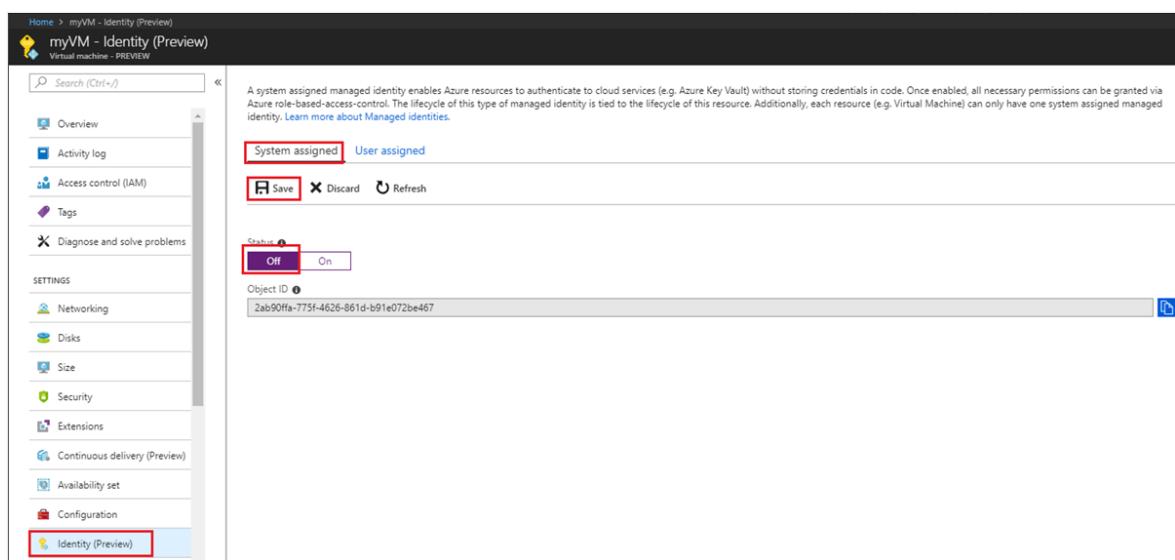


Remove system-assigned managed identity from a VM

To remove system-assigned managed identity from a VM, your account needs the [Virtual Machine Contributor](#) role assignment. No other Microsoft Entra directory role assignments are required.

If you have a Virtual Machine that no longer needs system-assigned managed identity:

1. Sign in to the [Azure portal](#) using an account associated with the Azure subscription that contains the VM.
2. Navigate to the desired Virtual Machine and select **Identity**.
3. Under **System assigned**, **Status**, select **Off** and then click **Save**:



User-assigned managed identity

In this section, you learn how to add and remove a user-assigned managed identity from a VM using the Azure portal.

Assign a user-assigned identity during the creation of a VM

To assign a user-assigned identity to a VM, your account needs the [Virtual Machine Contributor](#) and [Managed Identity Operator](#) role assignments. No other Microsoft Entra directory role assignments are required.

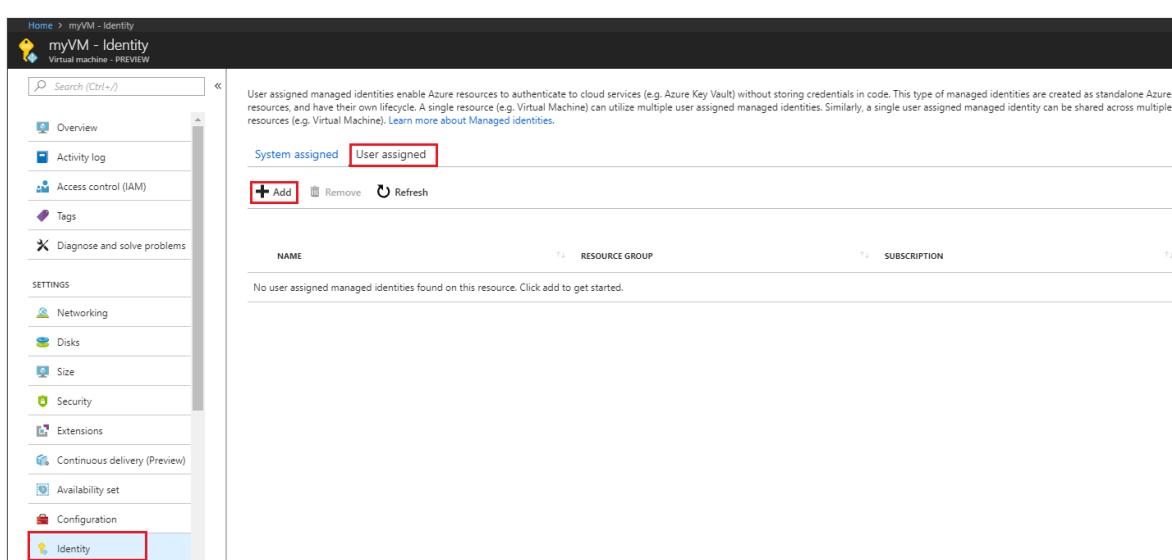
Currently, the Azure portal does not support assigning a user-assigned managed identity during the creation of a VM. Instead, refer to one of the following VM creation Quickstart articles to first create a VM, and then proceed to the next section for details on assigning a user-assigned managed identity to the VM:

- [Create a Windows virtual machine with the Azure portal](#)
- [Create a Linux virtual machine with the Azure portal](#)

Assign a user-assigned managed identity to an existing VM

To assign a user-assigned identity to a VM, your account needs the [Virtual Machine Contributor](#) and [Managed Identity Operator](#) role assignments. No other Microsoft Entra directory role assignments are required.

1. Sign in to the [Azure portal](#) using an account associated with the Azure subscription that contains the VM.
2. Navigate to the desired VM and click **Identity**, **User assigned** and then **+Add**.



3. Click the user-assigned identity you want to add to the VM and then click **Add**.

The screenshot shows the 'Add user assigned managed identity' dialog box. At the top, it says 'PREVIEW'. Below that, there's a dropdown for 'Subscription' set to 'Woodgrove IT Production Environment'. A search bar labeled 'Filter by identity name and/or resource group name' is present. A list of identities is shown, with 'ID1' selected and highlighted with a red box. The other identities listed are 'cyibarravmexid', 'devtestmsi', 'platAppIdentity', 'MKTG-UA-01', and 'MKTG-UA-02'. Below the list, a 'Selected identities:' section contains 'ID1 TestRG', also highlighted with a red box. To the right of this section is a 'Remove' button. At the bottom of the dialog is a blue 'Add' button, which is also highlighted with a red box.

Remove a user-assigned managed identity from a VM

To remove a user-assigned identity from a VM, your account needs the [Virtual Machine Contributor](#) role assignment. No other Microsoft Entra directory role assignments are required.

1. Sign in to the [Azure portal](#) using an account associated with the Azure subscription that contains the VM.

2. Navigate to the desired VM and select **Identity**, **User assigned**, the name of the user-assigned managed identity you want to delete and then click **Remove** (click **Yes** in the confirmation pane).

The screenshot shows the Azure portal interface for managing identities of a virtual machine named 'myVM'. On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, and Configuration. The main area is titled 'myVM - Identity' and has a sub-header 'Virtual machine - PREVIEW'. It displays information about user-assigned managed identities. A note at the top explains that these identities enable authentication to cloud services without storing credentials in code. Below this, there are tabs for 'System assigned' and 'User assigned', with 'User assigned' being the active tab. Underneath are buttons for '+ Add', 'Remove', and 'Refresh'. A table lists identities, showing columns for NAME, RESOURCE GROUP, and SUBSCRIPTION. The identity 'IDI' is listed with a checkmark in the NAME column and is highlighted with a red box. The RESOURCE GROUP is 'TestRG' and the SUBSCRIPTION is '<SUBSCRIPTION ID>'. At the bottom of the table, there's a 'Delete' button.

Next steps

- Using the Azure portal, give an Azure VM's managed identity [access to another Azure resource](#).

Connect to and query Azure SQL Database using .NET and Entity Framework Core

Article • 05/21/2024

Applies to:  Azure SQL Database

This quickstart describes how to connect an application to a database in Azure SQL Database and perform queries using .NET and Entity Framework Core. This quickstart follows the recommended passwordless approach to connect to the database. You can learn more about passwordless connections on the [passwordless hub](#).

Prerequisites

- An [Azure subscription](#).
- A SQL database configured for authentication with Microsoft Entra ID ([formerly Azure Active Directory](#)). You can create one using the [Create database quickstart](#).
- [.NET 7.0](#) or later.
- [Visual Studio](#) or later with the **ASP.NET and web development** workload.
- The latest version of the [Azure CLI](#).
- The latest version of the Entity Framework Core tools:
 - Visual Studio users should install the [Package Manager Console tools for Entity Framework Core](#).
 - .NET CLI users should install the [.NET CLI tools for Entity Framework Core](#).

Configure the database server

Secure, passwordless connections to Azure SQL Database require certain database configurations. Verify the following settings on your [logical server in Azure](#) to properly connect to Azure SQL Database in both local and hosted environments:

1. For local development connections, make sure your logical server is configured to allow your local machine IP address and other Azure services to connect:
 - Navigate to the **Networking** page of your server.
 - Toggle the **Selected networks** radio button to show additional configuration options.

- Select **Add your client IPv4 address(xx.xx.xx.xx)** to add a firewall rule that will enable connections from your local machine IPv4 address. Alternatively, you can also select **+ Add a firewall rule** to enter a specific IP address of your choice.
- Make sure the **Allow Azure services and resources to access this server** checkbox is selected.

The screenshot shows the Azure portal's 'Networking' settings for a database. The left sidebar shows various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Settings (selected), Azure Active Directory, SQL databases, SQL elastic pools, DTU quota, Properties, Locks, Data management (Backups, Deleted databases, Failover groups, Import/Export history), Security (Networking selected), Microsoft Defender for Cloud, Transparent data encryption, Identity, Auditing, and Intelligent Performance. The main pane shows 'Public access' tab selected. Under 'Public network access', the 'Selected networks' radio button is selected (highlighted by a red box). Under 'Virtual networks', there is a link to 'Add a virtual network rule'. Under 'Firewall rules', there is a link to 'Add your client IPv4 address' (highlighted by a red box). Under 'Exceptions', the 'Allow Azure services and resources to access this server' checkbox is checked (highlighted by a red box). A search icon is visible in the bottom right corner.

⚠ Warning

Enabling the **Allow Azure services and resources to access this server** setting is not a recommended security practice for production scenarios. Real applications should implement more secure approaches, such as stronger firewall restrictions or virtual network configurations.

You can read more about database security configurations on the following resources:

- [Configure Azure SQL Database firewall rules](#).
- [Configure a virtual network with private endpoints](#).

2. The server must also have Microsoft Entra authentication enabled and have a Microsoft Entra admin account assigned. For local development connections, the Microsoft Entra admin account should be an account you can also log into Visual Studio or the Azure CLI with locally. You can verify whether your server has Microsoft Entra authentication enabled on the **Microsoft Entra ID** page of your logical server.

The screenshot shows the Microsoft Entra admin center interface. At the top, there's a search bar and navigation links for 'Set admin', 'Remove admin', and 'Save'. Below this, the 'Microsoft Entra admin' section is displayed, with a note about centrally managing identity and access to Azure SQL Database. It shows an 'Admin name' as testing123@microsoft.com. Under 'Microsoft Entra authentication only', it states that only Microsoft Entra ID will be used for authentication, and SQL authentication will be disabled. A checkbox for 'Support only Microsoft Entra authentication for this server' is checked. On the left, a sidebar lists 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', and 'Settings'. The 'Microsoft Entra ID' option under 'Settings' is highlighted with a red box. On the right, there's a magnifying glass icon.

3. If you're using a personal Azure account, make sure you have [Microsoft Entra setup and configured for Azure SQL Database](#) in order to assign your account as a server admin. If you're using a corporate account, Microsoft Entra ID will most likely already be configured for you.

Create the project

The steps in this section create a .NET Minimal Web API by using either the .NET CLI or Visual Studio 2022.

Visual Studio

1. In the Visual Studio menu bar, navigate to **File > New > Project...**
2. In the dialog window, enter *ASP.NET* into the project template search box and select the ASP.NET Core Web API result. Choose **Next** at the bottom of the dialog.
3. For the **Project Name**, enter *DotNetSQL*. Leave the default values for the rest of the fields and select **Next**.
4. For the **Framework**, select *.NET 7.0* and uncheck **Use controllers (uncheck to use minimal APIs)**. This quickstart uses a Minimal API template to streamline endpoint creation and configuration.
5. Choose **Create**. The new project opens inside the Visual Studio environment.

Add Entity Framework Core to the project

To connect to Azure SQL Database by using .NET and Entity Framework Core, you need to add three NuGet packages to your project using one of the following methods:

1. In the **Solution Explorer** window, right-click the project's **Dependencies** node and select **Manage NuGet Packages**.
2. In the resulting window, search for *EntityFrameworkCore*. Locate and install the following packages:
 - **Microsoft.EntityFrameworkCore**: Provides essential Entity Framework Core functionality
 - **Microsoft.EntityFrameworkCore.SqlServer**: Provides extra components to connect to the logical server
 - **Microsoft.EntityFrameworkCore.Design**: Provides support for running Entity Framework migrations

Alternatively, you can also run the `Install-Package` cmdlet in the **Package Manager Console** window:

PowerShell

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Design
```

Add the code to connect to Azure SQL Database

The Entity Framework Core libraries rely on the `Microsoft.Data.SqlClient` and `Azure.Identity` libraries to implement passwordless connections to Azure SQL Database. The `Azure.Identity` library provides a class called `DefaultAzureCredential` that handles passwordless authentication to Azure.

`DefaultAzureCredential` supports multiple authentication methods and determines which to use at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code. The [Azure Identity library overview](#) explains the order and locations in which `DefaultAzureCredential` looks for credentials.

Complete the following steps to connect to Azure SQL Database using Entity Framework Core and the underlying `DefaultAzureCredential` class:

1. Add a `ConnectionStrings` section to the `appsettings.Development.json` file so that it matches the following code. Remember to update the `<your database-server-name>` and `<your-database-name>` placeholders.

The passwordless connection string includes a configuration value of `Authentication=Active Directory Default`, which enables Entity Framework Core to use `DefaultAzureCredential` to connect to Azure services. When the app runs locally, it authenticates with the user you're signed into Visual Studio with. Once the app deploys to Azure, the same code discovers and applies the managed identity that is associated with the hosted app, which you'll configure later.

 **Note**

Passwordless connection strings are safe to commit to source control, since they do not contain any secrets such as usernames, passwords, or access keys.

JSON

```
{  
    "Logging": {  
        "LogLevel": {  
            "Default": "Information",  
            "Microsoft.AspNetCore": "Warning"  
        }  
    },  
    "ConnectionStrings": {  
        "AZURE_SQL_CONNECTIONSTRING": "Data  
Source=passwordlessdbserver.database.windows.net;  
Initial Catalog=passwordlessdb; Authentication=Active  
Directory Default; Encrypt=True;"  
    }  
}
```

2. Add the following code to the `Program.cs` file above the line of code that reads

```
var app = builder.Build();
```

. This code performs the following configurations:

- Retrieves the passwordless database connection string from the `appsettings.Development.json` file for local development, or from the environment variables for hosted production scenarios.
- Registers the Entity Framework Core `DbContext` class with the .NET dependency injection container.

C#

```

var connection = String.Empty;
if (builder.Environment.IsDevelopment())
{
    connection =
        builder.Configuration.AddEnvironmentVariables().AddJsonFile("appsettings.Development.json");
}
else
{
    connection =
        Environment.GetEnvironmentVariable("AZURE_SQL_CONNECTIONSTRING");
}

builder.Services.AddDbContext<PersonDbContext>(options =>
    options.UseSqlServer(connection));

```

3. Add the following endpoints to the bottom of the `Program.cs` file above `app.Run()` to retrieve and add entities in the database using the `PersonDbContext` class.

C#

```

app.MapGet("/Person", (PersonDbContext context) =>
{
    return context.Person.ToList();
})
.WithName("GetPersons")
.WithOpenApi();

app.MapPost("/Person", (Person person, PersonDbContext context) =>
{
    context.Add(person);
    context.SaveChanges();
})
.WithName("CreatePerson")
.WithOpenApi();

```

Finally, add the `Person` and `PersonDbContext` classes to the bottom of the `Program.cs` file. The `Person` class represents a single record in the database's `Persons` table. The `PersonDbContext` class represents the `Person` database and allows you to perform operations on it through code. You can read more about `DbContext` in the [Getting Started](#) documentation for Entity Framework Core.

C#

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

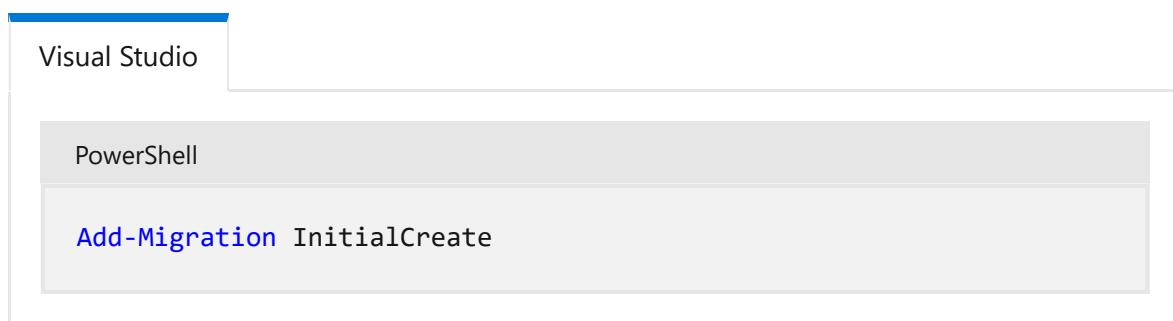
public class PersonDbContext : DbContext
{
    public PersonDbContext(DbContextOptions<PersonDbContext> options)
        : base(options)
    {
    }

    public DbSet<Person> Person { get; set; }
}
```

Run the migrations to create the database

To update the database schema to match your data model using Entity Framework Core, you must use a migration. Migrations can create and incrementally update a database schema to keep it in sync with your application's data model. You can learn more about this pattern in the [migrations overview](#).

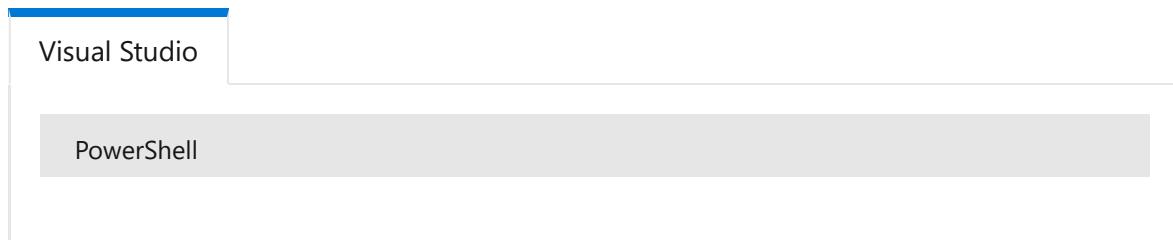
1. Open a terminal window to the root of your project.
2. Run the following command to generate an initial migration that can create the database:



```
Visual Studio

PowerShell
Add-Migration InitialCreate
```

3. A `Migrations` folder should appear in your project directory, along with a file called `InitialCreate` with unique numbers prepended. Run the migration to create the database using the following command:



```
Visual Studio

PowerShell
Add-Migration InitialCreate
```

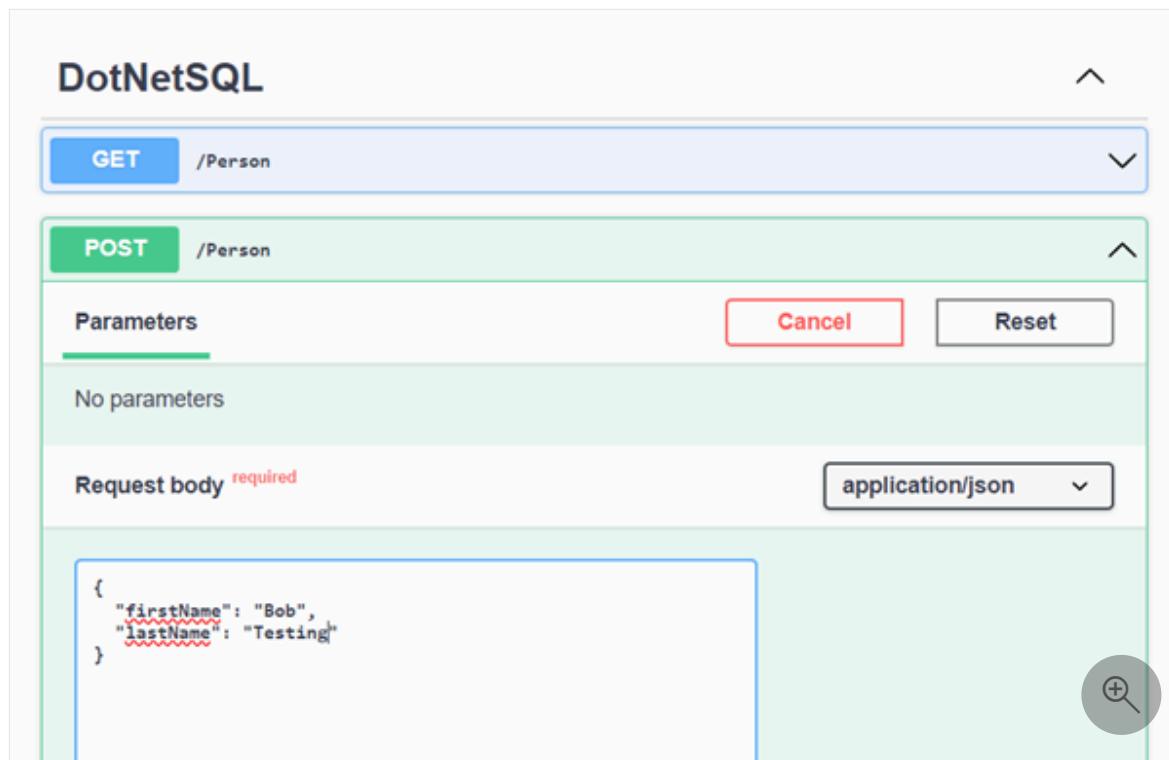
Update-Database

The Entity Framework Core tooling will create the database schema in Azure defined by the `PersonDbContext` class.

Test the app locally

The app is ready to be tested locally. Make sure you're signed in to Visual Studio or the Azure CLI with the same account you set as the admin for your database.

1. Press the run button at the top of Visual Studio to launch the API project.
2. On the Swagger UI page, expand the POST method and select Try it.
3. Modify the sample JSON to include values for the first and last name. Select Execute to add a new record to the database. The API returns a successful response.

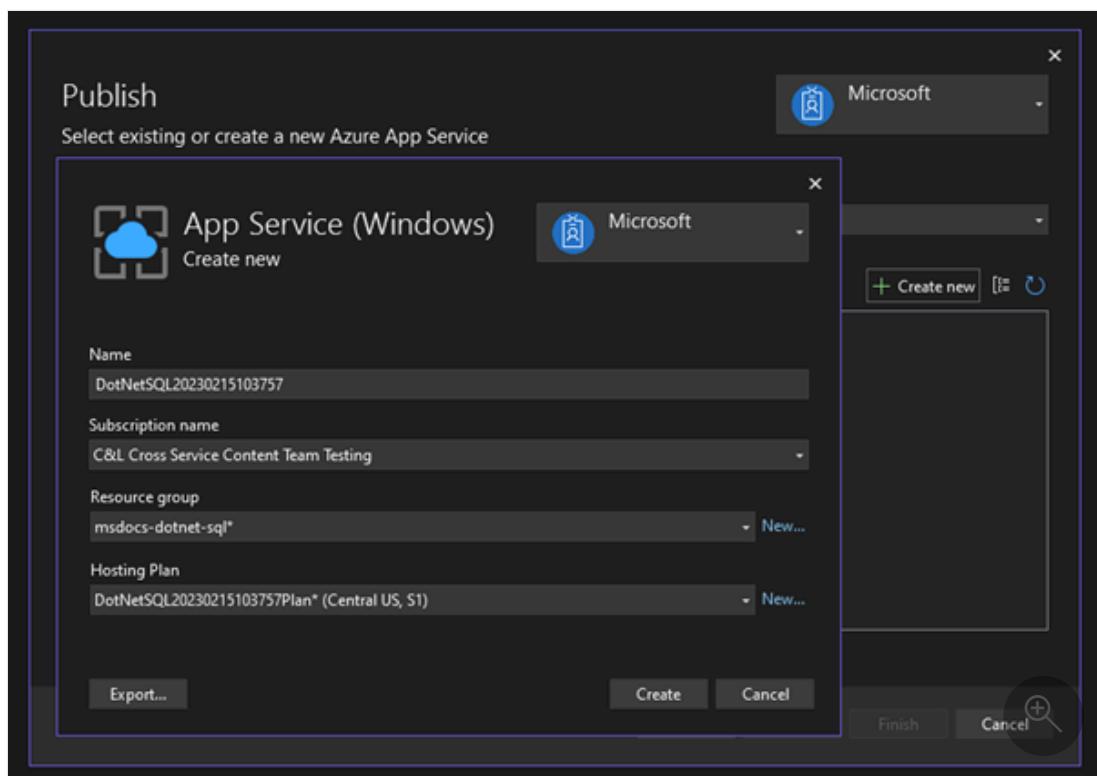


4. Expand the **GET** method on the Swagger UI page and select **Try it**. Select **Execute**, and the person you just created is returned.

Deploy to Azure App Service

The app is ready to be deployed to Azure. Visual Studio can create an Azure App Service and deploy your application in a single workflow.

1. Make sure the app is stopped and builds successfully.
2. In Visual Studio's **Solution Explorer** window, right-click on the top-level project node and select **Publish**.
3. In the publishing dialog, select **Azure** as the deployment target, and then select **Next**.
4. For the specific target, select **Azure App Service (Windows)**, and then select **Next**.
5. Select the green + icon to create a new App Service to deploy to and enter the following values:
 - **Name:** Leave the default value.
 - **Subscription name:** Select the subscription to deploy to.
 - **Resource group:** Select **New** and create a new resource group called *msdocs-dotnet-sql*.
 - **Hosting Plan:** Select **New** to open the hosting plan dialog. Leave the default values and select **OK**.
 - Select **Create** to close the original dialog. Visual Studio creates the App Service resource in Azure.



- Once the resource is created, make sure it's selected in the list of app services, and then select **Next**.
- On the **API Management** step, select the **Skip this step** checkbox at the bottom and then select **Finish**.
- Select **Publish** in the upper right of the publishing profile summary to deploy the app to Azure.

When the deployment finishes, Visual Studio launches the browser to display the hosted app, but at this point the app doesn't work correctly on Azure. You still need to configure the secure connection between the App Service and the SQL database to retrieve your data.

Connect the App Service to Azure SQL Database

The following steps are required to connect the App Service instance to Azure SQL Database:

- Create a managed identity for the App Service. The `Microsoft.Data.SqlClient` library included in your app will automatically discover the managed identity, just like it discovered your local Visual Studio user.
- Create a SQL database user and associate it with the App Service managed identity.
- Assign SQL roles to the database user that allow for read, write, and potentially other permissions.

There are multiple tools available to implement these steps:

Service Connector (Recommended)

Service Connector is a tool that streamlines authenticated connections between different services in Azure. Service Connector currently supports connecting an App Service to a SQL database via the Azure CLI using the `az webapp connection create sql` command. This single command completes the three steps mentioned above for you.

Azure CLI

```
az webapp connection create sql
-g <your-resource-group>
-n <your-app-service-name>
```

```
--tg <your-database-server-resource-group>
--server <your-database-server-name>
--database <your-database-name>
--system-identity
```

You can verify the changes made by Service Connector on the App Service settings.

1. Navigate to the **Identity** page for your App Service. Under the **System assigned** tab, the **Status** should be set to **On**. This value means that a system-assigned managed identity was enabled for your app.
2. Navigate to the **Configuration** page for your App Service. Under the **Connection strings** tab, you should see a connection string called **AZURE_SQL_CONNECTIONSTRING**. Select the **Click to show value** text to view the generated passwordless connection string. The name of this connection string aligns with the one you configured in your app, so it will be discovered automatically when running in Azure.

Important

Although this solution provides a simple approach for getting started, it is not a best practice for enterprise production environments. In those scenarios the app should not perform all operations using a single, elevated identity. You should try to implement the principle of least privilege by configuring multiple identities with specific permissions for specific tasks.

You can read more about configuring database roles and security on the following resources:

[Tutorial: Secure a database in Azure SQL Database](#)

[Authorize database access to SQL Database](#)

Test the deployed application

Browse to the URL of the app to test that the connection to Azure SQL Database is working. You can locate the URL of your app on the App Service overview page. Append the `/person` path to the end of the URL to browse to the same endpoint you tested locally.

The person you created locally should display in the browser. Congratulations! Your application is now connected to Azure SQL Database in both local and hosted environments.

Clean up the resources

When you are finished working with the Azure SQL Database, delete the resource to avoid unintended costs.

Azure portal

1. In the Azure portal search bar, search for *Azure SQL* and select the matching result.
2. Locate and select your database in the list of databases.
3. On the **Overview** page of your Azure SQL Database, select **Delete**.
4. On the **Azure you sure you want to delete...** page that opens, type the name of your database to confirm, and then select **Delete**.

Related content

- [Tutorial: Secure a database in Azure SQL Database](#)
- [Authorize database access to SQL Database](#)
- [An overview of Azure SQL Database security capabilities](#)
- [Azure SQL Database security best practices](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Connect to and query Azure SQL Database using .NET and the Microsoft.Data.SqlClient library

Article • 07/11/2023

Applies to:  Azure SQL Database

This quickstart describes how to connect an application to a database in Azure SQL Database and perform queries using .NET and the [Microsoft.Data.SqlClient](#) library. This quickstart follows the recommended passwordless approach to connect to the database. You can learn more about passwordless connections on the [passwordless hub](#).

Prerequisites

- An [Azure subscription](#).
- An Azure SQL database configured for authentication with Microsoft Entra ID (formerly Azure Active Directory). You can create one using the [Create database quickstart](#).
- The latest version of the [Azure CLI](#).
- [Visual Studio](#) or later with the **ASP.NET and web development** workload.
- [.NET 7.0](#) or later.

Configure the database

Secure, passwordless connections to Azure SQL Database require certain database configurations. Verify the following settings on your [logical server in Azure](#) to properly connect to Azure SQL Database in both local and hosted environments:

1. For local development connections, make sure your logical server is configured to allow your local machine IP address and other Azure services to connect:
 - Navigate to the **Networking** page of your server.
 - Toggle the **Selected networks** radio button to show additional configuration options.
 - Select **Add your client IPv4 address(xx.xx.xx.xx)** to add a firewall rule that will enable connections from your local machine IPv4 address. Alternatively,

you can also select **+ Add a firewall rule** to enter a specific IP address of your choice.

- Make sure the **Allow Azure services and resources to access this server** checkbox is selected.

The screenshot shows the Azure portal's 'Public access' settings for a database. On the left, the 'Networking' section is highlighted with a red box. In the main area, the 'Selected networks' radio button is selected, indicated by a red box. Below it, the 'Add a virtual network rule' button is also highlighted with a red box. Under the 'Firewall rules' section, the 'Allow Azure services and resources to access this server' checkbox is checked and highlighted with a red box. A magnifying glass icon is in the bottom right corner.

⚠️ Warning

Enabling the **Allow Azure services and resources to access this server** setting is not a recommended security practice for production scenarios. Real applications should implement more secure approaches, such as stronger firewall restrictions or virtual network configurations.

You can read more about database security configurations on the following resources:

- [Configure Azure SQL Database firewall rules.](#)
- [Configure a virtual network with private endpoints.](#)

2. The server must also have Microsoft Entra authentication enabled and have a Microsoft Entra admin account assigned. For local development connections, the Microsoft Entra admin account should be an account you can also log into Visual Studio or the Azure CLI with locally. You can verify whether your server has Microsoft Entra authentication enabled on the **Microsoft Entra ID** page of your logical server.

The screenshot shows the Microsoft Entra admin center interface. At the top, there's a search bar and navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, and Settings. Under Settings, the 'Microsoft Entra ID' option is selected and highlighted with a red box. On the right, there's a section titled 'Microsoft Entra authentication only' with a checkbox labeled 'Support only Microsoft Entra authentication for this server'. A large circular button with a magnifying glass icon is visible in the bottom right corner.

3. If you're using a personal Azure account, make sure you have [Microsoft Entra setup and configured for Azure SQL Database](#) in order to assign your account as a server admin. If you're using a corporate account, Microsoft Entra ID will most likely already be configured for you.

Create the project

For the steps ahead, create a .NET Minimal Web API using either the .NET CLI or Visual Studio 2022.

The screenshot shows the Visual Studio interface with a 'Visual Studio' tab selected. A numbered list of steps is overlaid on the screen:

1. In the Visual Studio menu, navigate to **File > New > Project...**
2. In the dialog window, enter *ASP.NET* into the project template search box and select the ASP.NET Core Web API result. Choose **Next** at the bottom of the dialog.
3. For the **Project Name**, enter *DotNetSQL*. Leave the default values for the rest of the fields and select **Next**.
4. For the **Framework**, select .NET 7.0 and uncheck **Use controllers (uncheck to use minimal APIs)**. This quickstart uses a Minimal API template to streamline endpoint creation and configuration.
5. Choose **Create**. The new project opens inside the Visual Studio environment.

Add the Microsoft.Data.SqlClient library

To connect to Azure SQL Database by using .NET, install [Microsoft.Data.SqlClient](#). This package acts as a data provider for connecting to databases, executing commands, and

retrieving results.

ⓘ Note

Make sure to install `Microsoft.Data.SqlClient` and not `System.Data.SqlClient`. `Microsoft.Data.SqlClient` is a newer version of the SQL client library that provides additional capabilities.

Visual Studio

1. In the **Solution Explorer** window, right-click the project's **Dependencies** node and select **Manage NuGet Packages**.
2. In the resulting window, search for *SqlClient*. Locate the `Microsoft.Data.SqlClient` result and select **Install**.

Configure the connection string

Passwordless (Recommended)

For local development with passwordless connections to Azure SQL Database, add the following `ConnectionStrings` section to the `appsettings.json` file. Replace the `<database-server-name>` and `<database-name>` placeholders with your own values.

JSON

```
"ConnectionStrings": {  
    "AZURE_SQL_CONNECTIONSTRING": "Server=tcp:<database-server-name>.database.windows.net,1433;Initial Catalog=<database-name>;Encrypt=True;TrustServerCertificate=False;ConnectionTimeout=30;Authentication=\"Active Directory Default\";"  
}
```

The passwordless connection string sets a configuration value of `Authentication="Active Directory Default"`, which instructs the `Microsoft.Data.SqlClient` library to connect to Azure SQL Database using a class called `DefaultAzureCredential`. `DefaultAzureCredential` enables passwordless connections to Azure services and is provided by the Azure Identity library on which the SQL client library depends. `DefaultAzureCredential` supports multiple

authentication methods and determines which to use at runtime for different environments.

For example, when the app runs locally, `DefaultAzureCredential` authenticates via the user you're signed into Visual Studio with, or other local tools like the Azure CLI. Once the app deploys to Azure, the same code discovers and applies the managed identity that is associated with the hosted app, which you'll configure later. The [Azure Identity library overview](#) explains the order and locations in which `DefaultAzureCredential` looks for credentials.

 **Note**

Passwordless connection strings are safe to commit to source control, since they don't contain secrets such as usernames, passwords, or access keys.

Add the code to connect to Azure SQL Database

Replace the contents of the `Program.cs` file with the following code, which performs the following important steps:

- Retrieves the passwordless connection string from `appsettings.json`
- Creates a `Persons` table in the database during startup (for testing scenarios only)
- Creates an HTTP GET endpoint to retrieve all records stored in the `Persons` table
- Creates an HTTP POST endpoint to add new records to the `Persons` table

C#

```
using Microsoft.Data.SqlClient;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// For production scenarios, consider keeping Swagger configurations behind
// the environment check
// if (app.Environment.IsDevelopment())
// {
    app.UseSwagger();
    app.UseSwaggerUI();
```

```
// }

app.UseHttpsRedirection();

string connectionString =
app.Configuration.GetConnectionString("AZURE_SQL_CONNECTIONSTRING")!;

try
{
    // Table would be created ahead of time in production
    using var conn = new SqlConnection(connectionString);
    conn.Open();

    var command = new SqlCommand(
        "CREATE TABLE Persons (ID int NOT NULL PRIMARY KEY IDENTITY,
FirstName varchar(255), LastName varchar(255));",
        conn);
    using SqlDataReader reader = command.ExecuteReader();
}

catch (Exception e)
{
    // Table may already exist
    Console.WriteLine(e.Message);
}

app.MapGet("/Person", () => {
    var rows = new List<string>();

    using var conn = new SqlConnection(connectionString);
    conn.Open();

    var command = new SqlCommand("SELECT * FROM Persons", conn);
    using SqlDataReader reader = command.ExecuteReader();

    if (reader.HasRows)
    {
        while (reader.Read())
        {
            rows.Add($"{reader.GetInt32(0)}, {reader.GetString(1)},
{reader.GetString(2)}");
        }
    }
    return rows;
})
.WithName("GetPersons")
.WithOpenApi();

app.MapPost("/Person", (Person person) => {
    using var conn = new SqlConnection(connectionString);
    conn.Open();

    var command = new SqlCommand(
        "INSERT INTO Persons (firstName, lastName) VALUES (@firstName,
@lastName)",
```

```
conn);

command.Parameters.Clear();
command.Parameters.AddWithValue("@firstName", person.FirstName);
command.Parameters.AddWithValue("@lastName", person.LastName);

using SqlDataReader reader = command.ExecuteReader();
}

.WithName("CreatePerson")
.WithOpenApi();

app.Run();
```

Finally, add the `Person` class to the bottom of the `Program.cs` file. This class represents a single record in the database's `Persons` table.

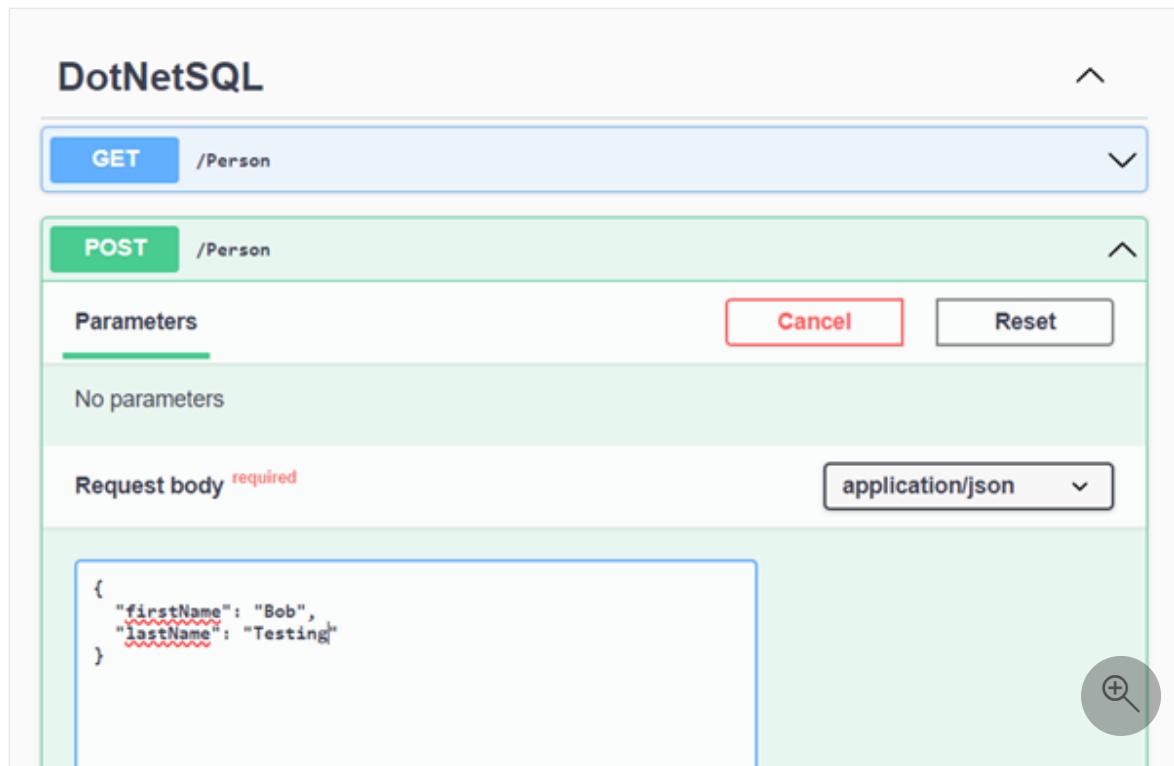
C#

```
public class Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
}
```

Run and test the app locally

The app is ready to be tested locally. Make sure you're signed in to Visual Studio or the Azure CLI with the same account you set as the admin for your database.

1. Press the run button at the top of Visual Studio to launch the API project.
2. On the Swagger UI page, expand the POST method and select **Try it**.
3. Modify the sample JSON to include values for the first and last name. Select **Execute** to add a new record to the database. The API returns a successful response.



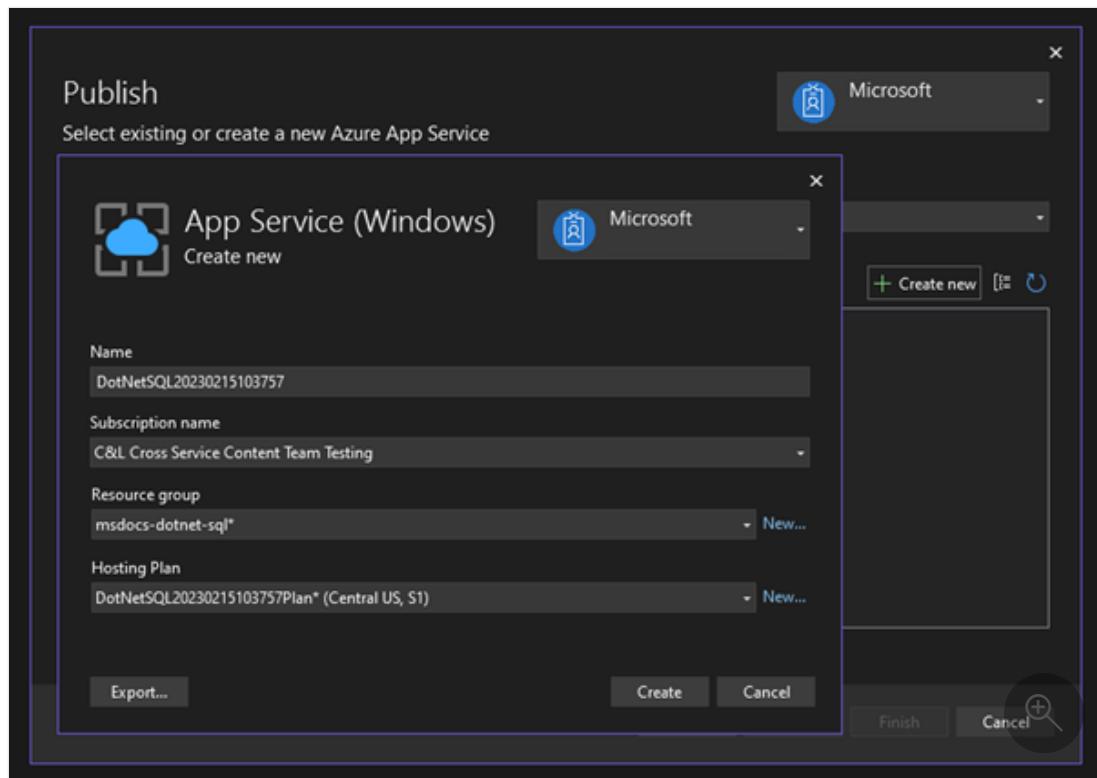
4. Expand the **GET** method on the Swagger UI page and select **Try it**. Choose **Execute**, and the person you just created is returned.

Deploy to Azure App Service

The app is ready to be deployed to Azure. Visual Studio can create an Azure App Service and deploy your application in a single workflow.

1. Make sure the app is stopped and builds successfully.
2. In Visual Studio's **Solution Explorer** window, right-click on the top-level project node and select **Publish**.
3. In the publishing dialog, select **Azure** as the deployment target, and then select **Next**.
4. For the specific target, select **Azure App Service (Windows)**, and then select **Next**.
5. Select the + icon to create a new App Service to deploy to and enter the following values:
 - **Name:** Leave the default value.
 - **Subscription name:** Select the subscription to deploy to.
 - **Resource group:** Select **New** and create a new resource group called *msdocs-dotnet-sql*.

- **Hosting Plan:** Select **New** to open the hosting plan dialog. Leave the default values and select **OK**.
- Select **Create** to close the original dialog. Visual Studio creates the App Service resource in Azure.



6. Once the resource is created, make sure it's selected in the list of app services, and then select **Next**.
7. On the **API Management** step, select the **Skip this step** checkbox at the bottom and then choose **Finish**.
8. On the **Finish** step, select **Close** if the dialog does not close automatically.
9. Select **Publish** in the upper right of the publishing profile summary to deploy the app to Azure.

When the deployment finishes, Visual Studio launches the browser to display the hosted app, but at this point the app doesn't work correctly on Azure. You still need to configure the secure connection between the App Service and the SQL database to retrieve your data.

Connect the App Service to Azure SQL Database

Passwordless (Recommended)

The following steps are required to create a passwordless connection between the App Service instance and Azure SQL Database:

1. Create a managed identity for the App Service. The `Microsoft.Data.SqlClient` library included in your app will automatically discover the managed identity, just like it discovered your local Visual Studio user.
2. Create a SQL database user and associate it with the App Service managed identity.
3. Assign SQL roles to the database user that allow for read, write, and potentially other permissions.

There are multiple tools available to implement these steps:

Service Connector (Recommended)

Service Connector is a tool that streamlines authenticated connections between different services in Azure. Service Connector currently supports connecting an App Service to a SQL database via the Azure CLI using the `az webapp connection create sql` command. This single command completes the three steps mentioned above for you.

Azure CLI

```
az webapp connection create sql \
-g <app-service-resource-group> \
-n <app-service-name> \
--tg <database-server-resource-group> \
--server <database-server-name> \
--database <database-name> \
--system-identity
```

You can verify the changes made by Service Connector on the App Service settings.

1. Navigate to the **Identity** page for your App Service. Under the **System assigned** tab, the **Status** should be set to **On**. This value means that a system-assigned managed identity was enabled for your app.
2. Navigate to the **Configuration** page for your App Service. Under the **Connection strings** tab, you should see a connection string called **AZURE_SQL_CONNECTIONSTRING**. Select the **Click to show value** text to view the generated passwordless connection string. The name of this

connection string matches the one you configured in your app, so it will be discovered automatically when running in Azure.

Important

Although this solution provides a simple approach for getting started, it's not a best practice for production-grade environments. In those scenarios, the app shouldn't perform all operations using a single, elevated identity. You should try to implement the principle of least privilege by configuring multiple identities with specific permissions for specific tasks.

You can read more about configuring database roles and security on the following resources:

- [Tutorial: Secure a database in Azure SQL Database](#)
- [Authorize database access to SQL Database](#)

Test the deployed application

1. Select the **Browse** button at the top of App Service overview page to launch the root url of your app.
2. Append the `/swagger/index.html` path to the URL to load the same Swagger test page you used locally.
3. Execute test GET and POST requests to verify that the endpoints work as expected.

Tip

If you receive a 500 Internal Server error while testing, it may be due to your database networking configurations. Verify that your logical server is configured with the settings outlined in the [Configure the database](#) section.

Congratulations! Your application is now connected to Azure SQL Database in both local and hosted environments.

Clean up the resources

When you are finished working with the Azure SQL Database, delete the resource to avoid unintended costs.

Azure portal

1. In the Azure portal search bar, search for *Azure SQL* and select the matching result.
2. Locate and select your database in the list of databases.
3. On the **Overview** page of your Azure SQL Database, select **Delete**.
4. On the **Azure you sure you want to delete...** page that opens, type the name of your database to confirm, and then select **Delete**.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Quickstart: Azure Cosmos DB for NoSQL library for .NET

Article • 01/08/2024

APPLIES TO:  NoSQL

Get started with the Azure Cosmos DB for NoSQL client library for .NET to query data in your containers and perform common operations on individual items. Follow these steps to deploy a minimal solution to your environment using the Azure Developer CLI.

[API reference documentation](#) | [Library source code](#) | [Package \(NuGet\)](#) | [Azure Developer CLI](#)

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [GitHub account](#)

Setting up

Deploy this project's development container to your environment. Then, use the Azure Developer CLI (`azd`) to create an Azure Cosmos DB for NoSQL account and deploy a containerized sample application. The sample application uses the client library to manage, create, read, and query sample data.

 GITHUB CODESPACES OPEN 

Important

GitHub accounts include an entitlement of storage and core hours at no cost. For more information, see [included storage and core hours for GitHub accounts](#).

1. Open a terminal in the root directory of the project.
2. Authenticate to the Azure Developer CLI using `azd auth login`. Follow the steps specified by the tool to authenticate to the CLI using your preferred Azure credentials.

Azure CLI

```
azd auth login
```

3. Use `azd init` to initialize the project.

```
Azure CLI
```

```
azd init
```

4. During initialization, configure a unique environment name.

 **Tip**

The environment name will also be used as the target resource group name. For this quickstart, consider using `msdocs-cosmos-db-nosql`.

5. Deploy the Azure Cosmos DB for NoSQL account using `azd up`. The Bicep templates also deploy a sample web application.

```
Azure CLI
```

```
azd up
```

6. During the provisioning process, select your subscription and desired location. Wait for the provisioning process to complete. The process can take **approximately five minutes**.
7. Once the provisioning of your Azure resources is done, a URL to the running web application is included in the output.

```
Output
```

```
Deploying services (azd deploy)
```

```
(✓) Done: Deploying service web
- Endpoint: <https://[container-app-sub-domain].azurecontainerapps.io>
```

```
SUCCESS: Your application was provisioned and deployed to Azure in 5
minutes 0 seconds.
```

8. Use the URL in the console to navigate to your web application in the browser. Observe the output of the running app.

Azure Cosmos DB for NoSQL | Go Quickstart

```
Current Status: Starting...
Get database: cosmicworks
Get container: products
Upserter item: {70b63682-b93a-4c77-aad2-65501347265f gear-surf-surfboards Yamba Surfboard 12 850 false}
Status code: 201
Request charge: 7.24
Upserter item: {25a68543-b90c-439d-8332-7ef41e06a0e0 gear-surf-surfboards Kiama Classic Surfboard 25 790 true}
Status code: 201
Request charge: 7.24
Read item id: 70b63682-b93a-4c77-aad2-65501347265f
Read item: {70b63682-b93a-4c77-aad2-65501347265f gear-surf-surfboards Yamba Surfboard 12 850 false}
Status code: 200
Request charge: 1.00
```

Install the client library

The client library is available through NuGet, as the `Microsoft.Azure.Cosmos` package.

1. Open a terminal and navigate to the `/src/web` folder.

```
Bash
```

```
cd ./src/web
```

2. If not already installed, install the `Microsoft.Azure.Cosmos` package using `dotnet add package`.

```
Bash
```

```
dotnet add package Microsoft.Azure.Cosmos
```

3. Also, install the `Azure.Identity` package if not already installed.

```
Bash
```

```
dotnet add package Azure.Identity
```

4. Open and review the `src/web/Cosmos.Samples.NoSQL.Quickstart.Web.csproj` file to validate that the `Microsoft.Azure.Cosmos` and `Azure.Identity` entries both exist.

Object model

[] Expand table

Name	Description
CosmosClient	This class is the primary client class and is used to manage account-wide

Name	Description
	metadata or databases.
Database	This class represents a database within the account.
Container	This class is primarily used to perform read, update, and delete operations on either the container or the items stored within the container.
PartitionKey	This class represents a logical partition key. This class is required for many common operations and queries.

Code examples

- [Authenticate the client](#)
- [Get a database](#)
- [Get a container](#)
- [Create an item](#)
- [Get an item](#)
- [Query items](#)

The sample code in the template uses a database named `cosmicworks` and container named `products`. The `products` container contains details such as name, category, quantity, a unique identifier, and a sale flag for each product. The container uses the `/category` property as a logical partition key.

Authenticate the client

Application requests to most Azure services must be authorized. Use the `DefaultAzureCredential` type as the preferred way to implement a passwordless connection between your applications and Azure Cosmos DB for NoSQL. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime.

ⓘ Important

You can also authorize requests to Azure services using passwords, connection strings, or other credentials directly. However, this approach should be used with caution. Developers must be diligent to never expose these secrets in an unsecure location. Anyone who gains access to the password or secret key is able to authenticate to the database service. `DefaultAzureCredential` offers improved

management and security benefits over the account key to allow passwordless authentication without the risk of storing keys.

This sample creates a new instance of the `CosmosClient` class and authenticates using a `DefaultAzureCredential` instance.

C#

```
CosmosClient client = new(
    accountEndpoint:
builder.Configuration["AZURE_COSMOS_DB_NOSQL_ENDPOINT"]!,
    tokenCredential: new DefaultAzureCredential()
);
```

Get a database

Use `client.GetDatabase` to retrieve the existing database named `cosmicworks`.

C#

```
Database database = client.GetDatabase("cosmicworks");
```

Get a container

Retrieve the existing `products` container using `database.GetContainer`.

C#

```
Container container = database.GetContainer("products");
```

Create an item

Build a C# record type with all of the members you want to serialize into JSON. In this example, the type has a unique identifier, and fields for category, name, quantity, price, and sale.

C#

```
public record Product(
    string id,
    string category,
    string name,
```

```
    int quantity,  
    decimal price,  
    bool clearance  
);
```

Create an item in the container using `container.UpsertItem`. This method "upserts" the item effectively replacing the item if it already exists.

C#

```
Product item = new(  
    id: "68719518391",  
    category: "gear-surf-surfboards",  
    name: "Yamba Surfboard",  
    quantity: 12,  
    price: 850.00m,  
    clearance: false  
);  
  
ItemResponse<Product> response = await container.UpsertItemAsync<Product>(  
    item:  
    partitionKey: new PartitionKey("gear-surf-surfboards")  
);
```

Read an item

Perform a point read operation by using both the unique identifier (`id`) and partition key fields. Use `container.ReadItem` to efficiently retrieve the specific item.

C#

```
ItemResponse<Product> response = await container.ReadItemAsync<Product>(  
    id: "68719518391",  
    partitionKey: new PartitionKey("gear-surf-surfboards")  
);
```

Query items

Perform a query over multiple items in a container using `container.GetItemQueryIterator`. Find all items within a specified category using this parameterized query:

nosql

```
SELECT * FROM products p WHERE p.category = @category
```

C#

```
var query = new QueryDefinition(
    query: "SELECT * FROM products p WHERE p.category = @category"
)
    .WithParameter("@category", "gear-surf-surfboards");

using FeedIterator<Product> feed = container.GetItemQueryIterator<Product>(
    queryDefinition: query
);
```

Parse the paginated results of the query by looping through each page of results using `feed.ReadNextAsync`. Use `feed.HasMoreResults` to determine if there are any results left at the start of each loop.

C#

```
List<Product> items = new();
double requestCharge = 0d;
while (feed.HasMoreResults)
{
    FeedResponse<Product> response = await feed.ReadNextAsync();
    foreach (Product item in response)
    {
        items.Add(item);
    }
    requestCharge += response.RequestCharge;
}
```

Clean up resources

When you no longer need the sample application or resources, remove the corresponding deployment and all resources.

Azure CLI

```
azd down
```

In GitHub Codespaces, delete the running codespace to maximize your storage and core entitlements.

Related content

- [JavaScript/Node.js Quickstart](#)
- [Java Quickstart](#)

- [Python Quickstart](#)
- [Go Quickstart](#)

Next step

[Tutorial: Develop a .NET console application](#)

Quickstart: Send events to and receive events from Azure Event Hubs using .NET

Article • 04/05/2024

In this quickstart, you learn how to send events to an event hub and then receive those events from the event hub using the `Azure.Messaging.EventHubs` .NET library.

ⓘ Note

Quickstarts are for you to quickly ramp up on the service. If you are already familiar with the service, you might want to see .NET samples for Event Hubs in our .NET SDK repository on GitHub: [Event Hubs samples on GitHub](#), [Event processor samples on GitHub](#).

Prerequisites

If you're new to Azure Event Hubs, see [Event Hubs overview](#) before you go through this quickstart.

To complete this quickstart, you need the following prerequisites:

- **Microsoft Azure subscription.** To use Azure services, including Azure Event Hubs, you need a subscription. If you don't have an existing Azure account, you can sign up for a [free trial](#) or use your MSDN subscriber benefits when you [create an account](#).
- **Microsoft Visual Studio 2022.** The Azure Event Hubs client library makes use of new features that were introduced in C# 8.0. You can still use the library with previous C# language versions, but the new syntax isn't available. To make use of the full syntax, we recommend that you compile with the [.NET Core SDK](#) 3.0 or higher and [language version](#) set to `latest`. If you're using Visual Studio, versions before Visual Studio 2022 aren't compatible with the tools needed to build C# 8.0 projects. Visual Studio 2022, including the free Community edition, can be downloaded [here](#).
- **Create an Event Hubs namespace and an event hub.** The first step is to use the Azure portal to create an Event Hubs namespace and an event hub in the namespace. Then, obtain the management credentials that your application needs

to communicate with the event hub. To create a namespace and an event hub, see [Quickstart: Create an event hub using Azure portal](#).

Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Event Hubs:

- Passwordless (Microsoft Entra authentication)
- Connection string

The first option shows you how to use your security principal in Azure **Active Directory** and **role-based access control (RBAC)** to connect to an Event Hubs namespace. You don't need to worry about having hard-coded connection strings in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a **connection string** to connect to an Event Hubs namespace. If you're new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless

Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Event Hubs has the correct permissions. You'll need the [Azure Event Hubs Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Event Hubs Data Owner` role to your user account, which provides full access to Azure Event Hubs resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

Azure built-in roles for Azure Event Hubs

For Azure Event Hubs, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to an Event Hubs namespace:

- [Azure Event Hubs Data Owner](#): Enables data access to Event Hubs namespace and its entities (queues, topics, subscriptions, and filters)
- [Azure Event Hubs Data Sender](#): Use this role to give the sender access to Event Hubs namespace and its entities.
- [Azure Event Hubs Data Receiver](#): Use this role to give the receiver access to Event Hubs namespace and its entities.

If you want to create a custom role, see [Rights required for Event Hubs operations](#).

 **Important**

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

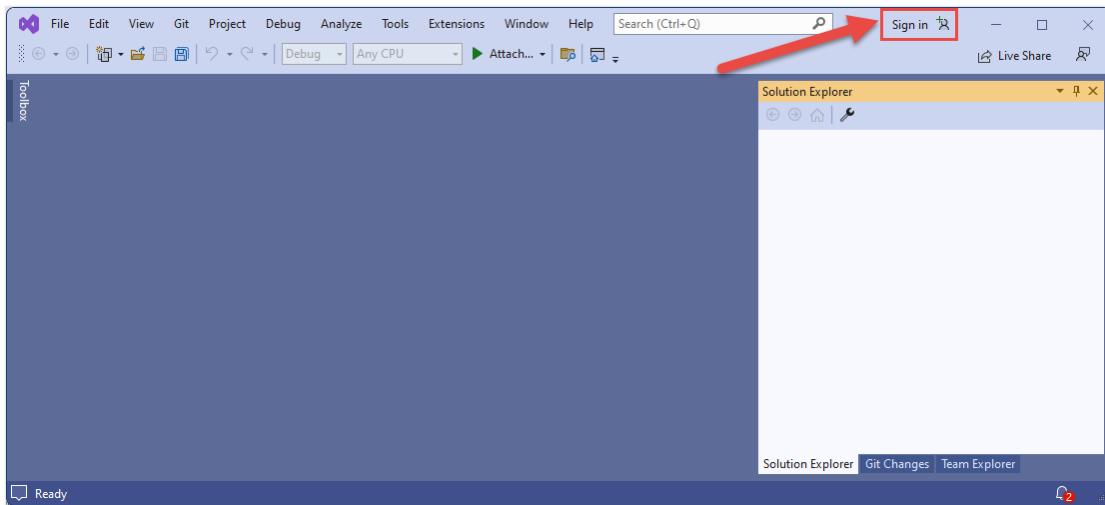
1. In the Azure portal, locate your Event Hubs namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Event Hubs Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

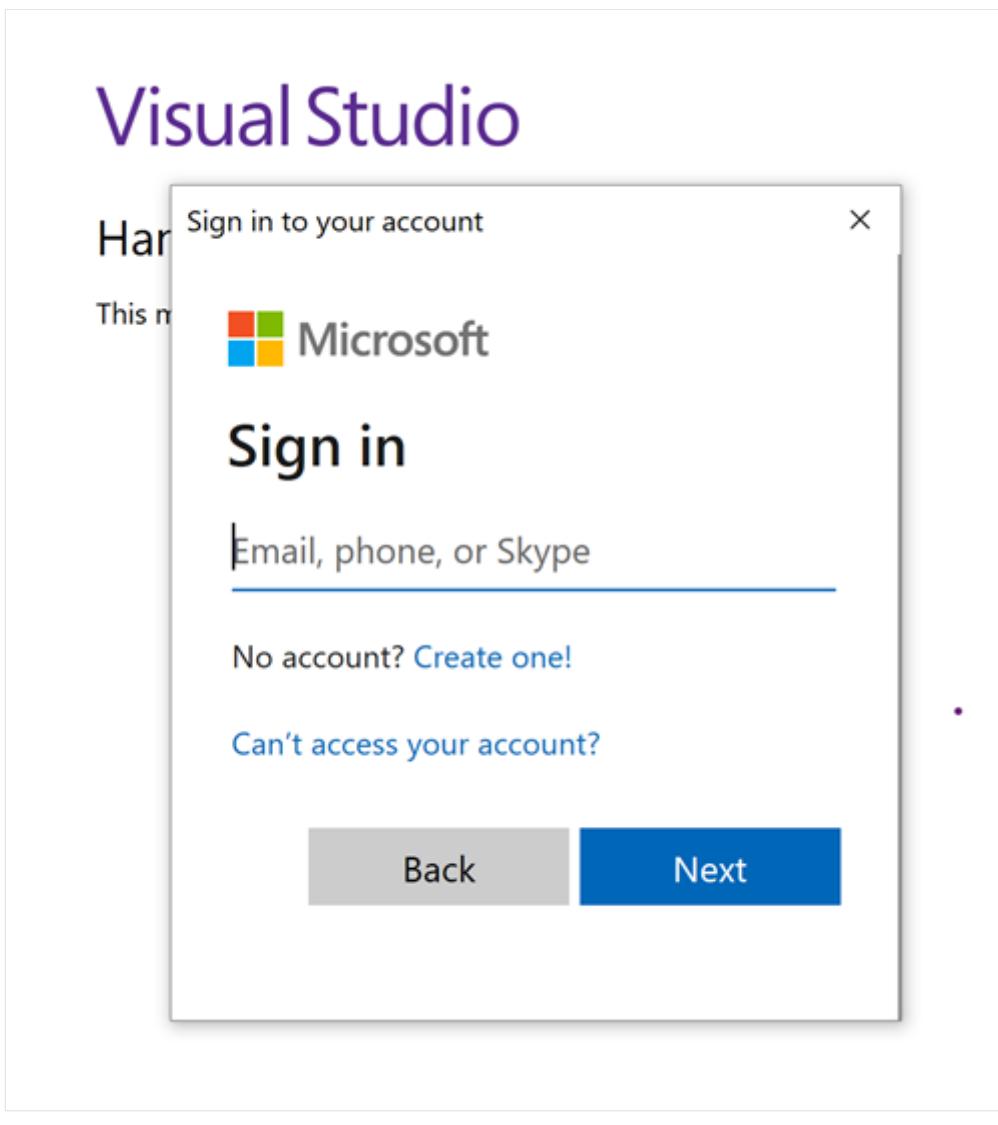
Launch Visual Studio and sign-in to Azure

You can authorize access to the service bus namespace using the following steps:

1. Launch Visual Studio. If you see the **Get started** window, select the **Continue without code** link in the right pane.
2. Select the **Sign in** button in the top right of Visual Studio.



3. Sign-in using the Microsoft Entra account you assigned a role to previously.

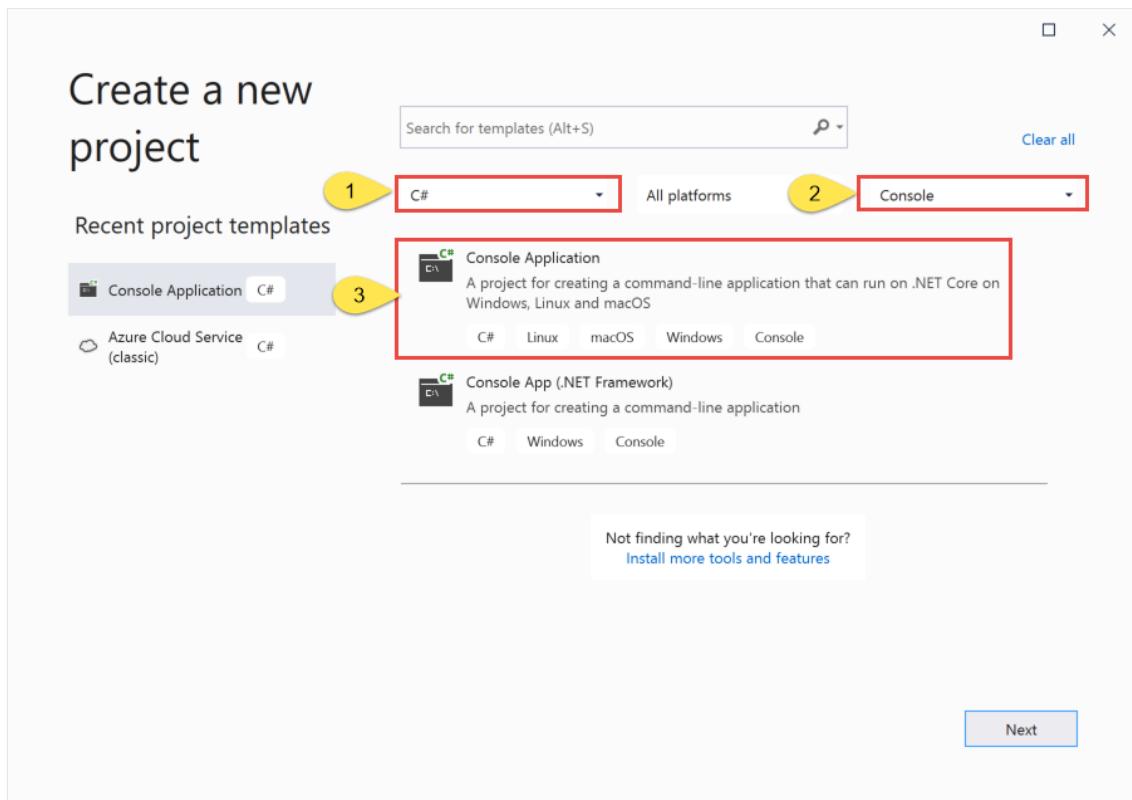


Send events to the event hub

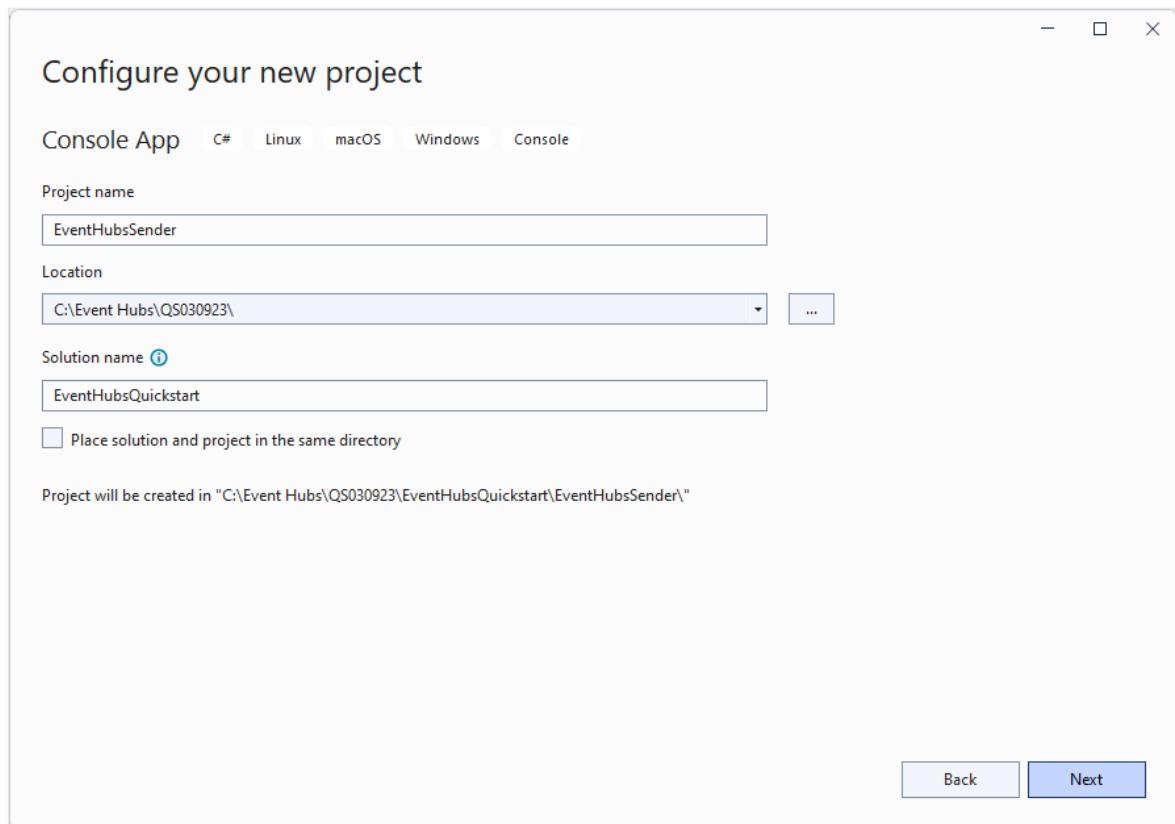
This section shows you how to create a .NET Core console application to send events to the event hub you created.

Create a console application

1. If you have Visual Studio 2022 open already, select **File** on the menu, select **New**, and then select **Project**. Otherwise, launch Visual Studio 2022 and select **Create a new project** if you see a popup window.
2. On the **Create a new project** dialog box, do the following steps: If you don't see this dialog box, select **File** on the menu, select **New**, and then select **Project**.
 - a. Select **C#** for the programming language.
 - b. Select **Console** for the type of the application.
 - c. Select **Console Application** from the results list.
 - d. Then, select **Next**.



3. Enter **EventHubsSender** for the project name, **EventHubsQuickStart** for the solution name, and then select **Next**.



4. On the Additional information page, select **Create**.

Add the NuGet packages to the project

Passwordless (Recommended)

1. Select **Tools > NuGet Package Manager > Package Manager Console** from the menu.
2. Run the following commands to install **Azure.Messaging.EventHubs** and **Azure.Identity** NuGet packages. Press **ENTER** to run the second command.

PowerShell

```
Install-Package Azure.Messaging.EventHubs
Install-Package Azure.Identity
```

Write code to send events to the event hub

Passwordless (Recommended)

1. Replace the existing code in the `Program.cs` file with the following sample code. Then, replace `<EVENT_HUB_NAMESPACE>` and `<HUB_NAME>` placeholder values for the `EventHubProducerClient` parameters with the names of your Event Hubs namespace and the event hub. For example:
`"spehubns0309.servicebus.windows.net"` and `"spehub"`.

Here are the important steps from the code:

- a. Creates an `EventHubProducerClient` object using the namespace and the event hub name.
- b. Invokes the `CreateBatchAsync` method on the `EventHubProducerClient` object to create an `EventDataBatch` object.
- c. Add events to the batch using the `EventDataBatch.TryAdd` method.
- d. Sends the batch of messages to the event hub using the `EventHubProducerClient.SendAsync` method.

C#

```
using Azure.Identity;
using Azure.Messaging.EventHubs;
using Azure.Messaging.EventHubs.Producer;
using System.Text;

// number of events to be sent to the event hub
int numOfEvents = 3;

// The Event Hubs client types are safe to cache and use as a
// singleton for the lifetime
// of the application, which is best practice when events are being
// published or read regularly.
// TODO: Replace the <EVENT_HUB_NAMESPACE> and <HUB_NAME>
// placeholder values
EventHubProducerClient producerClient = new EventHubProducerClient(
    "<EVENT_HUB_NAMESPACE>.servicebus.windows.net",
    "<HUB_NAME>",
    new DefaultAzureCredential());

// Create a batch of events
using EventDataBatch eventBatch = await
producerClient.CreateBatchAsync();

for (int i = 1; i <= numOfEvents; i++)
{
    if (!eventBatch.TryAdd(new
EventData(Encoding.UTF8.GetBytes($"Event {i}"))))
    {
        // if it is too large for the batch
        throw new Exception($"Event {i} is too large for the batch
and cannot be sent.");
    }
}
```

```
        }

    try
    {
        // Use the producer client to send the batch of events to the
        event hub
        await producerClient.SendAsync(eventBatch);
        Console.WriteLine($"A batch of {numOfEvents} events has been
published.");
        Console.ReadLine();
    }
    finally
    {
        await producerClient.DisposeAsync();
    }
}
```

2. Build the project, and ensure that there are no errors.

3. Run the program and wait for the confirmation message.

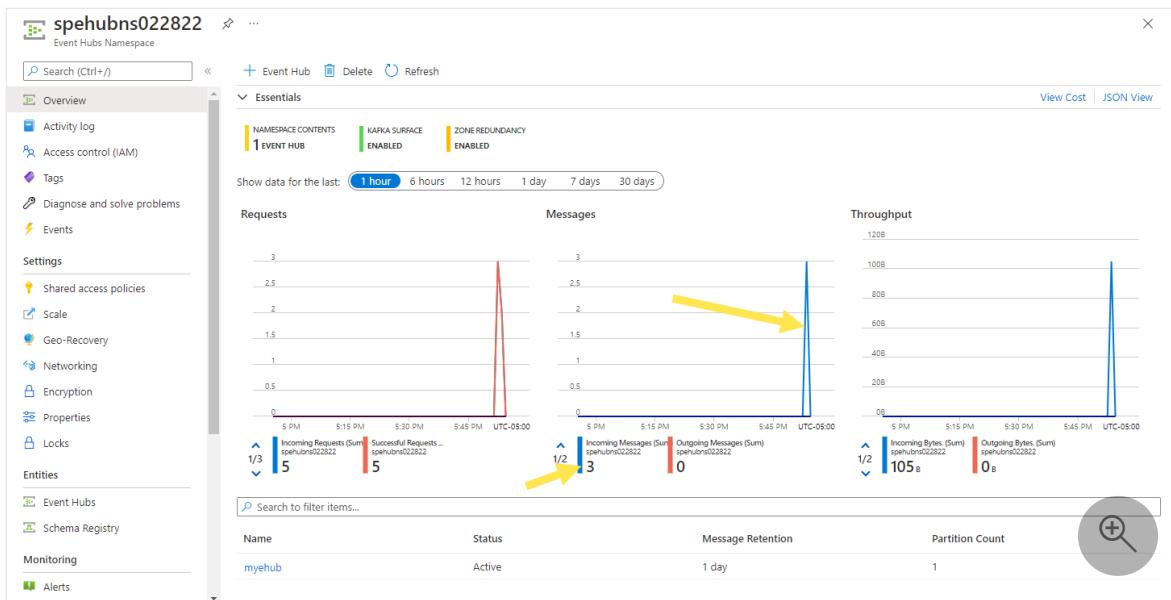
C#

A batch of 3 events has been published.

Important

If you are using the Passwordless (Azure Active Directory's Role-based Access Control) authentication, select **Tools**, then select **Options**. In the **Options** window, expand **Azure Service Authentication**, and select **Account Selection**. Confirm that you are using the account that was added to the **Azure Event Hubs Data Owner** role on the Event Hubs namespace.

4. On the **Event Hubs Namespace** page in the Azure portal, you see three incoming messages in the **Messages** chart. Refresh the page to update the chart if needed. It might take a few seconds for it to show that the messages have been received.



ⓘ Note

For the complete source code with more informational comments, see [this file on the GitHub](#)

Receive events from the event hub

This section shows how to write a .NET Core console application that receives events from an event hub using an event processor. The event processor simplifies receiving events from event hubs.

Create an Azure Storage Account and a blob container

In this quickstart, you use Azure Storage as the checkpoint store. Follow these steps to create an Azure Storage account.

1. [Create an Azure Storage account](#)
2. [Create a blob container](#)
3. Authenticate to the blob container using either Microsoft Entra ID (passwordless) authentication or a connection string to the namespace.

Follow these recommendations when using Azure Blob Storage as a checkpoint store:

- Use a separate container for each consumer group. You can use the same storage account, but use one container per each group.
- Don't use the container for anything else, and don't use the storage account for anything else.

- Storage account should be in the same region as the deployed application is located in. If the application is on-premises, try to choose the closest region possible.

On the **Storage account** page in the Azure portal, in the **Blob service** section, ensure that the following settings are disabled.

- Hierarchical namespace
- Blob soft delete
- Versioning

Passwordless (Recommended)

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

ⓘ Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.

2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'identitymigrationstorage | Access Control (IAM)' page. The left sidebar has a red box around the 'Access Control (IAM)' section. The top navigation bar has a red box around the '+ Add' button. A dropdown menu is open, with 'Add role assignment' highlighted and also has a red box around it. Other options in the dropdown include 'Add co-administrator', 'Assignments', 'Roles', 'Deny assignments', and 'Classic administrators'. The main content area includes sections for 'My access', 'Check access', and 'Grant access to this resource' with a 'Add role assignment' button.

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Create a project for the receiver

1. In the Solution Explorer window, right-click the **EventHubQuickStart** solution, point to **Add**, and select **New Project**.
2. Select **Console application**, and select **Next**.
3. Enter **EventHubsReceiver** for the **Project name**, and select **Create**.

4. In the Solution Explorer window, right-click **EventHubsReceiver**, and select **Set as a Startup Project**.

Add the NuGet packages to the project

Passwordless (Recommended)

1. Select **Tools > NuGet Package Manager > Package Manager Console** from the menu.
2. In the **Package Manager Console** window, confirm that **EventHubsReceiver** is selected for the **Default project**. If not, use the drop-down list to select **EventHubsReceiver**.
3. Run the following command to install the **Azure.Messaging.EventHubs** and the **Azure.Identity** NuGet packages. Press **ENTER** to run the last command.

PowerShell

```
Install-Package Azure.Messaging.EventHubs
Install-Package Azure.Messaging.EventHubs.Processor
Install-Package Azure.Identity
```

Update the code

Replace the contents of **Program.cs** with the following code:

Passwordless (Recommended)

1. Replace the existing code in the **Program.cs** file with the following sample code. Then, replace the `<STORAGE_ACCOUNT_NAME>` and `<BLOB_CONTAINER_NAME>` placeholder values for the `BlobContainerClient` URL. Replace the `<EVENT_HUB_NAMESPACE>` and `<HUB_NAME>` placeholder values for the `EventProcessorClient` as well.

Here are the important steps from the code:

- a. Creates an `EventProcessorClient` object using the Event Hubs namespace and the event hub name. You need to build `BlobContainerClient` object for the container in the Azure storage you created earlier.

- b. Specifies handlers for the `ProcessEventAsync` and `ProcessErrorAsync` events of the `EventProcessorClient` object.
- c. Starts processing events by invoking the `StartProcessingAsync` on the `EventProcessorClient` object.
- d. Stops processing events after 30 seconds by invoking `StopProcessingAsync` on the `EventProcessorClient` object.

C#

```

using Azure.Identity;
using Azure.Messaging.EventHubs;
using Azure.Messaging.EventHubs.Consumer;
using Azure.Messaging.EventHubs.Processor;
using Azure.Storage.Blobs;
using System.Text;

// Create a blob container client that the event processor will use
// TODO: Replace <STORAGE_ACCOUNT_NAME> and <BLOB_CONTATINAER_NAME>
// with actual names
BlobContainerClient storageClient = new BlobContainerClient(
    new
Uri("https://<STORAGE_ACCOUNT_NAME>.blob.core.windows.net/<BLOB_CON
TAINER_NAME>"),
    new DefaultAzureCredential());

// Create an event processor client to process events in the event
hub
// TODO: Replace the <EVENT_HUBS_NAMESPACE> and <HUB_NAME>
placeholder values
var processor = new EventProcessorClient(
    storageClient,
    EventHubConsumerClient.DefaultConsumerGroupName,
    "<EVENT_HUB_NAMESPACE>.servicebus.windows.net",
    "<HUB_NAME>",
    new DefaultAzureCredential());

// Register handlers for processing events and handling errors
processor.ProcessEventAsync += ProcessEventHandler;
processor.ProcessErrorAsync += ProcessErrorHandler;

// Start the processing
await processor.StartProcessingAsync();

// Wait for 30 seconds for the events to be processed
await Task.Delay(TimeSpan.FromSeconds(30));

// Stop the processing
await processor.StopProcessingAsync();

Task ProcessEventHandler(ProcessEventArgs eventArgs)
{
    // Write the body of the event to the console window
}

```

```
        Console.WriteLine("\tReceived event: {0}",  
Encoding.UTF8.GetString(eventArgs.Data.Body.ToArray()));  
        Console.ReadLine();  
        return Task.CompletedTask;  
    }  
  
    Task ProcessErrorHandler(ProcessErrorEventArgs eventArgs)  
{  
    // Write details about the error to the console window  
    Console.WriteLine($"\\tPartition '{eventArgs.PartitionId}': an  
unhandled exception was encountered. This was not expected to  
happen.");  
    Console.WriteLine(eventArgs.Exception.Message);  
    Console.ReadLine();  
    return Task.CompletedTask;  
}
```

2. Build the project, and ensure that there are no errors.

 **Note**

For the complete source code with more informational comments, see [this file on the GitHub ↗](#).

3. Run the receiver application.

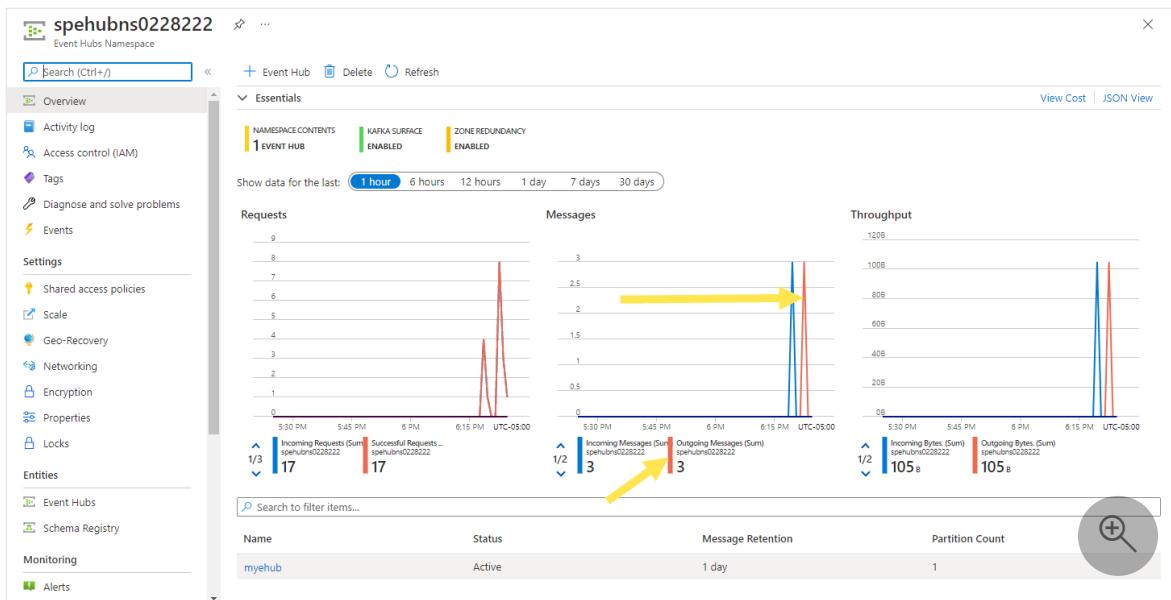
4. You should see a message that the events have been received. Press ENTER after you see a received event message.

Bash

```
Received event: Event 1  
Received event: Event 2  
Received event: Event 3
```

These events are the three events you sent to the event hub earlier by running the sender program.

5. In the Azure portal, you can verify that there are three outgoing messages, which Event Hubs sent to the receiving application. Refresh the page to update the chart. It might take a few seconds for it to show that the messages have been received.



Schema validation for Event Hubs SDK based applications

You can use Azure Schema Registry to perform schema validation when you stream data with your Event Hubs SDK-based applications. Azure Schema Registry of Event Hubs provides a centralized repository for managing schemas and you can seamlessly connect your new or existing applications with Schema Registry.

To learn more, see [Validate schemas with Event Hubs SDK](#).

Samples and reference

This quick start provides step-by-step instructions to implement a scenario of sending a batch of events to an event hub and then receiving them. For more samples, select the following links.

- [Event Hubs samples on GitHub ↗](#)
- [Event processor samples on GitHub ↗](#)
- [Azure role-based access control \(Azure RBAC\) sample ↗](#)

For complete .NET library reference, see our [SDK documentation](#).

Clean up resources

Delete the resource group that has the Event Hubs namespace or delete only the namespace if you want to keep the resource group.

Related content

See the following tutorial:

[Tutorial: Visualize data anomalies in real-time events sent to Azure Event Hubs](#)

Quickstart: Azure Key Vault certificate client library for .NET

Article • 04/07/2024

Get started with the Azure Key Vault certificate client library for .NET. [Azure Key Vault](#) is a cloud service that provides a secure store for certificates. You can securely store keys, passwords, certificates, and other secrets. Azure key vaults may be created and managed through the Azure portal. In this quickstart, you learn how to create, retrieve, and delete certificates from an Azure key vault using the .NET client library

Key Vault client library resources:

[API reference documentation](#) | [Library source code](#) | [Package \(NuGet\)](#)

For more information about Key Vault and certificates, see:

- [Key Vault Overview](#)
- [Certificates Overview.](#)

Prerequisites

- An Azure subscription - [create one for free](#)
- [.NET 6 SDK or later](#)
- [Azure CLI](#)
- A Key Vault - you can create one using [Azure portal](#), [Azure CLI](#), or [Azure PowerShell](#).

This quickstart is using `dotnet` and Azure CLI

Setup

This quickstart is using Azure Identity library with Azure CLI to authenticate user to Azure Services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls, for more information, see [Authenticate the client with Azure Identity client library](#).

Sign in to Azure

1. Run the `login` command.

Azure CLI

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

Create new .NET console app

1. In a command shell, run the following command to create a project named `key-vault-console-app`:

.NET CLI

```
dotnet new console --name key-vault-console-app
```

2. Change to the newly created `key-vault-console-app` directory, and run the following command to build the project:

```
.NET CLI
```

```
dotnet build
```

The build output should contain no warnings or errors.

```
Console
```

```
Build succeeded.  
0 Warning(s)  
0 Error(s)
```

Install the packages

From the command shell, install the Azure Key Vault certificate client library for .NET:

```
.NET CLI
```

```
dotnet add package Azure.Security.KeyVault.Certificates
```

For this quickstart, you'll also need to install the Azure Identity client library:

```
.NET CLI
```

```
dotnet add package Azure.Identity
```

Set environment variables

This application is using key vault name as an environment variable called

`KEY_VAULT_NAME`.

Windows

```
Windows Command Prompt
```

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

Windows PowerShell

```
PowerShell
```

```
$Env:KEY_VAULT_NAME="<your-key-vault-name>"
```

macOS or Linux

Bash

```
export KEY_VAULT_NAME=<your-key-vault-name>
```

Object model

The Azure Key Vault certificate client library for .NET allows you to manage certificates.

The [Code examples](#) section shows how to create a client, set a certificate, retrieve a certificate, and delete a certificate.

Code examples

Add directives

Add the following directives to the top of *Program.cs*:

C#

```
using System;
using Azure.Identity;
using Azure.Security.KeyVault.Certificates;
```

Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) class provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this example, the name of your key vault is expanded to the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

C#

```
string keyVaultName = Environment.GetEnvironmentVariable("KEY_VAULT_NAME");
var kvUri = "https://" + keyVaultName + ".vault.azure.net";

var client = new CertificateClient(new Uri(kvUri), new
DefaultAzureCredential());
```

Save a certificate

In this example, for simplicity you can use self-signed certificate with default issuance policy. For this task, use the [StartCreateCertificateAsync](#) method. The method's parameters accepts a certificate name and the [certificate policy](#).

C#

```
var operation = await client.StartCreateCertificateAsync("myCertificate",
CertificatePolicy.Default);
var certificate = await operation.WaitForCompletionAsync();
```

ⓘ Note

If certificate name exists, above code will create new version of that certificate.

Retrieve a certificate

You can now retrieve the previously created certificate with the [GetCertificateAsync](#) method.

C#

```
var certificate = await client.GetCertificateAsync("myCertificate");
```

Delete a certificate

Finally, let's delete and purge the certificate from your key vault with the [StartDeleteCertificateAsync](#) and [PurgeDeletedCertificateAsync](#) methods.

C#

```
var operation = await client.StartDeleteCertificateAsync("myCertificate");

// You only need to wait for completion if you want to purge or recover the
// certificate.
await operation.WaitForCompletionAsync();

var certificate = operation.Value;
await client.PurgeDeletedCertificateAsync("myCertificate");
```

Sample code

Modify the .NET console app to interact with the Key Vault by completing the following steps:

- Replace the code in *Program.cs* with the following code:

C#

```
using System;
using System.Threading.Tasks;
using Azure.Identity;
using Azure.Security.KeyVault.Certificates;

namespace key_vault_console_app
{
    class Program
    {
        static async Task Main(string[] args)
        {
            const string certificateName = "myCertificate";
            var keyVaultName =
Environment.GetEnvironmentVariable("KEY_VAULT_NAME");
            var kvUri = $"https://{{keyVaultName}}.vault.azure.net";

            var client = new CertificateClient(new Uri(kvUri), new
DefaultAzureCredential());

            Console.WriteLine($"Creating a certificate in {{keyVaultName}}
called '{certificateName}' ...");
            CertificateOperation operation = await
client.StartCreateCertificateAsync(certificateName,
CertificatePolicy.Default);
            await operation.WaitForCompletionAsync();
            Console.WriteLine(" done.");

            Console.WriteLine($"Retrieving your certificate from
{{keyVaultName}}.");
            var certificate = await
```

```
client.GetCertificateAsync(certificateName);
    Console.WriteLine($"Your certificate version is
'{certificate.Value.Properties.Version}'.");

    Console.Write($"Deleting your certificate from
{keyVaultName} ...");
    DeleteCertificateOperation deleteOperation = await
client.StartDeleteCertificateAsync(certificateName);
    // You only need to wait for completion if you want to
purge or recover the certificate.
    await deleteOperation.WaitForCompletionAsync();
Console.WriteLine(" done.");

    Console.Write($"Purging your certificate from
{keyVaultName} ...");
    await client.PurgeDeletedCertificateAsync(certificateName);
Console.WriteLine(" done.");
}
}
```

Test and verify

Execute the following command to build the project

.NET CLI

```
dotnet build
```

A variation of the following output appears:

Console

```
Creating a certificate in mykeyvault called 'myCertificate' ... done.
Retrieving your certificate from mykeyvault.
Your certificate version is '8532359bcfed24e4bb2525f2d2050738a'.
Deleting your certificate from mykeyvault ... done
Purging your certificate from mykeyvault ... done
```

Next steps

In this quickstart, you created a key vault, stored a certificate, and retrieved that certificate.

To learn more about Key Vault and how to integrate it with your apps, see the following articles:

- Read an [Overview of Azure Key Vault](#)
- Read an [Overview of certificates](#)
- See an [Access Key Vault from App Service Application Tutorial](#)
- See an [Access Key Vault from Virtual Machine Tutorial](#)
- See the [Azure Key Vault developer's guide](#)
- Review the [Key Vault security overview](#)

Quickstart: Azure Key Vault key client library for .NET

Article • 04/07/2024

Get started with the Azure Key Vault key client library for .NET. [Azure Key Vault](#) is a cloud service that provides a secure store for cryptographic keys. You can securely store cryptographic keys, passwords, certificates, and other secrets. Azure key vaults may be created and managed through the Azure portal. In this quickstart, you learn how to create, retrieve, and delete keys from an Azure key vault using the .NET key client library

Key Vault key client library resources:

[API reference documentation](#) | [Library source code](#) | [Package \(NuGet\)](#)

For more information about Key Vault and keys, see:

- [Key Vault Overview](#)
- [Keys Overview.](#)

Prerequisites

- An Azure subscription - [create one for free](#)
- [.NET 6 SDK or later](#)
- [Azure CLI](#)
- A Key Vault - you can create one using [Azure portal](#), [Azure CLI](#), or [Azure PowerShell](#).

This quickstart is using `dotnet` and Azure CLI

Setup

This quickstart is using Azure Identity library with Azure CLI to authenticate user to Azure Services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls, for more information, see [Authenticate the client with Azure Identity client library](#).

Sign in to Azure

1. Run the `login` command.

Azure CLI

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

Create new .NET console app

1. In a command shell, run the following command to create a project named `key-vault-console-app`:

.NET CLI

```
dotnet new console --name key-vault-console-app
```

2. Change to the newly created `key-vault-console-app` directory, and run the following command to build the project:

```
.NET CLI
```

```
dotnet build
```

The build output should contain no warnings or errors.

```
Console
```

```
Build succeeded.  
0 Warning(s)  
0 Error(s)
```

Install the packages

From the command shell, install the Azure Key Vault key client library for .NET:

```
.NET CLI
```

```
dotnet add package Azure.Security.KeyVault.Keys
```

For this quickstart, you'll also need to install the Azure Identity client library:

```
.NET CLI
```

```
dotnet add package Azure.Identity
```

Set environment variables

This application is using key vault name as an environment variable called `KEY_VAULT_NAME`.

Windows

```
Windows Command Prompt
```

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

Windows PowerShell

```
PowerShell
```

```
$Env:KEY_VAULT_NAME="<your-key-vault-name>"
```

macOS or Linux

```
Bash
```

```
export KEY_VAULT_NAME=<your-key-vault-name>
```

Object model

The Azure Key Vault key client library for .NET allows you to manage keys. The [Code examples](#) section shows how to create a client, set a key, retrieve a key, and delete a key.

Code examples

Add directives

Add the following directives to the top of *Program.cs*:

```
C#
```

```
using System;
using Azure.Identity;
using Azure.Security.KeyVault.Keys;
```

Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) class provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this example, the name of your key vault is expanded to the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

C#

```
var keyVaultName = Environment.GetEnvironmentVariable("KEY_VAULT_NAME");
var kvUri = $"https://{{keyVaultName}}.vault.azure.net";

var client = new KeyClient(new Uri(kvUri), new DefaultAzureCredential());
```

Save a key

For this task, use the [CreateKeyAsync](#) method. The method's parameters accepts a key name and the [key type](#).

C#

```
var key = await client.CreateKeyAsync("myKey", KeyType.Rsa);
```

ⓘ Note

If key name exists, this code will create new version of that key.

Retrieve a key

You can now retrieve the previously created key with the [GetKeyAsync](#) method.

C#

```
var key = await client.GetKeyAsync("myKey");
```

Delete a key

Finally, let's delete and purge the key from your key vault with the [StartDeleteKeyAsync](#) and [PurgeDeletedKeyAsync](#) methods.

C#

```
var operation = await client.StartDeleteKeyAsync("myKey");

// You only need to wait for completion if you want to purge or recover the
```

```
key.  
await operation.WaitForCompletionAsync();  
  
var key = operation.Value;  
await client.PurgeDeletedKeyAsync("myKey");
```

Sample code

Modify the .NET console app to interact with the Key Vault by completing the following steps:

- Replace the code in *Program.cs* with the following code:

```
C#  
  
using System;  
using System.Threading.Tasks;  
using Azure.Identity;  
using Azure.Security.KeyVault.Keys;  
  
namespace key_vault_console_app  
{  
    class Program  
    {  
        static async Task Main(string[] args)  
        {  
            const string keyName = "myKey";  
            var keyVaultName =  
Environment.GetEnvironmentVariable("KEY_VAULT_NAME");  
            var kvUri = $"https://{{keyVaultName}}.vault.azure.net";  
  
            var client = new KeyClient(new Uri(kvUri), new  
DefaultAzureCredential());  
  
            Console.WriteLine($"Creating a key in {{keyVaultName}} called  
'{{keyName}}' ...");  
            var createdKey = await client.CreateKeyAsync(keyName,  
KeyType.Rsa);  
            Console.WriteLine("done.");  
  
            Console.WriteLine($"Retrieving your key from  
{{keyVaultName}}.");  
            var key = await client.GetKeyAsync(keyName);  
            Console.WriteLine($"Your key version is  
'{{key.Value.Properties.Version}}'.");  
  
            Console.WriteLine($"Deleting your key from {{keyVaultName}}  
...");  
            var deleteOperation = await  
client.StartDeleteKeyAsync(keyName);  
            // You only need to wait for completion if you want to
```

```
purge or recover the key.  
        await deleteOperation.WaitForCompletionAsync();  
        Console.WriteLine("done.");  
  
        Console.Write($"Purging your key from {keyVaultName} ...");  
        await client.PurgeDeletedKeyAsync(keyName);  
        Console.WriteLine(" done.");  
    }  
}  
}
```

Test and verify

1. Execute the following command to build the project

```
.NET CLI
```

```
dotnet build
```

2. Execute the following command to run the app.

```
.NET CLI
```

```
dotnet run
```

3. When prompted, enter a secret value. For example, mySecretPassword.

A variation of the following output appears:

```
Console
```

```
Creating a key in mykeyvault called 'myKey' ... done.  
Retrieving your key from mykeyvault.  
Your key version is '8532359bc...'.  
Deleting your key from jl-kv ... done  
Purging your key from <your-unique-keyvault-name> ... done.
```

Next steps

In this quickstart, you created a key vault, stored a key, and retrieved that key.

To learn more about Key Vault and how to integrate it with your apps, see the following articles:

- Read an [Overview of Azure Key Vault](#)

- Read an [Overview of keys](#)
- See an [Access Key Vault from App Service Application Tutorial](#)
- See an [Access Key Vault from Virtual Machine Tutorial](#)
- See the [Azure Key Vault developer's guide](#)
- Review the [Key Vault security overview](#)

Quickstart: Azure Key Vault secret client library for .NET

Article • 04/07/2024

Get started with the Azure Key Vault secret client library for .NET. [Azure Key Vault](#) is a cloud service that provides a secure store for secrets. You can securely store keys, passwords, certificates, and other secrets. Azure key vaults may be created and managed through the Azure portal. In this quickstart, you learn how to create, retrieve, and delete secrets from an Azure key vault using the .NET client library

Key Vault client library resources:

[API reference documentation](#) | [Library source code](#) | [Package \(NuGet\)](#)

For more information about Key Vault and secrets, see:

- [Key Vault Overview](#)
- [Secrets Overview](#).

Prerequisites

- An Azure subscription - [create one for free](#)
- [.NET 6 SDK or later](#)
- [Azure CLI](#) or [Azure PowerShell](#)
- A Key Vault - you can create one using [Azure portal](#), [Azure CLI](#), or [Azure PowerShell](#)

This quickstart is using `dotnet` and Azure CLI or Azure PowerShell.

Setup

Azure CLI

This quickstart is using Azure Identity library with Azure CLI to authenticate user to Azure Services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls, for more information, see [Authenticate the client with Azure Identity client library](#).

Sign in to Azure

1. Run the `az login` command.

Azure CLI

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

Create new .NET console app

1. In a command shell, run the following command to create a project named `key-vault-console-app`:

.NET CLI

```
dotnet new console --name key-vault-console-app
```

2. Change to the newly created *key-vault-console-app* directory, and run the following command to build the project:

```
.NET CLI
```

```
dotnet build
```

The build output should contain no warnings or errors.

```
Console
```

```
Build succeeded.  
0 Warning(s)  
0 Error(s)
```

Install the packages

From the command shell, install the Azure Key Vault secret client library for .NET:

```
.NET CLI
```

```
dotnet add package Azure.Security.KeyVault.Secrets
```

For this quickstart, you'll also need to install the Azure Identity client library:

```
.NET CLI
```

```
dotnet add package Azure.Identity
```

Set environment variables

This application is using key vault name as an environment variable called `KEY_VAULT_NAME`.

Windows

```
Windows Command Prompt
```

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

Windows PowerShell

```
PowerShell
```

```
$Env:KEY_VAULT_NAME=<your-key-vault-name>"
```

macOS or Linux

Bash

```
export KEY_VAULT_NAME=<your-key-vault-name>
```

Object model

The Azure Key Vault secret client library for .NET allows you to manage secrets. The [Code examples](#) section shows how to create a client, set a secret, retrieve a secret, and delete a secret.

Code examples

Add directives

Add the following directives to the top of *Program.cs*:

C#

```
using System;
using Azure.Identity;
using Azure.Security.KeyVault.Secrets;
```

Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) class provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can

automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this example, the name of your key vault is expanded to the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

```
C#
```

```
string keyVaultName = Environment.GetEnvironmentVariable("KEY_VAULT_NAME");
var kvUri = "https://" + keyVaultName + ".vault.azure.net";

var client = new SecretClient(new Uri(kvUri), new DefaultAzureCredential());
```

Save a secret

Now that the console app is authenticated, add a secret to the key vault. For this task, use the [SetSecretAsync](#) method.

The method's first parameter accepts a name for the secret. In this sample, the variable `secretName` stores the string "mySecret".

The method's second parameter accepts a value for the secret. In this sample, the secret is input by the user via the commandline and stored in the variable `secretValue`.

```
C#
```

```
await client.SetSecretAsync(secretName, secretValue);
```

ⓘ Note

If secret name exists, the code will create new version of that secret.

Retrieve a secret

You can now retrieve the previously set value with the [GetSecretAsync](#) method.

```
C#
```

```
var secret = await client.GetSecretAsync(secretName);
```

Your secret is now saved as `secret.Value`.

Delete a secret

Finally, let's delete the secret from your key vault with the [StartDeleteSecretAsync](#) and [PurgeDeletedSecretAsync](#) methods.

C#

```
var operation = await client.StartDeleteSecretAsync(secretName);
// You only need to wait for completion if you want to purge or recover the
key.
await operation.WaitForCompletionAsync();

await client.PurgeDeletedSecretAsync(secretName);
```

Sample code

Modify the .NET console app to interact with the Key Vault by completing the following steps:

1. Replace the code in *Program.cs* with the following code:

C#

```
using System;
using System.Threading.Tasks;
using Azure.Identity;
using Azure.Security.KeyVault.Secrets;

namespace key_vault_console_app
{
    class Program
    {
        static async Task Main(string[] args)
        {
            const string secretName = "mySecret";
            var keyVaultName =
Environment.GetEnvironmentVariable("KEY_VAULT_NAME");
            var kvUri = $"https://'{keyVaultName}'.vault.azure.net";

            var client = new SecretClient(new Uri(kvUri), new
DefaultAzureCredential());

            Console.Write("Input the value of your secret > ");
            var secretValue = Console.ReadLine();

            Console.WriteLine($"Creating a secret in {keyVaultName} called
```

```

'{secretName}' with the value '{secretValue}' ...");
await client.SetSecretAsync(secretName, secretValue);
Console.WriteLine(" done.");

Console.WriteLine("Forgetting your secret.");
secretValue = string.Empty;
Console.WriteLine($"Your secret is '{secretValue}'.");

Console.WriteLine($"Retrieving your secret from
{keyVaultName}.");
var secret = await client.GetSecretAsync(secretName);
Console.WriteLine($"Your secret is
'{secret.Value.Value}'.");

Console.Write($"Deleting your secret from {keyVaultName}
...");
DeleteSecretOperation operation = await
client.StartDeleteSecretAsync(secretName);
// You only need to wait for completion if you want to
purge or recover the secret.
await operation.WaitForCompletionAsync();
Console.WriteLine(" done.");

Console.Write($"Purging your secret from {keyVaultName}
...");
await client.PurgeDeletedSecretAsync(secretName);
Console.WriteLine(" done.");
}

}
}

```

Test and verify

1. Execute the following command to run the app.

.NET CLI

`dotnet run`

2. When prompted, enter a secret value. For example, mySecretPassword.

A variation of the following output appears:

Console

```

Input the value of your secret > mySecretPassword
Creating a secret in <your-unique-keyvault-name> called 'mySecret' with the
value 'mySecretPassword' ... done.
Forgetting your secret.
Your secret is ''.

```

```
Retrieving your secret from <your-unique-keyvault-name>.  
Your secret is 'mySecretPassword'.  
Deleting your secret from <your-unique-keyvault-name> ... done.  
Purging your secret from <your-unique-keyvault-name> ... done.
```

Next steps

To learn more about Key Vault and how to integrate it with your apps, see the following articles:

- Read an [Overview of Azure Key Vault](#)
- See an [Access Key Vault from App Service Application Tutorial](#)
- See an [Access Key Vault from Virtual Machine Tutorial](#)
- See the [Azure Key Vault developer's guide](#)
- Review the [Key Vault security overview](#)

Quickstart: Send and receive messages from an Azure Service Bus queue (.NET)

Article • 12/05/2023

In this quickstart, you'll do the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus queue, using the Azure portal.
3. Write a .NET console application to send a set of messages to the queue.
4. Write a .NET console application to receive those messages from the queue.

! Note

This quick start provides step-by-step instructions to implement a simple scenario of sending a batch of messages to a Service Bus queue and then receiving them. For an overview of the .NET client library, see [Azure Service Bus client library for .NET](#). For more samples, see [Service Bus .NET samples on GitHub](#).

Prerequisites

If you're new to the service, see [Service Bus overview](#) before you do this quickstart.

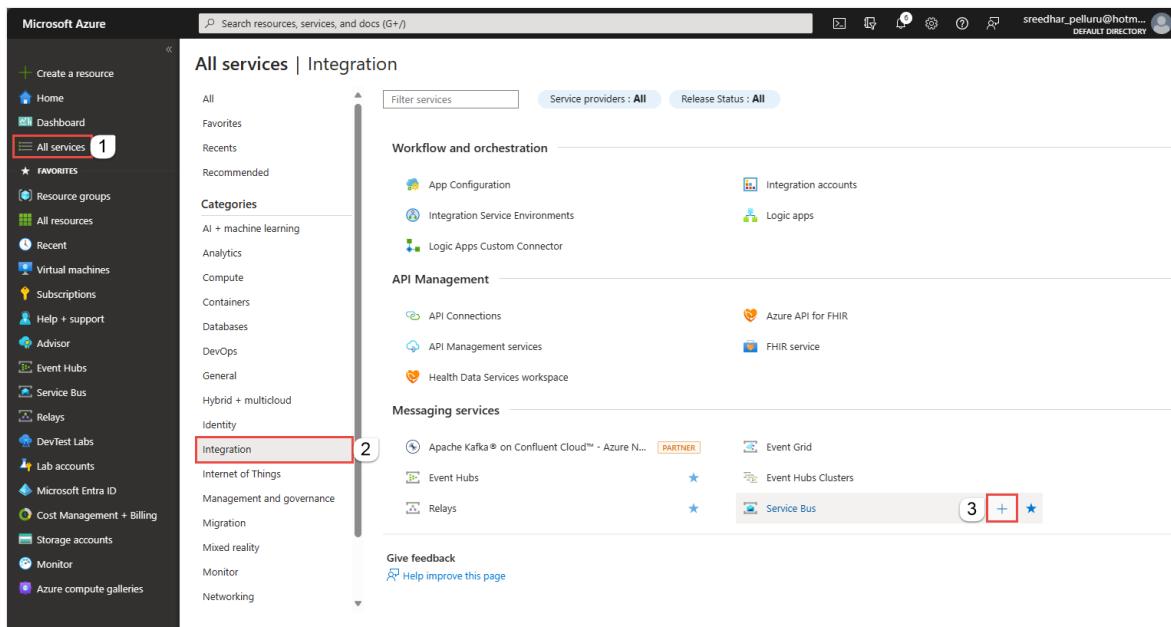
- **Azure subscription.** To use Azure services, including Azure Service Bus, you need a subscription. If you don't have an existing Azure account, you can sign up for a [free trial](#).
- **Visual Studio 2022.** The sample application makes use of new features that were introduced in C# 10. You can still use the Service Bus client library with previous C# language versions, but the syntax might vary. To use the latest syntax, we recommend that you install .NET 6.0, or higher and set the language version to `latest`. If you're using Visual Studio, versions before Visual Studio 2022 aren't compatible with the tools needed to build C# 10 projects.

Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.

To create a namespace:

1. Sign in to the [Azure portal](#).
2. In the left navigation pane of the portal, select **All services**, select **Integration** from the list of categories, hover the mouse over **Service Bus**, and then select + button on the Service Bus tile.



3. In the **Basics** tag of the [Create namespace](#) page, follow these steps:
 - a. For **Subscription**, choose an Azure subscription in which to create the namespace.
 - b. For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
 - c. Enter a **name for the namespace**. The namespace name should adhere to the following naming conventions:
 - The name must be unique across Azure. The system immediately checks to see if the name is available.
 - The name length is at least 6 and at most 50 characters.
 - The name can contain only letters, numbers, hyphens “-”.
 - The name must start with a letter and end with a letter or number.
 - The name doesn't end with “-sb” or “-mgmt”.
 - d. For **Location**, choose the region in which your namespace should be hosted.

- e. For **Pricing tier**, select the pricing tier (Basic, Standard, or Premium) for the namespace. For this quickstart, select **Standard**.

ⓘ Important

If you want to use **topics and subscriptions**, choose either Standard or Premium. Topics/subscriptions aren't supported in the Basic pricing tier.

If you selected the **Premium** pricing tier, specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, 4, 8 or 16 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

- f. Select **Review + create** at the bottom of the page.

The screenshot shows the 'Create namespace' dialog box. The 'Basics' tab is selected. In the 'Project Details' section, the 'Subscription' dropdown is set to 'Visual Studio Enterprise Subscription'. Below it, the 'Resource group' dropdown shows '(New) spsbusrg' with a 'Create new' link. In the 'Instance Details' section, the 'Namespace name' input field contains 'contosoordersns' with a green checkmark and '.servicebus.windows.net' suffix. The 'Location' dropdown is set to 'East US'. The 'Pricing tier' dropdown is set to 'Standard' with a link to 'Browse the available plans and their features'. At the bottom, there are three buttons: 'Review + create' (highlighted in blue), '< Previous', and 'Next: Advanced >'.

- g. On the **Review + create** page, review settings, and select **Create**.
4. Once the deployment of the resource is successful, select **Go to resource** on the deployment page.

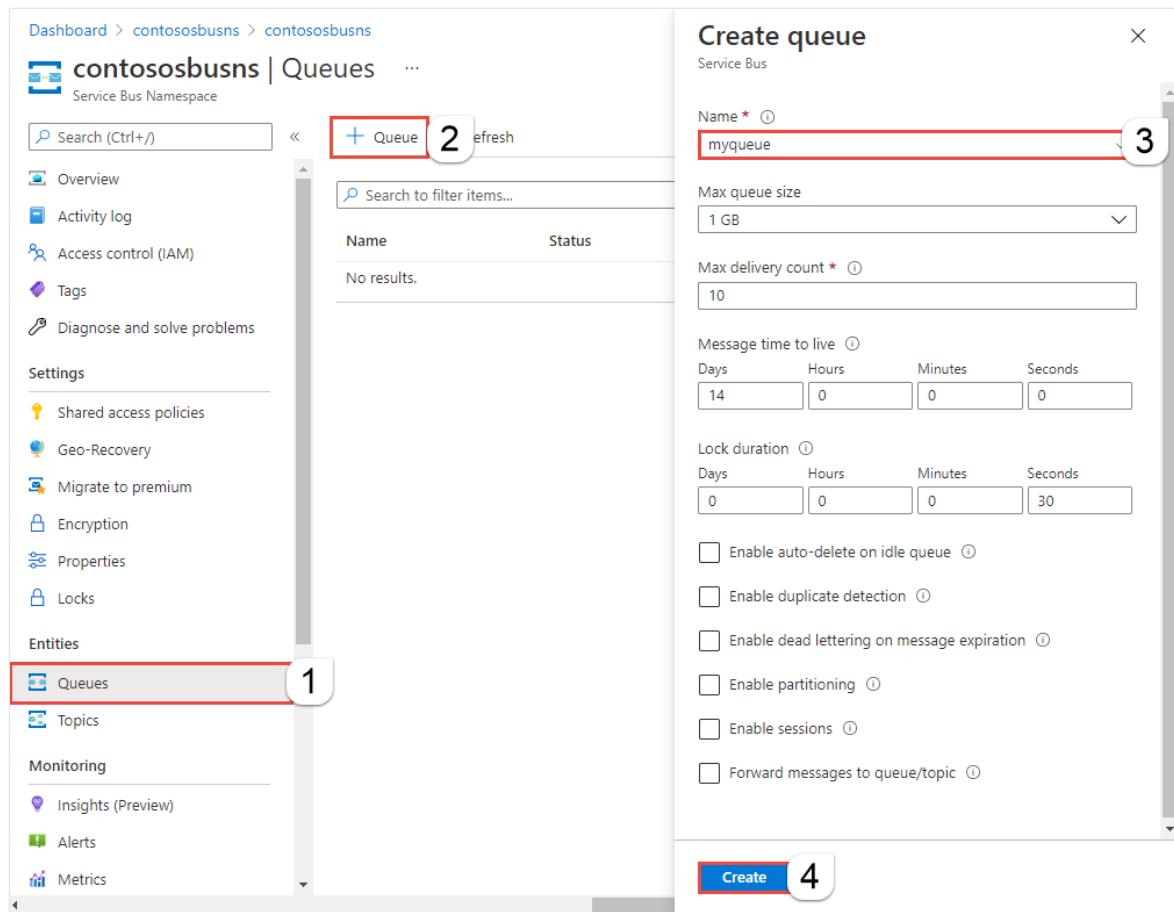
The screenshot shows the Azure portal's deployment overview for the service bus namespace 'contosoordersns'. The main message is 'Your deployment is complete'. Deployment details include the name 'contosoordersns', subscription 'Visual Studio Enterprise Subscription', and resource group 'spsbusrg'. The start time was 10/20/2022, 4:45:03 PM, and the correlation ID is a453ace1-bab9-4c4a-81ad-a1c5366460ea. A 'Go to resource' button is highlighted in red.

5. You see the home page for your service bus namespace.

The screenshot shows the Azure portal's Service Bus Namespace overview for 'spsbusns1128'. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Geo-Recovery, Migrate to premium, Encryption, Configuration, Properties, Locks), Entities (Queues, Topics), Monitoring (Insights (Preview), Alerts), and Queues (0) / Topics (0). The main area displays 'Essentials' information such as Resource group (spsbusrg), Status (Active), Location (East US), Subscription (Visual Studio Enterprise Subscription), Subscription ID, Tags, and various metrics for Requests and Messages over the last hour.

Create a queue in the Azure portal

1. On the Service Bus Namespace page, select **Queues** in the left navigational menu.
2. On the **Queues** page, select **+ Queue** on the toolbar.
3. Enter a name for the queue, and leave the other values with their defaults.
4. Now, select **Create**.



ⓘ Important

If you are new to Azure, you might find the **Connection String** option easier to follow. Select the **Connection String** tab to see instructions on using a connection string in this quickstart. We recommend that you use the **Passwordless** option in real-world applications and production environments.

Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Service Bus: **passwordless** and **connection string**.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace. You don't need to worry about having hard-coded connection string in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a connection string to connect to a Service Bus namespace. If you are new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication](#)

and authorization. You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Service Bus has the correct permissions. You'll need the [Azure Service Bus Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Service Bus Data Owner` role to your user account, which provides full access to Azure Service Bus resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

Azure built-in roles for Azure Service Bus

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters). A member of this role can send and receive messages from queues or topics/subscriptions.
- [Azure Service Bus Data Sender](#): Use this role to give the send access to Service Bus namespace and its entities.
- [Azure Service Bus Data Receiver](#): Use this role to give the receive access to Service Bus namespace and its entities.

If you want to create a custom role, see [Rights required for Service Bus operations](#).

Add Microsoft Entra user to Azure Service Bus Owner role

Add your Microsoft Entra user name to the **Azure Service Bus Data Owner** role at the Service Bus namespace level. It will allow an app running in the context of your user account to send messages to a queue or a topic, and receive messages from a queue or a topic's subscription.

ⓘ Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. If you don't have the Service Bus Namespace page open in the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the Azure Service Bus Namespace Access control (IAM) page. The left sidebar has a red box around the 'Access control (IAM)' menu item. The top navigation bar has a yellow circle with '2' over the '+ Add' button, and another yellow circle with '3' over the 'Add role assignment' option in the dropdown menu. The 'Role assignments' tab is highlighted with a yellow circle and labeled '1'. The 'My access' section is visible below.

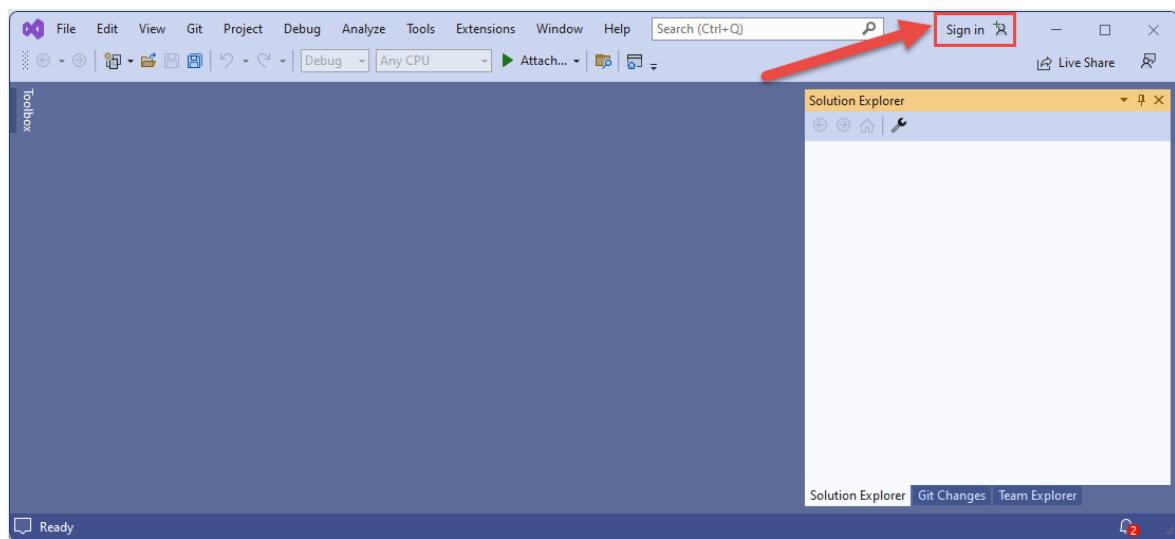
5. Use the search box to filter the results to the desired role. For this example, search for **Azure Service Bus Data Owner** and select the matching result. Then choose **Next**.

6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Launch Visual Studio and sign-in to Azure

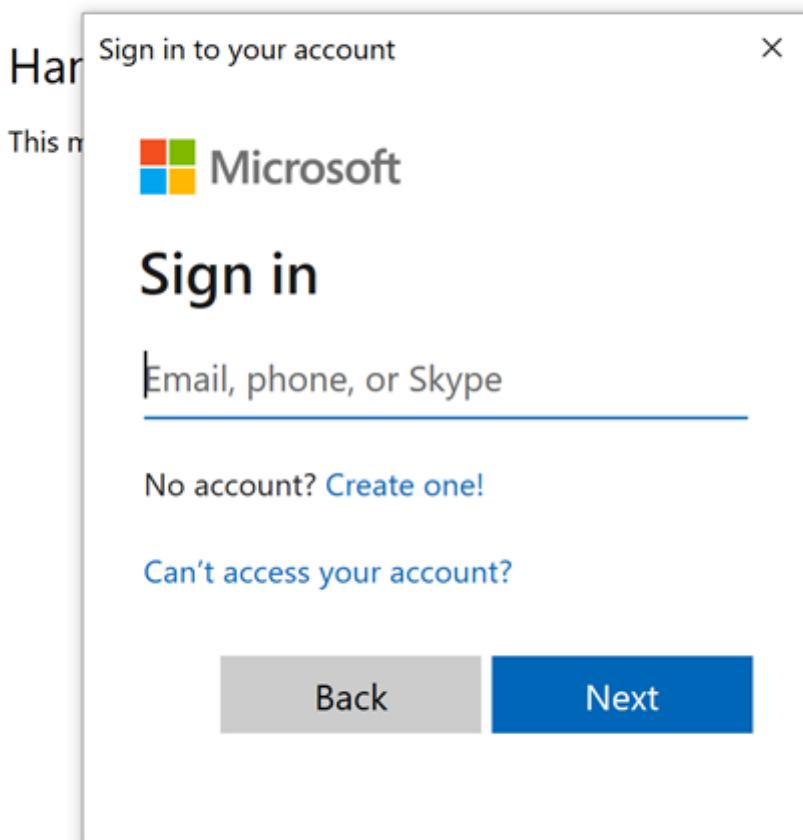
You can authorize access to the service bus namespace using the following steps:

1. Launch Visual Studio. If you see the **Get started** window, select the **Continue without code** link in the right pane.
2. Select the **Sign in** button in the top right of Visual Studio.



3. Sign-in using the Microsoft Entra account you assigned a role to previously.

Visual Studio



Send messages to the queue

This section shows you how to create a .NET console application to send messages to a Service Bus queue.

! Note

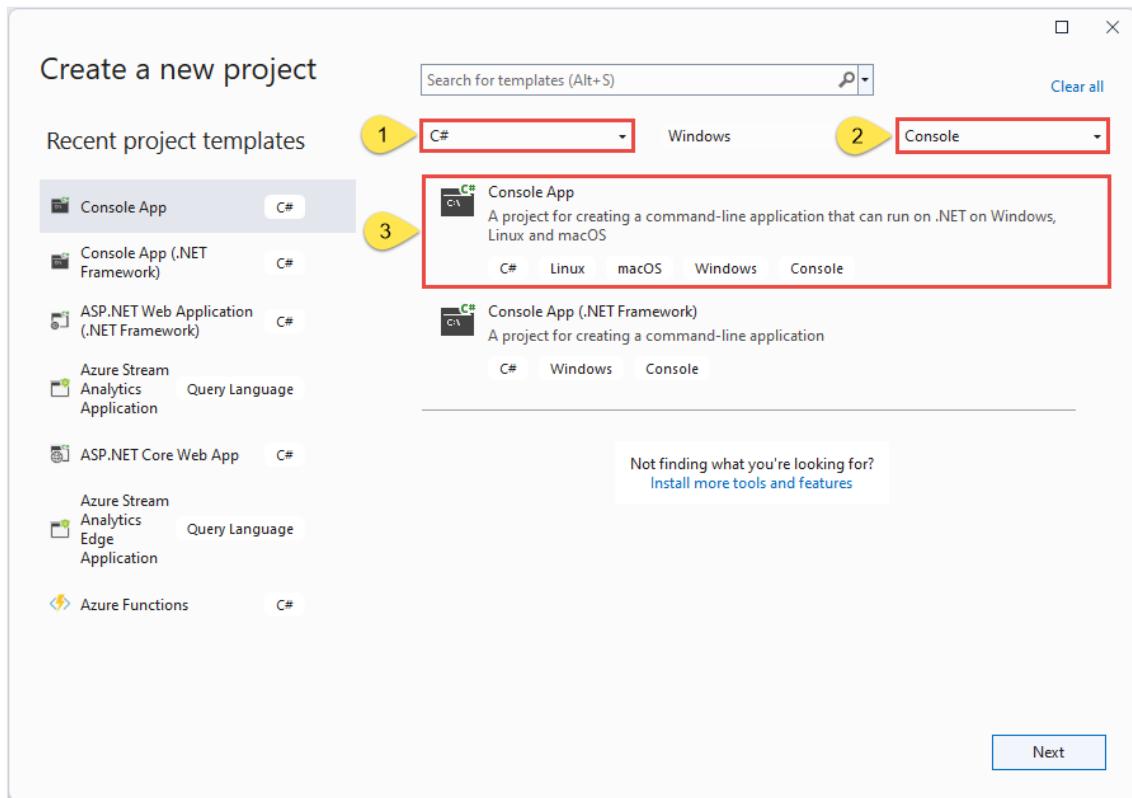
This quick start provides step-by-step instructions to implement a simple scenario of sending a batch of messages to a Service Bus queue and then receiving them. For more samples on other and advanced scenarios, see [Service Bus .NET samples on GitHub](#).

Create a console application

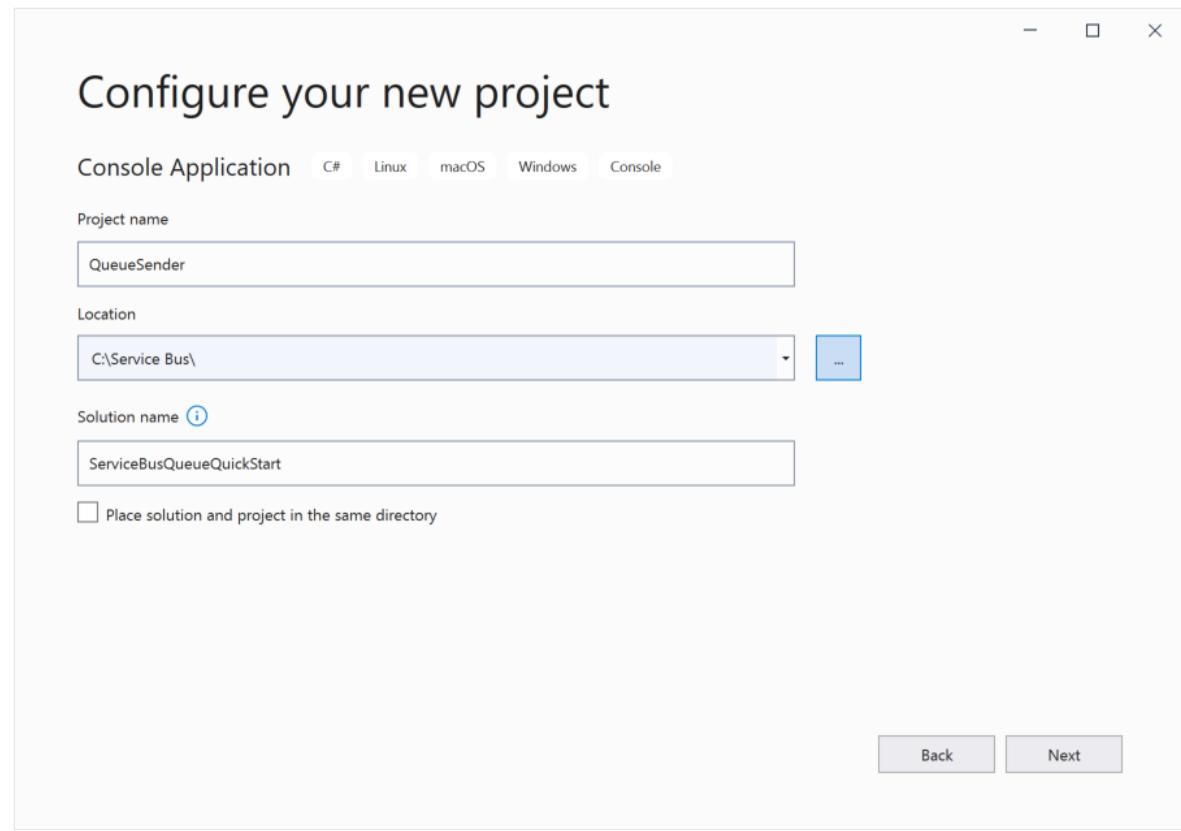
1. In Visual Studio, select **File -> New -> Project** menu.

2. On the **Create a new project** dialog box, do the following steps: If you don't see this dialog box, select **File** on the menu, select **New**, and then select **Project**.

- a. Select **C#** for the programming language.
- b. Select **Console** for the type of the application.
- c. Select **Console App** from the results list.
- d. Then, select **Next**.



3. Enter **QueueSender** for the project name, **ServiceBusQueueQuickStart** for the solution name, and then select **Next**.



4. On the **Additional information** page, select **Create** to create the solution and the project.

Add the NuGet packages to the project

Passwordless

1. Select **Tools > NuGet Package Manager > Package Manager Console** from the menu.
2. Run the following command to install the **Azure.Messaging.ServiceBus** NuGet package.

```
PowerShell  
Install-Package Azure.Messaging.ServiceBus
```
3. Run the following command to install the **Azure.Identity** NuGet package.

```
PowerShell  
Install-Package Azure.Identity
```

Add code to send messages to the queue

1. Replace the contents of `Program.cs` with the following code. The important steps are outlined in the following section, with additional information in the code comments.

Passwordless

- Creates a `ServiceBusClient` object using the `DefaultAzureCredential` object. `DefaultAzureCredential` automatically discovers and uses the credentials of your Visual Studio sign-in to authenticate to Azure Service Bus.
- Invokes the `CreateSender` method on the `ServiceBusClient` object to create a `ServiceBusSender` object for the specific Service Bus queue.
- Creates a `ServiceBusMessageBatch` object by using the `ServiceBusSender.CreateMessageBatchAsync` method.
- Add messages to the batch using the `ServiceBusMessageBatch.TryAddMessage`.
- Sends the batch of messages to the Service Bus queue using the `ServiceBusSender.SendMessagesAsync` method.

ⓘ Important

Update placeholder values (`<NAMESPACE-NAME>` and `<QUEUE-NAME>`) in the code snippet with names of your Service Bus namespace and queue.

C#

```
using Azure.Messaging.ServiceBus;
using Azure.Identity;

// name of your Service Bus queue
// the client that owns the connection and can be used to create
// senders and receivers
ServiceBusClient client;

// the sender used to publish messages to the queue
ServiceBusSender sender;

// number of messages to be sent to the queue
const int numMessages = 3;

// The Service Bus client types are safe to cache and use as a
// singleton for the lifetime
```

```
// of the application, which is best practice when messages are
// being published or read
// regularly.
//
// Set the transport type to AmqpWebSockets so that the
ServiceBusClient uses the port 443.
// If you use the default AmqpTcp, ensure that ports 5671 and 5672
are open.
var clientOptions = new ServiceBusClientOptions
{
    TransportType = ServiceBusTransportType.AmqpWebSockets
};
//TODO: Replace the "<NAMESPACE-NAME>" and "<QUEUE-NAME>"
placeholders.
client = new ServiceBusClient(
    "<NAMESPACE-NAME>.servicebus.windows.net",
    new DefaultAzureCredential(),
    clientOptions);
sender = client.CreateSender("<QUEUE-NAME>");

// create a batch
using ServiceBusMessageBatch messageBatch = await
sender.CreateMessageBatchAsync();

for (int i = 1; i <= numOfMessages; i++)
{
    // try adding a message to the batch
    if (!messageBatch.TryAddMessage(new ServiceBusMessage($"Message
{i}")))
    {
        // if it is too large for the batch
        throw new Exception($"The message {i} is too large to fit
in the batch.");
    }
}

try
{
    // Use the producer client to send the batch of messages to the
Service Bus queue
    await sender.SendMessagesAsync(messageBatch);
    Console.WriteLine($"A batch of {numOfMessages} messages has
been published to the queue.");
}
finally
{
    // Calling DisposeAsync on client types is required to ensure
that network
    // resources and other unmanaged objects are properly cleaned
up.
    await sender.DisposeAsync();
    await client.DisposeAsync();
}
```

```
Console.WriteLine("Press any key to end the application");
Console.ReadKey();
```

2. Build the project, and ensure that there are no errors.
3. Run the program and wait for the confirmation message.

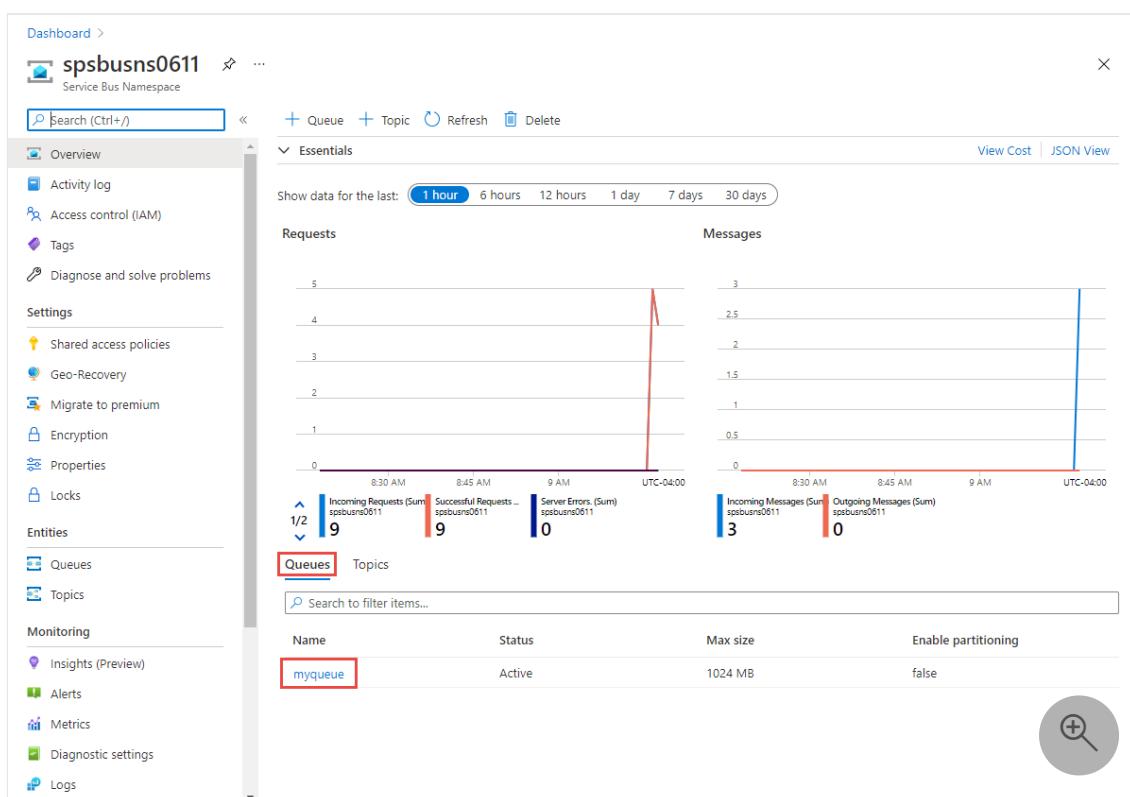
Bash

```
A batch of 3 messages has been published to the queue
```

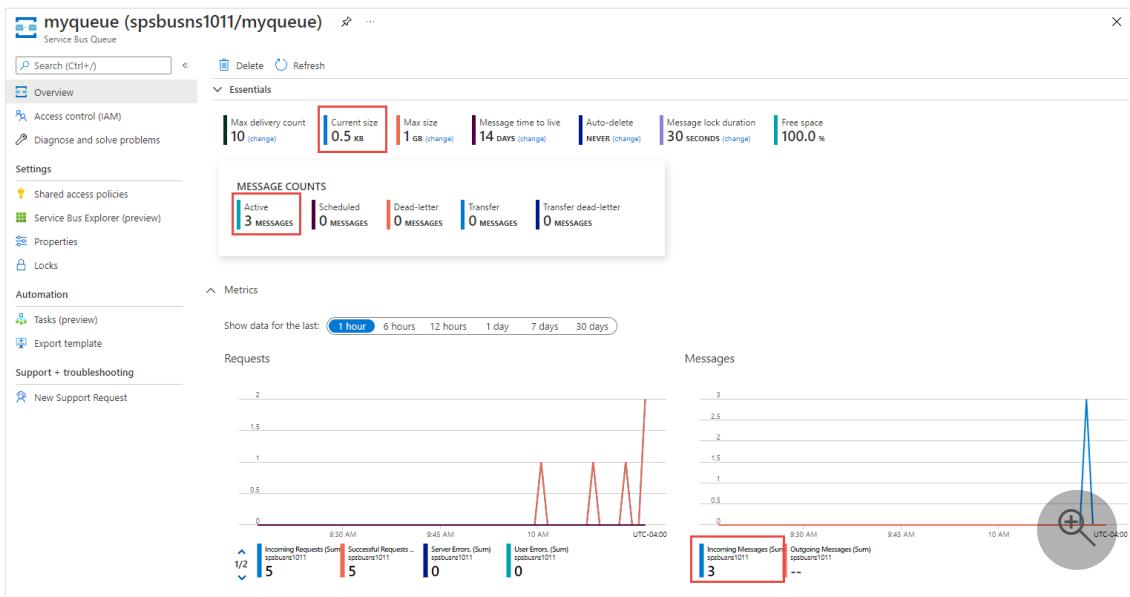
ⓘ Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it might take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

4. In the Azure portal, follow these steps:
 - a. Navigate to your Service Bus namespace.
 - b. On the **Overview** page, select the queue in the bottom-middle pane.



- c. Notice the values in the **Essentials** section.



Notice the following values:

- The **Active** message count value for the queue is now **3**. Each time you run this sender app without retrieving the messages, this value increases by 3.
- The **current size** of the queue increments each time the app adds messages to the queue.
- In the **Messages** chart in the bottom **Metrics** section, you can see that there are three incoming messages for the queue.

Receive messages from the queue

In this section, you create a .NET console application that receives messages from the queue.

! Note

This quickstart provides step-by-step instructions to implement a scenario of sending a batch of messages to a Service Bus queue and then receiving them. For more samples on other and advanced scenarios, see [Service Bus .NET samples on GitHub ↗](#).

Create a project for the receiver

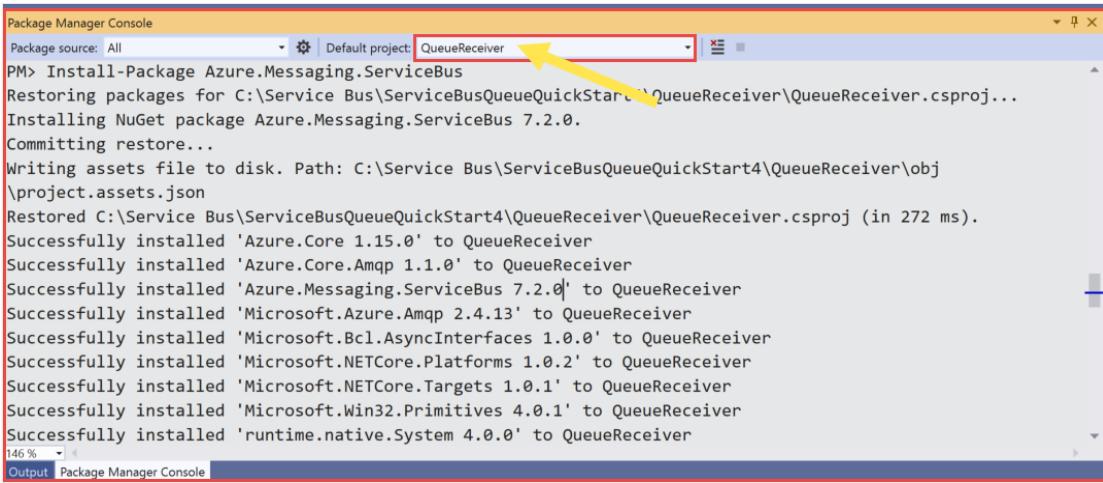
1. In the Solution Explorer window, right-click the **ServiceBusQueueQuickStart** solution, point to **Add**, and select **New Project**.
2. Select **Console application**, and select **Next**.
3. Enter **QueueReceiver** for the **Project name**, and select **Create**.

4. In the Solution Explorer window, right-click QueueReceiver, and select Set as a Startup Project.

Add the NuGet packages to the project

1. Select Tools > NuGet Package Manager > Package Manager Console from the menu.

2. Select QueueReceiver for Default project.



3. Run the following command to install the **Azure.Messaging.ServiceBus** NuGet package.

```
PowerShell
Install-Package Azure.Messaging.ServiceBus
```

4. Run the following command to install the **Azure.Identity** NuGet package.

```
PowerShell
Install-Package Azure.Identity
```

Add the code to receive messages from the queue

In this section, you add code to retrieve messages from the queue.

1. Within the `Program` class, add the following code:

Passwordless

C#

```
using System.Threading.Tasks;
using Azure.Identity;
using Azure.Messaging.ServiceBus;

// the client that owns the connection and can be used to create
// senders and receivers
ServiceBusClient client;

// the processor that reads and processes messages from the queue
ServiceBusProcessor processor;
```

2. Append the following methods to the end of the `Program` class.

C#

```
// handle received messages
async Task MessageHandler(ProcessMessageEventArgs args)
{
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Received: {body}");

    // complete the message. message is deleted from the queue.
    await args.CompleteMessageAsync(args.Message);
}

// handle any errors when receiving messages
Task ErrorHandler(ProcessErrorEventArgs args)
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
}
```

3. Append the following code to the end of the `Program` class. The important steps are outlined in the following section, with additional information in the code comments.

Passwordless

- Creates a `ServiceBusClient` object using the `DefaultAzureCredential` object. `DefaultAzureCredential` automatically discovers and uses the

credentials of your Visual Studio sign in to authenticate to Azure Service Bus.

- Invokes the `CreateProcessor` method on the `ServiceBusClient` object to create a `ServiceBusProcessor` object for the specified Service Bus queue.
- Specifies handlers for the `ProcessMessageAsync` and `ProcessErrorAsync` events of the `ServiceBusProcessor` object.
- Starts processing messages by invoking the `StartProcessingAsync` on the `ServiceBusProcessor` object.
- When user presses a key to end the processing, invokes the `StopProcessingAsync` on the `ServiceBusProcessor` object.

ⓘ Important

Update placeholder values (`<NAMESPACE-NAME>` and `<QUEUE-NAME>`) in the code snippet with names of your Service Bus namespace and queue.

C#

```
// The Service Bus client types are safe to cache and use as a
// singleton for the lifetime
// of the application, which is best practice when messages are
// being published or read
// regularly.
//
// Set the transport type to AmqpWebSockets so that the
// ServiceBusClient uses port 443.
// If you use the default AmqpTcp, make sure that ports 5671 and
// 5672 are open.

// TODO: Replace the <NAMESPACE-NAME> placeholder
var clientOptions = new ServiceBusClientOptions()
{
    TransportType = ServiceBusTransportType.AmqpWebSockets
};
client = new ServiceBusClient(
    "<NAMESPACE-NAME>.servicebus.windows.net",
    new DefaultAzureCredential(),
    clientOptions);

// create a processor that we can use to process the messages
// TODO: Replace the <QUEUE-NAME> placeholder
processor = client.CreateProcessor("<QUEUE-NAME>", new
ServiceBusProcessorOptions());

try
{
    // add handler to process messages
    processor.ProcessMessageAsync += MessageHandler;
```

```

    // add handler to process any errors
    processor.ProcessErrorAsync += ErrorHandler;

    // start processing
    await processor.StartProcessingAsync();

    Console.WriteLine("Wait for a minute and then press any key to
end the processing");
    Console.ReadKey();

    // stop processing
    Console.WriteLine("\nStopping the receiver...");
    await processor.StopProcessingAsync();
    Console.WriteLine("Stopped receiving messages");
}

finally
{
    // Calling DisposeAsync on client types is required to ensure
that network
    // resources and other unmanaged objects are properly cleaned
up.
    await processor.DisposeAsync();
    await client.DisposeAsync();
}

```

4. The completed `Program` class should match the following code:

Passwordless

```

C#

using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
using Azure.Identity;

// the client that owns the connection and can be used to create
senders and receivers
ServiceBusClient client;

// the processor that reads and processes messages from the queue
ServiceBusProcessor processor;

// The Service Bus client types are safe to cache and use as a
singleton for the lifetime
// of the application, which is best practice when messages are
being published or read
// regularly.
//
// Set the transport type to AmqpWebSockets so that the
ServiceBusClient uses port 443.

```

```
// If you use the default AmqpTcp, make sure that ports 5671 and
// 5672 are open.

// TODO: Replace the <NAMESPACE-NAME> and <QUEUE-NAME> placeholders
var clientOptions = new ServiceBusClientOptions()
{
    TransportType = ServiceBusTransportType.AmqpWebSockets
};
client = new ServiceBusClient("<NAMESPACE-NAME>.servicebus.windows.net",
    new DefaultAzureCredential(), clientOptions);

// create a processor that we can use to process the messages
// TODO: Replace the <QUEUE-NAME> placeholder
processor = client.CreateProcessor("<QUEUE-NAME>", new
ServiceBusProcessorOptions());

try
{
    // add handler to process messages
    processor.ProcessMessageAsync += MessageHandler;

    // add handler to process any errors
    processor.ProcessErrorAsync += ErrorHandler;

    // start processing
    await processor.StartProcessingAsync();

    Console.WriteLine("Wait for a minute and then press any key to
end the processing");
    Console.ReadKey();

    // stop processing
    Console.WriteLine("\nStopping the receiver...");
    await processor.StopProcessingAsync();
    Console.WriteLine("Stopped receiving messages");
}

finally
{
    // Calling DisposeAsync on client types is required to ensure
    // that network
    // resources and other unmanaged objects are properly cleaned
    // up.
    await processor.DisposeAsync();
    await client.DisposeAsync();
}

// handle received messages
async Task MessageHandler(ProcessMessageEventArgs args)
{
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Received: {body}");

    // complete the message. message is deleted from the queue.
    await args.CompleteMessageAsync(args.Message);
```

```
}

// handle any errors when receiving messages
Task ErrorHandler(ProcessErrorEventArgs args)
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
}
```

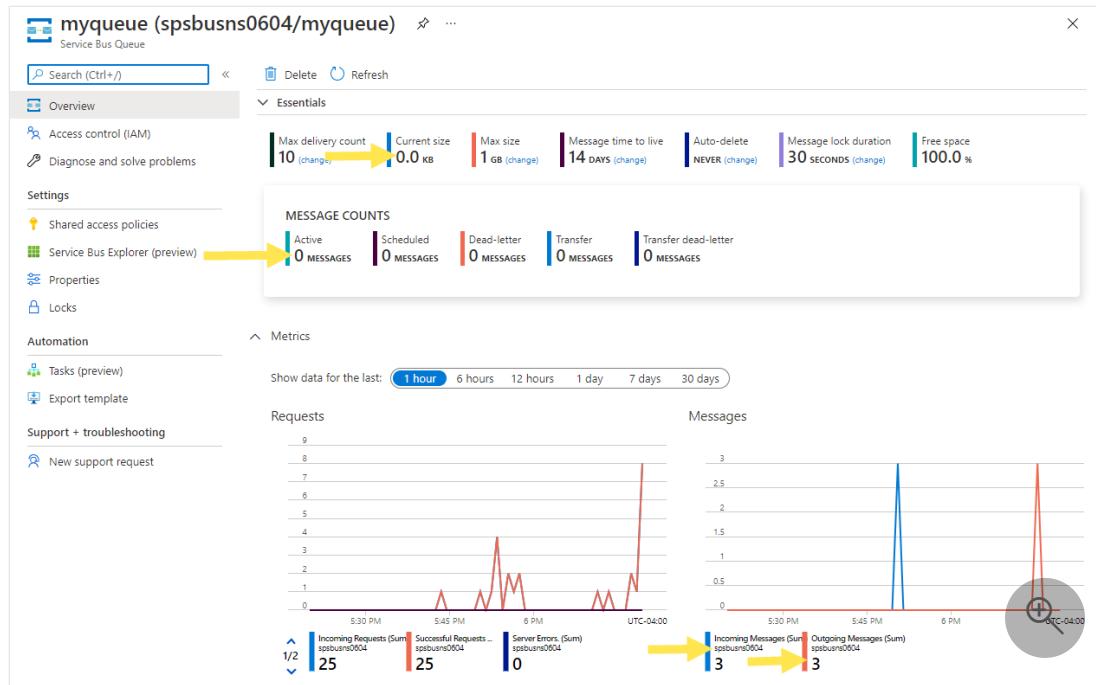
5. Build the project, and ensure that there are no errors.
6. Run the receiver application. You should see the received messages. Press any key to stop the receiver and the application.

Console

```
Wait for a minute and then press any key to end the processing
Received: Message 1
Received: Message 2
Received: Message 3

Stopping the receiver...
Stopped receiving messages
```

7. Check the portal again. Wait for a few minutes and refresh the page if you don't see **0** for **Active** messages.
 - The **Active** message count and **Current size** values are now **0**.
 - In the **Messages** chart in the bottom **Metrics** section, you can see that there are three incoming messages and three outgoing messages for the queue.



Clean up resources

Navigate to your Service Bus namespace in the Azure portal, and select **Delete** on the Azure portal to delete the namespace and the queue in it.

See also

See the following documentation and samples:

- [Azure Service Bus client library for .NET - Readme ↗](#)
- [Samples on GitHub ↗](#)
- [.NET API reference](#)
- [Abstract away infrastructure concerns with higher-level frameworks like NServiceBus](#)

Next steps

[Get started with Azure Service Bus topics and subscriptions \(.NET\)](#)

Get started with Azure Service Bus topics and subscriptions (.NET)

Article • 12/07/2023

This quickstart shows how to send messages to a Service Bus topic and receive messages from a subscription to that topic by using the [Azure.Messaging.ServiceBus](#) .NET library.

In this quickstart, you'll do the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus topic, using the Azure portal.
3. Create a Service Bus subscription to that topic, using the Azure portal.
4. Write a .NET console application to send a set of messages to the topic.
5. Write a .NET console application to receive those messages from the subscription.

ⓘ Note

This quick start provides step-by-step instructions to implement a simple scenario of sending a batch of messages to a Service Bus topic and receiving those messages from a subscription of the topic. For more samples on other and advanced scenarios, see [Service Bus .NET samples on GitHub](#).

- This quick start shows you two ways of connecting to Azure Service Bus: **connection string** and **passwordless**. The first option shows you how to use a connection string to connect to a Service Bus namespace. The second option shows you how to use your security principal in Microsoft Entra ID and the role-based access control (RBAC) to connect to a Service Bus namespace. You don't need to worry about having hard-coded connection string in your code or in a configuration file or in secure storage like Azure Key Vault. If you are new to Azure, you might find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#).

Prerequisites

If you're new to the service, see [Service Bus overview](#) before you do this quickstart.

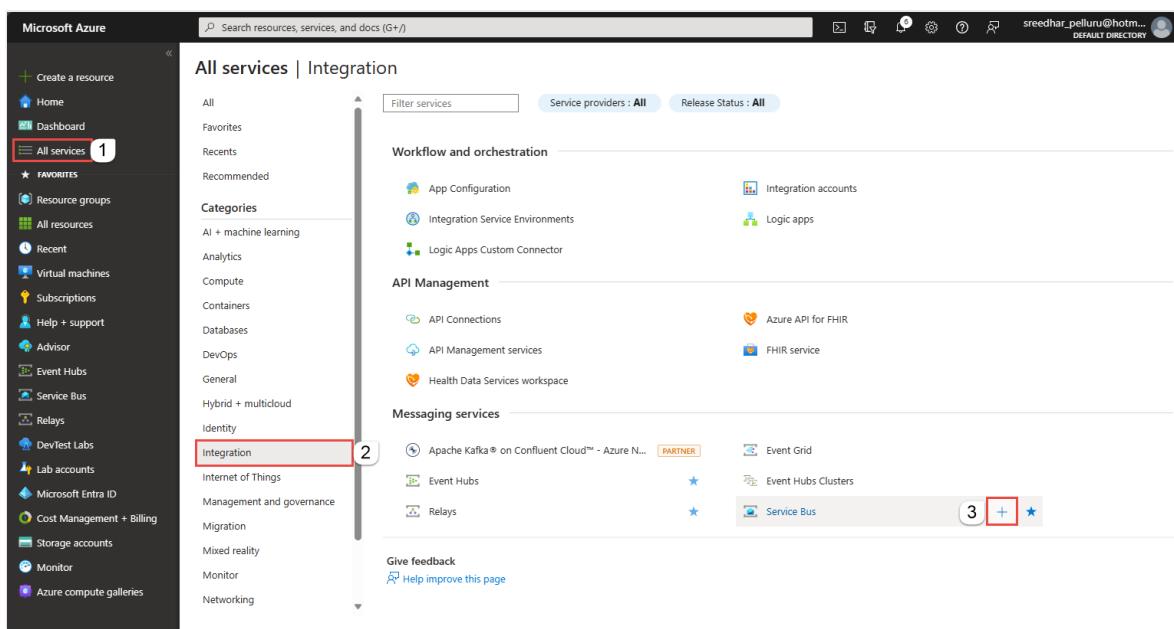
- **Azure subscription.** To use Azure services, including Azure Service Bus, you need a subscription. If you don't have an existing Azure account, you can sign up for a [free trial](#).
- **Visual Studio 2022.** The sample application makes use of new features that were introduced in C# 10. You can still use the Service Bus client library with previous C# language versions, but the syntax might vary. To use the latest syntax, we recommend that you install .NET 6.0, or higher and set the language version to `latest`. If you're using Visual Studio, versions before Visual Studio 2022 aren't compatible with the tools needed to build C# 10 projects.

Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.

To create a namespace:

1. Sign in to the [Azure portal](#).
2. In the left navigation pane of the portal, select **All services**, select **Integration** from the list of categories, hover the mouse over **Service Bus**, and then select **+** button on the Service Bus tile.



3. In the **Basics** tag of the [Create namespace](#) page, follow these steps:
 - a. For **Subscription**, choose an Azure subscription in which to create the namespace.

b. For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.

c. Enter a **name for the namespace**. The namespace name should adhere to the following naming conventions:

- The name must be unique across Azure. The system immediately checks to see if the name is available.
- The name length is at least 6 and at most 50 characters.
- The name can contain only letters, numbers, hyphens “-”.
- The name must start with a letter and end with a letter or number.
- The name doesn't end with “-sb” or “-mgmt”.

d. For **Location**, choose the region in which your namespace should be hosted.

e. For **Pricing tier**, select the pricing tier (Basic, Standard, or Premium) for the namespace. For this quickstart, select **Standard**.

 **Important**

If you want to use [topics and subscriptions](#), choose either Standard or Premium. Topics/subscriptions aren't supported in the Basic pricing tier.

If you selected the **Premium** pricing tier, specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, 4, 8 or 16 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

f. Select **Review + create** at the bottom of the page.

 **Create namespace** ...

Service Bus

Basics Advanced Networking Tags Review + create

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * Visual Studio Enterprise Subscription

Resource group * (New) spsbusrg [Create new](#)

Instance Details

Enter required settings for this namespace.

Namespace name * contosoordersns .servicebus.windows.net

Location * East US

Pricing tier * Standard [Browse the available plans and their features](#)

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

g. On the **Review + create** page, review settings, and select **Create**.

4. Once the deployment of the resource is successful, select **Go to resource** on the deployment page.

 **contosoordersns | Overview** ⚡ ...

Deployment

Search [Delete](#) [Cancel](#) [Redeploy](#) [Download](#) [Refresh](#)

Overview  Your deployment is complete

Deployment name: contosoordersns
Subscription: Visual Studio Enterprise Subscription
Resource group: spsbusrg

Start time: 10/20/2022, 4:45:03 PM
Correlation ID: a453ace1-bab9-4c4a-81ad-a1c5366460ea [Copy](#)

[Deployment details](#) [Next steps](#)

[Go to resource](#)

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

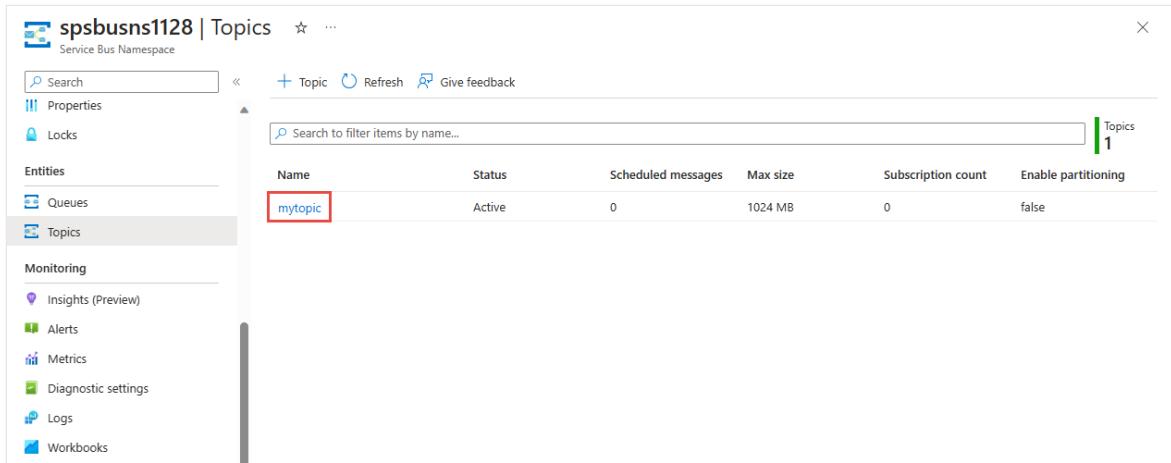
5. You see the home page for your service bus namespace.

Create a topic using the Azure portal

1. On the Service Bus Namespace page, select **Topics** on the left menu.
2. Select **+ Topic** on the toolbar.
3. Enter a **name** for the topic. Leave the other options with their default values.
4. Select **Create**.

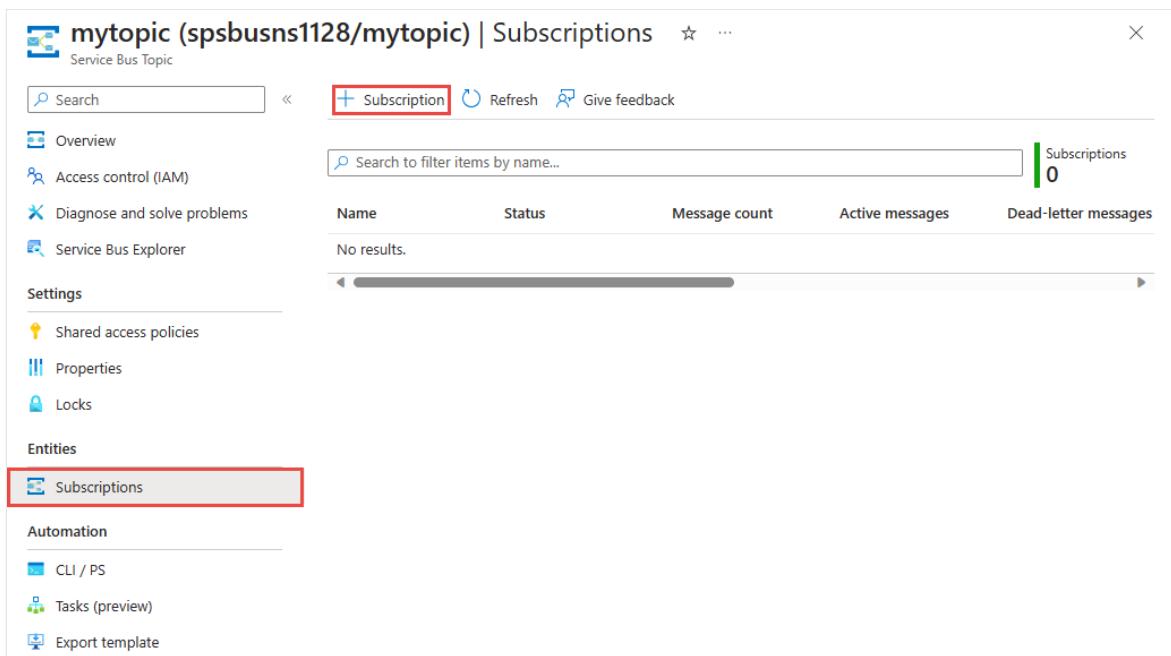
Create a subscription to the topic

1. Select the **topic** that you created in the previous section.



The screenshot shows the 'Topics' blade in the Azure Service Bus Namespace 'spsbusns1128'. On the left, there's a navigation menu with options like 'Search', 'Properties', 'Locks', 'Entities' (Queues and Topics), 'Monitoring' (Insights, Alerts, Metrics, Diagnostic settings, Logs, Workbooks), and 'Service Bus Explorer'. The 'Topics' item under 'Entities' is selected and highlighted with a red box. In the main area, there's a search bar and a table with one row. The table columns are 'Name', 'Status', 'Scheduled messages', 'Max size', 'Subscription count', and 'Enable partitioning'. The single row in the table has 'mytopic' in the 'Name' column, 'Active' in 'Status', '0' in 'Scheduled messages', '1024 MB' in 'Max size', '0' in 'Subscription count', and 'false' in 'Enable partitioning'. A green box highlights the number '1' in the top right corner of the table area.

2. On the Service Bus Topic page, select **+ Subscription** on the toolbar.



The screenshot shows the 'Subscriptions' blade for the Service Bus Topic 'mytopic'. On the left, there's a navigation menu with 'Overview', 'Access control (IAM)', 'Diagnose and solve problems', 'Service Bus Explorer', 'Settings' (Shared access policies, Properties, Locks), 'Entities' (Subscriptions), and 'Automation' (CLI / PS, Tasks (preview), Export template). The 'Subscriptions' item under 'Entities' is selected and highlighted with a red box. The '+ Subscription' button on the toolbar is also highlighted with a red box. In the main area, there's a search bar and a table with one row. The table columns are 'Name', 'Status', 'Message count', 'Active messages', and 'Dead-letter messages'. The single row in the table has 'No results.' in all columns. A green box highlights the number '0' in the top right corner of the table area.

3. On the **Create subscription** page, follow these steps:

- Enter **S1** for **name** of the subscription.
- Enter **3** for **Max delivery count**.
- Then, select **Create** to create the subscription.

Create subscription

Service Bus

Name * ⓘ

S1



Max delivery count * ⓘ

10

Auto-delete after idle for ⓘ

Days

Hours

Minutes

Seconds

14

0

0

0

Never auto-delete

Forward messages to queue/topic ⓘ

MESSAGE SESSIONS

Service bus sessions allow ordered handling of unbounded sequences of related messages. With sessions enabled a subscription can guarantee first-in-first-out delivery of messages. [Learn more.](#)

Enable sessions

MESSAGE TIME TO LIVE AND DEAD-LETTERING

Message time to live (default) ⓘ

Days

Hours

Minutes

Seconds

14

0

0

0

Enable dead lettering on message expiration

Move messages that cause filter evaluation exceptions to the dead-letter subqueue

MESSAGE LOCK DURATION

Lock duration ⓘ

Days

Hours

Minutes

Seconds

0

0

1

0

Create

Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Service Bus: **passwordless** and **connection string**.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace. You don't

need to worry about having hard-coded connection string in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a connection string to connect to a Service Bus namespace. If you are new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Service Bus has the correct permissions. You'll need the [Azure Service Bus Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Service Bus Data Owner` role to your user account, which provides full access to Azure Service Bus resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

Azure built-in roles for Azure Service Bus

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters). A member of this role can send and receive messages from queues or topics/subscriptions.
- [Azure Service Bus Data Sender](#): Use this role to give the send access to Service Bus namespace and its entities.

- **Azure Service Bus Data Receiver:** Use this role to give the receive access to Service Bus namespace and its entities.

If you want to create a custom role, see [Rights required for Service Bus operations](#).

Add Microsoft Entra user to Azure Service Bus Owner role

Add your Microsoft Entra user name to the **Azure Service Bus Data Owner** role at the Service Bus namespace level. It will allow an app running in the context of your user account to send messages to a queue or a topic, and receive messages from a queue or a topic's subscription.

Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. If you don't have the Service Bus Namespace page open in the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

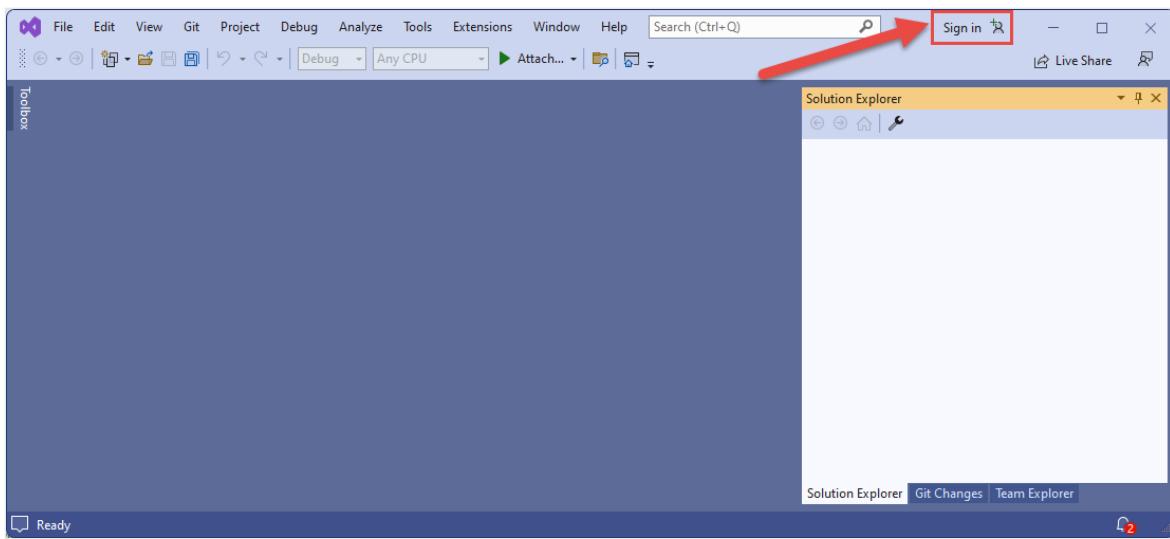
The screenshot shows the 'Access control (IAM)' section of the Azure Service Bus Namespace. The left sidebar has a red box around the 'Access control (IAM)' item. A yellow arrow labeled '1' points to this item. Above the 'Add' button, there is a tooltip with the number '2' and the text 'Download role assignments'. Another tooltip on the 'Add role assignment' button has the number '3'.

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Service Bus Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

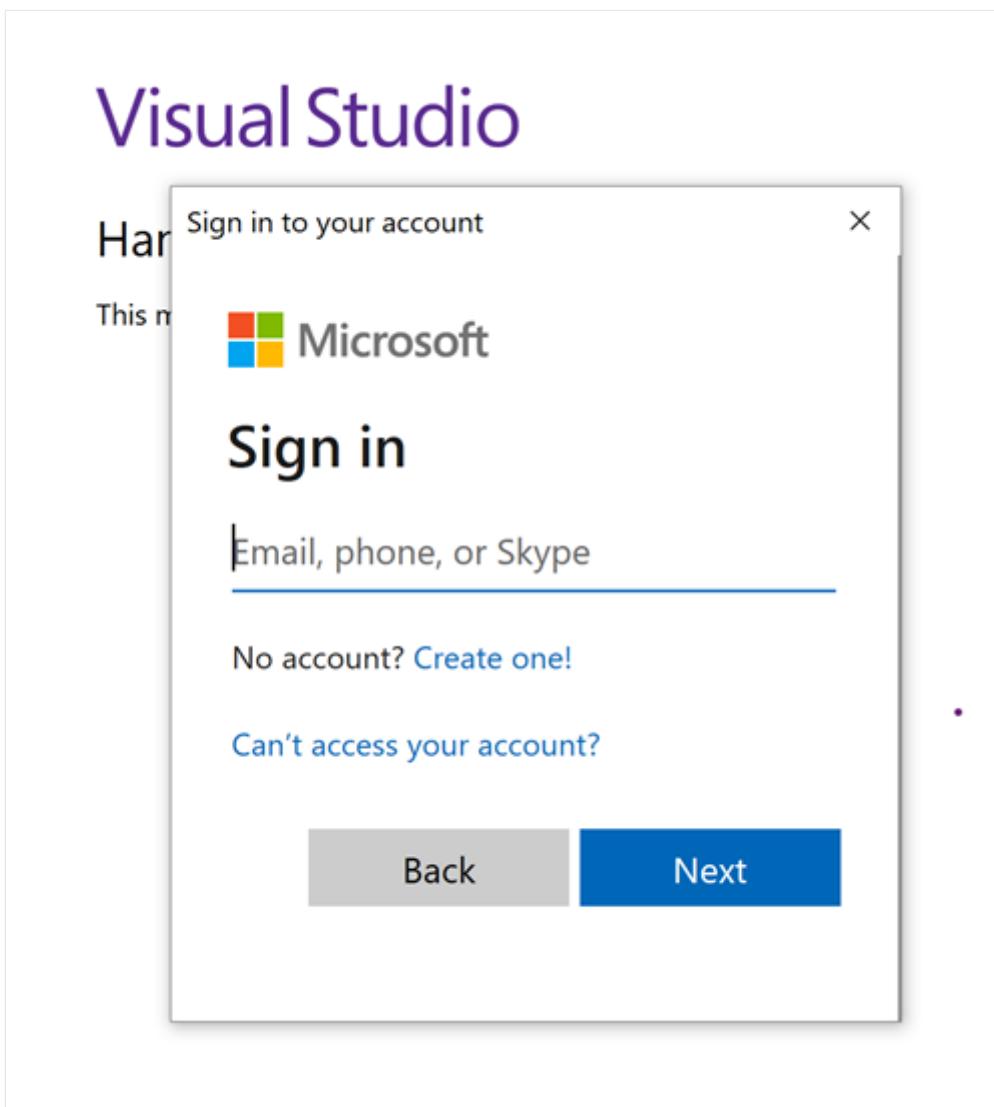
Launch Visual Studio and sign-in to Azure

You can authorize access to the service bus namespace using the following steps:

1. Launch Visual Studio. If you see the **Get started** window, select the **Continue without code** link in the right pane.
2. Select the **Sign in** button in the top right of Visual Studio.



3. Sign-in using the Microsoft Entra account you assigned a role to previously.



Send messages to the topic

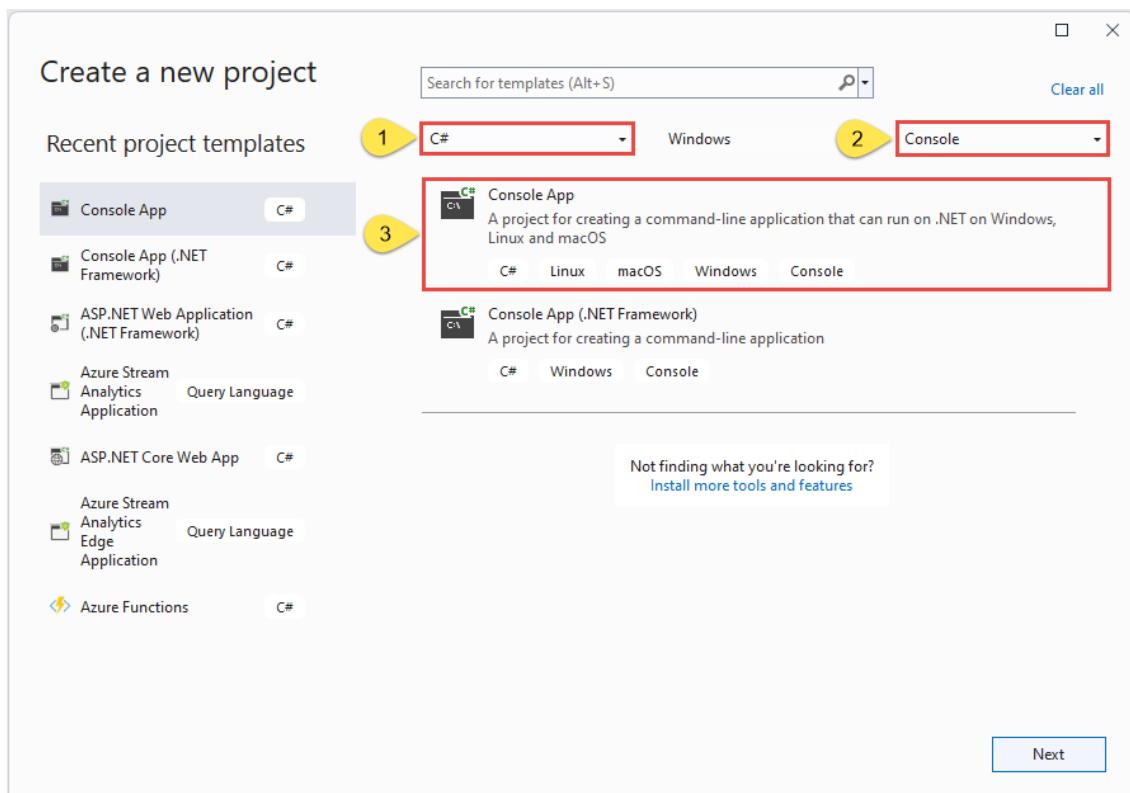
This section shows you how to create a .NET console application to send messages to a Service Bus topic.

⚠ Note

This quick start provides step-by-step instructions to implement a simple scenario of sending a batch of messages to a Service Bus topic and receiving those messages from a subscription of the topic. For more samples on other and advanced scenarios, see [Service Bus .NET samples on GitHub](#).

Create a console application

1. In Visual Studio, select **File** -> **New** -> **Project** menu.
2. On the **Create a new project** dialog box, do the following steps: If you don't see this dialog box, select **File** on the menu, select **New**, and then select **Project**.
 - a. Select **C#** for the programming language.
 - b. Select **Console** for the type of the application.
 - c. Select **Console App** from the results list.
 - d. Then, select **Next**.



3. Enter **TopicSender** for the project name, **ServiceBusTopicQuickStart** for the solution name, and then select **Next**.
4. On the **Additional information** page, select **Create** to create the solution and the project.

Add the NuGet packages to the project

Passwordless

1. Select **Tools > NuGet Package Manager > Package Manager Console** from the menu.
2. Run the following command to install the **Azure.Messaging.ServiceBus** NuGet package.

PowerShell

```
Install-Package Azure.Messaging.ServiceBus
```

3. Run the following command to install the **Azure.Identity** NuGet package.

PowerShell

```
Install-Package Azure.Identity
```

Add code to send messages to the topic

1. Replace the contents of Program.cs with the following code. The important steps are outlined in this section, with additional information in the code comments.

Passwordless

- a. Creates a **ServiceBusClient** object using the **DefaultAzureCredential** object. **DefaultAzureCredential** automatically discovers and uses the credentials of your Visual Studio sign-in to authenticate to Azure Service Bus.
- b. Invokes the **CreateSender** method on the **ServiceBusClient** object to create a **ServiceBusSender** object for the specific Service Bus topic.
- c. Creates a **ServiceBusMessageBatch** object by using the **ServiceBusSender.CreateMessageBatchAsync**.
- d. Add messages to the batch using the **ServiceBusMessageBatch.TryAddMessage**.
- e. Sends the batch of messages to the Service Bus topic using the **ServiceBusSender.SendMessagesAsync** method.

ⓘ Important

Update placeholder values (<NAMESPACE-NAME> and <TOPIC-NAME>) in the code snippet with names of your Service Bus namespace and topic.

C#

```
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
using Azure.Identity;

// the client that owns the connection and can be used to create
// senders and receivers
ServiceBusClient client;

// the sender used to publish messages to the topic
ServiceBusSender sender;

// number of messages to be sent to the topic
const int numOfMessages = 3;

// The Service Bus client types are safe to cache and use as a
// singleton for the lifetime
// of the application, which is best practice when messages are
// being published or read
// regularly.

//TODO: Replace the "<NAMESPACE-NAME>" and "<TOPIC-NAME>"
//placeholders.
client = new ServiceBusClient(
    "<NAMESPACE-NAME>.servicebus.windows.net",
    new DefaultAzureCredential());
sender = client.CreateSender("<TOPIC-NAME>");

// create a batch
using ServiceBusMessageBatch messageBatch = await
sender.CreateMessageBatchAsync();

for (int i = 1; i <= numOfMessages; i++)
{
    // try adding a message to the batch
    if (!messageBatch.TryAddMessage(new ServiceBusMessage($"Message
{i}")))
    {
        // if it is too large for the batch
        throw new Exception($"The message {i} is too large to fit
in the batch.");
    }
}

try
{
```

```
// Use the producer client to send the batch of messages to the
// Service Bus topic
    await sender.SendMessagesAsync(messageBatch);
    Console.WriteLine($"A batch of {numOfMessages} messages has
been published to the topic.");
}
finally
{
    // Calling DisposeAsync on client types is required to ensure
    // that network
    // resources and other unmanaged objects are properly cleaned
    up.
    await sender.DisposeAsync();
    await client.DisposeAsync();
}

Console.WriteLine("Press any key to end the application");
Console.ReadKey();
```

2. Build the project, and ensure that there are no errors.
3. Run the program and wait for the confirmation message.

Bash

```
A batch of 3 messages has been published to the topic
```

 **Important**

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it might take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

4. In the Azure portal, follow these steps:
 - a. Navigate to your Service Bus namespace.
 - b. On the **Overview** page, in the bottom-middle pane, switch to the **Topics** tab, and select the Service Bus topic. In the following example, it's `mytopic`.

- c. On the **Service Bus Topic** page, In the **Messages** chart in the bottom **Metrics** section, you can see that there are three incoming messages for the topic. If you don't see the value, wait for a few minutes, and refresh the page to see the updated chart.

- d. Select the subscription in the bottom pane. In the following example, it's S1. On the **Service Bus Subscription** page, you see the **Active message count** as 3. The subscription has received the three messages that you sent to the topic, but no receiver has picked them yet.

The screenshot shows the Azure Service Bus Subscription Overview page for a subscription named S1. The left sidebar includes links for Overview, Settings, Automation, Support + troubleshooting, and New support request. The main content area displays subscription details like Namespace (spibusn0609), Topic (mytopic), Status (Active), and various message counts (Max delivery count: 3, Active message count: 3 messages, Message time to live: 14 days, etc.). A yellow arrow highlights the 'Active message count' section.

Receive messages from a subscription

In this section, you create a .NET console application that receives messages from the subscription to the Service Bus topic.

Note

This quick start provides step-by-step instructions to implement a simple scenario of sending a batch of messages to a Service Bus topic and receiving those messages from a subscription of the topic. For more samples on other and advanced scenarios, see [Service Bus .NET samples on GitHub](#).

Create a project for the receiver

1. In the Solution Explorer window, right-click the **ServiceBusTopicQuickStart** solution, point to **Add**, and select **New Project**.
2. Select **Console application**, and select **Next**.
3. Enter **SubscriptionReceiver** for the **Project name**, and select **Next**.
4. On the **Additional information** page, select **Create**.
5. In the **Solution Explorer** window, right-click **SubscriptionReceiver**, and select **Set as a Startup Project**.

Add the NuGet packages to the project

>Passwordless

1. Select **Tools > NuGet Package Manager > Package Manager Console** from the menu.
2. Select **SubscriptionReceiver** for **Default project** drop-down list.
3. Run the following command to install the **Azure.Messaging.ServiceBus** NuGet package.

```
PowerShell
```

```
Install-Package Azure.Messaging.ServiceBus
```

4. Run the following command to install the **Azure.Identity** NuGet package.

```
PowerShell
```

```
Install-Package Azure.Identity
```

Add code to receive messages from the subscription

In this section, you add code to retrieve messages from the subscription.

1. Replace the existing contents of `Program.cs` with the following properties and methods:

```
Passwordless
```

```
C#
```

```
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
using Azure.Identity;

// the client that owns the connection and can be used to create
// senders and receivers
ServiceBusClient client;

// the processor that reads and processes messages from the
// subscription
ServiceBusProcessor processor;

// handle received messages
async Task MessageHandler(ProcessMessageEventArgs args)
```

```
{  
    string body = args.Message.Body.ToString();  
    Console.WriteLine($"Received: {body} from subscription.");  
  
    // complete the message. messages is deleted from the  
    // subscription.  
    await args.CompleteMessageAsync(args.Message);  
}  
  
// handle any errors when receiving messages  
Task ErrorHandler(ProcessErrorEventArgs args)  
{  
    Console.WriteLine(args.Exception.ToString());  
    return Task.CompletedTask;  
}
```

2. Append the following code to the end of `Program.cs`.

Passwordless

- Creates a `ServiceBusClient` object using the `DefaultAzureCredential` object. `DefaultAzureCredential` automatically discovers and uses the credentials of your Visual Studio sign-in to authenticate to Azure Service Bus.
- Invokes the `CreateProcessor` method on the `ServiceBusClient` object to create a `ServiceBusProcessor` object for the specified Service Bus topic.
- Specifies handlers for the `ProcessMessageAsync` and `ProcessErrorAsync` events of the `ServiceBusProcessor` object.
- Starts processing messages by invoking the `StartProcessingAsync` on the `ServiceBusProcessor` object.
- When user presses a key to end the processing, invokes the `StopProcessingAsync` on the `ServiceBusProcessor` object.

ⓘ Important

Update placeholder values (<NAMESPACE-NAME>, <TOPIC-NAME>, <SUBSCRIPTION-NAME>) in the code snippet with names of your Service Bus namespace, topic, and subscription.

For more information, see code comments.

C#

```

// The Service Bus client types are safe to cache and use as a
singleton for the lifetime
// of the application, which is best practice when messages are
being published or read
// regularly.
//
// Create the clients that we'll use for sending and processing
messages.
// TODO: Replace the <NAMESPACE-NAME> placeholder
client = new ServiceBusClient(
    "<NAMESPACE-NAME>.servicebus.windows.net",
    new DefaultAzureCredential());

// create a processor that we can use to process the messages
// TODO: Replace the <TOPIC-NAME> and <SUBSCRIPTION-NAME>
placeholders
processor = client.CreateProcessor("<TOPIC-NAME>", "<SUBSCRIPTION-
NAME>", new ServiceBusProcessorOptions());

try
{
    // add handler to process messages
    processor.ProcessMessageAsync += MessageHandler;

    // add handler to process any errors
    processor.ProcessErrorAsync += ErrorHandler;

    // start processing
    await processor.StartProcessingAsync();

    Console.WriteLine("Wait for a minute and then press any key to
end the processing");
    Console.ReadKey();

    // stop processing
    Console.WriteLine("\nStopping the receiver...");
    await processor.StopProcessingAsync();
    Console.WriteLine("Stopped receiving messages");
}
finally
{
    // Calling DisposeAsync on client types is required to ensure
that network
    // resources and other unmanaged objects are properly cleaned
up.
    await processor.DisposeAsync();
    await client.DisposeAsync();
}

```

3. Here's what your `Program.cs` should look like:

C#

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
using Azure.Identity;

// the client that owns the connection and can be used to create
// senders and receivers
ServiceBusClient client;

// the processor that reads and processes messages from the
// subscription
ServiceBusProcessor processor;

// handle received messages
async Task MessageHandler(ProcessMessageEventArgs args)
{
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Received: {body} from subscription.");

    // complete the message. messages is deleted from the
    // subscription.
    await args.CompleteMessageAsync(args.Message);
}

// handle any errors when receiving messages
Task ErrorHandler(ProcessErrorEventArgs args)
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
}

// The Service Bus client types are safe to cache and use as a
// singleton for the lifetime
// of the application, which is best practice when messages are
// being published or read
// regularly.
//
// Create the clients that we'll use for sending and processing
// messages.
// TODO: Replace the <NAMESPACE-NAME> placeholder
client = new ServiceBusClient(
    "<NAMESPACE-NAME>.servicebus.windows.net",
    new DefaultAzureCredential());

// create a processor that we can use to process the messages
// TODO: Replace the <TOPIC-NAME> and <SUBSCRIPTION-NAME>
// placeholders
processor = client.CreateProcessor("<TOPIC-NAME>", "<SUBSCRIPTION-
NAME>", new ServiceBusProcessorOptions());
```

```

try
{
    // add handler to process messages
    processor.ProcessMessageAsync += MessageHandler;

    // add handler to process any errors
    processor.ProcessErrorAsync += ErrorHandler;

    // start processing
    await processor.StartProcessingAsync();

    Console.WriteLine("Wait for a minute and then press any key to
end the processing");
    Console.ReadKey();

    // stop processing
    Console.WriteLine("\nStopping the receiver...");
    await processor.StopProcessingAsync();
    Console.WriteLine("Stopped receiving messages");
}
finally
{
    // Calling DisposeAsync on client types is required to ensure
    // that network
    // resources and other unmanaged objects are properly cleaned
    up.
    await processor.DisposeAsync();
    await client.DisposeAsync();
}

```

4. Build the project, and ensure that there are no errors.

5. Run the receiver application. You should see the received messages. Press any key to stop the receiver and the application.

Console

```

Wait for a minute and then press any key to end the processing
Received: Message 1 from subscription: S1
Received: Message 2 from subscription: S1
Received: Message 3 from subscription: S1

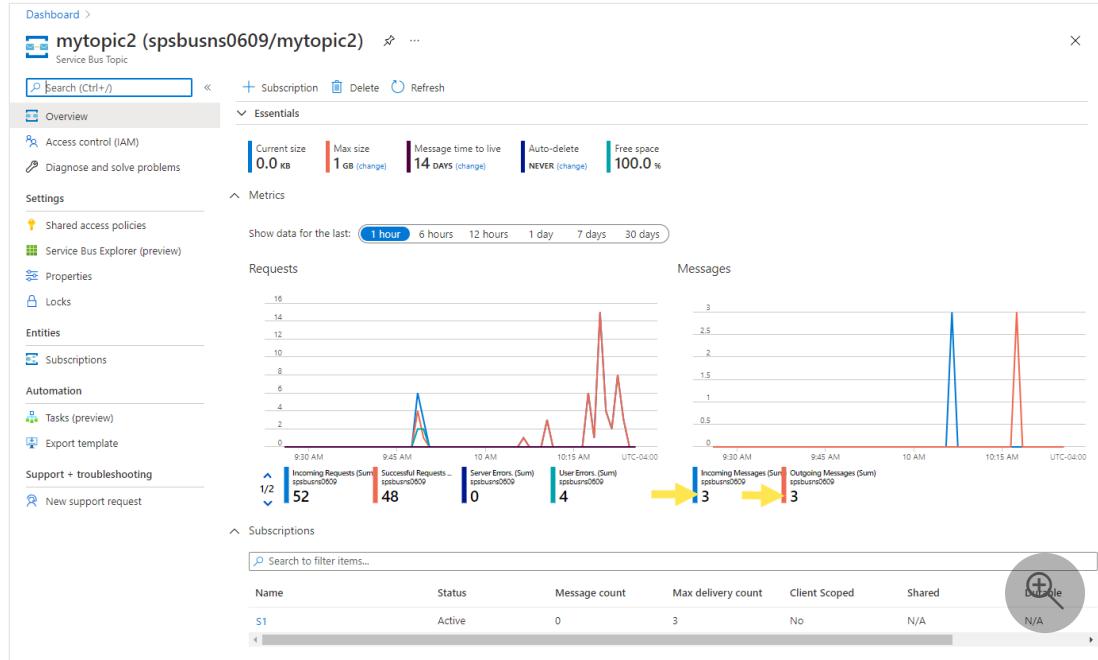
Stopping the receiver...
Stopped receiving messages

```

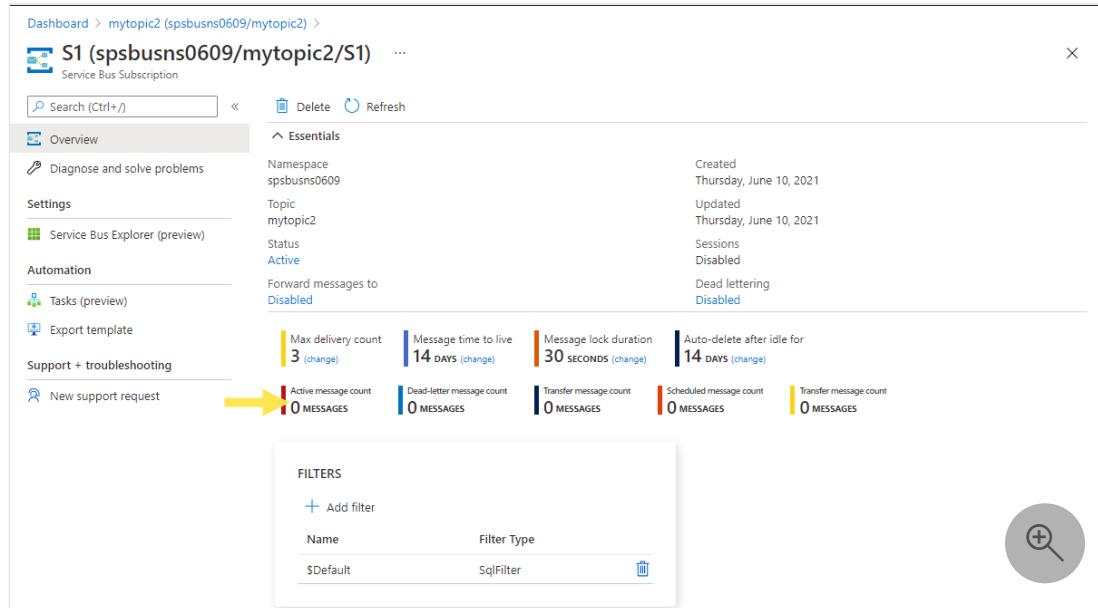
6. Check the portal again.

- On the Service Bus Topic page, in the **Messages** chart, you see three incoming messages and three outgoing messages. If you don't see these

numbers, wait for a few minutes, and refresh the page to see the updated chart.



- On the Service Bus Subscription page, you see the Active message count as zero. It's because a receiver has received messages from this subscription and completed the messages.



Next steps

See the following documentation and samples:

- [Azure Service Bus client library for .NET - Readme ↗](#)
- [.NET samples for Azure Service Bus on GitHub ↗](#)
- [.NET API reference](#)

Quickstart: Azure Blob Storage client library for .NET

Article • 02/06/2024

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

ⓘ Note

The **Build from scratch** option walks you step by step through the process of creating a new project, installing packages, writing the code, and running a basic console app. This approach is recommended if you want to understand all the details involved in creating an app that connects to Azure Blob Storage. If you prefer to automate deployment tasks and start with a completed project, choose [Start with a template](#).

Get started with the Azure Blob Storage client library for .NET. Azure Blob Storage is Microsoft's object storage solution for the cloud, and is optimized for storing massive amounts of unstructured data.

In this article, you follow steps to install the package and try out example code for basic tasks.

[API reference documentation](#) | [Library source code](#) ↗ | [Package \(NuGet\)](#) ↗ | [Samples](#)

This video shows you how to start using the Azure Blob Storage client library for .NET.
<https://learn-video.azurefd.net/vod/player?id=cdae65e7-1892-48fe-934a-70edfbe147be&locale=en-us&embedUrl=%2Fazure%2Fstorage%2Fblobs%2Fstorage-quickstart-blobs-dotnet> ↗

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

Prerequisites

- Azure subscription - [create one for free](#) ↗
- Azure storage account - [create a storage account](#)
- Latest [.NET SDK](#) ↗ for your operating system. Be sure to get the SDK and not the runtime.

Setting up

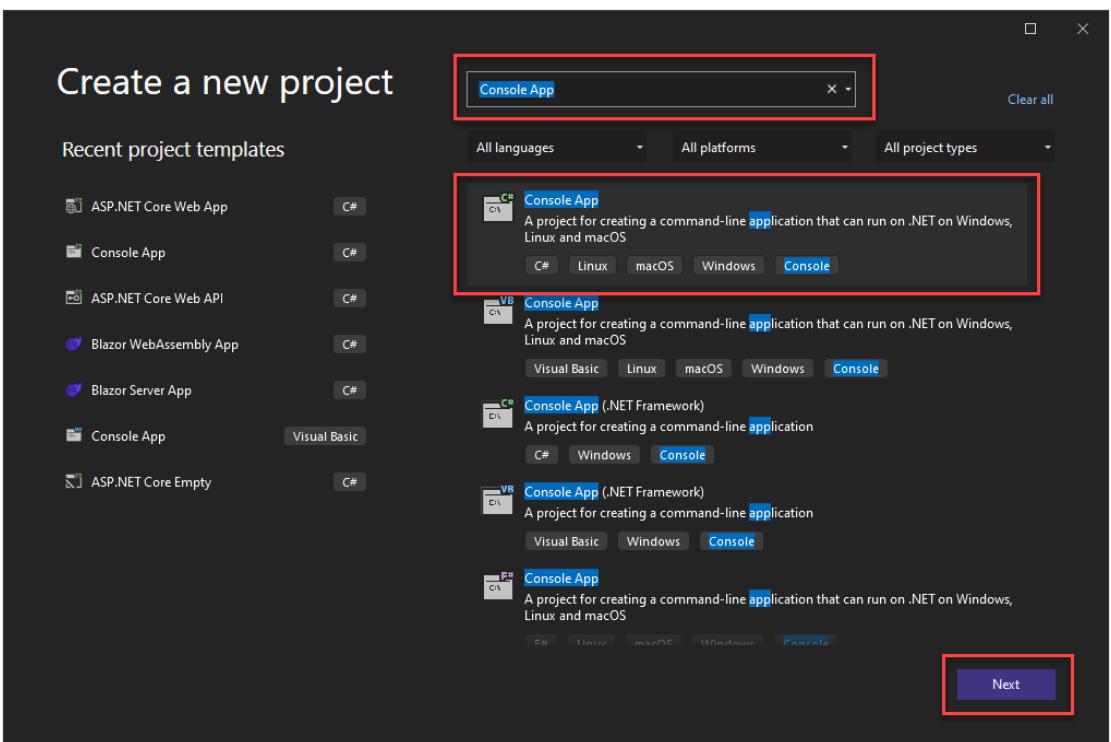
This section walks you through preparing a project to work with the Azure Blob Storage client library for .NET.

Create the project

Create a .NET console app using either the .NET CLI or Visual Studio 2022.

Visual Studio 2022

1. At the top of Visual Studio, navigate to **File > New > Project...**
2. In the dialog window, enter *console app* into the project template search box and select the first result. Choose **Next** at the bottom of the dialog.

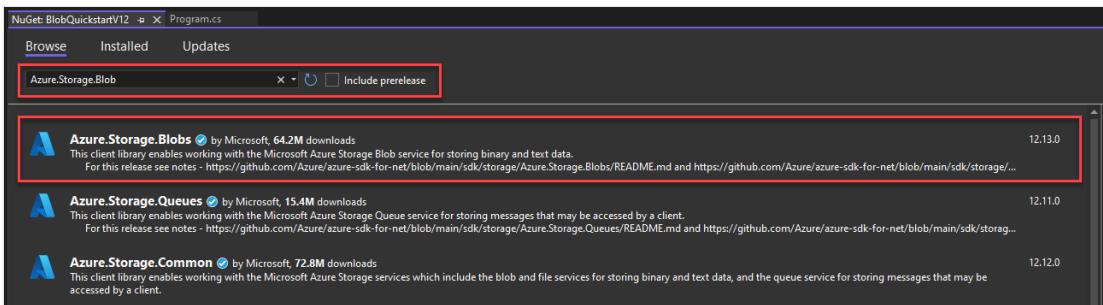


3. For the **Project Name**, enter *BlobQuickstart*. Leave the default values for the rest of the fields and select **Next**.
4. For the **Framework**, ensure the latest installed version of .NET is selected. Then choose **Create**. The new project opens inside the Visual Studio environment.

Install the package

To interact with Azure Blob Storage, install the Azure Blob Storage client library for .NET.

1. In Solution Explorer, right-click the **Dependencies** node of your project. Select **Manage NuGet Packages**.
2. In the resulting window, search for *Azure.Storage.Blobs*. Select the appropriate result, and select **Install**.



Set up the app code

Replace the starting code in the `Program.cs` file so that it matches the following example, which includes the necessary `using` statements for this exercise.

C#

```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using System;
using System.IO;

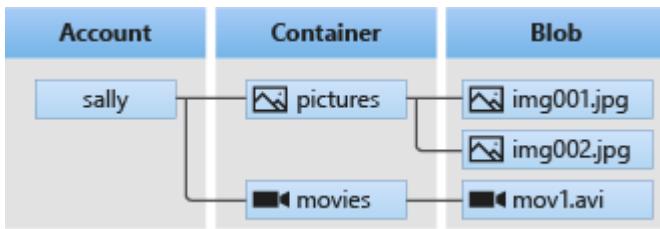
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Object model

Azure Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data doesn't adhere to a particular data model or definition, such as text or binary data. Blob storage offers three types of resources:

- The storage account
- A container in the storage account
- A blob in the container

The following diagram shows the relationship between these resources.



Use the following .NET classes to interact with these resources:

- [BlobServiceClient](#): The `BlobServiceClient` class allows you to manipulate Azure Storage resources and blob containers.
- [BlobContainerClient](#): The `BlobContainerClient` class allows you to manipulate Azure Storage containers and their blobs.
- [BlobClient](#): The `BlobClient` class allows you to manipulate Azure Storage blobs.

Code examples

The sample code snippets in the following sections demonstrate how to perform the following tasks with the Azure Blob Storage client library for .NET:

- [Authenticate to Azure and authorize access to blob data](#)
- [Create a container](#)
- [Upload a blob to a container](#)
- [List blobs in a container](#)
- [Download a blob](#)
- [Delete a container](#)

Important

Make sure you've installed the correct NuGet packages and added the necessary `using` statements in order for the code samples to work, as described in the [setting up](#) section.

Authenticate to Azure and authorize access to blob data

Application requests to Azure Blob Storage must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code, including Blob Storage.

You can also authorize requests to Azure Blob Storage by using the account access key. However, this approach should be used with caution. Developers must be diligent to

never expose the access key in an unsecure location. Anyone who has the access key is able to authorize requests against the storage account, and effectively has access to all the data. `DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

Passwordless (Recommended)

`DefaultAzureCredential` is a class provided by the Azure Identity client library for .NET, which you can learn more about on the [DefaultAzureCredential overview](#).

`DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` looks for credentials can be found in the [Azure Identity library overview](#).

For example, your app can authenticate using your Visual Studio sign-in credentials with when developing locally. Your app can then use a [managed identity](#) once it has been deployed to Azure. No code changes are required for this transition.

Assign roles to your Microsoft Entra user account

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

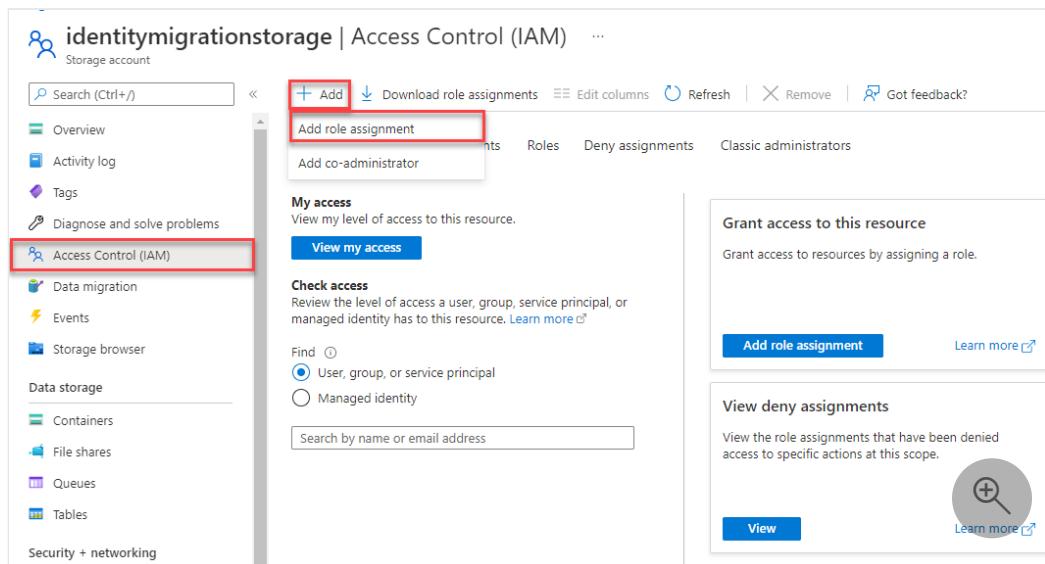
The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

ⓘ Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.

dialog.

8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Sign in and connect your app code to Azure using DefaultAzureCredential

You can authorize access to data in your storage account using the following steps:

1. For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

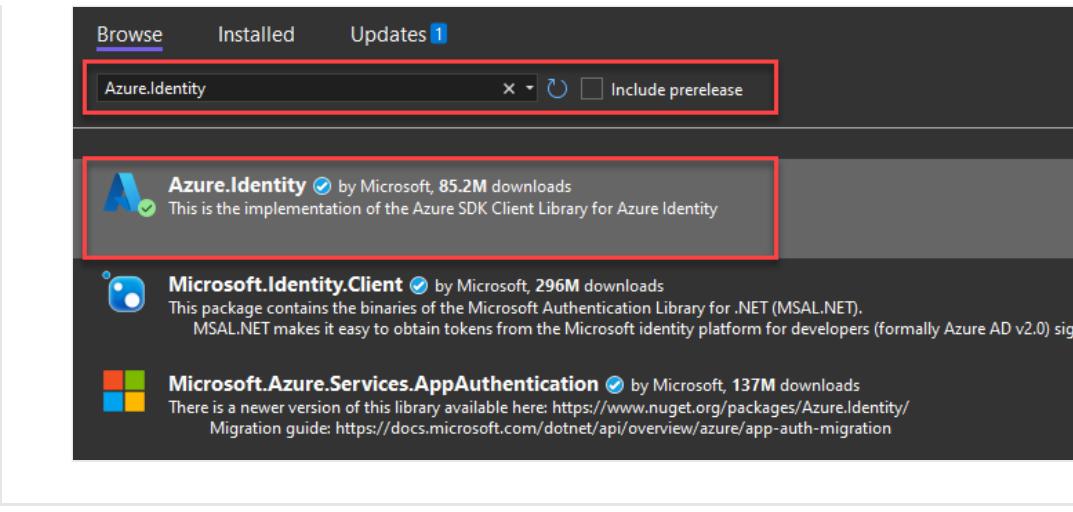
Azure CLI

```
az login
```

2. To use `DefaultAzureCredential`, add the **Azure.Identity** package to your application.

Visual Studio

- a. In **Solution Explorer**, right-click the **Dependencies** node of your project. Select **Manage NuGet Packages**.
- b. In the resulting window, search for *Azure.Identity*. Select the appropriate result, and select **Install**.



3. Update your *Program.cs* code to match the following example. When the code is run on your local workstation during development, it will use the developer credentials of the prioritized tool you're logged into to authenticate to Azure, such as the Azure CLI or Visual Studio.

C#

```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using System;
using System.IO;
using Azure.Identity;

// TODO: Replace <storage-account-name> with your actual storage
account name
var blobServiceClient = new BlobServiceClient(
    new Uri("https://<storage-account-
name>.blob.core.windows.net"),
    new DefaultAzureCredential());
```

4. Make sure to update the storage account name in the URI of your `BlobServiceClient`. The storage account name can be found on the overview page of the Azure portal.

identitymigrationstorage

Storage account

Search (Ctrl+ /)

Upload Open in Explorer Delete Move Refresh

Overview Activity log Tags Diagnose and solve problems Access Control (IAM) Data migration Events Storage browser

Resource group (move) : alexw-identity-revamp

Location : East US

Primary/Secondary Location : Primary: East US, Secondary: West US

Subscription (move) : C&L Cross Service Content Team Testing

Subscription ID :

Disk state : Primary: Available, Secondary: Available

Tags (edit) : Click here to add tags

① Note

When deployed to Azure, this same code can be used to authorize requests to Azure Storage from an application running in Azure. However, you'll need to enable managed identity on your app in Azure. Then configure your storage account to allow that managed identity to connect. For detailed instructions on configuring this connection between Azure services, see the [Auth from Azure-hosted apps](#) tutorial.

Create a container

Create a new container in your storage account by calling the `CreateBlobContainerAsync` method on the `blobServiceClient` object. In this example, the code appends a GUID value to the container name to ensure that it's unique.

Add the following code to the end of the `Program.cs` file:

C#

```
// TODO: Replace <storage-account-name> with your actual storage account
name
var blobServiceClient = new BlobServiceClient(
    new Uri("https://<storage-account-name>.blob.core.windows.net"),
    new DefaultAzureCredential());

//Create a unique name for the container
string containerName = "quickstartblobs" + Guid.NewGuid().ToString();

// Create the container and return a container client object
```

```
BlobContainerClient containerClient = await  
blobServiceClient.CreateBlobContainerAsync(containerName);
```

To learn more about creating a container, and to explore more code samples, see [Create a blob container with .NET](#).

ⓘ Important

Container names must be lowercase. For more information about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Upload a blob to a container

Upload a blob to a container using [UploadAsync](#). The example code creates a text file in the local *data* directory to upload to the container.

Add the following code to the end of the `Program.cs` file:

C#

```
// Create a local file in the ./data/ directory for uploading and  
downloading  
string localPath = "data";  
Directory.CreateDirectory(localPath);  
string fileName = "quickstart" + Guid.NewGuid().ToString() + ".txt";  
string localFilePath = Path.Combine(localPath, fileName);  
  
// Write text to the file  
await File.WriteAllTextAsync(localFilePath, "Hello, World!");  
  
// Get a reference to a blob  
BlobClient blobClient = containerClient.GetBlobClient(fileName);  
  
Console.WriteLine("Uploading to Blob storage as blob:\n\t {0}\n",  
blobClient.Uri);  
  
// Upload data from the local file, overwrite the blob if it already exists  
await blobClient.UploadAsync(localFilePath, true);
```

To learn more about uploading blobs, and to explore more code samples, see [Upload a blob with .NET](#).

List blobs in a container

List the blobs in the container by calling the [GetBlobsAsync](#) method.

Add the following code to the end of the `Program.cs` file:

```
C#  
  
Console.WriteLine("Listing blobs...");  
  
// List all blobs in the container  
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())  
{  
    Console.WriteLine("\t" + blobItem.Name);  
}
```

To learn more about listing blobs, and to explore more code samples, see [List blobs with .NET](#).

Download a blob

Download the blob we created earlier by calling the [DownloadToAsync](#) method. The example code appends the string "DOWNLOADED" to the file name so that you can see both files in local file system.

Add the following code to the end of the `Program.cs` file:

```
C#  
  
// Download the blob to a local file  
// Append the string "DOWNLOADED" before the .txt extension  
// so you can compare the files in the data directory  
string downloadFilePath = localFilePath.Replace(".txt", "DOWNLOADED.txt");  
  
Console.WriteLine("\nDownloading blob to\n\t{0}\n", downloadFilePath);  
  
// Download the blob's contents and save it to a file  
await blobClient.DownloadToAsync(downloadFilePath);
```

To learn more about downloading blobs, and to explore more code samples, see [Download a blob with .NET](#).

Delete a container

The following code cleans up the resources the app created by deleting the container using [DeleteAsync](#). The code example also deletes the local files created by the app.

The app pauses for user input by calling `Console.ReadLine` before it deletes the blob, container, and local files. This is a good chance to verify that the resources were created correctly, before they're deleted.

Add the following code to the end of the `Program.cs` file:

```
C#  
  
// Clean up  
Console.Write("Press any key to begin clean up");  
Console.ReadLine();  
  
Console.WriteLine("Deleting blob container...");  
await containerClient.DeleteAsync();  
  
Console.WriteLine("Deleting the local source and downloaded files...");  
File.Delete(localFilePath);  
File.Delete(downloadFilePath);  
  
Console.WriteLine("Done");
```

To learn more about deleting a container, and to explore more code samples, see [Delete and restore a blob container with .NET](#).

The completed code

After completing these steps, the code in your `Program.cs` file should now resemble the following:

Passwordless (Recommended)

```
C#  
  
using Azure.Storage.Blobs;  
using Azure.Storage.Blobs.Models;  
using Azure.Identity;  
  
// TODO: Replace <storage-account-name> with your actual storage account  
name  
var blobServiceClient = new BlobServiceClient(  
    new Uri("https://<storage-account-name>.blob.core.windows.net"),  
    new DefaultAzureCredential());  
  
//Create a unique name for the container  
string containerName = "quickstartblobs" + Guid.NewGuid().ToString();  
  
// Create the container and return a container client object  
BlobContainerClient containerClient = await
```

```
blobServiceClient.CreateBlobContainerAsync(containerName);

// Create a local file in the ./data/ directory for uploading and
// downloading
string localPath = "data";
Directory.CreateDirectory(localPath);
string fileName = "quickstart" + Guid.NewGuid().ToString() + ".txt";
string localFilePath = Path.Combine(localPath, fileName);

// Write text to the file
await File.WriteAllTextAsync(localFilePath, "Hello, World!");

// Get a reference to a blob
BlobClient blobClient = containerClient.GetBlobClient(fileName);

Console.WriteLine("Uploading to Blob storage as blob:\n\t {0}\n",
blobClient.Uri);

// Upload data from the local file
await blobClient.UploadAsync(localFilePath, true);

Console.WriteLine("Listing blobs...");

// List all blobs in the container
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())
{
    Console.WriteLine("\t" + blobItem.Name);
}

// Download the blob to a local file
// Append the string "DOWNLOADED" before the .txt extension
// so you can compare the files in the data directory
string downloadFilePath = localFilePath.Replace(".txt",
"DOWNLOADED.txt");

Console.WriteLine("\nDownloading blob to\n\t{0}\n", downloadFilePath);

// Download the blob's contents and save it to a file
await blobClient.DownloadToAsync(downloadFilePath);

// Clean up
Console.Write("Press any key to begin clean up");
Console.ReadLine();

Console.WriteLine("Deleting blob container...");
await containerClient.DeleteAsync();

Console.WriteLine("Deleting the local source and downloaded files...");
File.Delete(localFilePath);
File.Delete(downloadFilePath);

Console.WriteLine("Done");
```

Run the code

This app creates a test file in your local *data* folder and uploads it to Blob storage. The example then lists the blobs in the container and downloads the file with a new name so that you can compare the old and new files.

If you're using Visual Studio, press F5 to build and run the code and interact with the console app. If you're using the .NET CLI, navigate to your application directory, then build and run the application.

```
Console
```

```
dotnet build
```

```
Console
```

```
dotnet run
```

The output of the app is similar to the following example (GUID values omitted for readability):

```
Output
```

```
Azure Blob Storage - .NET quickstart sample
```

```
Uploading to Blob storage as blob:
```

```
https://mystorageacct.blob.core.windows.net/quickstartblobsGUID/quickstartGUID.txt
```

```
Listing blobs...
```

```
    quickstartGUID.txt
```

```
Downloading blob to
```

```
    ./data/quickstartGUIDDOWNLOADED.txt
```

```
Press any key to begin clean up
```

```
Deleting blob container...
```

```
Deleting the local source and downloaded files...
```

```
Done
```

Before you begin the clean-up process, check your *data* folder for the two files. You can open them and observe that they're identical.

Clean up resources

After you verify the files and finish testing, press the **Enter** key to delete the test files along with the container you created in the storage account. You can also use [Azure CLI](#) to delete resources.

Next steps

In this quickstart, you learned how to upload, download, and list blobs using .NET.

To see Blob storage sample apps, continue to:

Azure Blob Storage library for .NET samples

- To learn more, see the [Azure Blob Storage client libraries for .NET](#).
- For tutorials, samples, quick starts and other documentation, visit [Azure for .NET developers](#).
- To learn more about .NET, see [Get started with .NET in 10 minutes](#).

Quickstart: Azure Queue Storage client library for .NET

Article • 06/29/2023

Get started with the Azure Queue Storage client library for .NET. Azure Queue Storage is a service for storing large numbers of messages for later retrieval and processing. Follow these steps to install the package and try out example code for basic tasks.

[API reference documentation](#) | [Library source code](#) | [Package \(NuGet\)](#) | [Samples](#)

Use the Azure Queue Storage client library for .NET to:

- Create a queue
- Add messages to a queue
- Peek at messages in a queue
- Update a message in a queue
- Get the queue length
- Receive messages from a queue
- Delete messages from a queue
- Delete a queue

Prerequisites

- Azure subscription - [create one for free](#)
- Azure Storage account - [create a storage account](#)
- Current [.NET SDK](#) for your operating system. Be sure to get the SDK and not the runtime.

Setting up

This section walks you through preparing a project to work with the Azure Queue Storage client library for .NET.

Create the project

Create a .NET application named `QueuesQuickstart`.

1. In a console window (such as cmd, PowerShell, or Bash), use the `dotnet new` command to create a new console app with the name `QueuesQuickstart`. This

command creates a simple "hello world" C# project with a single source file named *Program.cs*.

```
Console  
dotnet new console -n QueuesQuickstart
```

2. Switch to the newly created `QueuesQuickstart` directory.

```
Console  
cd QueuesQuickstart
```

Install the packages

While still in the application directory, install the Azure Queue Storage client library for .NET package by using the `dotnet add package` command.

```
Console  
dotnet add package Azure.Storage.Queues
```

The Azure Identity client library package is also needed for passwordless connections to Azure services.

```
Console  
dotnet add package Azure.Identity
```

Set up the app framework

1. Open the project in your editor of choice
2. Open the *Program.cs* file
3. Update the existing code to match the following:

```
C#  
  
using Azure;  
using Azure.Identity;  
using Azure.Storage.Queues;  
using Azure.Storage.Queues.Models;  
using System;  
using System.Threading.Tasks;
```

```
Console.WriteLine("Azure Queue Storage client library - .NET quickstart sample");  
// Quickstart code goes here
```

Authenticate to Azure

Application requests to most Azure services must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code.

You can also authorize requests to Azure services using passwords, connection strings, or other credentials directly. However, this approach should be used with caution. Developers must be diligent to never expose these secrets in an unsecure location. Anyone who gains access to the password or secret key is able to authenticate. `DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

Passwordless (Recommended)

`DefaultAzureCredential` is a class provided by the Azure Identity client library for .NET. To learn more about `DefaultAzureCredential`, see the [DefaultAzureCredential overview](#). `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

For example, your app can authenticate using your Visual Studio sign-in credentials when developing locally, and then use a [managed identity](#) once it has been deployed to Azure. No code changes are required for this transition.

When developing locally, make sure that the user account that is accessing the queue data has the correct permissions. You'll need [Storage Queue Data Contributor](#) to read and write queue data. To assign yourself this role, you'll need to be assigned the [User Access Administrator](#) role, or another role that includes the [Microsoft.Authorization/roleAssignments/write](#) action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

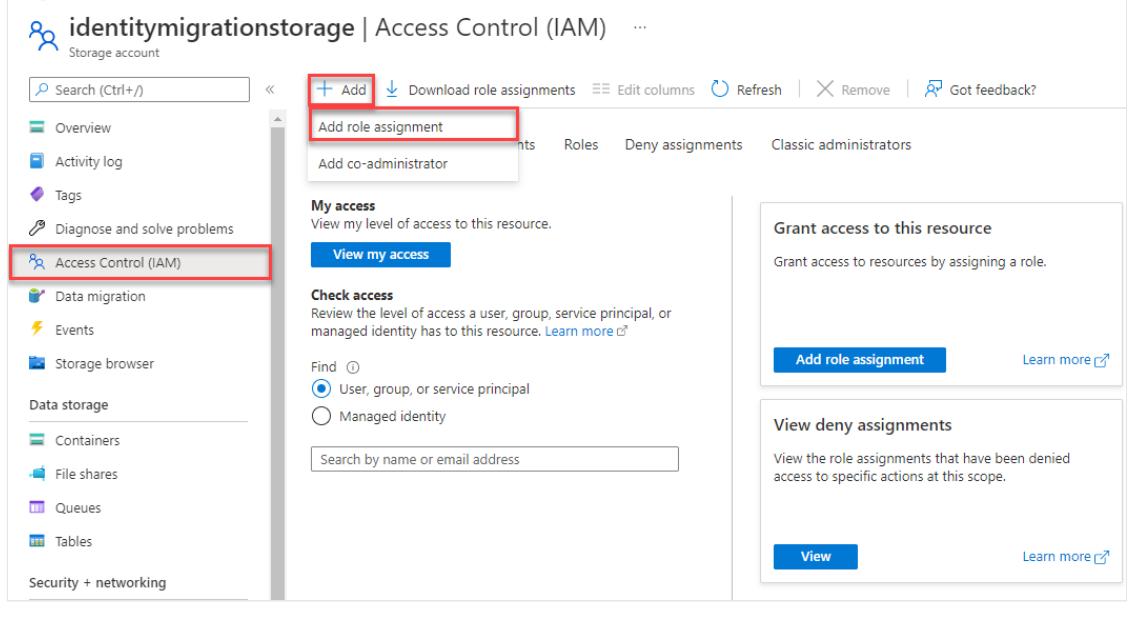
The following example will assign the **Storage Queue Data Contributor** role to your user account, which provides both read and write access to queue data in your storage account.

ⓘ Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



The screenshot shows the 'identitymigrationstorage | Access Control (IAM)' page. The left sidebar has 'Access Control (IAM)' selected. The top navigation bar includes a search bar, '+ Add', 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. A red box highlights the '+ Add' button. A dropdown menu is open over the '+ Add' button, with 'Add role assignment' also highlighted by a red box. The 'My access' and 'Check access' sections are visible on the left, and 'Grant access to this resource' and 'View deny assignments' sections are on the right.

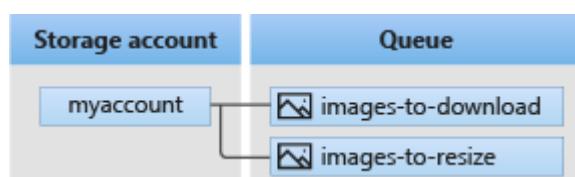
5. Use the search box to filter the results to the desired role. For this example, search for *Storage Queue Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Object model

Azure Queue Storage is a service for storing large numbers of messages. A queue message can be up to 64 KB in size. A queue may contain millions of messages, up to the total capacity limit of a storage account. Queues are commonly used to create a backlog of work to process asynchronously. Queue Storage offers three types of resources:

- **Storage account:** All access to Azure Storage is done through a storage account. For more information about storage accounts, see [Storage account overview](#)
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. A message can remain in the queue for a maximum of 7 days. For version 2017-07-29 or later, the maximum time-to-live can be any positive number, or -1 indicating that the message doesn't expire. If this parameter is omitted, the default time-to-live is seven days.

The following diagram shows the relationship between these resources.



Use the following .NET classes to interact with these resources:

- [QueueServiceClient](#): The `QueueServiceClient` allows you to manage all the queues in your storage account.
- [QueueClient](#): The `QueueClient` class allows you to manage and manipulate an individual queue and its messages.
- [QueueMessage](#): The `QueueMessage` class represents the individual objects returned when calling `ReceiveMessages` on a queue.

Code examples

These example code snippets show you how to perform the following actions with the Azure Queue Storage client library for .NET:

- [Authorize access and create a client object](#)
- [Create a queue](#)
- [Add messages to a queue](#)
- [Peek at messages in a queue](#)
- [Update a message in a queue](#)
- [Get the queue length](#)
- [Receive messages from a queue](#)
- [Delete messages from a queue](#)
- [Delete a queue](#)

Passwordless (Recommended)

Authorize access and create a client object

For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

Azure CLI

```
az login
```

Once authenticated, you can create and authorize a `QueueClient` object using `DefaultAzureCredential` to access queue data in the storage account. `DefaultAzureCredential` automatically discovers and uses the account you signed in with in the previous step.

To authorize using `DefaultAzureCredential`, make sure you've added the `Azure.Identity` package, as described in [Install the packages](#). Also, be sure to add a using directive for the `Azure.Identity` namespace in the `Program.cs` file:

```
C#
```

```
using Azure.Identity;
```

Next, decide on a name for the queue and create an instance of the `QueueClient` class, using `DefaultAzureCredential` for authorization. We use this client object to create and interact with the queue resource in the storage account.

ⓘ Important

Queue names may only contain lowercase letters, numbers, and hyphens, and must begin with a letter or a number. Each hyphen must be preceded and followed by a non-hyphen character. The name must also be between 3 and 63 characters long. For more information, see [Naming queues and metadata](#).

Add the following code to the end of the `Program.cs` file. Make sure to replace the `<storage-account-name>` placeholder value:

```
C#
```

```
// Create a unique name for the queue
// TODO: Replace the <storage-account-name> placeholder
string queueName = "quickstartqueues-" + Guid.NewGuid().ToString();
string storageAccountName = "<storage-account-name>";

// Instantiate a QueueClient to create and interact with the queue
QueueClient queueClient = new QueueClient(
    new Uri($"https://{{storageAccountName}}.queue.core.windows.net/{{queueName}}"),
    new DefaultAzureCredential());
```

ⓘ Note

Messages sent using the `QueueClient` class must be in a format that can be included in an XML request with UTF-8 encoding. You can optionally set the `MessageEncoding` option to `Base64` to handle non-compliant messages.

Create a queue

Using the `QueueClient` object, call the `CreateAsync` method to create the queue in your storage account.

Add this code to the end of the `Program.cs` method:

C#

```
Console.WriteLine($"Creating queue: {queueName}");

// Create the queue
await queueClient.CreateAsync();
```

Add messages to a queue

The following code snippet asynchronously adds messages to queue by calling the `SendMessageAsync` method. It also saves a `SendReceipt` returned from a `SendMessageAsync` call. The receipt is used to update the message later in the program.

Add this code to the end of the `Program.cs` file:

C#

```
Console.WriteLine("\nAdding messages to the queue...");

// Send several messages to the queue
await queueClient.SendMessageAsync("First message");
await queueClient.SendMessageAsync("Second message");

// Save the receipt so we can update this message later
SendReceipt receipt = await queueClient.SendMessageAsync("Third message");
```

Peek at messages in a queue

Peek at the messages in the queue by calling the `PeekMessagesAsync` method. This method retrieves one or more messages from the front of the queue but doesn't alter the visibility of the message.

Add this code to the end of the *Program.cs* file:

```
C#  
  
Console.WriteLine("\nPeek at the messages in the queue...");  
  
// Peek at messages in the queue  
PeekedMessage[] peekedMessages = await  
queueClient.PeekMessagesAsync(maxMessages: 10);  
  
foreach (PeekedMessage peekedMessage in peekedMessages)  
{  
    // Display the message  
    Console.WriteLine($"Message: {peekedMessage.MessageText}");  
}
```

Update a message in a queue

Update the contents of a message by calling the [UpdateMessageAsync](#) method. This method can change a message's visibility timeout and contents. The message content must be a UTF-8 encoded string that is up to 64 KB in size. Along with the new content for the message, pass in the values from the `SendReceipt` that was saved earlier in the code. The `SendReceipt` values identify which message to update.

```
C#  
  
Console.WriteLine("\nUpdating the third message in the queue...");  
  
// Update a message using the saved receipt from sending the message  
await queueClient.UpdateMessageAsync(receipt.MessageId, receipt.PopReceipt,  
"Third message has been updated");
```

Get the queue length

You can get an estimate of the number of messages in a queue. The [GetProperties](#) method returns queue properties including the message count. The [ApproximateMessagesCount](#) property contains the approximate number of messages in the queue. This number isn't lower than the actual number of messages in the queue, but could be higher.

Add this code to the end of the *Program.cs* file:

```
C#
```

```
QueueProperties properties = queueClient.GetProperties();

// Retrieve the cached approximate message count
int cachedMessagesCount = properties.ApproximateMessagesCount;

// Display number of messages
Console.WriteLine($"Number of messages in queue: {cachedMessagesCount}");
```

Receive messages from a queue

Download previously added messages by calling the [ReceiveMessagesAsync](#) method.

Add this code to the end of the *Program.cs* file:

C#

```
Console.WriteLine("\nReceiving messages from the queue...");

// Get messages from the queue
QueueMessage[] messages = await
queueClient.ReceiveMessagesAsync(maxMessages: 10);
```

You can optionally specify a value for `maxMessages`, which is the number of messages to retrieve from the queue. The default is 1 message and the maximum is 32 messages. You can also specify a value for `visibilityTimeout`, which hides the messages from other operations for the timeout period. The default is 30 seconds.

Delete messages from a queue

Delete messages from the queue after they've been processed. In this case, processing is just displaying the message on the console.

The app pauses for user input by calling `Console.ReadLine` before it processes and deletes the messages. Verify in your [Azure portal](#) that the resources were created correctly, before they're deleted. Any messages not explicitly deleted eventually become visible in the queue again for another chance to process them.

Add this code to the end of the *Program.cs* file:

C#

```
Console.WriteLine("\nPress Enter key to 'process' messages and delete them
from the queue...");

Console.ReadLine();
```

```
// Process and delete messages from the queue
foreach (QueueMessage message in messages)
{
    // "Process" the message
    Console.WriteLine($"Message: {message.MessageText}");

    // Let the service know we're finished with
    // the message and it can be safely deleted.
    await queueClient.DeleteMessageAsync(message.MessageId,
    message.PopReceipt);
}
```

Delete a queue

The following code cleans up the resources the app created by deleting the queue using the [DeleteAsync](#) method.

Add this code to the end of the *Program.cs* file:

C#

```
Console.WriteLine("\nPress Enter key to delete the queue...");
Console.ReadLine();

// Clean up
Console.WriteLine($"Deleting queue: {queueClient.Name}");
await queueClient.DeleteAsync();

Console.WriteLine("Done");
```

Run the code

This app creates and adds three messages to an Azure queue. The code lists the messages in the queue, then retrieves and deletes them, before finally deleting the queue.

In your console window, navigate to your application directory, then build and run the application.

Console

```
dotnet build
```

Console

```
dotnet run
```

The output of the app is similar to the following example:

Output

```
Azure Queue Storage client library - .NET quickstart sample

Creating queue: quickstartqueues-5c72da2c-30cc-4f09-b05c-a95d9da52af2

Adding messages to the queue...

Peek at the messages in the queue...
Message: First message
Message: Second message
Message: Third message

Updating the third message in the queue...

Receiving messages from the queue...

Press Enter key to 'process' messages and delete them from the queue...

Message: First message
Message: Second message
Message: Third message has been updated

Press Enter key to delete the queue...

Deleting queue: quickstartqueues-5c72da2c-30cc-4f09-b05c-a95d9da52af2
Done
```

When the app pauses before receiving messages, check your storage account in the [Azure portal](#). Verify the messages are in the queue.

Press the `Enter` key to receive and delete the messages. When prompted, press the `Enter` key again to delete the queue and finish the demo.

Next steps

In this quickstart, you learned how to create a queue and add messages to it using asynchronous .NET code. Then you learned to peek, retrieve, and delete messages. Finally, you learned how to delete a message queue.

For tutorials, samples, quick starts and other documentation, visit:

- For related code samples using deprecated .NET version 11.x SDKs, see [Code samples using .NET version 11.x](#).
- To learn more, see the [Azure Storage libraries for .NET](#).
- For more Azure Queue Storage sample apps, see [Azure Queue Storage client library for .NET samples](#).
- To learn more about .NET Core, see [Get started with .NET in 10 minutes](#).

Tutorial: Use identity-based connections instead of secrets with triggers and bindings

Article • 11/03/2022

This tutorial shows you how to configure Azure Functions to connect to Azure Service Bus queues using managed identities instead of secrets stored in the function app settings. The tutorial is a continuation of the [Create a function app without default storage secrets in its definition](#) tutorial. To learn more about identity-based connections, see [Configure an identity-based connection..](#)

While the procedures shown work generally for all languages, this tutorial currently supports C# class library functions on Windows specifically.

In this tutorial, you'll learn how to:

- ✓ Create a Service Bus namespace and queue.
- ✓ Configure your function app with managed identity
- ✓ Create a role assignment granting that identity permission to read from the Service Bus queue
- ✓ Create and deploy a function app with a Service Bus trigger.
- ✓ Verify your identity-based connection to Service Bus

Prerequisite

Complete the previous tutorial: [Create a function app with identity-based connections.](#)

Create a service bus and queue

1. In the [Azure portal](#), choose **Create a resource (+)**.
2. On the **Create a resource** page, select **Integration > Service Bus**.
3. On the **Basics** page, use the following table to configure the Service Bus namespace settings. Use the default values for the remaining options.

Option	Suggested value	Description
Subscription	Your subscription	The subscription under which your resources are created.
Resource group	myResourceGroup	The resource group you created with your function app.
Namespace name	Globally unique name	The namespace of your instance from which to trigger your function. Because the namespace is publicly accessible, you must use a name that is globally unique across Azure. The name must also be between 6 and 50 characters in length, contain only alphanumeric characters and dashes, and can't start with a number.
Location	myFunctionRegion	The region where you created your function app.
Pricing tier	Basic	The basic Service Bus tier.

4. Select **Review + create**. After validation finishes, select **Create**.

5. After deployment completes, select **Go to resource**.
6. In your new Service Bus namespace, select **+ Queue** to add a queue.
7. Type `myinputqueue` as the new queue's name and select **Create**.

Now, that you have a queue, you will add a role assignment to the managed identity of your function app.

Configure your Service Bus trigger with a managed identity

To use Service Bus triggers with identity-based connections, you will need to add the **Azure Service Bus Data Receiver** role assignment to the managed identity in your function app. This role is required when using managed identities to trigger off of your service bus namespace. You can also add your own account to this role, which makes it possible to connect to the service bus namespace during local testing.

ⓘ Note

Role requirements for using identity-based connections vary depending on the service and how you are connecting to it. Needs vary across triggers, input bindings, and output bindings. For more details on specific role requirements, please refer to the trigger and binding documentation for the service.

1. In your service bus namespace that you just created, select **Access Control (IAM)**. This is where you can view and configure who has access to the resource.
2. Click **Add** and select **add role assignment**.
3. Search for **Azure Service Bus Data Receiver**, select it, and click **Next**.
4. On the **Members** tab, under **Assign access to**, choose **Managed Identity**
5. Click **Select members** to open the **Select managed identities** panel.
6. Confirm that the **Subscription** is the one in which you created the resources earlier.
7. In the **Managed identity** selector, choose **Function App** from the **System-assigned managed identity** category. The label "Function App" may have a number in parentheses next to it, indicating the number of apps in the subscription with system-assigned identities.
8. Your app should appear in a list below the input fields. If you don't see it, you can use the **Select** box to filter the results with your app's name.
9. Click on your application. It should move down into the **Selected members** section. Click **Select**.
10. Back on the **Add role assignment** screen, click **Review + assign**. Review the configuration, and then click **Review + assign**.

You've granted your function app access to the service bus namespace using managed identities.

Connect to Service Bus in your function app

1. In the portal, search for the function app you created in the [previous tutorial](#), or browse to it in the [Function App](#) page.
2. In your function app, select **Configuration** under **Settings**.
3. In **Application settings**, select **+ New application setting** to create the new setting in the following table.

Name	Value	Description
ServiceBusConnection_fullyQualifiedNamespace	<SERVICE_BUS_NAMESPACE>.servicebus.windows.net	This setting connects your function app to the Service Bus using an identity-based connection instead of secrets.

4. After you create the two settings, select **Save > Confirm**.

! **Note**

When using **Azure App Configuration** or **Key Vault** to provide settings for Managed Identity connections, setting names should use a valid key separator such as `:` or `/` in place of the `_` to ensure names are resolved correctly.

For example, `ServiceBusConnection:fullyQualifiedNamespace`.

Now that you've prepared the function app to connect to the service bus namespace using a managed identity, you can add a new function that uses a Service Bus trigger to your local project.

Add a Service Bus triggered function

1. Run the `func init` command, as follows, to create a functions project in a folder named `LocalFunctionProj` with the specified runtime:

```
C#
func init LocalFunctionProj --dotnet
```

2. Navigate into the project folder:

```
Console
cd LocalFunctionProj
```

3. In the root project folder, run the following commands:

```
command
```

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.ServiceBus --version 5.2.0
```

This replaces the default version of the Service Bus extension package with a version that supports managed identities.

4. Run the following command to add a Service Bus triggered function to the project:

```
C#
```

```
func new --name ServiceBusTrigger --template ServiceBusQueueTrigger
```

This adds the code for a new Service Bus trigger and a reference to the extension package. You need to add a service bus namespace connection setting for this trigger.

5. Open the new ServiceBusTrigger.cs project file and replace the `ServiceBusTrigger` class with the following code:

```
C#
```

```
public static class ServiceBusTrigger
{
    [FunctionName("ServiceBusTrigger")]
    public static void Run([ServiceBusTrigger("myinputqueue",
        Connection = "ServiceBusConnection")]string myQueueItem, ILogger log)
    {
        log.LogInformation($"C# ServiceBus queue trigger function processed message:
{myQueueItem}");
    }
}
```

This code sample updates the queue name to `myinputqueue`, which is the same name as you queue you created earlier. It also sets the name of the Service Bus connection to `ServiceBusConnection`. This is the Service Bus namespace used by the identity-based connection `ServiceBusConnection__fullyQualifiedNamespace` you configured in the portal.

ⓘ Note

If you try to run your functions now using `func start` you'll receive an error. This is because you don't have an identity-based connection defined locally. If you want to run your function locally, set the app setting `ServiceBusConnection__fullyQualifiedNamespace` in `local.settings.json` as you did in [the previous section](#). In addition, you'll need to assign the role to your developer identity. For more details, please refer to the [local development with identity-based connections documentation](#).

ⓘ Note

When using [Azure App Configuration](#) or [Key Vault](#) to provide settings for Managed Identity connections, setting names should use a valid key separator such as `:` or `/` in place of the `_` to ensure names are resolved correctly.

For example, `ServiceBusConnection:fullyQualifiedNamespace`.

Publish the updated project

1. Run the following command to locally generate the files needed for the deployment package:

Console

```
dotnet publish --configuration Release
```

2. Browse to the `\bin\Release\netcoreapp3.1\publish` subfolder and create a .zip file from its contents.

3. Publish the .zip file by running the following command, replacing the `FUNCTION_APP_NAME`, `RESOURCE_GROUP_NAME`, and `PATH_TO_ZIP` parameters as appropriate:

Azure CLI

```
az functionapp deploy -n FUNCTION_APP_NAME -g RESOURCE_GROUP_NAME --src-path PATH_TO_ZIP
```

Now that you have updated the function app with the new trigger, you can verify that it works using the identity.

Validate your changes

1. In the portal, search for `Application Insights` and select **Application Insights** under Services.
2. In **Application Insights**, browse or search for your named instance.
3. In your instance, select **Live Metrics** under **Investigate**.
4. Keep the previous tab open, and open the Azure portal in a new tab. In your new tab, navigate to your Service Bus namespace, select **Queues** from the left blade.
5. Select your queue named `myinputqueue`.
6. Select **Service Bus Explorer** from the left blade.
7. Send a test message.
8. Select your open **Live Metrics** tab and see the Service Bus queue execution.

Congratulations! You have successfully set up your Service Bus queue trigger with a managed identity!

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, you can delete them by deleting the resource group.

From the Azure portal menu or **Home** page, select **Resource groups**. Then, on the **Resource groups** page, select `myResourceGroup`.

On the **myResourceGroup** page, make sure that the listed resources are the ones you want to delete.

Select **Delete resource group**, type **myResourceGroup** in the text box to confirm, and then select **Delete**.

Next steps

In this tutorial, you created a function app with identity-based connections.

Use the following links to learn more Azure Functions with identity-based connections:

- [Managed identity in Azure Functions](#)
- [Identity-based connections in Azure Functions](#)
- [Functions documentation for local development](#)

Tutorial: Use a managed identity to connect Key Vault to an Azure web app in .NET

Article • 02/20/2024

Azure Key Vault provides a way to store credentials and other secrets with increased security. But your code needs to authenticate to Key Vault to retrieve them. Managed identities for Azure resources help to solve this problem by giving Azure services an automatically managed identity in Microsoft Entra ID. You can use this identity to authenticate to any service that supports Microsoft Entra authentication, including Key Vault, without having to display credentials in your code.

In this tutorial, you'll create and deploy Azure web application to [Azure App Service](#). You'll use a managed identity to authenticate your Azure web app with an Azure key vault using [Azure Key Vault secret client library for .NET](#) and the [Azure CLI](#). The same basic principles apply when you use the development language of your choice, Azure PowerShell, and/or the Azure portal.

For more information about Azure App service web applications and deployment presented in this tutorial, see:

- [App Service overview](#)
- [Create an ASP.NET Core web app in Azure App Service](#)
- [Local Git deployment to Azure App Service](#)

Prerequisites

To complete this tutorial, you need:

- An Azure subscription. [Create one for free.](#)
- The [.NET Core 3.1 SDK \(or later\)](#).
- A [Git](#) installation of version 2.28.0 or greater.
- The [Azure CLI](#) or [Azure PowerShell](#).
- [Azure Key Vault](#). You can create a key vault by using the [Azure portal](#), the [Azure CLI](#), or [Azure PowerShell](#).
- A Key Vault [secret](#). You can create a secret by using the [Azure portal](#), [PowerShell](#), or the [Azure CLI](#).

If you already have your web application deployed in Azure App Service, you can skip to [configure web app access to a key vault](#) and [modify web application code](#) sections.

Create a .NET Core app

In this step, you'll set up the local .NET Core project.

In a terminal window on your machine, create a directory named `akvwebapp` and make it the current directory:

```
Bash
```

```
mkdir akvwebapp  
cd akvwebapp
```

Create a .NET Core app by using the [dotnet new web](#) command:

```
Bash
```

```
dotnet new web
```

Run the application locally so you know how it should look when you deploy it to Azure:

```
Bash
```

```
dotnet run
```

In a web browser, go to the app at `http://localhost:5000`.

You'll see the "Hello World!" message from the sample app displayed on the page.

For more information about creating web applications for Azure, see [Create an ASP.NET Core web app in Azure App Service](#)

Deploy the app to Azure

In this step, you'll deploy your .NET Core application to Azure App Service by using local Git. For more information on how to create and deploy applications, see [Create an ASP.NET Core web app in Azure](#).

Configure the local Git deployment

In the terminal window, select **Ctrl+C** to close the web server. Initialize a Git repository for the .NET Core project:

```
Bash
```

```
git init --initial-branch=main  
git add .  
git commit -m "first commit"
```

You can use FTP and local Git to deploy an Azure web app by using a *deployment user*. After you configure your deployment user, you can use it for all your Azure deployments. Your account-level deployment user name and password are different from your Azure subscription credentials.

To configure the deployment user, run the [az webapp deployment user set](#) command. Choose a user name and password that adheres to these guidelines:

- The user name must be unique within Azure. For local Git pushes, it can't contain the at sign symbol (@).
- The password must be at least eight characters long and contain two of the following three elements: letters, numbers, and symbols.

Azure CLI

```
az webapp deployment user set --user-name "<username>" --password "<password>"
```

The JSON output shows the password as `null`. If you get a `'Conflict'. Details: 409` error, change the user name. If you get a `'Bad Request'. Details: 400` error, use a stronger password.

Record your user name and password so you can use it to deploy your web apps.

Create a resource group

A resource group is a logical container into which you deploy Azure resources and manage them. Create a resource group to contain both your key vault and your web app by using the [az group create](#) command:

Azure CLI

```
az group create --name "myResourceGroup" -l "EastUS"
```

Create an App Service plan

Create an [App Service plan](#) by using the Azure CLI [az appservice plan create](#) command. This following example creates an App Service plan named `myAppServicePlan` in the

FREE pricing tier:

Azure CLI

```
az appservice plan create --name myAppServicePlan --resource-group  
myResourceGroup --sku FREE
```

When the App Service plan is created, the Azure CLI displays information similar to what you see here:

```
{  
    "adminSiteName": null,  
    "appServicePlanName": "myAppServicePlan",  
    "geoRegion": "West Europe",  
    "hostingEnvironmentProfile": null,  
    "id": "/subscriptions/0000-  
0000/resourceGroups/myResourceGroup/providers/Microsoft.Web/serverfarms/myAp-  
pServicePlan",  
    "kind": "app",  
    "location": "West Europe",  
    "maximumNumberOfWorkers": 1,  
    "name": "myAppServicePlan",  
    < JSON data removed for brevity. >  
    "targetWorkerSizeId": 0,  
    "type": "Microsoft.Web/serverfarms",  
    "workerTierName": null  
}
```

For more information, see [Manage an App Service plan in Azure](#).

Create a web app

Create an [Azure web app](#) in the `myAppServicePlan` App Service plan.

Important

Like a key vault, an Azure web app must have a unique name. Replace `<your-webapp-name>` with the name of your web app in the following examples.

Azure CLI

```
az webapp create --resource-group "myResourceGroup" --plan  
"myAppServicePlan" --name "<your-webapp-name>" --deployment-local-git
```

When the web app is created, the Azure CLI shows output similar to what you see here:

```
Local git is configured with url of 'https://<username>@<your-webapp-name>.scm.azurewebsites.net/<your-webapp-name>.git'
{
  "availabilityState": "Normal",
  "clientAffinityEnabled": true,
  "clientCertEnabled": false,
  "clientCertExclusionPaths": null,
  "cloningInfo": null,
  "containerSize": 0,
  "dailyMemoryTimeQuota": 0,
  "defaultHostName": "<your-webapp-name>.azurewebsites.net",
  "deploymentLocalGitUrl": "https://<username>@<your-webapp-name>.scm.azurewebsites.net/<your-webapp-name>.git",
  "enabled": true,
  < JSON data removed for brevity. >
}
```

The URL of the Git remote is shown in the `deploymentLocalGitUrl` property, in the format `https://<username>@<your-webapp-name>.scm.azurewebsites.net/<your-webapp-name>.git`. Save this URL. You'll need it later.

Now configure your web app to deploy from the `main` branch:

Azure CLI

```
az webapp config appsettings set -g MyResourceGroup --name "<your-webapp-name>" --settings deployment_branch=main
```

Go to your new app by using the following command. Replace `<your-webapp-name>` with your app name.

Bash

```
https://<your-webapp-name>.azurewebsites.net
```

You'll see the default webpage for a new Azure web app.

Deploy your local app

Back in the local terminal window, add an Azure remote to your local Git repository. In the following command, replace `<deploymentLocalGitUrl-from-create-step>` with the URL of the Git remote that you saved in the [Create a web app](#) section.

Bash

```
git remote add azure <deploymentLocalGitUrl-from-create-step>
```

Use the following command to push to the Azure remote to deploy your app. When Git Credential Manager prompts you for credentials, use the credentials you created in the [Configure the local Git deployment](#) section.

Bash

```
git push azure main
```

This command might take a few minutes to run. While it runs, it displays information similar to what you see here:

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 285 bytes | 95.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Deploy Async
remote: Updating branch 'main'.
remote: Updating submodules.
remote: Preparing deployment for commit id 'd6b54472f7'.
remote: Repository path is /home/site/repository
remote: Running oryx build...
remote: Build orchestrated by Microsoft Oryx,
https://github.com/Microsoft/Oryx
remote: You can report issues at https://github.com/Microsoft/Oryx/issues
remote:
remote: Oryx Version      : 0.2.20200114.13, Commit:
204922f30f8e8d41f5241b8c218425ef89106d1d, ReleaseTagName: 20200114.13
remote: Build Operation ID: |imoMY2y77/s=.40ca2a87_
remote: Repository Commit : d6b54472f7e8e9fd885ffafaa64522e74cf370e1
.
.
.
remote: Deployment successful.
remote: Deployment Logs : 'https://<your-webapp-
name>.scm.azurewebsites.net/newui/jsonviewer?
view_url=/api/deployments/d6b54472f7e8e9fd885ffafaa64522e74cf370e1/log'
To https://<your-webapp-name>.scm.azurewebsites.net:443/<your-webapp-
name>.git
d87e6ca..d6b5447  main -> main
```

Go to (or refresh) the deployed application by using your web browser:

Bash

```
http://<your-webapp-name>.azurewebsites.net
```

You'll see the "Hello World!" message you saw earlier when you visited

```
http://localhost:5000.
```

For more information about deploying web application using Git, see [Local Git deployment to Azure App Service](#)

Configure the web app to connect to Key Vault

In this section, you'll configure web access to Key Vault and update your application code to retrieve a secret from Key Vault.

Create and assign a managed identity

In this tutorial, we'll use [managed identity](#) to authenticate to Key Vault. Managed identity automatically manages application credentials.

In the Azure CLI, to create the identity for the application, run the [az webapp-identity assign](#) command:

```
Azure CLI
```

```
az webapp identity assign --name "<your-webapp-name>" --resource-group  
"myResourceGroup"
```

The command will return this JSON snippet:

```
JSON
```

```
{  
  "principalId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",  
  "tenantId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",  
  "type": "SystemAssigned"  
}
```

To give your web app permission to do [get](#) and [list](#) operations on your key vault, pass the `principalId` to the Azure CLI [az keyvault set-policy](#) command:

```
Azure CLI
```

```
az keyvault set-policy --name "<your-keyvault-name>" --object-id "
```

```
<principalId>" --secret-permissions get list
```

You can also assign access policies by using the [Azure portal](#) or [PowerShell](#).

Modify the app to access your key vault

In this tutorial, you'll use [Azure Key Vault secret client library](#) for demonstration purposes. You can also use [Azure Key Vault certificate client library](#), or [Azure Key Vault key client library](#).

Install the packages

From the terminal window, install the Azure Key Vault secret client library for .NET and Azure Identity client library packages:

Console

```
dotnet add package Azure.Identity  
dotnet add package Azure.Security.KeyVault.Secrets
```

Update the code

Find and open the Startup.cs file for .NET 5.0 or earlier, or Program.cs file for .NET 6.0 in your akvwebapp project.

Add these lines to the header:

C#

```
using Azure.Identity;  
using Azure.Security.KeyVault.Secrets;  
using Azure.Core;
```

Add the following lines before the `app.UseEndpoints` call (.NET 5.0 or earlier) or `app.MapGet` call (.NET 6.0), updating the URI to reflect the `vaultUri` of your key vault. This code uses [DefaultAzureCredential\(\)](#) to authenticate to Key Vault, which uses a token from managed identity to authenticate. For more information about authenticating to Key Vault, see the [Developer's Guide](#). The code also uses exponential backoff for retries in case Key Vault is being throttled. For more information about Key Vault transaction limits, see [Azure Key Vault throttling guidance](#).

C#

```
SecretClientOptions options = new SecretClientOptions()
{
    Retry =
    {
        Delay= TimeSpan.FromSeconds(2),
        MaxDelay = TimeSpan.FromSeconds(16),
        MaxRetries = 5,
        Mode = RetryMode.Exponential
    }
};

var client = new SecretClient(new Uri("https://<your-unique-key-vault-name>.vault.azure.net/"), new DefaultAzureCredential(),options);

KeyVaultSecret secret = client.GetSecret("<mySecret>");

string secretValue = secret.Value;
```

.NET 5.0 or earlier

Update the line `await context.Response.WriteAsync("Hello World!");` to look like this line:

C#

```
await context.Response.WriteAsync(secretValue);
```

.NET 6.0

Update the line `app.MapGet("/", () => "Hello World!");` to look like this line:

C#

```
app.MapGet("/", () => secretValue);
```

Be sure to save your changes before continuing to the next step.

Redeploy your web app

Now that you've updated your code, you can redeploy it to Azure by using these Git commands:

Bash

```
git add .
git commit -m "Updated web app to access my key vault"
git push azure main
```

Go to your completed web app

Bash

```
http://<your-webapp-name>.azurewebsites.net
```

Where before you saw "Hello World!", you should now see the value of your secret displayed.

Next steps

- [Use Azure Key Vault with applications deployed to a virtual machine in .NET](#)
- Learn more about [managed identities for Azure resources](#)
- View the [Developer's Guide](#)
- [Secure access to a key vault](#)

Use Azure OpenAI without keys

Article • 05/20/2024

Application requests to most Azure services must be authenticated with keys or [passwordless connections](#). Developers must be diligent to never expose the keys in an unsecure location. Anyone who gains access to the key is able to authenticate to the service. Passwordless authentication offers improved management and security benefits over the account key because there's no key (or connection string) to store.

Passwordless connections are enabled with the following steps:

- Configure your authentication.
- Set environment variables, as needed.
- Use an Azure Identity library credential type to create an Azure OpenAI client object.

Authentication

Authentication to Microsoft Entra ID is required to use the Azure client libraries.

Authentication differs based on the environment in which the app is running:

- [Local development](#)
- [Azure](#)

Authenticate for local development

.NET

Select a tool for [authentication during local development](#).

Authenticate for Azure-hosted environments

.NET

Learn about how to manage the [DefaultAzureCredential](#) for applications deployed to Azure.

Configure roles for authorization

1. Find the [role](#) for your usage of Azure OpenAI. Depending on how you intend to set that role, you'll need either the name or ID.

[+] [Expand table](#)

Role name	Role ID
For Azure CLI or Azure PowerShell, you can use role name. For Bicep, you need the role ID.	

2. For many Azure OpenAI chat completion use cases, the following role and ID are suggested.

[+] [Expand table](#)

Role name	Role ID
Cognitive Services OpenAI User	5e0bd9bd-7b93-4f28-af87-19fc36ad61bd

3. Select an identity type to use.

- **Personal identity:** This is your personal identity tied to your sign in to Azure.
- **Managed identity:** This is an identity managed by and created for use on Azure. For [managed identity](#), create a [user-assigned managed identity](#). When you create the managed identity, you need the `Client ID`, also known as the `app ID`.

4. To find your personal identity, use one of the following commands. Use the ID as the `<identity-id>` in the next step.

Azure CLI

For local development, to get your own identity ID, use the following command. You need to sign in with `az login` before using this command.

Azure CLI

```
az ad signed-in-user show \
--query id -o tsv
```

5. Assign the role-based access control (RBAC) role to the identity for the resource group.

Azure CLI

To grant your identity permissions to your resource through RBAC, assign a role using the Azure CLI command [az role assignment create](#).

Azure CLI

```
az role assignment create \
    --role "Cognitive Services OpenAI User" \
    --assignee "<identity-id>" \
    --scope "/subscriptions/<subscription-
    id>/resourceGroups/<resource-group-name>"
```

Where applicable, replace `<identity-id>`, `<subscription-id>`, and `<resource-group-name>` with your actual values.

Configure environment variables

To connect to Azure OpenAI, your code needs to know your resource endpoint, and *may* need additional environment variables.

1. Create an environment variable for your Azure OpenAI endpoint.

- `AZURE_OPENAI_ENDPOINT`: This URL is the access point for your Azure OpenAI resource.

2. Create environment variables based on the location in which your app runs:

[+] [Expand table](#)

Location	Identity	Description
Local	Personal	For local runtimes with your personal identity , sign in to create your credential with a tool.
Azure cloud	User-assigned managed identity	Create an <code>AZURE_CLIENT_ID</code> environment variable containing the client ID of the user-assigned managed identity to authenticate as.

Install Azure Identity client library

.NET

Install the .NET Azure Identity client library [↗](#):

.NET CLI

```
dotnet add package Azure.Identity
```

Use DefaultAzureCredential

The Azure Identity library's `DefaultAzureCredential` allows the customer to run the same code in the local development environment and in the Azure Cloud.

.NET

For more information on `DefaultAzureCredential` for .NET, see [Azure Identity client library for .NET](#).

C#

```
using Azure;
using Azure.AI.OpenAI;
using Azure.Identity;
using System;
using static System.Environment;

string endpoint = GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT");

OpenAIClient client = new(new Uri(endpoint), new
DefaultAzureCredential());
```

Additional resources

- [Passwordless connections developer guide](#)

Feedback

Was this page helpful?

Yes

No

Get help at [Microsoft Q&A](#)

How to use managed identities for App Service and Azure Functions

Article • 10/12/2023

This article shows you how to create a managed identity for App Service and Azure Functions applications and how to use it to access other resources.

ⓘ Important

Because [managed identities don't support cross-directory scenarios](#), they won't behave as expected if your app is migrated across subscriptions or tenants. To recreate the managed identities after such a move, see [Will managed identities be recreated automatically if I move a subscription to another directory?](#).

Downstream resources also need to have access policies updated to use the new identity.

ⓘ Note

Managed identities are not available for apps deployed in Azure Arc.

A managed identity from Microsoft Entra ID allows your app to easily access other Microsoft Entra protected resources such as Azure Key Vault. The identity is managed by the Azure platform and does not require you to provision or rotate any secrets. For more about managed identities in Microsoft Entra ID, see [Managed identities for Azure resources](#).

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to your application and is deleted if your 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 app. An app can have multiple user-assigned identities.

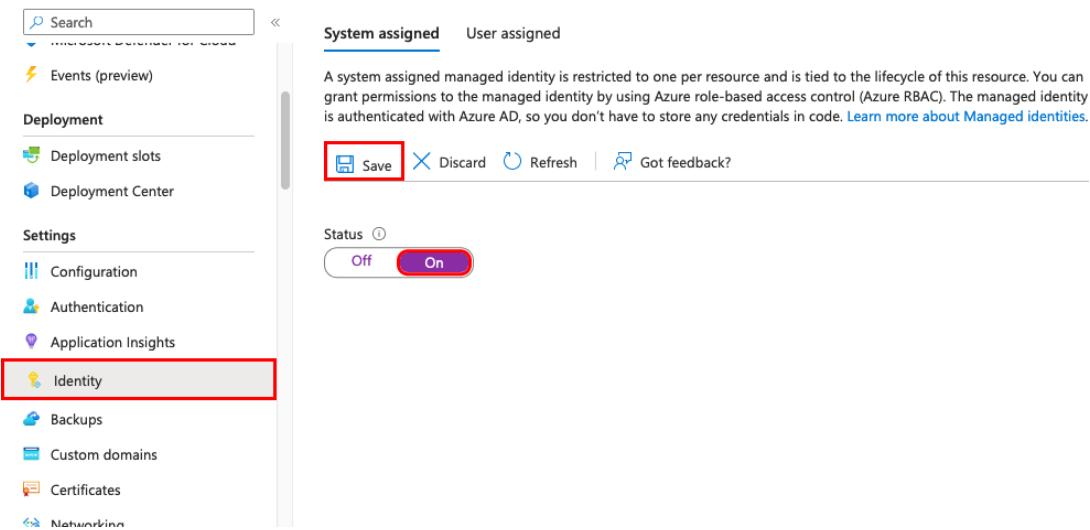
The managed identity configuration is specific to the slot. To configure a managed identity for a deployment slot in the portal, navigate to the slot first. To find the managed identity for your web app or deployment slot in your Microsoft Entra tenant from the Azure portal, search for it directly from the **Overview** page of your tenant.

Usually, the slot name is similar to `<app-name>/slots/<slot-name>`.

Add a system-assigned identity

Azure portal

1. In the left navigation of your app's page, scroll down to the **Settings** group.
2. Select **Identity**.
3. Within the **System assigned** tab, switch **Status** to **On**. Click **Save**.



Add a user-assigned identity

Creating an app with a user-assigned identity requires that you create the identity and then add its resource identifier to your app config.

Azure portal

First, you'll need to create a user-assigned identity resource.

1. Create a user-assigned managed identity resource according to [these instructions](#).
2. In the left navigation for your app's page, scroll down to the **Settings** group.
3. Select **Identity**.
4. Select **User assigned > Add**.
5. Search for the identity you created earlier, select it, and select **Add**.

The screenshot shows two side-by-side windows from the Azure portal. The left window is titled 'my-demo-app | Identity' and shows the 'Identity' section selected in the sidebar. The right window is titled 'Add user assigned managed i...' and shows the configuration of a user assigned identity. In the 'User assigned managed identities' list, 'test' is selected. Below it, 'test-identity' is listed under 'Selected identities'. A red box highlights the 'Add' button at the bottom of the list.

Once you select **Add**, the app restarts.

Configure target resource

You may need to configure the target resource to allow access from your app or function. For example, if you [request a token](#) to access Key Vault, you must also add an access policy that includes the managed identity of your app or function. Otherwise, your calls to Key Vault will be rejected, even if you use a valid token. The same is true for Azure SQL Database. To learn more about which resources support Microsoft Entra tokens, see [Azure services that support Microsoft Entra 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 its managed identity, an app can obtain tokens for Azure resources that are protected by Microsoft Entra ID, 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.

App Service and Azure Functions provide an internally accessible [REST endpoint](#) for token retrieval. 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 and simplifies the development experience. Connecting to other Azure services is as simple as adding a credential object to the service-specific client.

HTTP GET

A raw HTTP GET request looks like the following example:

HTTP

```
GET /MSI/token?resource=https://vault.azure.net&api-version=2019-08-01
HTTP/1.1
Host: localhost:4141
X-IDENTITY-HEADER: 853b9a84-5bfa-4b22-a3f3-0b9a43d9ad8a
```

And a sample response might look like the following:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "access_token": "eyJ0eXAi...",
    "expires_on": "1586984735",
    "resource": "https://vault.azure.net",
    "token_type": "Bearer",
    "client_id": "5E29463D-71DA-4FE0-8E69-999B57DB23B0"
}
```

This response is the same as the [response for the Microsoft Entra service-to-service access token request](#). To access Key Vault, you will then add the value of `access_token` to a client connection with the vault.

For more information on the REST endpoint, see [REST endpoint reference](#).

Remove an identity

When you remove a system-assigned identity, it's deleted from Microsoft Entra ID. System-assigned identities are also automatically removed from Microsoft Entra ID

when you delete the app resource itself.

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**. Click **Save**.
 - **User-assigned identity:** Select the **User assigned** tab, select the checkbox for the identity, and select **Remove**. Select **Yes** to confirm.

ⓘ Note

There is also an application setting that can be set, WEBSITE_DISABLE_MSI, which just disables the local token service. However, it leaves the identity in place, and tooling will still show the managed identity as "on" or "enabled." As a result, use of this setting is not recommended.

REST endpoint reference

An app with a managed identity makes this endpoint available by defining two environment variables:

- **IDENTITY_ENDPOINT** - the URL to the local token service.
- **IDENTITY_HEADER** - a header used to help mitigate server-side request forgery (SSRF) attacks. The value is rotated by the platform.

The **IDENTITY_ENDPOINT** is a local URL from which your app can request tokens. To get a token for a resource, make an HTTP GET request to this endpoint, including the following parameters:

Parameter	In	Description
name		
resource	Query	The Microsoft Entra resource URI of the resource for which a token should be obtained. This could be one of the Azure services that support Microsoft Entra authentication or any other resource URI.
api-version	Query	The version of the token API to be used. Use <code>2019-08-01</code> .

Parameter	In	Description
<code>name</code>		
X-IDENTITY-HEADER	Header	The value of the IDENTITY_HEADER environment variable. This header is used to help mitigate server-side request forgery (SSRF) attacks.
<code>client_id</code>	Query	(Optional) The client ID of the user-assigned identity to be used. Cannot be used on a request that includes <code>principal_id</code> , <code>mi_res_id</code> , or <code>object_id</code> . If all ID parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.
<code>principal_id</code>	Query	(Optional) The principal ID of the user-assigned identity to be used. <code>object_id</code> is an alias that may be used instead. Cannot be used on a request that includes <code>client_id</code> , <code>mi_res_id</code> , or <code>object_id</code> . If all ID parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.
<code>mi_res_id</code>	Query	(Optional) The Azure resource ID of the user-assigned identity to be used. Cannot be used on a request that includes <code>principal_id</code> , <code>client_id</code> , or <code>object_id</code> . If all ID parameters (<code>client_id</code> , <code>principal_id</code> , <code>object_id</code> , and <code>mi_res_id</code>) are omitted, the system-assigned identity is used.

ⓘ Important

If you are attempting to obtain tokens for user-assigned identities, you must include one of the optional properties. Otherwise the token service will attempt to obtain a token for a system-assigned identity, which may or may not exist.

Next steps

- [Tutorial: Connect to SQL Database from App Service without secrets using a managed identity](#)
- [Access Azure Storage securely using a managed identity](#)
- [Call Microsoft Graph securely using a managed identity](#)
- [Connect securely to services with Key Vault secrets](#)

Assign an Azure role for access to blob data

Article • 02/16/2024

Microsoft Entra authorizes access rights to secured resources through [Azure role-based access control \(Azure RBAC\)](#). Azure Storage defines a set of Azure built-in roles that encompass common sets of permissions used to access blob data.

When an Azure role is assigned to a Microsoft Entra security principal, Azure grants access to those resources for that security principal. A Microsoft Entra security principal can be a user, a group, an application service principal, or a [managed identity for Azure resources](#).

To learn more about using Microsoft Entra ID to authorize access to blob data, see [Authorize access to blobs using Microsoft Entra ID](#).

ⓘ Note

This article shows how to assign an Azure role for access to blob data in a storage account. To learn about assigning roles for management operations in Azure Storage, see [Use the Azure Storage resource provider to access management resources](#).

Assign an Azure role

You can use the Azure portal, PowerShell, Azure CLI, or an Azure Resource Manager template to assign a role for data access.

Azure portal

To access blob data in the Azure portal with Microsoft Entra credentials, a user must have the following role assignments:

- A data access role, such as **Storage Blob Data Reader** or **Storage Blob Data Contributor**
- The Azure Resource Manager **Reader** role, at a minimum

To learn how to assign these roles to a user, follow the instructions provided in [Assign Azure roles using the Azure portal](#).

The **Reader** role is an Azure Resource Manager role that permits users to view storage account resources, but not modify them. It doesn't provide read permissions to data in Azure Storage, but only to account management resources. The **Reader** role is necessary so that users can navigate to blob containers in the Azure portal.

For example, if you assign the **Storage Blob Data Contributor** role to user Mary at the level of a container named **sample-container**, then Mary is granted read, write, and delete access to all of the blobs in that container. However, if Mary wants to view a blob in the Azure portal, then the **Storage Blob Data Contributor** role by itself won't provide sufficient permissions to navigate through the portal to the blob in order to view it. The additional permissions are required to navigate through the portal and view the other resources that are visible there.

A user must be assigned the **Reader** role to use the Azure portal with Microsoft Entra credentials. However, if a user is assigned a role with **Microsoft.Storage/storageAccounts/listKeys/action** permissions, then the user can use the portal with the storage account keys, via Shared Key authorization. To use the storage account keys, Shared Key access must be permitted for the storage account. For more information on permitting or disallowing Shared Key access, see [Prevent Shared Key authorization for an Azure Storage account](#).

You can also assign an Azure Resource Manager role that provides additional permissions beyond the **Reader** role. Assigning the least possible permissions is recommended as a security best practice. For more information, see [Best practices for Azure RBAC](#).

Note

Prior to assigning yourself a role for data access, you will be able to access data in your storage account via the Azure portal because the Azure portal can also use the account key for data access. For more information, see [Choose how to authorize access to blob data in the Azure portal](#).

Keep in mind the following points about Azure role assignments in Azure Storage:

- When you create an Azure Storage account, you aren't automatically assigned permissions to access data via Microsoft Entra ID. You must explicitly assign yourself an Azure role for Azure Storage. You can assign it at the level of your subscription, resource group, storage account, or container.

- If the storage account is locked with an Azure Resource Manager read-only lock, then the lock prevents the assignment of Azure roles that are scoped to the storage account or a container.
- If you set the appropriate allow permissions to access data via Microsoft Entra ID and are unable to access the data, for example you're getting an "AuthorizationPermissionMismatch" error. Be sure to allow enough time for the permissions changes you made in Microsoft Entra ID to replicate, and be sure that you don't have any deny assignments that block your access, see [Understand Azure deny assignments](#).

 **Note**

You can create custom Azure RBAC roles for granular access to blob data. For more information, see [Azure custom roles](#).

Next steps

- [What is Azure role-based access control \(Azure RBAC\)?](#)
- [Best practices for Azure RBAC](#)

Managed identities in Azure Container Apps

Article • 10/25/2023

A managed identity from Microsoft Entra ID allows your container app to access other Microsoft Entra protected resources. For more about managed identities in Microsoft Entra ID, 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 Microsoft Entra 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.

[Init containers](#) can't access managed identities.

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 navigation options: Overview, Access control (IAM), Tags, Diagnose and solve problems, Settings, Secrets, Ingress, Continuous deployment, Identity (which is highlighted in grey), and Locks. The main content area is titled 'System assigned' and contains a brief description of system-assigned managed identities. At the bottom of this section is a 'Status' toggle switch, which is currently set to 'On' (indicated by a purple button). A red box is drawn around this switch. Below the status section 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 Microsoft 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 managed identities' with one row: 'No results'. On the right, a modal window titled 'Add user assigned managed identity...' is open. It has a dropdown for 'Subscription' set to 'Demo-Subscription'. Below it is a list of identities: 'user-identity-1', 'user-identity-2', and 'user-identity3'. The identity 'music-store-user-identity' is selected and highlighted with a red box. At the bottom of the modal is a blue 'Add' button.

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 Microsoft Entra tokens, see [Azure services that support Microsoft Entra 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 Microsoft Entra ID, 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 Microsoft Entra ID. System-assigned identities are also automatically removed from Microsoft Entra ID when you delete the container app resource itself. Removing user-assigned managed identities from your container app doesn't remove them from Microsoft Entra ID.

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](#)

Configure role-based access control with Microsoft Entra ID for your Azure Cosmos DB account

Article • 10/12/2023

APPLIES TO:  NoSQL

Note

This article is about role-based access control for data plane operations in Azure Cosmos DB. If you are using management plane operations, see [role-based access control applied to your management plane operations article](#).

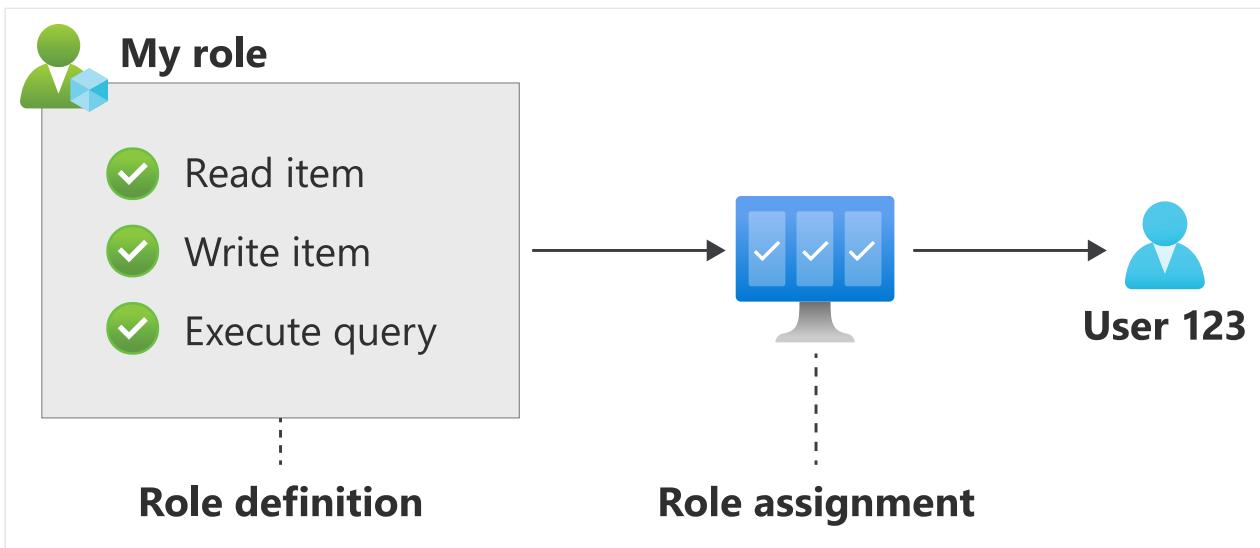
Azure Cosmos DB exposes a built-in role-based access control system that lets you:

- Authenticate your data requests with a Microsoft Entra identity.
- Authorize your data requests with a fine-grained, role-based permission model.

Concepts

The Azure Cosmos DB data plane role-based access control is built on concepts that are commonly found in other role-based access control systems like [Azure role-based access control](#):

- The [permission model](#) is composed of a set of [actions](#); each of these actions maps to one or multiple database operations. Some examples of actions include reading an item, writing an item, or executing a query.
- Azure Cosmos DB users create [role definitions](#) containing a list of allowed actions.
- Role definitions get assigned to specific Microsoft Entra identities through [role assignments](#). A role assignment also defines the scope that the role definition applies to; currently, three scopes are currently:
 - An Azure Cosmos DB account,
 - An Azure Cosmos DB database,
 - An Azure Cosmos DB container.



Permission model

i Important

This permission model covers only database operations that involve reading and writing data. It **does not** cover any kind of management operations on management resources, including:

- Create/Replace/Delete Database
- Create/Replace/Delete Container
- Read/Replace Container Throughput
- Create/Replace/Delete/Read Stored Procedures
- Create/Replace/Delete/Read Triggers
- Create/Replace/Delete/Read User Defined Functions

You *cannot* use any Azure Cosmos DB data plane SDK to authenticate management operations with a Microsoft Entra identity. Instead, you must use [Azure role-based access control](#) through one of the following options:

- [Azure Resource Manager templates \(ARM templates\)](#)
- [Azure PowerShell scripts](#)
- [Azure CLI scripts](#)
- Azure management libraries available in:
 - [.NET](#)
 - [Java](#)
 - [Python](#)

Read Database and Read Container are considered [metadata requests](#). Access to these operations can be granted as stated in the following section.

This table lists all the actions exposed by the permission model.

Name	Corresponding database operation(s)
Microsoft.DocumentDB/databaseAccounts/readMetadata	Read account metadata. See Metadata requests for details.
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/create	Create a new item.
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/read	Read an individual item by its ID and partition key (point-read).
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/replace	Replace an existing item.
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/upsert	"Upsert" an item. This operation creates an item if it doesn't already exist, or to replace the item if it does exist.
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/delete	Delete an item.
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/executeQuery	Execute a SQL query .
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/readChangeFeed	Read from the container's change feed . Execute SQL queries using the SDKs.
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/executeStoredProcedure	Execute a stored procedure .
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/manageConflicts	Manage conflicts for multi-write region

Name	Corresponding database operation(s)
	accounts (that is, list and delete items from the conflict feed).

ⓘ Note

When executing queries through the SDKs, both

`Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/executeQuery` and

`Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/readChangeFeed`

permissions are required.

Wildcards are supported at both *containers* and *items* levels:

- `Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/*`
- `Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/*`

Metadata requests

The Azure Cosmos DB SDKs issue read-only metadata requests during initialization and to serve specific data requests. These requests fetch various configuration details such as:

- The global configuration of your account, which includes the Azure regions the account is available in.
- The partition key of your containers or their indexing policy.
- The list of physical partitions that make a container and their addresses.

They **do not** fetch any of the data that you've stored in your account.

To ensure the best transparency of our permission model, these metadata requests are explicitly covered by the `Microsoft.DocumentDB/databaseAccounts/readMetadata` action. This action should be allowed in every situation where your Azure Cosmos DB account is accessed through one of the Azure Cosmos DB SDKs. It can be assigned (through a role assignment) at any level in the Azure Cosmos DB hierarchy (that is, account, database, or container).

The actual metadata requests allowed by the

`Microsoft.DocumentDB/databaseAccounts/readMetadata` action depend on the scope that the action is assigned to:

Scope	Requests allowed by the action
Account	<ul style="list-style-type: none"> • Listing the databases under the account • For each database under the account, the allowed actions at the database scope
Database	<ul style="list-style-type: none"> • Reading database metadata • Listing the containers under the database • For each container under the database, the allowed actions at the container scope
Container	<ul style="list-style-type: none"> • Reading container metadata • Listing physical partitions under the container • Resolving the address of each physical partition

ⓘ Important

Throughput is not included in the metadata for this action.

Built-in role definitions

Azure Cosmos DB exposes two built-in role definitions:

ⓘ Important

The term **role definitions** here refer to Azure Cosmos DB specific role definitions. These are distinct from Azure role-based access control role definitions.

ID	Name	Included actions
00000000-0000-0000-0000-000000000001	Cosmos DB	<code>Microsoft.DocumentDB/databaseAccounts/readMetadata</code>
0000-0000-0000-0000-000000000001	Built-in	<code>Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/read</code>
0000-0000-0000-0000-000000000001	Data	<code>Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/executeQuery</code>
0000000000000001	Reader	<code>Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/readChangeFeed</code>
00000000-0000-0000-0000-000000000001	Cosmos DB	<code>Microsoft.DocumentDB/databaseAccounts/readMetadata</code>
0000-0000-0000-0000-000000000001	Built-in	<code>Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/*</code>
0000-0000-0000-0000-000000000001	Data	<code>Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/*</code>
0000000000000002	Contributor	

Create custom role definitions

When creating a custom role definition, you need to provide:

- The name of your Azure Cosmos DB account.
- The resource group containing your account.
- The type of the role definition: `CustomRole`.

- The name of the role definition.
- A list of **actions** that you want the role to allow.
- One or multiple scope(s) that the role definition can be assigned at; supported scopes are:
 - / (account-level),
 - /dbs/<database-name> (database-level),
 - /dbs/<database-name>/colls/<container-name> (container-level).

 **Note**

The operations described are available in:

- Azure PowerShell: **Az.CosmosDB** version 1.2.0  or higher
- **Azure CLI**: version 2.24.0 or higher

Using Azure PowerShell

Create a role named *MyReadOnlyRole* that only contains read actions:

PowerShell

```
$resourceGroupName = "<myResourceGroup>"  
$accountName = "<myCosmosAccount>"  
New-AzCosmosDBSqlRoleDefinition -AccountName $accountName `  
-ResourceGroupName $resourceGroupName `  
-Type CustomRole -RoleName MyReadOnlyRole `  
-DataAction @(`  
    'Microsoft.DocumentDB/databaseAccounts/readMetadata',  
    'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/read', `  
    'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/executeQuery', `  
    'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/readChangeFeed' `)  
    -AssignableScope "/"
```

Create a role named *MyReadWriteRole* that contains all actions:

PowerShell

```
New-AzCosmosDBSqlRoleDefinition -AccountName $accountName `  
-ResourceGroupName $resourceGroupName `  
-Type CustomRole -RoleName MyReadWriteRole `  
-DataAction @(`  
    'Microsoft.DocumentDB/databaseAccounts/readMetadata',  
    'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/*', `
```

```
'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/*')`  
-AssignableScope "/"
```

List the role definitions you've created to fetch their IDs:

PowerShell

```
Get-AzCosmosDBSqlRoleDefinition -AccountName $accountName  
-ResourceGroupName $resourceGroupName
```

Output

```
RoleName      : MyReadWriteRole
Id           :
/subscriptions/<mySubscriptionId>/resourceGroups/<myResourceGroup>/providers/Microsoft.DocumentDB/databaseAccounts/<myCosmosAccount>/sqlRoleDefinitions/<roleDefinitionId>
Type         : CustomRole
Permissions   : {Microsoft.Azure.Management.CosmosDB.Models.Permission}
AssignableScopes :
{/subscriptions/<mySubscriptionId>/resourceGroups/<myResourceGroup>/providers/Microsoft.DocumentDB/databaseAccounts/<myCosmosAccount>}/sqlRoleDefinitions/<roleDefinitionId>
RoleName      : MyReadOnlyRole
Id           :
/subscriptions/<mySubscriptionId>/resourceGroups/<myResourceGroup>/providers/Microsoft.DocumentDB/databaseAccounts/<myCosmosAccount>/sqlRoleDefinitions/<roleDefinitionId>
Type         : CustomRole
Permissions   : {Microsoft.Azure.Management.CosmosDB.Models.Permission}
AssignableScopes :
{/subscriptions/<mySubscriptionId>/resourceGroups/<myResourceGroup>/providers/Microsoft.DocumentDB/databaseAccounts/<myCosmosAccount>}/sqlRoleDefinitions/<roleDefinitionId>
```

Using the Azure CLI

Create a role named `MyReadOnlyRole` that only contains read actions in a file named `role-definition-ro.json`:

JSON

```
{  
    "RoleName": "MyReadOnlyRole",  
    "Type": "CustomRole",  
    "AssignableScopes": ["/"],  
    "Permissions": [  
        "DataActions": [  
            "Microsoft.DocumentDB/databaseAccounts/readMetadata",  
            "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/read",  
            "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/write",  
            "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/insert",  
            "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/delete"  
        ]  
    ]  
}
```

```
"Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/executeQuery",  
  
"Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/readChangeFeed"  
    ]  
}  
}
```

Azure CLI

```
resourceGroupName='<myResourceGroup>'  
accountName='<myCosmosAccount>'  
az cosmosdb sql role definition create --account-name $accountName --resource-  
group $resourceGroupName --body @role-definition-ro.json
```

Create a role named *MyReadWriteRole* that contains all actions in a file named **role-definition-rw.json**:

JSON

```
{  
    "RoleName": "MyReadWriteRole",  
    "Type": "CustomRole",  
    "AssignableScopes": ["/"],  
    "Permissions": [  
        {  
            "DataActions": [  
                "Microsoft.DocumentDB/databaseAccounts/readMetadata",  
  
                "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/*",  
                "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/*"  
            ]  
        }]  
    ]  
}
```

Azure CLI

```
az cosmosdb sql role definition create --account-name $accountName --resource-  
group $resourceGroupName --body @role-definition-rw.json
```

List the role definitions you've created to fetch their IDs:

Azure CLI

```
az cosmosdb sql role definition list --account-name $accountName --resource-group  
$resourceGroupName
```

JSON

```
[  
    {
```

```
"assignableScopes": [  
  
    "/subscriptions/<mySubscriptionId>/resourceGroups/<myResourceGroup>/providers/Mic  
    rosoft.DocumentDB/databaseAccounts/<myCosmosAccount>"  
],  
"  
    "id":  
        "/subscriptions/<mySubscriptionId>/resourceGroups/<myResourceGroup>/providers/Mic  
        rosoft.DocumentDB/databaseAccounts/<myCosmosAccount>/sqlRoleDefinitions/<roleDefi  
        nitionId>",  
        "name": "<roleDefinitionId>",  
        "permissions": [  
            {  
                "dataActions": [  
                    "Microsoft.DocumentDB/databaseAccounts/readMetadata",  
  
                    "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/*",  
                    "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/*"  
                ],  
                "notDataActions": []  
            }  
        ],  
        "resourceGroup": "<myResourceGroup>",  
        "roleName": "MyReadWriteRole",  
        "sqlRoleDefinitionGetResultsType": "CustomRole",  
        "type": "Microsoft.DocumentDB/databaseAccounts/sqlRoleDefinitions"  
},  
{  
    "assignableScopes": [  
  
        "/subscriptions/<mySubscriptionId>/resourceGroups/<myResourceGroup>/providers/Mic  
        rosoft.DocumentDB/databaseAccounts/<myCosmosAccount>"  
],  
    "  
        "id":  
            "/subscriptions/<mySubscriptionId>/resourceGroups/<myResourceGroup>/providers/Mic  
            rosoft.DocumentDB/databaseAccounts/<myCosmosAccount>/sqlRoleDefinitions/<roleDefi  
            nitionId>",  
            "name": "<roleDefinitionId>",  
            "permissions": [  
                {  
                    "dataActions": [  
                        "Microsoft.DocumentDB/databaseAccounts/readMetadata",  
  
                        "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/items/read",  
  
                        "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/executeQuery",  
  
                        "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/readChangeFeed"  
                    ],  
                    "notDataActions": []  
                }  
            ],  
            "resourceGroup": "<myResourceGroup>",  
            "roleName": "MyReadOnlyRole",  
            "sqlRoleDefinitionGetResultsType": "CustomRole",  
            "type": "Microsoft.DocumentDB/databaseAccounts/sqlRoleDefinitions"  
}  
]
```

Using Azure Resource Manager templates

For a reference and examples of using Azure Resource Manager templates to create role definitions, see [Microsoft.DocumentDB databaseAccounts/sqlRoleDefinitions](#).

Create role assignments

You can associate built-in or custom role definitions with your Microsoft Entra identities. When creating a role assignment, you need to provide:

- The name of your Azure Cosmos DB account.
- The resource group containing your account.
- The ID of the role definition to assign.
- The principal ID of the identity that the role definition should be assigned to.
- The scope of the role assignment; supported scopes are:
 - `/` (account-level)
 - `/dbs/<database-name>` (database-level)
 - `/dbs/<database-name>/colls/<container-name>` (container-level)

The scope must match or be a subscope of one of the role definition's assignable scopes.

ⓘ Note

If you want to create a role assignment for a service principal, make sure to use its **Object ID** as found in the **Enterprise applications** section of the **Microsoft Entra ID** portal blade.

ⓘ Note

The operations described are available in:

- Azure PowerShell: `Az.CosmosDB` version 1.2.0  or higher
- Azure CLI: version 2.24.0 or higher

Using Azure PowerShell

Assign a role to an identity:

```
PowerShell
```

```

$resourceGroupName = "<myResourceGroup>"
$accountName = "<myCosmosAccount>" 
$readOnlyRoleDefinitionId = "<roleDefinitionId>" # as fetched above
# For Service Principals make sure to use the Object ID as found in the
Enterprise applications section of the Azure Active Directory portal blade.
$principalId = "<aadPrincipalId>" 
New-AzCosmosDBSqlRoleAssignment -AccountName $accountName
    -ResourceGroupName $resourceGroupName
    -RoleDefinitionId $readOnlyRoleDefinitionId
    -Scope "/"
    -PrincipalId $principalId

```

Using the Azure CLI

Assign a role to an identity:

Azure CLI

```

resourceGroupName='<myResourceGroup>' 
accountName='<myCosmosAccount>' 
readOnlyRoleDefinitionId='<roleDefinitionId>' # as fetched above
# For Service Principals make sure to use the Object ID as found in the
Enterprise applications section of the Azure Active Directory portal blade.
principalId='<aadPrincipalId>' 
az cosmosdb sql role assignment create --account-name $accountName --resource-
group $resourceGroupName --scope "/" --principal-id $principalId --role-
definition-id $readOnlyRoleDefinitionId

```

Using Bicep/Azure Resource Manager templates

For a built-in assignment using a Bicep template:

```

resource sqlRoleAssignment
'Microsoft.DocumentDB/databaseAccounts/sqlRoleAssignments@2023-04-15' = {
  name: guid(<roleDefinitionId>, <aadPrincipalId>, <databaseAccountResourceId>)
  parent: databaseAccount
  properties: {
    principalId: <aadPrincipalId>
    roleDefinitionId:
      '${subscription().id}/resourceGroups/<databaseAccountResourceGroup>/providers/Mi
crosoft.DocumentDB/databaseAccounts/<myCosmosAccount>/sqlRoleDefinitions/<roleDef
initionId>'
    scope: <databaseAccountResourceId>
  }
}

```

For a reference and examples of using Azure Resource Manager templates to create role assignments, see [Microsoft.DocumentDB databaseAccounts/sqlRoleAssignments](#).

Initialize the SDK with Microsoft Entra ID

To use the Azure Cosmos DB role-based access control in your application, you have to update the way you initialize the Azure Cosmos DB SDK. Instead of passing your account's primary key, you have to pass an instance of a `TokenCredential` class. This instance provides the Azure Cosmos DB SDK with the context required to fetch a Microsoft Entra token on behalf of the identity you wish to use.

The way you create a `TokenCredential` instance is beyond the scope of this article. There are many ways to create such an instance depending on the type of Microsoft Entra identity you want to use (user principal, service principal, group etc.). Most importantly, your `TokenCredential` instance must resolve to the identity (principal ID) that you've assigned your roles to. You can find examples of creating a `TokenCredential` class:

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

The following examples use a service principal with a `ClientSecretCredential` instance.

In .NET

The Azure Cosmos DB role-based access control is currently supported in the [.NET SDK V3](#).

C#

```
TokenCredential servicePrincipal = new ClientSecretCredential(
    "<azure-ad-tenant-id>",
    "<client-application-id>",
    "<client-application-secret>");
CosmosClient client = new CosmosClient("<account-endpoint>", servicePrincipal);
```

In Java

The Azure Cosmos DB role-based access control is currently supported in the [Java SDK V4](#).

Java

```
TokenCredential ServicePrincipal = new ClientSecretCredentialBuilder()
    .authorityHost("https://login.microsoftonline.com")
    .tenantId("<azure-ad-tenant-id>")
    .clientId("<client-application-id>")
    .clientSecret("<client-application-secret>")
    .build();
CosmosAsyncClient Client = new CosmosClientBuilder()
    .endpoint("<account-endpoint>")
```

```
.credential(ServicePrincipal)  
.build();
```

In JavaScript

The Azure Cosmos DB role-based access control is currently supported in the [JavaScript SDK V3](#).

JavaScript

```
const servicePrincipal = new ClientSecretCredential(  
    "<azure-ad-tenant-id>",  
    "<client-application-id>",  
    "<client-application-secret>");  
const client = new CosmosClient({  
    endpoint: "<account-endpoint>",  
    aadCredentials: servicePrincipal  
});
```

In Python

The Azure Cosmos DB role-based access control is supported in the [Python SDK versions 4.3.0b4](#) and higher.

Python

```
aad_credentials = ClientSecretCredential(  
    tenant_id="<azure-ad-tenant-id>",  
    client_id="<client-application-id>",  
    client_secret="<client-application-secret>")  
client = CosmosClient("<account-endpoint>", aad_credentials)
```

Authenticate requests on the REST API

When constructing the [REST API authorization header](#), set the **type** parameter to **Microsoft Entra ID** and the hash signature (**sig**) to the [OAuth token](#) as shown in the following example:

```
type=aad&ver=1.0&sig=<token-from-oauth>
```

Use data explorer

ⓘ Note

The data explorer exposed in the Azure portal does not support the Azure Cosmos DB role-based access control yet. To use your Microsoft Entra identity when exploring your

data, you must use the [Azure Cosmos DB Explorer](#) instead.

When you access the [Azure Cosmos DB Explorer](#) with the specific `?feature.enableAadDataPlane=true` query parameter and sign in, the following logic is used to access your data:

1. A request to fetch the account's primary key is attempted on behalf of the identity signed in. If this request succeeds, the primary key is used to access the account's data.
2. If the identity signed in isn't allowed to fetch the account's primary key, this identity is directly used to authenticate data access. In this mode, the identity must be [assigned with proper role definitions](#) to ensure data access.

Audit data requests

[Diagnostic logs](#) get augmented with identity and authorization information for each data operation when using Azure Cosmos DB role-based access control. This augmentation lets you perform detailed auditing and retrieve the Microsoft Entra identity used for every data request sent to your Azure Cosmos DB account.

This additional information flows in the **DataPlaneRequests** log category and consists of two extra columns:

- `aadPrincipalId_g` shows the principal ID of the Microsoft Entra identity that was used to authenticate the request.
- `aadAppliedRoleIdAssignmentId_g` shows the [role assignment](#) that was honored when authorizing the request.

Enforcing role-based access control as the only authentication method

In situations where you want to force clients to connect to Azure Cosmos DB through role-based access control exclusively, you can disable the account's primary/secondary keys. When doing so, any incoming request using either a primary/secondary key or a resource token is actively rejected.

Use Azure Resource Manager templates

When creating or updating your Azure Cosmos DB account using Azure Resource Manager templates, set the `disableLocalAuth` property to `true`:

JSON

```
"resources": [
  {
    "type": "Microsoft.DocumentDB/databaseAccounts",
    "properties": {
      "disableLocalAuth": true,
      // ...
    },
    // ...
  },
  // ...
]
```

Limits

- You can create up to 100 role definitions and 2,000 role assignments per Azure Cosmos DB account.
- You can only assign role definitions to Microsoft Entra identities belonging to the same Microsoft Entra tenant as your Azure Cosmos DB account.
- Microsoft Entra group resolution isn't currently supported for identities that belong to more than 200 groups.
- The Microsoft Entra token is currently passed as a header with each individual request sent to the Azure Cosmos DB service, increasing the overall payload size.

Frequently asked questions

This section includes frequently asked questions about role-based access control and Azure Cosmos DB.

Which Azure Cosmos DB APIs support role-based access control?

The API for NoSQL is supported. Support for the API for MongoDB is in preview.

Is it possible to manage role definitions and role assignments from the Azure portal?

Azure portal support for role management isn't available yet.

Which SDKs in Azure Cosmos DB API for NoSQL support role-based access control?

The [.NET V3](#), [Java V4](#), [JavaScript V3](#) and [Python V4.3+](#) SDKs are currently supported.

Is the Microsoft Entra token automatically refreshed by the Azure Cosmos DB SDKs when it expires?

Yes.

Is it possible to disable the usage of the account primary/secondary keys when using role-based access control?

Yes, see [Enforcing role-based access control as the only authentication method](#).

Next steps

- Get an overview of [secure access to data in Azure Cosmos DB](#).
- Learn more about [role-based access control for Azure Cosmos DB management](#).

How to use managed identities with Azure Container Instances

Article • 10/11/2023

Use [managed identities for Azure resources](#) to run code in Azure Container Instances that interacts with other Azure services - without maintaining any secrets or credentials in code. The feature provides an Azure Container Instances deployment with an automatically managed identity in Microsoft Entra ID.

In this article, you learn more about managed identities in Azure Container Instances and:

- ✓ Enable a user-assigned or system-assigned identity in a container group
- ✓ Grant the identity access to an Azure key vault
- ✓ Use the managed identity to access a key vault from a running container

Adapt the examples to enable and use identities in Azure Container Instances to access other Azure services. These examples are interactive. However, in practice your container images would run code to access Azure services.

Why use a managed identity?

Use a managed identity in a running container to authenticate to any [service that supports Microsoft Entra authentication](#) without managing credentials in your container code. For services that don't support AD authentication, you can store secrets in an Azure key vault and use the managed identity to access the key vault to retrieve credentials. For more information about using a managed identity, see [What is managed identities for Azure resources?](#)

Enable a managed identity

When you create a container group, enable one or more managed identities by setting a [ContainerGroupIdentity](#) property. You can also enable or update managed identities after a container group is running - either action causes the container group to restart. To set the identities on a new or existing container group, use the Azure CLI, a Resource Manager template, a YAML file, or another Azure tool.

Azure Container Instances supports both types of managed Azure identities: user-assigned and system-assigned. On a container group, you can enable a system-assigned

identity, one or more user-assigned identities, or both types of identities. If you're unfamiliar with managed identities for Azure resources, see the [overview](#).

Use a managed identity

To use a managed identity, the identity must be granted access to one or more Azure service resources (such as a web app, a key vault, or a storage account) in the subscription. Using a managed identity in a running container is similar to using an identity in an Azure VM. See the VM guidance for using a [token](#), [Azure PowerShell](#) or [Azure CLI](#), or the [Azure SDKs](#).

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the button is a small blue square icon with a white arrow pointing outwards.
 - If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.
- This article requires version 2.0.49 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Create an Azure key vault

The examples in this article use a managed identity in Azure Container Instances to access an Azure key vault secret.

First, create a resource group named `myResourceGroup` in the `eastus` location with the following `az group create` command:

Azure CLI

```
az group create --name myResourceGroup --location eastus
```

Use the [az keyvault create](#) command to create a key vault. Be sure to specify a unique key vault name.

Azure CLI

```
az keyvault create \
--name mykeyvault \
--resource-group myResourceGroup \
--location eastus
```

Store a sample secret in the key vault using the [az keyvault secret set](#) command:

Azure CLI

```
az keyvault secret set \
--name SampleSecret \
--value "Hello Container Instances" \
--description ACIsecret --vault-name mykeyvault
```

Continue with the following examples to access the key vault using either a user-assigned or system-assigned managed identity in Azure Container Instances.

Example 1: Use a user-assigned identity to access Azure key vault

Create an identity

First create an identity in your subscription using the [az identity create](#) command. You can use the same resource group used to create the key vault, or use a different one.

Azure CLI

```
az identity create \
--resource-group myResourceGroup \
--name myACIID
```

To use the identity in the following steps, use the [az identity show](#) command to store the identity's service principal ID and resource ID in variables.

Azure CLI

```
# Get service principal ID of the user-assigned identity
SP_ID=$(az identity show \
    --resource-group myResourceGroup \
    --name myACIId \
    --query principalId --output tsv)

# Get resource ID of the user-assigned identity
RESOURCE_ID=$(az identity show \
    --resource-group myResourceGroup \
    --name myACIId \
    --query id --output tsv)
```

Grant user-assigned identity access to the key vault

Run the following [az keyvault set-policy](#) command to set an access policy on the key vault. The following example allows the user-assigned identity to get secrets from the key vault:

Azure CLI

```
az keyvault set-policy \
    --name mykeyvault \
    --resource-group myResourceGroup \
    --object-id $SP_ID \
    --secret-permissions get
```

Enable user-assigned identity on a container group

Run the following [az container create](#) command to create a container instance based on Microsoft's `azure-cli` image. This example provides a single-container group that you can use interactively to run the Azure CLI to access other Azure services. In this section, only the base operating system is used. For an example to use the Azure CLI in the container, see [Enable system-assigned identity on a container group](#).

The `--assign-identity` parameter passes your user-assigned managed identity to the group. The long-running command keeps the container running. This example uses the same resource group used to create the key vault, but you could specify a different one.

Azure CLI

```
az container create \
    --resource-group myResourceGroup \
    --name mycontainer \
```

```
--image mcr.microsoft.com/azure-cli \
--assign-identity $RESOURCE_ID \
--command-line "tail -f /dev/null"
```

Within a few seconds, you should get a response from the Azure CLI indicating that the deployment has completed. Check its status with the [az container show](#) command.

Azure CLI

```
az container show \
--resource-group myResourceGroup \
--name mycontainer
```

The `identity` section in the output looks similar to the following, showing the identity is set in the container group. The `principalID` under `userAssignedIdentities` is the service principal of the identity you created in Microsoft Entra ID:

Output

```
[...]
"identity": {
    "principalId": "null",
    "tenantId": "xxxxxxxx-f292-4e60-9122-xxxxxxxxxxxx",
    "type": "UserAssigned",
    "userAssignedIdentities": {
        "/subscriptions/xxxxxxxx-0903-4b79-a55a-
xxxxxxxxxxxx/resourcegroups/danlep1018/providers/Microsoft.ManagedIdentity/u
serAssignedIdentities/myACIId": {
            "clientId": "xxxxxxxx-5523-45fc-9f49-xxxxxxxxxxxx",
            "principalId": "xxxxxxxx-f25b-4895-b828-xxxxxxxxxxxx"
        }
    }
},
[...]
```

Use user-assigned identity to get secret from key vault

Now you can use the managed identity within the running container instance to access the key vault. First launch a bash shell in the container:

Azure CLI

```
az container exec \
--resource-group myResourceGroup \
--name mycontainer \
--exec-command "/bin/bash"
```

Run the following commands in the bash shell in the container. To get an access token to use Microsoft Entra ID to authenticate to key vault, run the following command:

Bash

```
client_id="xxxxxxxx-5523-45fc-9f49-xxxxxxxxxxxx"
curl "http://169.254.169.254/metadata/identity/oauth2/token?api-
version=2018-02-
01&resource=https%3A%2F%2Fvault.azure.net&client_id=$client_id" -H
Metadata:true -s
```

Output:

Bash

```
{"access_token": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1QiLCJhbGciOiJSUzI1NiIsIng1
dCI6Imk2bEdrM0ZaenhSY1ViMkMzbkVRN3N5SEpsWSIsImtpZCI6Imk2bEdrM0ZaenhSY1ViMkMz
bkVRN3N5SEpsWSJ9.....xxxxxxxxxxxxxx", "refresh_token": "", "expires_in": "28
799", "expires_on": "1539927532", "not_before": "1539898432", "resource": "https:/
/vault.azure.net/", "token_type": "Bearer"}
```

To store the access token in a variable to use in subsequent commands to authenticate, run the following command:

Bash

```
TOKEN=$(curl 'http://169.254.169.254/metadata/identity/oauth2/token?api-
version=2018-02-01&resource=https%3A%2F%2Fvault.azure.net' -H Metadata:true
| jq -r '.access_token')
```

Now use the access token to authenticate to key vault and read a secret. Be sure to substitute the name of your key vault in the URL ([https://mykeyvault.vault.azure.net/...](https://mykeyvault.vault.azure.net/)):

Bash

```
curl https://mykeyvault.vault.azure.net/secrets/SampleSecret/?api-
version=7.4 -H "Authorization: Bearer $TOKEN"
```

The response looks similar to the following, showing the secret. In your code, you would parse this output to obtain the secret. Then, use the secret in a subsequent operation to access another Azure resource.

Bash

```
{"value": "Hello Container Instances", "contentType": "ACIsecret", "id": "https://mykeyvault.vault.azure.net/secrets/SampleSecret/xxxxxxxxxxxxxxxxxxxx", "attributes": {"enabled": true, "created": 1539965967, "updated": 1539965967, "recoveryLevel": "Purgeable"}, "tags": {"file-encoding": "utf-8"}}
```

Example 2: Use a system-assigned identity to access Azure key vault

Enable system-assigned identity on a container group

Run the following `az container create` command to create a container instance based on Microsoft's `azure-cli` image. This example provides a single-container group that you can use interactively to run the Azure CLI to access other Azure services.

The `--assign-identity` parameter with no additional value enables a system-assigned managed identity on the group. The identity is scoped to the resource group of the container group. The long-running command keeps the container running. This example uses the same resource group used to create the key vault, which is in the scope of the identity.

Azure CLI

```
# Get the resource ID of the resource group
RG_ID=$(az group show --name myResourceGroup --query id --output tsv)

# Create container group with system-managed identity
az container create \
    --resource-group myResourceGroup \
    --name mycontainer \
    --image mcr.microsoft.com/azure-cli \
    --assign-identity --scope $RG_ID \
    --command-line "tail -f /dev/null"
```

Within a few seconds, you should get a response from the Azure CLI indicating that the deployment has completed. Check its status with the `az container show` command.

Azure CLI

```
az container show \
    --resource-group myResourceGroup \
    --name mycontainer
```

The `identity` section in the output looks similar to the following, showing that a system-assigned identity is created in Microsoft Entra ID:

Output

```
[...]
"identity": {
    "principalId": "xxxxxxxx-528d-7083-b74c-xxxxxxxxxxxx",
    "tenantId": "xxxxxxxx-f292-4e60-9122-xxxxxxxxxxxx",
    "type": "SystemAssigned",
    "userAssignedIdentities": null
},
[...]
```

Set a variable to the value of `principalId` (the service principal ID) of the identity, to use in later steps.

Azure CLI

```
SP_ID=$(az container show \
--resource-group myResourceGroup \
--name mycontainer \
--query identity.principalId --out tsv)
```

Grant container group access to the key vault

Run the following [az keyvault set-policy](#) command to set an access policy on the key vault. The following example allows the system-managed identity to get secrets from the key vault:

Azure CLI

```
az keyvault set-policy \
--name mykeyvault \
--resource-group myResourceGroup \
--object-id $SP_ID \
--secret-permissions get
```

Use container group identity to get secret from key vault

Now you can use the managed identity to access the key vault within the running container instance. First launch a bash shell in the container:

Azure CLI

```
az container exec \  
  --resource-group myResourceGroup \  
  --name mycontainer \  
  --exec-command "/bin/bash"
```

Run the following commands in the bash shell in the container. First, sign in to the Azure CLI using the managed identity:

Azure CLI

```
az login --identity
```

From the running container, retrieve the secret from the key vault:

Azure CLI

```
az keyvault secret show \  
  --name SampleSecret \  
  --vault-name mykeyvault --query value
```

The value of the secret is retrieved:

Output

```
"Hello Container Instances"
```

Enable managed identity using Resource Manager template

To enable a managed identity in a container group using a [Resource Manager template](#), set the `identity` property of the `Microsoft.ContainerInstance/containerGroups` object with a `ContainerGroupIdentity` object. The following snippets show the `identity` property configured for different scenarios. See the [Resource Manager template reference](#). Specify a minimum `apiVersion` of `2018-10-01`.

User-assigned identity

A user-assigned identity is a resource ID of the form:

```
"/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{identityName}"
```

You can enable one or more user-assigned identities.

JSON

```
"identity": {  
    "type": "UserAssigned",  
    "userAssignedIdentities": {  
        "myResourceID1": {}  
    }  
}
```

System-assigned identity

JSON

```
"identity": {  
    "type": "SystemAssigned"  
}
```

System- and user-assigned identities

On a container group, you can enable both a system-assigned identity and one or more user-assigned identities.

JSON

```
"identity": {  
    "type": "System Assigned, UserAssigned",  
    "userAssignedIdentities": {  
        "myResourceID1": {}  
    }  
}  
...
```

Enable managed identity using YAML file

To enable a managed identity in a container group deployed using a [YAML file](#), include the following YAML. Specify a minimum `apiVersion` of `2018-10-01`.

User-assigned identity

A user-assigned identity is a resource ID of the form

```
'/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{identityName}'
```

You can enable one or more user-assigned identities.

YAML

```
identity:
  type: UserAssigned
  userAssignedIdentities:
    - myResourceId1: {}
```

System-assigned identity

YAML

```
identity:
  type: SystemAssigned
```

System- and user-assigned identities

On a container group, you can enable both a system-assigned identity and one or more user-assigned identities.

yml

```
identity:
  type: SystemAssigned, UserAssigned
  userAssignedIdentities:
    - myResourceId1: {}
```

Next steps

In this article, you learned about managed identities in Azure Container Instances and how to:

- ✓ Enable a user-assigned or system-assigned identity in a container group

- ✓ Grant the identity access to an Azure key vault
- ✓ Use the managed identity to access a key vault from a running container
 - Learn more about [managed identities for Azure resources](#).
 - See an [Azure Go SDK example ↗](#) of using a managed identity to access a key vault from Azure Container Instances.

Azure Service Bus JMS 2.0 developer guide

Article • 05/03/2023

This guide contains detailed information to help you succeed in communicating with Azure Service Bus using the Java Message Service (JMS) 2.0 API.

As a Java developer, if you're new to Azure Service Bus, please consider reading the below articles.

Getting started	Concepts
<ul style="list-style-type: none">• What is Azure Service Bus• Queues, Topics and Subscriptions	<ul style="list-style-type: none">• Azure Service Bus - Premium tier

Java Message Service (JMS) Programming model

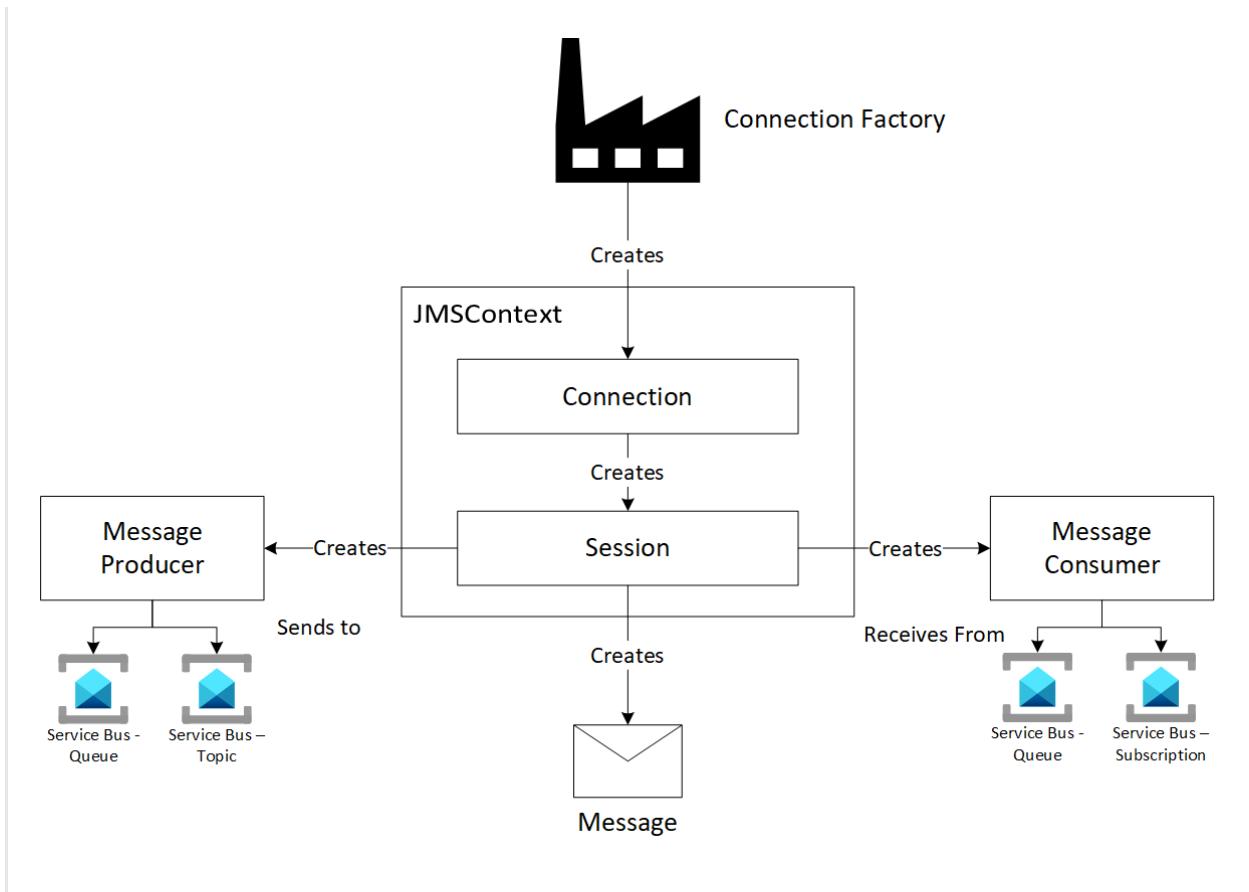
The Java Message Service API programming model is as shown below -

ⓘ Note

Azure Service Bus Premium tier supports JMS 1.1 and JMS 2.0.

Azure Service Bus - Standard tier supports limited JMS 1.1 functionality. For more details, please refer to this [documentation](#).

JMS 2.0 Programming model



JMS - Building blocks

The below building blocks are available to communicate with the JMS application.

! Note

The below guide has been adapted from the [Oracle Java EE 6 Tutorial for Java Message Service \(JMS\)](#)

Referring to this tutorial is recommended for better understanding of the Java Message Service (JMS).

Connection factory

The connection factory object is used by the client to connect with the JMS provider. The connection factory encapsulates a set of connection configuration parameters that are defined by the administrator.

Each connection factory is an instance of `ConnectionFactory`, `QueueConnectionFactory` or `TopicConnectionFactory` interface.

To simplify connecting with Azure Service Bus, these interfaces are implemented through `ServiceBusJmsConnectionFactory`, `ServiceBusJmsQueueConnectionFactory` and `ServiceBusJmsTopicConnectionFactory` respectively.

ⓘ Important

Java applications leveraging JMS 2.0 API can connect to Azure Service Bus using the connection string, or using a `TokenCredential` for leveraging Microsoft Entra backed authentication. When using Microsoft Entra backed authentication, ensure to **assign roles and permissions** to the identity as needed.

System Assigned Managed Identity

Create a [system assigned managed identity](#) on Azure, and use this identity to create a `TokenCredential`.

Java

```
TokenCredential tokenCredential = new  
DefaultAzureCredentialBuilder().build();
```

The Connection factory can then be instantiated with the below parameters.

- Token credential - Represents a credential capable of providing an OAuth token.
- Host - the hostname of the Azure Service Bus Premium tier namespace.
- `ServiceBusJmsConnectionFactorySettings` property bag, which contains
 - `connectionIdleTimeoutMS` - idle connection timeout in milliseconds.
 - `traceFrames` - boolean flag to collect AMQP trace frames for debugging.
 - *other configuration parameters*

The factory can be created as shown here. The token credential and host are required parameters, but the other properties are optional.

Java

```
String host = "<YourNamespaceName>.servicebus.windows.net";  
ConnectionFactory factory = new  
ServiceBusJmsConnectionFactory(tokenCredential, host, null);
```

JMS destination

A destination is the object a client uses to specify the target of the messages it produces and the source of the messages it consumes.

Destinations map to entities in Azure Service Bus - queues (in point to point scenarios) and topics (in pub-sub scenarios).

Connections

A connection encapsulates a virtual connection with a JMS provider. With Azure Service Bus, this represents a stateful connection between the application and Azure Service Bus over AMQP.

A connection is created from the connection factory as shown below.

```
Java
```

```
Connection connection = factory.createConnection();
```

Sessions

A session is a single-threaded context for producing and consuming messages. It can be utilized to create messages, message producers and consumers, but it also provides a transactional context to allow grouping of sends and receives into an atomic unit of work.

A session can be created from the connection object as shown below.

```
Java
```

```
Session session = connection.createSession(false,  
Session.CLIENT_ACKNOWLEDGE);
```

① Note

JMS API doesn't support receiving messages from service bus queues or topics with messaging sessions enabled.

Session modes

A session can be created with any of the below modes.

Session modes	Behavior
Session.AUTO_ACKNOWLEDGE	The session automatically acknowledges a client's receipt of a message either when the session has successfully returned from a call to receive or when the message listener the session has called to process the message successfully returns.
Session.CLIENT_ACKNOWLEDGE	The client acknowledges a consumed message by calling the message's acknowledge method.
Session.DUPS_OK_ACKNOWLEDGE	This acknowledgment mode instructs the session to lazily acknowledge the delivery of messages.
Session.SESSION_TRANSACTED	This value may be passed as the argument to the method createSession(int sessionMode) on the Connection object to specify that the session should use a local transaction.

When the session mode isn't specified, the `Session.AUTO_ACKNOWLEDGE` is picked by default.

JMSContext

 Note

JMSContext is defined as part of the JMS 2.0 specification.

JMSContext combines the functionality provided by the connection and session object. It can be created from the connection factory object.

Java

```
JMSContext context = connectionFactory.createContext();
```

JMSContext modes

Just like the `Session` object, the `JMSContext` can be created with the same acknowledge modes as mentioned in [Session modes](#).

Java

```
JMSContext context =
connectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE);
```

When the mode isn't specified, the `JMSContext.AUTO_ACKNOWLEDGE` is picked by default.

JMS message producers

A message producer is an object that is created using a `JMSContext` or a `Session` and used for sending messages to a destination.

It can be created either as a stand-alone object as below -

Java

```
JMSProducer producer = context.createProducer();
```

or created at runtime when a message is needed to be sent.

Java

```
context.createProducer().send(destination, message);
```

JMS message consumers

A message consumer is an object that is created by a `JMSContext` or a `Session` and used for receiving messages sent to a destination. It can be created as shown below -

Java

```
JMSConsumer consumer = context.createConsumer(dest);
```

Synchronous receives via `receive()` method

The message consumer provides a synchronous way to receive messages from the destination through the `receive()` method.

If no arguments/timeout is specified or a timeout of '0' is specified, then the consumer blocks indefinitely unless the message arrives, or the connection is broken (whichever is earlier).

Java

```
Message m = consumer.receive();
Message m = consumer.receive(0);
```

When a non-zero positive argument is provided, the consumer blocks until that timer expires.

Java

```
Message m = consumer.receive(1000); // time out after one second.
```

Asynchronous receives with JMS message listeners

A message listener is an object that is used for asynchronous handling of messages on a destination. It implements the `MessageListener` interface which contains the `onMessage` method where the specific business logic must live.

A message listener object must be instantiated and registered against a specific message consumer using the `setMessageListener` method.

Java

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

Consuming from topics

JMS Message Consumers are created against a `destination` which may be a queue or a topic.

Consumers on queues are simply client side objects that live in the context of the Session (and Connection) between the client application and Azure Service Bus.

Consumers on topics, however, have 2 parts -

- A **client side object** that lives in the context of the Session(or JMSContext), and,
- A **subscription** that is an entity on Azure Service Bus.

The subscriptions are documented [here](#) and can be one of the below -

- Shared durable subscriptions
- Shared non-durable subscriptions
- Unshared durable subscriptions
- Unshared non-durable subscriptions

JMS Queue Browsers

The JMS API provides a `QueueBrowser` object that allows the application to browse the messages in the queue and display the header values for each message.

A Queue Browser can be created using the `JMSContext` as below.

Java

```
QueueBrowser browser = context.createBrowser(queue);
```

ⓘ Note

JMS API doesn't provide an API to browse a topic.

This is because the topic itself doesn't store the messages. As soon as the message is sent to the topic, it is forwarded to the appropriate subscriptions.

JMS Message selectors

Message selectors can be used by receiving applications to filter the messages that are received. With message selectors, the receiving application offloads the work of filtering messages to the JMS provider (in this case, Azure Service Bus) instead of taking that responsibility itself.

Selectors can be utilized when creating any of the below consumers -

- Shared durable subscription
- Unshared durable subscription
- Shared non-durable subscription
- Unshared non-durable subscription
- Queue browser

AMQP disposition and Service Bus operation mapping

Here's how an AMQP disposition translates to a Service Bus operation:

Output

```
ACCEPTED = 1; -> Complete()
REJECTED = 2; -> DeadLetter()
RELEASED = 3; (just unlock the message in service bus, will then get
redelivered)
```

```
MODIFIED_FAILED = 4; -> Abandon() which increases delivery count  
MODIFIED_FAILED_UNDELIVERABLE = 5; -> Defer()
```

Summary

This developer guide showcased how Java client applications using Java Message Service (JMS) can connect with Azure Service Bus.

Next steps

For more information on Azure Service Bus and details about Java Message Service (JMS) entities, check out the links below -

- [Service Bus - Queues, Topics, and Subscriptions](#)
- [Service Bus - Java Message Service entities](#)
- [AMQP 1.0 support in Azure Service Bus](#)
- [Service Bus AMQP 1.0 Developer's Guide](#)
- [Get started with Service Bus queues](#)
- [Java Message Service API\(external Oracle doc\)](#) ↗
- [Learn how to migrate from ActiveMQ to Service Bus](#)

Tutorial: Deploy a Spring application to Azure Spring Apps with a passwordless connection to an Azure database

Article • 04/24/2023

This article shows you how to use passwordless connections to Azure databases in Spring Boot applications deployed to Azure Spring Apps.

In this tutorial, you complete the following tasks using the Azure portal or the Azure CLI. Both methods are explained in the following procedures.

- ✓ Provision an instance of Azure Spring Apps.
- ✓ Build and deploy apps to Azure Spring Apps.
- ✓ Run apps connected to Azure databases using managed identity.

ⓘ Note

This tutorial doesn't work for R2DBC.

Prerequisites

- An Azure subscription. If you don't already have one, create a [free account](#) before you begin.
- [Azure CLI](#) 2.45.0 or higher required.
- The Azure Spring Apps extension. You can install the extension by using the command: `az extension add --name spring`.
- [Java Development Kit \(JDK\)](#), version 8, 11, or 17.
- A [Git](#) client.
- [cURL](#) or a similar HTTP utility to test functionality.
- MySQL command line client if you choose to run Azure Database for MySQL. You can connect to your server with Azure Cloud Shell using a popular client tool, the [mysql.exe](#) command-line tool. Alternatively, you can use the `mysql` command line in your local environment.
- [ODBC Driver 18 for SQL Server](#) if you choose to run Azure SQL Database.

Prepare the working environment

First, set up some environment variables by using the following commands:

Bash

```
export AZ_RESOURCE_GROUP=passwordless-tutorial-rg
export AZ_DATABASE_SERVER_NAME=<YOUR_DATABASE_SERVER_NAME>
export AZ_DATABASE_NAME=demodb
export AZ_LOCATION=<YOUR_AZURE_REGION>
export AZ_SPRING_APPS_SERVICE_NAME=<YOUR_AZURE_SPRING_APPS_SERVICE_NAME>
export AZ_SPRING_APPS_APP_NAME=hellospring
export AZ_DB_ADMIN_USERNAME=<YOUR_DB_ADMIN_USERNAME>
export AZ_DB_ADMIN_PASSWORD=<YOUR_DB_ADMIN_PASSWORD>
export AZ_USER_IDENTITY_NAME=<YOUR_USER_ASSIGNED_MANAGEMED_IDENTITY_NAME>
```

Replace the placeholders with the following values, which are used throughout this article:

- <YOUR_DATABASE_SERVER_NAME>: The name of your Azure Database server, which should be unique across Azure.
- <YOUR_AZURE_REGION>: The Azure region you want to use. You can use `eastus` by default, but we recommend that you configure a region closer to where you live. You can see the full list of available regions by using `az account list-locations`.
- <YOUR_AZURE_SPRING_APPS_SERVICE_NAME>: The name of your Azure Spring Apps instance. The name must be between 4 and 32 characters long and can contain only lowercase letters, numbers, and hyphens. The first character of the service name must be a letter and the last character must be either a letter or a number.
- <AZ_DB_ADMIN_USERNAME>: The admin username of your Azure database server.
- <AZ_DB_ADMIN_PASSWORD>: The admin password of your Azure database server.
- <YOUR_USER_ASSIGNED_MANAGEMED_IDENTITY_NAME>: The name of your user assigned managed identity server, which should be unique across Azure.

Provision an instance of Azure Spring Apps

Use the following steps to provision an instance of Azure Spring Apps.

1. Update Azure CLI with the Azure Spring Apps extension by using the following command:

Azure CLI

```
az extension update --name spring
```

2. Sign in to the Azure CLI and choose your active subscription by using the following commands:

```
Azure CLI
```

```
az login  
az account list --output table  
az account set --subscription <name-or-ID-of-subscription>
```

3. Use the following commands to create a resource group to contain your Azure Spring Apps service and an instance of the Azure Spring Apps service:

```
Azure CLI
```

```
az group create \  
  --name $AZ_RESOURCE_GROUP \  
  --location $AZ_LOCATION  
az spring create \  
  --resource-group $AZ_RESOURCE_GROUP \  
  --name $AZ_SPRING_APPS_SERVICE_NAME
```

Create an Azure database instance

Use the following steps to provision an Azure Database instance.

```
Azure SQL Database
```

1. Create an Azure SQL Database server by using the following command:

```
Azure CLI
```

```
az sql server create \  
  --location $AZ_LOCATION \  
  --resource-group $AZ_RESOURCE_GROUP \  
  --name $AZ_DATABASE_SERVER_NAME \  
  --admin-user $AZ_DB_ADMIN_USERNAME \  
  --admin-password $AZ_DB_ADMIN_PASSWORD
```

2. The SQL server is empty, so create a new database by using the following command:

```
Azure CLI
```

```
az sql db create \  
  --resource-group $AZ_RESOURCE_GROUP \  
  --name $AZ_DATABASE_NAME
```

```
--server $AZ_DATABASE_SERVER_NAME \
--name $AZ_DATABASE_NAME
```

Create an app with a public endpoint assigned

Use the following command to create the app.

Azure CLI

```
az spring app create \
--resource-group $AZ_RESOURCE_GROUP \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--name $AZ_SPRING_APPS_APP_NAME \
--runtime-version=Java_17
--assign-endpoint true
```

Connect Azure Spring Apps to the Azure database

First, install the [Service Connector](#) passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

Azure SQL Database

ⓘ Note

Please make sure Azure CLI use the 64-bit Python, 32-bit Python has compatibility issue with the command's dependency [pyodbc](#). The Python information of Azure CLI can be got with command `az --version`. If it shows `[MSC v.1929 32 bit (Intel)]`, then it means it use 32-bit Python. The solution is to install 64-bit Python and install Azure CLI from [PyPI](#).

Use the following command to create a passwordless connection to the database.

Azure CLI

```
az spring connection create sql \
    --resource-group $AZ_RESOURCE_GROUP \
    --service $AZ_SPRING_APPS_SERVICE_NAME \
    --app $AZ_SPRING_APPS_APP_NAME \
    --target-resource-group $AZ_RESOURCE_GROUP \
    --server $AZ_DATABASE_SERVER_NAME \
    --database $AZ_DATABASE_NAME \
    --system-identity
```

This Service Connector command does the following tasks in the background:

- Enable system-assigned managed identity for the app `$AZ_SPRING_APPS_APP_NAME` hosted by Azure Spring Apps.
- Set the Microsoft Entra admin to current sign-in user.
- Add a database user named `$AZ_SPRING_APPS_SERVICE_NAME/apps/$AZ_SPRING_APPS_APP_NAME` for the managed identity created in step 1 and grant all privileges of the database `$AZ_DATABASE_NAME` to this user.
- Add one configuration to the app `$AZ_SPRING_APPS_APP_NAME`:
`spring.datasource.url`.

Note

If you see the error message `The subscription is not registered to use Microsoft.ServiceLinker`, run the command `az provider register --namespace Microsoft.ServiceLinker` to register the Service Connector resource provider, then run the connection command again.

Build and deploy the app

The following steps describe how to download, configure, build, and deploy the sample application.

1. Use the following command to clone the sample code repository:

Azure SQL Database

Bash

```
git clone https://github.com/Azure-Samples/quickstart-spring-data-jdbc-sql-server passwordless-sample
```

2. Add the following dependency to your *pom.xml* file:

Azure SQL Database

XML

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-identity</artifactId>
    <version>1.5.4</version>
</dependency>
```

There's currently no Spring Cloud Azure starter for Azure SQL Database, but the `azure-identity` dependency is required.

3. Use the following command to update the *application.properties* file:

Azure SQL Database

Bash

```
cat << EOF > passwordless-
sample/src/main/resources/application.properties

logging.level.org.springframework.jdbc.core=DEBUG
spring.sql.init.mode=always

EOF
```

4. Use the following commands to build the project using Maven:

Bash

```
cd passwordless-sample
./mvnw clean package -DskipTests
```

5. Use the following command to deploy the *target/demo-0.0.1-SNAPSHOT.jar* file for the app:

Azure CLI

```
az spring app deploy \
--name $AZ_SPRING_APPS_APP_NAME \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--resource-group $AZ_RESOURCE_GROUP \
--artifact-path target/demo-0.0.1-SNAPSHOT.jar
```

6. Query the app status after deployment by using the following command:

Azure CLI

```
az spring app list \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--resource-group $AZ_RESOURCE_GROUP \
--output table
```

You should see output similar to the following example.

Name	Location	ResourceGroup	Production Deployment
Public Url			Provisioning
Status	CPU	Memory	Running Instance Registered Instance
Persistent Storage			
<app name>	eastus	<resource group>	default
Succeeded	1	2	1/1
-			0/1

Test the application

To test the application, you can use cURL. First, create a new "todo" item in the database by using the following command:

Bash

```
curl --header "Content-Type: application/json" \
--request POST \
--data '{"description":"configuration","details":"congratulations, you
have set up JDBC correctly!","done": "true"}' \
https://${AZ_SPRING_APPS_SERVICE_NAME}-
hellospring.azuremicroservices.io
```

This command returns the created item, as shown in the following example:

JSON

```
{"id":1,"description":"configuration","details":"congratulations, you have  
set up JDBC correctly!","done":true}
```

Next, retrieve the data by using the following cURL request:

Bash

```
curl https://${AZ_SPRING_APPS_SERVICE_NAME}-  
hellospring.azuremicroservices.io
```

This command returns the list of "todo" items, including the item you've created, as shown in the following example:

JSON

```
[{"id":1,"description":"configuration","details":"congratulations, you have  
set up JDBC correctly!","done":true}]
```

Clean up resources

To clean up all resources used during this tutorial, delete the resource group by using the following command:

Azure CLI

```
az group delete \  
--name $AZ_RESOURCE_GROUP \  
--yes
```

Next steps

- [Spring Cloud Azure documentation](#)

Use Spring Data JDBC with Azure Database for MySQL

Article • 10/19/2023

This tutorial demonstrates how to store data in [Azure Database for MySQL](#) database using [Spring Data JDBC](#).

[JDBC](#) is the standard Java API to connect to traditional relational databases.

In this tutorial, we include two authentication methods: Microsoft Entra authentication and MySQL authentication. The **Passwordless** tab shows the Microsoft Entra authentication and the **Password** tab shows the MySQL authentication.

Microsoft Entra authentication is a mechanism for connecting to Azure Database for MySQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

MySQL authentication uses accounts stored in MySQL. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `user` table.

Because these passwords are stored in MySQL, you need to manage the rotation of the passwords by yourself.

Prerequisites

- An Azure subscription - [create one for free](#).
- Java Development Kit (JDK), version 8 or higher.
- Apache Maven.
- Azure CLI.
- MySQL command line client.
- If you don't have a Spring Boot application, create a Maven project with the [Spring Initializr](#). Be sure to select **Maven Project** and, under **Dependencies**, add the **Spring Web**, **Spring Data JDBC**, and **MySQL Driver** dependencies, and then select Java version 8 or higher.
- If you don't have one, create an Azure Database for MySQL Flexible Server instance named `mysqlflexibletest`. For instructions, see [Quickstart: Use the Azure portal to](#)

create an Azure Database for MySQL Flexible Server. Then, create a database named `demo`. For instructions, see [Create and manage databases for Azure Database for MySQL Flexible Server](#).

See the sample application

In this tutorial, you'll code a sample application. If you want to go faster, this application is already coded and available at [https://github.com/Azure-Samples/quickstart-spring-data-jdbc-mysql ↗](https://github.com/Azure-Samples/quickstart-spring-data-jdbc-mysql).

Configure a firewall rule for your MySQL server

Azure Database for MySQL instances are secured by default. They have a firewall that doesn't allow any incoming connection.

To be able to use your database, open the server's firewall to allow the local IP address to access the database server. For more information, see [Manage firewall rules for Azure Database for MySQL - Flexible Server using the Azure portal](#).

If you're connecting to your MySQL server from Windows Subsystem for Linux (WSL) on a Windows computer, you need to add the WSL host IP address to your firewall.

Create a MySQL non-admin user and grant permission

This step will create a non-admin user and grant all permissions on the `demo` database to it.

Passwordless (Recommended)

You can use the following method to create a non-admin user that uses a passwordless connection.

Service Connector (Recommended)

1. Use the following command to install the [Service Connector](#) passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

2. Use the following command to create the Microsoft Entra non-admin user:

Azure CLI

```
az connection create mysql-flexible \
    --resource-group <your_resource_group_name> \
    --connection mysql_conn \
    --target-resource-group <your_resource_group_name> \
    --server mysqlflexibletest \
    --database demo \
    --user-account mysql-identity-
    id=/subscriptions/<your_subscription_id>/resourcegroups/<your_r
    esource_group_name>/providers/Microsoft.ManagedIdentity/userAss
    ignedIdentities/<your_user_assigned_managed_identity_name> \
    --query authInfo.userName \
    --output tsv
```

When the command completes, take note of the username in the console output.

Store data from Azure Database for MySQL

Now that you have an Azure Database for MySQL Flexible server instance, you can store data by using Spring Cloud Azure.

To install the Spring Cloud Azure Starter JDBC MySQL module, add the following dependencies to your *pom.xml* file:

- The Spring Cloud Azure Bill of Materials (BOM):

XML

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.azure.spring</groupId>
            <artifactId>spring-cloud-azure-dependencies</artifactId>
            <version>4.13.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

```
</dependencies>  
</dependencyManagement>
```

ⓘ Note

If you're using Spring Boot 3.x, be sure to set the `spring-cloud-azure-dependencies` version to `5.7.0`. For more information about the `spring-cloud-azure-dependencies` version, see [Which Version of Spring Cloud Azure Should I Use ↗](#).

- The Spring Cloud Azure Starter JDBC MySQL artifact:

XML

```
<dependency>  
  <groupId>com.azure.spring</groupId>  
  <artifactId>spring-cloud-azure-starter-jdbc-mysql</artifactId>  
</dependency>
```

ⓘ Note

Passwordless connections have been supported since version `4.5.0`.

Configure Spring Boot to use Azure Database for MySQL

To store data from Azure Database for MySQL using Spring Data JDBC, follow these steps to configure the application:

1. Configure Azure Database for MySQL credentials by adding the following properties to your `application.properties` configuration file.

Passwordless (Recommended)

properties

```
logging.level.org.springframework.jdbc.core=DEBUG  
  
spring.datasource.url=jdbc:mysql://mysqlflexibletest.mysql.database.azure.com:3306/demo?serverTimezone=UTC  
spring.datasource.username=<your_mysql_ad_non_admin_username>  
spring.datasource.azure.passwordless-enabled=true
```

```
spring.sql.init.mode=always
```

⚠ Warning

The configuration property `spring.sql.init.mode=always` means that Spring Boot will automatically generate a database schema, using the `schema.sql` file that you'll create next, each time the server is started. This feature is great for testing, but remember that it will delete your data at each restart, so you shouldn't use it in production.

The configuration property `spring.datasource.url` has `?serverTimezone=UTC` appended to tell the JDBC driver to use the UTC date format (or Coordinated Universal Time) when connecting to the database. Without this parameter, your Java server wouldn't use the same date format as the database, which would result in an error.

2. Create the `src/main/resources/schema.sql` configuration file to configure the database schema, then add the following contents.

SQL

```
DROP TABLE IF EXISTS todo;
CREATE TABLE todo (id SERIAL PRIMARY KEY, description VARCHAR(255),
details VARCHAR(4096), done BOOLEAN);
```

3. Create a new `Todo` Java class. This class is a domain model mapped onto the `todo` table that will be created automatically by Spring Boot. The following code ignores the `getters` and `setters` methods.

Java

```
import org.springframework.data.annotation.Id;

public class Todo {

    public Todo() {
    }

    public Todo(String description, String details, boolean done) {
        this.description = description;
        this.details = details;
        this.done = done;
    }
}
```

```
@Id  
private Long id;  
  
private String description;  
  
private String details;  
  
private boolean done;  
  
}
```

4. Edit the startup class file to show the following content.

Java

```
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.boot.context.event.ApplicationReadyEvent;  
import org.springframework.context.ApplicationListener;  
import org.springframework.context.annotation.Bean;  
import org.springframework.data.repository.CrudRepository;  
  
import java.util.stream.Stream;  
  
@SpringBootApplication  
public class DemoApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(DemoApplication.class, args);  
    }  
  
    @Bean  
    ApplicationListener<ApplicationReadyEvent>  
    basicsApplicationListener(TodoRepository repository) {  
        return event->repository  
            .saveAll(Stream.of("A", "B", "C").map(name->new  
Todo("configuration", "congratulations, you have set up correctly!",  
true)).toList())  
            .forEach(System.out::println);  
    }  
  
}  
  
interface TodoRepository extends CrudRepository<Todo, Long> {  
}
```

 Tip

In this tutorial, there are no authentication operations in the configurations or the code. However, connecting to Azure services requires authentication. To complete the authentication, you need to use Azure Identity. Spring Cloud Azure uses `DefaultAzureCredential`, which the Azure Identity library provides to help you get credentials without any code changes.

`DefaultAzureCredential` supports multiple authentication methods and determines which method to use at runtime. This approach enables your app to use different authentication methods in different environments (such as local and production environments) without implementing environment-specific code. For more information, see [DefaultAzureCredential](#).

To complete the authentication in local development environments, you can use Azure CLI, Visual Studio Code, PowerShell, or other methods. For more information, see [Azure authentication in Java development environments](#). To complete the authentication in Azure hosting environments, we recommend using user-assigned managed identity. For more information, see [What are managed identities for Azure resources?](#)

5. Start the application. The application stores data into the database. You'll see logs similar to the following example:

shell

```
2023-02-01 10:22:36.701 DEBUG 7948 --- [main]
o.s.jdbc.core.JdbcTemplate : Executing prepared SQL statement [INSERT
INTO todo (description, details, done) VALUES (?, ?, ?)]
com.example.demo.Todo@4bdb04c8
```

Deploy to Azure Spring Apps

Now that you have the Spring Boot application running locally, it's time to move it to production. [Azure Spring Apps](#) makes it easy to deploy Spring Boot applications to Azure without any code changes. 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. To deploy your application to Azure Spring Apps, see [Deploy your first application to Azure Spring Apps](#).

Next steps

Azure for Spring developers

Spring Cloud Azure MySQL Samples

Use Spring Data JDBC with Azure Database for PostgreSQL

Article • 10/19/2023

This tutorial demonstrates how to store data in an [Azure Database for PostgreSQL](#) database using [Spring Data JDBC](#).

[JDBC](#) is the standard Java API to connect to traditional relational databases.

In this tutorial, we include two authentication methods: Microsoft Entra authentication and PostgreSQL authentication. The **Passwordless** tab shows the Microsoft Entra authentication and the **Password** tab shows the PostgreSQL authentication.

Microsoft Entra authentication is a mechanism for connecting to Azure Database for PostgreSQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

PostgreSQL authentication uses accounts stored in PostgreSQL. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `user` table. Because these passwords are stored in PostgreSQL, you need to manage the rotation of the passwords by yourself.

Prerequisites

- An Azure subscription - [create one for free](#).
- [Java Development Kit \(JDK\)](#), version 8 or higher.
- [Apache Maven](#).
- [Azure CLI](#).
- [PostgreSQL command line client](#).
- If you don't have a Spring Boot application, create a Maven project with the [Spring Initializr](#). Be sure to select **Maven Project** and, under **Dependencies**, add the **Spring Web**, **Spring Data JDBC**, and **PostgreSQL Driver** dependencies, and then select Java version 8 or higher.
- If you don't have one, create an Azure Database for PostgreSQL Flexible Server instance named `postgresqlflexibletest` and a database named `demo`. For

instructions, see [Quickstart: Create an Azure Database for PostgreSQL - Flexible Server in the Azure portal](#).

See the sample application

In this tutorial, you'll code a sample application. If you want to go faster, this application is already coded and available at <https://github.com/Azure-Samples/quickstart-spring-data-jdbc-postgresql>.

Configure a firewall rule for your PostgreSQL server

Azure Database for PostgreSQL instances are secured by default. They have a firewall that doesn't allow any incoming connection.

To be able to use your database, open the server's firewall to allow the local IP address to access the database server. For more information, see [Firewall rules in Azure Database for PostgreSQL - Flexible Server](#).

If you're connecting to your PostgreSQL server from Windows Subsystem for Linux (WSL) on a Windows computer, you need to add the WSL host ID to your firewall.

Create a PostgreSQL non-admin user and grant permission

Next, create a non-admin user and grant all permissions to the database.

Passwordless (Recommended)

You can use the following method to create a non-admin user that uses a passwordless connection.

Service Connector (Recommended)

1. Use the following command to install the [Service Connector](#) passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

2. Use the following command to create the Microsoft Entra non-admin user:

Azure CLI

```
az connection create postgres-flexible \
    --resource-group <your_resource_group_name> \
    --connection postgres_conn \
    --target-resource-group <your_resource_group_name> \
    --server postgresqlflexibletest \
    --database demo \
    --user-account \
    --query authInfo.userName \
    --output tsv
```

When the command completes, take note of the username in the console output.

Store data from Azure Database for PostgreSQL

Now that you have an Azure Database for PostgreSQL Flexible Server instance, you can store data by using Spring Cloud Azure.

To install the Spring Cloud Azure Starter JDBC PostgreSQL module, add the following dependencies to your *pom.xml* file:

- The Spring Cloud Azure Bill of Materials (BOM):

XML

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.azure.spring</groupId>
            <artifactId>spring-cloud-azure-dependencies</artifactId>
            <version>4.13.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

ⓘ Note

If you're using Spring Boot 3.x, be sure to set the `spring-cloud-azure-dependencies` version to `5.7.0`. For more information about the `spring-cloud-azure-dependencies` version, see [Which Version of Spring Cloud Azure Should I Use](#).

- The Spring Cloud Azure Starter JDBC PostgreSQL artifact:

XML

```
<dependency>
  <groupId>com.azure.spring</groupId>
  <artifactId>spring-cloud-azure-starter-jdbc-postgresql</artifactId>
</dependency>
```

ⓘ Note

Passwordless connections have been supported since version `4.5.0`.

Configure Spring Boot to use Azure Database for PostgreSQL

To store data from Azure Database for PostgreSQL using Spring Data JDBC, follow these steps to configure the application:

1. Configure Azure Database for PostgreSQL credentials by adding the following properties to your `application.properties` configuration file.

Passwordless (Recommended)

properties

```
logging.level.org.springframework.jdbc.core=DEBUG

spring.datasource.url=jdbc:postgresql://postgresqlflexibletest.post
gres.database.azure.com:5432/demo?sslmode=require
spring.datasource.username=<your_postgresql_ad_non_admin_username>
spring.datasource.azure.passwordless-enabled=true

spring.sql.init.mode=always
```

⚠ Warning

The configuration property `spring.sql.init.mode=always` means that Spring Boot will automatically generate a database schema, using the `schema.sql` file that you'll create next, each time the server is started. This feature is great for testing, but remember that it will delete your data at each restart, so you shouldn't use it in production.

2. Create the `src/main/resources/schema.sql` configuration file to configure the database schema, then add the following contents.

SQL

```
DROP TABLE IF EXISTS todo;
CREATE TABLE todo (id SERIAL PRIMARY KEY, description VARCHAR(255),
details VARCHAR(4096), done BOOLEAN);
```

3. Create a new `Todo` Java class. This class is a domain model mapped onto the `todo` table that will be created automatically by Spring Boot. The following code ignores the `getters` and `setters` methods.

Java

```
import org.springframework.data.annotation.Id;

public class Todo {

    public Todo() {
    }

    public Todo(String description, String details, boolean done) {
        this.description = description;
        this.details = details;
        this.done = done;
    }

    @Id
    private Long id;

    private String description;

    private String details;

    private boolean done;
}
```

4. Edit the startup class file to show the following content.

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.data.repository.CrudRepository;

import java.util.stream.Stream;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Bean
    ApplicationListener<ApplicationReadyEvent>
    basicsApplicationListener(TodoRepository repository) {
        return event->repository
            .saveAll(Stream.of("A", "B", "C").map(name->new
Todo("configuration", "congratulations, you have set up correctly!",
true)).toList())
            .forEach(System.out::println);
    }

}

interface TodoRepository extends CrudRepository<Todo, Long> {
```

💡 Tip

In this tutorial, there are no authentication operations in the configurations or the code. However, connecting to Azure services requires authentication. To complete the authentication, you need to use Azure Identity. Spring Cloud Azure uses `DefaultAzureCredential`, which the Azure Identity library provides to help you get credentials without any code changes.

`DefaultAzureCredential` supports multiple authentication methods and determines which method to use at runtime. This approach enables your app to use different authentication methods in different environments (such as local and production environments) without implementing environment-specific code. For more information, see [DefaultAzureCredential](#).

To complete the authentication in local development environments, you can use Azure CLI, Visual Studio Code, PowerShell, or other methods. For more information, see [Azure authentication in Java development environments](#). To complete the authentication in Azure hosting environments, we recommend using user-assigned managed identity. For more information, see [What are managed identities for Azure resources?](#)

5. Start the application. The application stores data into the database. You'll see logs similar to the following example:

```
shell
```

```
2023-02-01 10:22:36.701 DEBUG 7948 --- [main]
o.s.jdbc.core.JdbcTemplate : Executing prepared SQL statement [INSERT
INTO todo (description, details, done) VALUES (?, ?, ?)]
com.example.demo.Todo@4bdb04c8
```

Deploy to Azure Spring Apps

Now that you have the Spring Boot application running locally, it's time to move it to production. [Azure Spring Apps](#) makes it easy to deploy Spring Boot applications to Azure without any code changes. 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. To deploy your application to Azure Spring Apps, see [Deploy your first application to Azure Spring Apps](#).

Next steps

[Azure for Spring developers](#)[Spring Cloud Azure PostgreSQL Samples](#)

Configure passwordless database connections for Java apps on Oracle WebLogic Servers

Article • 10/19/2023

This article shows you how to configure passwordless database connections for Java apps on Oracle WebLogic Server offers with the Azure portal.

In this guide, you accomplish the following tasks:

- ✓ Provision database resources using Azure CLI.
- ✓ Enable the Microsoft Entra administrator in the database.
- ✓ Provision a user-assigned managed identity and create a database user for it.
- ✓ Configure a passwordless database connection in Oracle WebLogic offers with the Azure portal.
- ✓ Validate the database connection.

The offers support passwordless connections for PostgreSQL, MySQL and Azure SQL databases.

Prerequisites

- If you don't have an [Azure subscription](#), create a [free account](#) before you begin.
- Use [Azure Cloud Shell](#) using the Bash environment; make sure the Azure CLI version is 2.43.0 or higher.
A blue rectangular button with a white 'A' icon and the text 'Launch Cloud Shell'. To the right of the button is a small blue square with a white right-pointing arrow.
- If you prefer, [install the Azure CLI 2.43.0 or higher](#) to run Azure CLI commands.
 - If you're using a local install, sign in with Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. See [Sign in with Azure CLI](#) for other sign-in options.
 - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with Azure CLI](#).
 - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).
- Ensure the Azure identity you use to sign in and complete this article has either the [Owner](#) role in the current subscription or the [Contributor](#) and [User Access](#)

Administrator roles in the current subscription. For an overview of Azure roles, see [What is Azure role-based access control \(Azure RBAC\)?](#) For details on the specific roles required by Oracle WebLogic marketplace offer, see [Azure built-in roles](#).

Create a resource group

Create a resource group with [az group create](#). Because resource groups must be unique within a subscription, pick a unique name. An easy way to have unique names is to use a combination of your initials, today's date, and some identifier. For example, `abc1228rg`. This example creates a resource group named `abc1228rg` in the `eastus` location:

Azure CLI

```
export RESOURCE_GROUP_NAME="abc1228rg"
az group create \
--name ${RESOURCE_GROUP_NAME} \
--location eastus
```

Create a database server and a database

MySQL Flexible Server

Create a flexible server with the [az mysql flexible-server create](#) command. This example creates a flexible server named `mysql120221201` with admin user `azureuser` and admin password `Secret123456`. Replace the password with yours. For more information, see [Create an Azure Database for MySQL Flexible Server using Azure CLI](#).

Azure CLI

```
export MYSQL_NAME="mysql120221201"
export MYSQL_ADMIN_USER="azureuser"
export MYSQL_ADMIN_PASSWORD="Secret123456"

az mysql flexible-server create \
--resource-group $RESOURCE_GROUP_NAME \
--name $MYSQL_NAME \
--location eastus \
--admin-user $MYSQL_ADMIN_USER \
--admin-password $MYSQL_ADMIN_PASSWORD \
--public-access 0.0.0.0 \
--tier Burstable \
--sku-name Standard_B1ms
```

Create a database with [az mysql flexible-server db create](#).

Azure CLI

```
export DATABASE_NAME="contoso"

# create mysql database
az mysql flexible-server db create \
    --resource-group $RESOURCE_GROUP_NAME \
    --server-name $MYSQL_NAME \
    --database-name $DATABASE_NAME
```

When the command completes, you should see output similar to the following example:

Output

```
Creating database with utf8 charset and utf8_general_ci collation
{
  "charset": "utf8",
  "collation": "utf8_general_ci",
  "id": "/subscriptions/contoso-
hashcode/resourceGroups/abc1228rg/providers/Microsoft.DBforMySQL/flexibl
eServers/mysql20221201/databases/contoso",
  "name": "contoso",
  "resourceGroup": "abc1228rg",
  "systemData": null,
  "type": "Microsoft.DBforMySQL/flexibleServers/databases"
}
```

Configure a Microsoft Entra administrator to your database

Now that you've created the database, you need to make it ready to support passwordless connections. A passwordless connection requires a combination of managed identities for Azure resources and Microsoft Entra authentication. For an overview of managed identities for Azure resources, see [What are managed identities for Azure resources?](#)

MySQL Flexible Server

For information on how MySQL Flexible Server interacts with managed identities, see [Use Microsoft Entra ID for authentication with MySQL](#).

The following example configures the current Azure CLI user as a Microsoft Entra administrator account. To enable Azure authentication, it's necessary to assign an identity to MySQL Flexible Server.

First, create a managed identity with [az identity create](#) and assign the identity to MySQL server with [az mysql flexible-server identity assign](#).

Azure CLI

```
export MYSQL_UMI_NAME="id-mysql-aad-20221205"

# create a User Assigned Managed Identity for MySQL to be used for AAD
# authentication
az identity create \
    --resource-group $RESOURCE_GROUP_NAME \
    --name $MYSQL_UMI_NAME

## assign the identity to the MySQL server
az mysql flexible-server identity assign \
    --resource-group $RESOURCE_GROUP_NAME \
    --server-name $MYSQL_NAME \
    --identity $MYSQL_UMI_NAME
```

Then, set the current Azure CLI user as the Microsoft Entra administrator account with [az mysql flexible-server ad-admin create](#).

Azure CLI

```
export CURRENT_USER=$(az account show --query user.name --output tsv)
export CURRENT_USER_OBJECTID=$(az ad signed-in-user show --query id --
output tsv)

az mysql flexible-server ad-admin create \
    --resource-group $RESOURCE_GROUP_NAME \
    --server-name $MYSQL_NAME \
    --object-id $CURRENT_USER_OBJECTID \
    --display-name $CURRENT_USER \
    --identity $MYSQL_UMI_NAME
```

Create a user-assigned managed identity

Next, in Azure CLI, create an identity in your subscription by using the [az identity create](#) command. You use this managed identity to connect to your database.

Azure CLI

```
az identity create \
--resource-group ${RESOURCE_GROUP_NAME} \
--name myManagedIdentity
```

To configure the identity in the following steps, use the [az identity show](#) command to store the identity's client ID in a shell variable.

Azure CLI

```
# Get client ID of the user-assigned identity
export CLIENT_ID=$(az identity show \
--resource-group ${RESOURCE_GROUP_NAME} \
--name myManagedIdentity \
--query clientId \
--output tsv)
```

Create a database user for your managed identity

MySQL Flexible Server

Now, connect as the Microsoft Entra administrator user to your MySQL database, and create a MySQL user for your managed identity.

First, you're required to create a firewall rule to access the MySQL server from your CLI client. Run the following commands to get your current IP address.

Bash

```
export MY_IP=$(curl http://whatismyip.akamai.com)
```

If you're working on Windows Subsystem for Linux (WSL) with VPN enabled, the following command may return an incorrect IPv4 address. One way to get your IPv4 address is by visiting [whatismyipaddress.com](#). In any case, set the environment variable `MY_IP` as the IPv4 address from which you want to connect to the database.

Create a temporary firewall rule with [az mysql flexible-server firewall-rule create](#).

Azure CLI

```
az mysql flexible-server firewall-rule create \
--resource-group $RESOURCE_GROUP_NAME \
--name $MYSQL_NAME \
```

```
--rule-name AllowCurrentMachineToConnect \
--start-ip-address ${MY_IP} \
--end-ip-address ${MY_IP}
```

Then, prepare an *.sql* file to create a database user for the managed identity. The following example adds a user with login name `identity-contoso` and grants the user privileges to access database `contoso`.

Bash

```
export IDENTITY_LOGIN_NAME="identity-contoso"

cat <<EOF >createuser.sql
SET aad_auth_validate_oids_in_tenant = OFF;
DROP USER IF EXISTS '${IDENTITY_LOGIN_NAME}'@'%';
CREATE AADUSER '${IDENTITY_LOGIN_NAME}' IDENTIFIED BY '${CLIENT_ID}';
GRANT ALL PRIVILEGES ON ${DATABASE_NAME}.* TO
'${IDENTITY_LOGIN_NAME}'@'%';
FLUSH privileges;
EOF
```

Execute the *.sql* file with the command `az mysql flexible-server execute`. You can get your access token with the command `az account get-access-token`.

Azure CLI

```
export RDBMS_ACCESS_TOKEN=$(az account get-access-token \
    --resource-type oss-rdbms \
    --query accessToken \
    --output tsv)

az mysql flexible-server execute \
    --name ${MYSQL_NAME} \
    --admin-user ${CURRENT_USER} \
    --admin-password ${RDBMS_ACCESS_TOKEN} \
    --file-path "createuser.sql"
```

You may be prompted to install the `rdbms-connect` extension, as shown in the following output. Press `y` to continue. If you're not working with the `root` user, you need to input the user password.

Output

```
The command requires the extension rdbms-connect. Do you want to install it now? The command will continue to run after the extension is installed. (Y/n): y
Run 'az config set extension.use_dynamic_install=yes_without_prompt' to allow installing extensions without prompt.
```

```
This extension depends on gcc, libpq-dev, python3-dev and they will be  
installed first.
```

```
[sudo] password for user:
```

If the *.sql* file executes successfully, you find output that is similar to the following example:

Output

```
Running *.sql* file 'createuser.sql'...  
Successfully executed the file.  
Closed the connection to mysql20221201
```

The managed identity `myManagedIdentity` now has access to the database when authenticating with the username `identity-contoso`.

If you no longer want to access the server from this IP address, you can remove the firewall rule by using the following command.

Azure CLI

```
az mysql flexible-server firewall-rule delete \  
  --resource-group $RESOURCE_GROUP_NAME \  
  --name $MYSQL_NAME \  
  --rule-name AllowCurrentMachineToConnect \  
  --yes
```

Finally, use the following command to get the connection string that you use in the next section.

Azure CLI

```
export  
CONNECTION_STRING="jdbc:mysql://${MYSQL_NAME}.mysql.database.azure.com:3  
306/${DATABASE_NAME}?useSSL=true"  
echo ${CONNECTION_STRING}
```

Configure a passwordless database connection for Oracle WebLogic Server on Azure VMs

This section shows you how to configure the passwordless data source connection using the Azure Marketplace offers for Oracle WebLogic Server.

First, begin the process of deploying an offer. The following offers support passwordless database connections:

- [Oracle WebLogic Server on Azure Kubernetes Service ↗](#)
 - [Quickstart](#)
- [Oracle WebLogic Server Cluster on VMs ↗](#)
 - [Quickstart](#)
- [Oracle WebLogic Server with Admin Server on VMs ↗](#)
 - [Quickstart](#)
- [Oracle WebLogic Server Dynamic Cluster on VMs ↗](#)
 - [Quickstart](#)

Fill in the required information in the **Basics** pane and other panes if you want to enable the features. When you reach the **Database** pane, fill in the passwordless configuration as shown in the following steps.

MySQL Flexible Server

1. For **Connect to database?**, select **Yes**.
2. Under **Connection settings**, for **Choose database type**, open the dropdown menu and then select **MySQL (with support for passwordless connection)**.
3. For **JNDI Name**, input *testpasswordless* or your expected value.
4. For **DataSource Connection String**, input the connection string you obtained in the last section.
5. For **Database username**, input the database user name of your managed identity (the value of `${IDENTITY_LOGIN_NAME}`). In this example, the value is `identity-contoso`.
6. Select **Use passwordless datasource connection**.
7. For **User assigned managed identity**, select the managed identity you created previously. In this example, its name is `myManagedIdentity`.

The **Connection settings** section should look like the following screenshot, which uses [Oracle WebLogic Server Cluster on VMs ↗](#) as an example.

Connection settings

Choose database type * ⓘ MySQL (with support for passwordless connection) ▼

Info To support passwordless connection and various functionalities, the offer will upgrade the [Oracle WebLogic Server MySQL driver](#) with recent [MySQL Connector Java driver](#).

JNDI Name * ⓘ	testpasswordless ✓
DataSource Connection String * ⓘ	jdbc:mysql://mysql20221201.mysql.database.azure.com:3306/checklist... ✓
Global transactions protocol * ⓘ	OnePhaseCommit ▼
Database username * ⓘ	identity-contoso ✓
Use passwordless datasource connection ⓘ	<input checked="" type="checkbox"/> ✓

User assigned managed identity

Select a user assigned identity that has access to your database. For how to create a database user for your managed identity, see <https://aka.ms/javaee-db-identity>.

+ Add - Remove

Name	resource group	subscription
<input type="checkbox"/> myManagedIdentity	mydbrg20221201	↑ 🔍

You've now finished configuring the passwordless connection. You can continue to fill in the following panes or select **Review + create**, then **Create** to deploy the offer.

Verify the database connection

The database connection is configured successfully if the offer deployment completes without error.

Continuing to take [Oracle WebLogic Server Cluster on VMs](#) as an example, after the deployment completes, follow these steps in the Azure portal to find the Admin console URL.

1. Find the resource group in which you deployed WLS.
2. Under **Settings**, select **Deployments**.
3. Select the deployment with the longest **Duration**. This deployment should be at the bottom of the list.
4. Select **Outputs**.
5. The URL of the WebLogic Administration Console is the value of the **adminConsoleUrl** output.

6. Copy the value of the output variable `adminConsoleUrl`.
7. Paste the value into your browser address bar and press `Enter` to open the sign-in page of the WebLogic Administration Console.

Use the following steps to verify the database connection.

1. Sign in to the WebLogic Administration Console with the username and password you provided on the **Basics** pane.
2. Under the **Domain Structure**, select **Services**, **Data Sources**, then **testpasswordless**.
3. Select the **Monitoring** tab, where the state of the data source is *Running*, as shown in the following screenshot.

The screenshot shows the MySQL Flexible Server administration interface. On the left, there's a navigation tree under 'Domain Structure' for the 'wlsd' domain, with 'Data Sources' selected. The main panel shows the 'Monitoring' tab for the 'testpasswordless' data source. A table lists deployed instances, with one row for 'msp1' showing 'Enabled' as true and 'State' as 'Running'. A magnifying glass icon is visible at the bottom right of the table.

Server	Enabled	State	JDBC Driver
msp1	true	Running	com.mysql.cj.jdbc.Driver

4. Select the **Testing** tab, then select the radio button next to the desired server.
5. Select **Test Data Source**. You should see a message indicating a successful test, as shown in the following screenshot.

Messages

✓ Test of testpasswordless on server admin was successful.

Settings for testpasswordless

Configuration Targets **Monitoring** Control Security Notes

Statistics **Testing**

Use this page to test database connections in this JDBC data source.

▶ **Customize this table**

Test Data Source (Filtered - More Columns Exist)

Test Data Source		Showing 1 to 1 of 1 Previous Next
	Server	State
<input type="radio"/>	admin	Running

Test Data Source Showing 1 to 1 of 1 Previous | Next 

Clean up resources

If you don't need these resources, you can delete them by doing the following commands:

Azure CLI

```
az group delete --name ${RESOURCE_GROUP_NAME}  
az group delete --name <resource-group-name-that-deploys-the-offer>
```

Next steps

Learn more about running WLS on AKS or virtual machines by following these links:

[WLS on AKS](#)

[WLS on virtual machines](#)

[Passwordless Connections Samples for Java Apps](#)

Tutorial: Deploy a Spring application to Azure Spring Apps with a passwordless connection to an Azure database

Article • 04/24/2023

This article shows you how to use passwordless connections to Azure databases in Spring Boot applications deployed to Azure Spring Apps.

In this tutorial, you complete the following tasks using the Azure portal or the Azure CLI. Both methods are explained in the following procedures.

- ✓ Provision an instance of Azure Spring Apps.
- ✓ Build and deploy apps to Azure Spring Apps.
- ✓ Run apps connected to Azure databases using managed identity.

ⓘ Note

This tutorial doesn't work for R2DBC.

Prerequisites

- An Azure subscription. If you don't already have one, create a [free account](#) before you begin.
- [Azure CLI](#) 2.45.0 or higher required.
- The Azure Spring Apps extension. You can install the extension by using the command: `az extension add --name spring`.
- [Java Development Kit \(JDK\)](#), version 8, 11, or 17.
- A [Git](#) client.
- [cURL](#) or a similar HTTP utility to test functionality.
- MySQL command line client if you choose to run Azure Database for MySQL. You can connect to your server with Azure Cloud Shell using a popular client tool, the [mysql.exe](#) command-line tool. Alternatively, you can use the `mysql` command line in your local environment.
- [ODBC Driver 18 for SQL Server](#) if you choose to run Azure SQL Database.

Prepare the working environment

First, set up some environment variables by using the following commands:

Bash

```
export AZ_RESOURCE_GROUP=passwordless-tutorial-rg
export AZ_DATABASE_SERVER_NAME=<YOUR_DATABASE_SERVER_NAME>
export AZ_DATABASE_NAME=demodb
export AZ_LOCATION=<YOUR_AZURE_REGION>
export AZ_SPRING_APPS_SERVICE_NAME=<YOUR_AZURE_SPRING_APPS_SERVICE_NAME>
export AZ_SPRING_APPS_APP_NAME=hellospring
export AZ_DB_ADMIN_USERNAME=<YOUR_DB_ADMIN_USERNAME>
export AZ_DB_ADMIN_PASSWORD=<YOUR_DB_ADMIN_PASSWORD>
export AZ_USER_IDENTITY_NAME=<YOUR_USER_ASSIGNED_MANAGEMED_IDENTITY_NAME>
```

Replace the placeholders with the following values, which are used throughout this article:

- <YOUR_DATABASE_SERVER_NAME>: The name of your Azure Database server, which should be unique across Azure.
- <YOUR_AZURE_REGION>: The Azure region you want to use. You can use `eastus` by default, but we recommend that you configure a region closer to where you live. You can see the full list of available regions by using `az account list-locations`.
- <YOUR_AZURE_SPRING_APPS_SERVICE_NAME>: The name of your Azure Spring Apps instance. The name must be between 4 and 32 characters long and can contain only lowercase letters, numbers, and hyphens. The first character of the service name must be a letter and the last character must be either a letter or a number.
- <AZ_DB_ADMIN_USERNAME>: The admin username of your Azure database server.
- <AZ_DB_ADMIN_PASSWORD>: The admin password of your Azure database server.
- <YOUR_USER_ASSIGNED_MANAGEMED_IDENTITY_NAME>: The name of your user assigned managed identity server, which should be unique across Azure.

Provision an instance of Azure Spring Apps

Use the following steps to provision an instance of Azure Spring Apps.

1. Update Azure CLI with the Azure Spring Apps extension by using the following command:

Azure CLI

```
az extension update --name spring
```

2. Sign in to the Azure CLI and choose your active subscription by using the following commands:

```
Azure CLI
```

```
az login  
az account list --output table  
az account set --subscription <name-or-ID-of-subscription>
```

3. Use the following commands to create a resource group to contain your Azure Spring Apps service and an instance of the Azure Spring Apps service:

```
Azure CLI
```

```
az group create \  
  --name $AZ_RESOURCE_GROUP \  
  --location $AZ_LOCATION  
az spring create \  
  --resource-group $AZ_RESOURCE_GROUP \  
  --name $AZ_SPRING_APPS_SERVICE_NAME
```

Create an Azure database instance

Use the following steps to provision an Azure Database instance.

```
Azure SQL Database
```

1. Create an Azure SQL Database server by using the following command:

```
Azure CLI
```

```
az sql server create \  
  --location $AZ_LOCATION \  
  --resource-group $AZ_RESOURCE_GROUP \  
  --name $AZ_DATABASE_SERVER_NAME \  
  --admin-user $AZ_DB_ADMIN_USERNAME \  
  --admin-password $AZ_DB_ADMIN_PASSWORD
```

2. The SQL server is empty, so create a new database by using the following command:

```
Azure CLI
```

```
az sql db create \  
  --resource-group $AZ_RESOURCE_GROUP \  
  --name $AZ_DATABASE_NAME
```

```
--server $AZ_DATABASE_SERVER_NAME \
--name $AZ_DATABASE_NAME
```

Create an app with a public endpoint assigned

Use the following command to create the app.

Azure CLI

```
az spring app create \
--resource-group $AZ_RESOURCE_GROUP \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--name $AZ_SPRING_APPS_APP_NAME \
--runtime-version=Java_17
--assign-endpoint true
```

Connect Azure Spring Apps to the Azure database

First, install the [Service Connector](#) passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

Azure SQL Database

ⓘ Note

Please make sure Azure CLI use the 64-bit Python, 32-bit Python has compatibility issue with the command's dependency [pyodbc](#). The Python information of Azure CLI can be got with command `az --version`. If it shows `[MSC v.1929 32 bit (Intel)]`, then it means it use 32-bit Python. The solution is to install 64-bit Python and install Azure CLI from [PyPI](#).

Use the following command to create a passwordless connection to the database.

Azure CLI

```
az spring connection create sql \
    --resource-group $AZ_RESOURCE_GROUP \
    --service $AZ_SPRING_APPS_SERVICE_NAME \
    --app $AZ_SPRING_APPS_APP_NAME \
    --target-resource-group $AZ_RESOURCE_GROUP \
    --server $AZ_DATABASE_SERVER_NAME \
    --database $AZ_DATABASE_NAME \
    --system-identity
```

This Service Connector command does the following tasks in the background:

- Enable system-assigned managed identity for the app `$AZ_SPRING_APPS_APP_NAME` hosted by Azure Spring Apps.
- Set the Microsoft Entra admin to current sign-in user.
- Add a database user named `$AZ_SPRING_APPS_SERVICE_NAME/apps/$AZ_SPRING_APPS_APP_NAME` for the managed identity created in step 1 and grant all privileges of the database `$AZ_DATABASE_NAME` to this user.
- Add one configuration to the app `$AZ_SPRING_APPS_APP_NAME`:
`spring.datasource.url`.

ⓘ Note

If you see the error message `The subscription is not registered to use Microsoft.ServiceLinker`, run the command `az provider register --namespace Microsoft.ServiceLinker` to register the Service Connector resource provider, then run the connection command again.

Build and deploy the app

The following steps describe how to download, configure, build, and deploy the sample application.

1. Use the following command to clone the sample code repository:

Azure SQL Database

Bash

```
git clone https://github.com/Azure-Samples/quickstart-spring-data-jdbc-sql-server passwordless-sample
```

2. Add the following dependency to your *pom.xml* file:

Azure SQL Database

XML

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-identity</artifactId>
    <version>1.5.4</version>
</dependency>
```

There's currently no Spring Cloud Azure starter for Azure SQL Database, but the `azure-identity` dependency is required.

3. Use the following command to update the *application.properties* file:

Azure SQL Database

Bash

```
cat << EOF > passwordless-
sample/src/main/resources/application.properties

logging.level.org.springframework.jdbc.core=DEBUG
spring.sql.init.mode=always

EOF
```

4. Use the following commands to build the project using Maven:

Bash

```
cd passwordless-sample
./mvnw clean package -DskipTests
```

5. Use the following command to deploy the *target/demo-0.0.1-SNAPSHOT.jar* file for the app:

Azure CLI

```
az spring app deploy \
--name $AZ_SPRING_APPS_APP_NAME \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--resource-group $AZ_RESOURCE_GROUP \
--artifact-path target/demo-0.0.1-SNAPSHOT.jar
```

6. Query the app status after deployment by using the following command:

Azure CLI

```
az spring app list \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--resource-group $AZ_RESOURCE_GROUP \
--output table
```

You should see output similar to the following example.

Name	Location	ResourceGroup	Production Deployment
Public Url			Provisioning
Status	CPU	Memory	Running Instance Registered Instance
Persistent Storage			
<app name>	eastus	<resource group>	default
Succeeded	1	2	1/1
-			0/1

Test the application

To test the application, you can use cURL. First, create a new "todo" item in the database by using the following command:

Bash

```
curl --header "Content-Type: application/json" \
--request POST \
--data '{"description":"configuration","details":"congratulations, you
have set up JDBC correctly!","done": "true"}' \
https://${AZ_SPRING_APPS_SERVICE_NAME}-
hellospring.azuremicroservices.io
```

This command returns the created item, as shown in the following example:

JSON

```
{"id":1,"description":"configuration","details":"congratulations, you have  
set up JDBC correctly!","done":true}
```

Next, retrieve the data by using the following cURL request:

Bash

```
curl https://${AZ_SPRING_APPS_SERVICE_NAME}-  
hellospring.azuremicroservices.io
```

This command returns the list of "todo" items, including the item you've created, as shown in the following example:

JSON

```
[{"id":1,"description":"configuration","details":"congratulations, you have  
set up JDBC correctly!","done":true}]
```

Clean up resources

To clean up all resources used during this tutorial, delete the resource group by using the following command:

Azure CLI

```
az group delete \  
--name $AZ_RESOURCE_GROUP \  
--yes
```

Next steps

- [Spring Cloud Azure documentation](#)

Use Spring Data JPA with Azure Database for MySQL

Article • 10/19/2023

This tutorial demonstrates how to store data in [Azure Database for MySQL](#) database using [Spring Data JPA](#).

The [Java Persistence API \(JPA\)](#) is the standard Java API for object-relational mapping.

In this tutorial, we include two authentication methods: Microsoft Entra authentication and MySQL authentication. The **Passwordless** tab shows the Microsoft Entra authentication and the **Password** tab shows the MySQL authentication.

Microsoft Entra authentication is a mechanism for connecting to Azure Database for MySQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

MySQL authentication uses accounts stored in MySQL. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `user` table.

Because these passwords are stored in MySQL, you need to manage the rotation of the passwords by yourself.

Prerequisites

- An Azure subscription - [create one for free](#).
- [Java Development Kit \(JDK\)](#), version 8 or higher.
- [Apache Maven](#).
- [Azure CLI](#).
- [MySQL command line client](#).
- If you don't have a Spring Boot application, create a Maven project with the [Spring Initializr](#). Be sure to select **Maven Project** and, under **Dependencies**, add the **Spring Web**, **Spring Data JPA**, and **MySQL Driver** dependencies, and then select Java version 8 or higher.
- If you don't have one, create an Azure Database for MySQL Flexible Server instance named `mysqlflexibletest`. For instructions, see [Quickstart: Use the Azure portal to](#)

create an Azure Database for MySQL Flexible Server. Then, create a database named `demo`. For instructions, see [Create and manage databases for Azure Database for MySQL Flexible Server](#).

ⓘ Important

To use passwordless connections, create a Microsoft Entra admin user for your Azure Database for MySQL instance. For instructions, see the [Configure the Microsoft Entra Admin](#) section of [Set up Microsoft Entra authentication for Azure Database for MySQL - Flexible Server](#).

See the sample application

In this tutorial, you'll code a sample application. If you want to go faster, this application is already coded and available at [https://github.com/Azure-Samples/quickstart-spring-data-jpa-mysql ↗](https://github.com/Azure-Samples/quickstart-spring-data-jpa-mysql).

Configure a firewall rule for your MySQL server

Azure Database for MySQL instances are secured by default. They have a firewall that doesn't allow any incoming connection.

To be able to use your database, open the server's firewall to allow the local IP address to access the database server. For more information, see [Manage firewall rules for Azure Database for MySQL - Flexible Server using the Azure portal](#).

If you're connecting to your MySQL server from Windows Subsystem for Linux (WSL) on a Windows computer, you need to add the WSL host IP address to your firewall.

Create a MySQL non-admin user and grant permission

This step will create a non-admin user and grant all permissions on the `demo` database to it.

>Passwordless (Recommended)

You can use the following method to create a non-admin user that uses a passwordless connection.

Service Connector (Recommended)

1. Use the following command to install the [Service Connector](#) passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

2. Use the following command to create the Microsoft Entra non-admin user:

Azure CLI

```
az connection create mysql-flexible \
    --resource-group <your_resource_group_name> \
    --connection mysql_conn \
    --target-resource-group <your_resource_group_name> \
    --server mysqlflexibletest \
    --database demo \
    --user-account mysql-identity-
    id=/subscriptions/<your_subscription_id>/resourcegroups/<your_r
    esource_group_name>/providers/Microsoft.ManagedIdentity/userAss
    ignedIdentities/<your_user_assigned_managed_identity_name> \
    --query authInfo.userName \
    --output tsv
```

When the command completes, take note of the username in the console output.

Store data from Azure Database for MySQL

Now that you have an Azure Database for MySQL Flexible server instance, you can store data by using Spring Cloud Azure.

To install the Spring Cloud Azure Starter JDBC MySQL module, add the following dependencies to your *pom.xml* file:

- The Spring Cloud Azure Bill of Materials (BOM):

XML

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.azure.spring</groupId>
      <artifactId>spring-cloud-azure-dependencies</artifactId>
      <version>4.13.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

ⓘ Note

If you're using Spring Boot 3.x, be sure to set the `spring-cloud-azure-dependencies` version to `5.7.0`. For more information about the `spring-cloud-azure-dependencies` version, see [Which Version of Spring Cloud Azure Should I Use](#).

- The Spring Cloud Azure Starter JDBC MySQL artifact:

XML

```
<dependency>
  <groupId>com.azure.spring</groupId>
  <artifactId>spring-cloud-azure-starter-jdbc-mysql</artifactId>
</dependency>
```

ⓘ Note

Passwordless connections have been supported since version `4.5.0`.

Configure Spring Boot to use Azure Database for MySQL

To store data from Azure Database for MySQL using Spring Data JPA, follow these steps to configure the application:

1. Configure Azure Database for MySQL credentials by adding the following properties to your `application.properties` configuration file.

Passwordless (Recommended)

properties

```
logging.level.org.hibernate.SQL=DEBUG  
  
spring.datasource.azure.passwordless-enabled=true  
spring.datasource.url=jdbc:mysql://mysqlflexibletest.mysql.database.  
.azure.com:3306/demo?serverTimezone=UTC  
spring.datasource.username=<your_mysql_ad_non_admin_username>  
  
spring.jpa.hibernate.ddl-auto=create-drop  
spring.jpa.properties.hibernate.dialect  
=org.hibernate.dialect.MySQL8Dialect
```

⚠ Warning

The configuration property `spring.datasource.url` has `?serverTimezone=UTC` appended to tell the JDBC driver to use the UTC date format (or Coordinated Universal Time) when connecting to the database. Without this parameter, your Java server wouldn't use the same date format as the database, which would result in an error.

2. Create a new `Todo` Java class. This class is a domain model mapped onto the `todo` table that will be created automatically by JPA. The following code ignores the `getters` and `setters` methods.

Java

```
package com.example.demo;  
  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.Id;  
  
@Entity  
public class Todo {  
  
    public Todo() {}  
  
    public Todo(String description, String details, boolean done) {  
        this.description = description;  
        this.details = details;  
        this.done = done;  
    }  
  
    @Id
```

```

    @GeneratedValue
    private Long id;

    private String description;

    private String details;

    private boolean done;

}

```

3. Edit the startup class file to show the following content.

Java

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.stream.Collectors;
import java.util.stream.Stream;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Bean
    ApplicationListener<ApplicationReadyEvent>
    basicsApplicationListener(TodoRepository repository) {
        return event->repository
            .saveAll(Stream.of("A", "B", "C").map(name->new
Todo("configuration", "congratulations, you have set up correctly!",
true)).collect(Collectors.toList()))
            .forEach(System.out::println);
    }

}

interface TodoRepository extends JpaRepository<Todo, Long> {
}

```

 Tip

In this tutorial, there are no authentication operations in the configurations or the code. However, connecting to Azure services requires authentication. To complete the authentication, you need to use Azure Identity. Spring Cloud Azure uses `DefaultAzureCredential`, which the Azure Identity library provides to help you get credentials without any code changes.

`DefaultAzureCredential` supports multiple authentication methods and determines which method to use at runtime. This approach enables your app to use different authentication methods in different environments (such as local and production environments) without implementing environment-specific code. For more information, see [DefaultAzureCredential](#).

To complete the authentication in local development environments, you can use Azure CLI, Visual Studio Code, PowerShell, or other methods. For more information, see [Azure authentication in Java development environments](#). To complete the authentication in Azure hosting environments, we recommend using user-assigned managed identity. For more information, see [What are managed identities for Azure resources?](#)

4. Start the application. You'll see logs similar to the following example:

```
shell
```

```
2023-02-01 10:29:19.763 DEBUG 4392 --- [main] org.hibernate.SQL :  
insert into todo (description, details, done, id) values (?, ?, ?, ?)  
com.example.demo.Todo@1f
```

Deploy to Azure Spring Apps

Now that you have the Spring Boot application running locally, it's time to move it to production. [Azure Spring Apps](#) makes it easy to deploy Spring Boot applications to Azure without any code changes. 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. To deploy your application to Azure Spring Apps, see [Deploy your first application to Azure Spring Apps](#).

Next steps

Azure for Spring developers

Spring Cloud Azure MySQL Samples

Use Spring Data JPA with Azure Database for PostgreSQL

Article • 10/19/2023

This tutorial demonstrates how to store data in [Azure Database for PostgreSQL](#) using [Spring Data JPA](#).

The [Java Persistence API \(JPA\)](#) is the standard Java API for object-relational mapping.

In this tutorial, we include two authentication methods: Microsoft Entra authentication and PostgreSQL authentication. The **Passwordless** tab shows the Microsoft Entra authentication and the **Password** tab shows the PostgreSQL authentication.

Microsoft Entra authentication is a mechanism for connecting to Azure Database for PostgreSQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

PostgreSQL authentication uses accounts stored in PostgreSQL. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `user` table. Because these passwords are stored in PostgreSQL, you need to manage the rotation of the passwords by yourself.

Prerequisites

- An Azure subscription - [create one for free](#).
- [Java Development Kit \(JDK\)](#), version 8 or higher.
- [Apache Maven](#).
- [Azure CLI](#).
- [PostgreSQL command line client](#).
- If you don't have a Spring Boot application, create a Maven project with the [Spring Initializr](#). Be sure to select **Maven Project** and, under **Dependencies**, add the **Spring Web**, **Spring Data JDBC**, and **PostgreSQL Driver** dependencies, and then select Java version 8 or higher.
- If you don't have one, create an Azure Database for PostgreSQL Flexible Server instance named `postgresqlflexibletest` and a database named `demo`. For

instructions, see [Quickstart: Create an Azure Database for PostgreSQL - Flexible Server in the Azure portal](#).

Important

To use passwordless connections, configure the Microsoft Entra admin user for your Azure Database for PostgreSQL Flexible Server instance. For more information, see [Manage Microsoft Entra roles in Azure Database for PostgreSQL - Flexible Server](#).

See the sample application

In this tutorial, you'll code a sample application. If you want to go faster, this application is already coded and available at <https://github.com/Azure-Samples/quickstart-spring-data-jpa-postgresql>.

Configure a firewall rule for your PostgreSQL server

Azure Database for PostgreSQL instances are secured by default. They have a firewall that doesn't allow any incoming connection.

To be able to use your database, open the server's firewall to allow the local IP address to access the database server. For more information, see [Firewall rules in Azure Database for PostgreSQL - Flexible Server](#).

If you're connecting to your PostgreSQL server from Windows Subsystem for Linux (WSL) on a Windows computer, you need to add the WSL host ID to your firewall.

Create a PostgreSQL non-admin user and grant permission

Next, create a non-admin user and grant all permissions to the database.

Passwordless (Recommended)

You can use the following method to create a non-admin user that uses a passwordless connection.

Service Connector (Recommended)

1. Use the following command to install the [Service Connector](#) passwordless extension for the Azure CLI:

```
Azure CLI
```

```
az extension add --name serviceconnector-passwordless --upgrade
```

2. Use the following command to create the Microsoft Entra non-admin user:

```
Azure CLI
```

```
az connection create postgres-flexible \
--resource-group <your_resource_group_name> \
--connection postgres_conn \
--target-resource-group <your_resource_group_name> \
--server postgresqlflexibletest \
--database demo \
--user-account \
--query authInfo.userName \
--output tsv
```

When the command completes, take note of the username in the console output.

Store data from Azure Database for PostgreSQL

Now that you have an Azure Database for PostgreSQL Flexible Server instance, you can store data by using Spring Cloud Azure.

To install the Spring Cloud Azure Starter JDBC PostgreSQL module, add the following dependencies to your *pom.xml* file:

- The Spring Cloud Azure Bill of Materials (BOM):

```
XML
```

```
<dependencyManagement>
<dependencies>
  <dependency>
    <groupId>com.azure.spring</groupId>
    <artifactId>spring-cloud-azure-dependencies</artifactId>
```

```
<version>4.13.0</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

ⓘ Note

If you're using Spring Boot 3.x, be sure to set the `spring-cloud-azure-dependencies` version to `5.7.0`. For more information about the `spring-cloud-azure-dependencies` version, see [Which Version of Spring Cloud Azure Should I Use](#).

- The Spring Cloud Azure Starter JDBC PostgreSQL artifact:

XML

```
<dependency>
  <groupId>com.azure.spring</groupId>
  <artifactId>spring-cloud-azure-starter-jdbc-postgresql</artifactId>
</dependency>
```

ⓘ Note

Passwordless connections have been supported since version `4.5.0`.

Configure Spring Boot to use Azure Database for PostgreSQL

To store data from Azure Database for PostgreSQL using Spring Data JPA, follow these steps to configure the application:

1. Configure Azure Database for PostgreSQL credentials by adding the following properties to your `application.properties` configuration file.

Passwordless (Recommended)

properties

```
logging.level.org.hibernate.SQL=DEBUG
```

```
spring.datasource.url=jdbc:postgresql://postgresqlflexibletest.post  
gres.database.azure.com:5432/demo?sslmode=require  
spring.datasource.username=<your_postgresql_ad_non_admin_username>  
spring.datasource.azure.passwordless-enabled=true  
  
spring.jpa.hibernate.ddl-auto=create-drop  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.Postg  
reSQLDialect
```

2. Create a new `Todo` Java class. This class is a domain model mapped onto the `todo` table that will be created automatically by JPA. The following code ignores the `getters` and `setters` methods.

Java

```
package com.example.demo;  
  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.Id;  
  
@Entity  
public class Todo {  
  
    public Todo() {  
    }  
  
    public Todo(String description, String details, boolean done) {  
        this.description = description;  
        this.details = details;  
        this.done = done;  
    }  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String description;  
  
    private String details;  
  
    private boolean done;  
}
```

3. Edit the startup class file to show the following content.

Java

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.stream.Collectors;
import java.util.stream.Stream;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Bean
    ApplicationListener<ApplicationReadyEvent>
    basicsApplicationListener(TodoRepository repository) {
        return event->repository
            .saveAll(Stream.of("A", "B", "C").map(name->new
Todo("configuration", "congratulations, you have set up correctly!",
true)).collect(Collectors.toList()))
            .forEach(System.out::println);
    }

}

interface TodoRepository extends JpaRepository<Todo, Long> {
}

```

💡 Tip

In this tutorial, there are no authentication operations in the configurations or the code. However, connecting to Azure services requires authentication. To complete the authentication, you need to use Azure Identity. Spring Cloud Azure uses `DefaultAzureCredential`, which the Azure Identity library provides to help you get credentials without any code changes.

`DefaultAzureCredential` supports multiple authentication methods and determines which method to use at runtime. This approach enables your app to use different authentication methods in different environments (such as local and production environments) without implementing environment-specific code. For more information, see [DefaultAzureCredential](#).

To complete the authentication in local development environments, you can use Azure CLI, Visual Studio Code, PowerShell, or other methods. For more information, see [Azure authentication in Java development environments](#). To complete the authentication in Azure hosting environments, we recommend using user-assigned managed identity. For more information, see [What are managed identities for Azure resources?](#)

4. Start the application. You'll see logs similar to the following example:

```
shell
```

```
2023-02-01 10:29:19.763 DEBUG 4392 --- [main] org.hibernate.SQL :  
insert into todo (description, details, done, id) values (?, ?, ?, ?)  
com.example.demo.Todo@1f
```

Deploy to Azure Spring Apps

Now that you have the Spring Boot application running locally, it's time to move it to production. [Azure Spring Apps](#) makes it easy to deploy Spring Boot applications to Azure without any code changes. 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. To deploy your application to Azure Spring Apps, see [Deploy your first application to Azure Spring Apps](#).

Next steps

[Azure for Spring developers](#)[Spring Cloud Azure PostgreSQL Samples](#)

Use Spring Kafka with Azure Event Hubs for Kafka API

Article • 10/19/2023

This tutorial shows you how to configure a Java-based Spring Cloud Stream Binder to use Azure Event Hubs for Kafka for sending and receiving messages with Azure Event Hubs. For more information, see [Use Azure Event Hubs from Apache Kafka applications](#)

In this tutorial, we'll include two authentication methods: [Microsoft Entra authentication](#) and [Shared Access Signatures \(SAS\) authentication](#). The **Passwordless** tab shows the Microsoft Entra authentication and the **Connection string** tab shows the SAS authentication.

Microsoft Entra authentication is a mechanism for connecting to Azure Event Hubs for Kafka using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

SAS authentication uses the connection string of your Azure Event Hubs namespace for the delegated access to Event Hubs for Kafka. If you choose to use Shared Access Signatures as credentials, you need to manage the connection string by yourself.

Prerequisites

- An Azure subscription - [create one for free](#).
- [Java Development Kit \(JDK\)](#) version 8 or higher.
- [Apache Maven](#), version 3.2 or higher.
- [cURL](#) or a similar HTTP utility to test functionality.
- [Azure Cloud Shell](#) or [Azure CLI](#) 2.37.0 or higher.
- An Azure Event hub. If you don't have one, [create an event hub using Azure portal](#).
- A Spring Boot application. If you don't have one, create a Maven project with the [Spring Initializr](#). Be sure to select **Maven Project** and, under **Dependencies**, add the **Spring Web**, **Spring for Apache Kafka**, and **Cloud Stream** dependencies, then select Java version 8 or higher.

 **Important**

Spring Boot version 2.5 or higher is required to complete the steps in this tutorial.

Prepare credentials

Passwordless (Recommended)

Azure Event Hubs supports using Microsoft Entra ID to authorize requests to Event Hubs resources. With Microsoft Entra ID, you can use [Azure role-based access control \(Azure RBAC\)](#) to grant permissions to a [security principal](#), which may be a user or an application service principal.

If you want to run this sample locally with Microsoft Entra authentication, be sure your user account has authenticated via Azure Toolkit for IntelliJ, Visual Studio Code Azure Account plugin, or Azure CLI. Also, be sure the account has been granted sufficient permissions.

ⓘ Note

When using passwordless connections, you need to grant your account access to resources. In Azure Event Hubs, assign the [Azure Event Hubs Data Receiver](#) and [Azure Event Hubs Data Sender](#) role to the Microsoft Entra account you're currently using. For more information about granting access roles, see [Assign Azure roles using the Azure portal](#) and [Authorize access to Event Hubs resources using Microsoft Entra ID](#).

Send and receive messages from Azure Event Hubs

With an Azure Event hub, you can send and receive messages using Spring Cloud Azure.

To install the Spring Cloud Azure Starter module, add the following dependencies to your *pom.xml* file:

- The Spring Cloud Azure Bill of Materials (BOM):

XML

```
<dependencyManagement>
  <dependencies>
    <dependency>
```

```
<groupId>com.azure.spring</groupId>
<artifactId>spring-cloud-azure-dependencies</artifactId>
<version>4.13.0</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

ⓘ Note

If you're using Spring Boot 3.x, be sure to set the `spring-cloud-azure-dependencies` version to `5.7.0`. For more information about the `spring-cloud-azure-dependencies` version, see [Which Version of Spring Cloud Azure Should I Use](#).

- The Spring Cloud Azure Starter artifact:

XML

```
<dependency>
  <groupId>com.azure.spring</groupId>
  <artifactId>spring-cloud-azure-starter</artifactId>
</dependency>
```

Code the application

Use the following steps to configure your application to produce and consume messages using Azure Event Hubs.

1. Configure the Event hub credentials by adding the following properties to your `application.properties` file.

Passwordless (Recommended)

properties

```
spring.cloud.stream.kafka.binder.brokers=${AZ_EVENTHUBS_NAMESPACE_NAME}.servicebus.windows.net:9093
spring.cloud.function.definition=consume;supply
spring.cloud.stream.bindings.consume-in-0.destination=${AZ_EVENTHUB_NAME}
spring.cloud.stream.bindings.consume-in-0.group=$Default
```

```
spring.cloud.stream.bindings.supply-out-  
0.destination=${AZ_EVENTHUB_NAME}
```

💡 Tip

If you're using version `spring-cloud-azure-dependencies:4.3.0`, then you should add the property `spring.cloud.stream.binders.<kafka-binder-name>.environment.spring.main.sources` with the value `com.azure.spring.cloud.autoconfigure.kafka.AzureKafkaSpringCloudStreamConfiguration`.

Since `4.4.0`, this property will be added automatically, so there's no need to add it manually.

The following table describes the fields in the configuration:

Field	Description
<code>spring.cloud.stream.kafka.binder.brokers</code>	Specifies the Azure Event Hubs endpoint.
<code>spring.cloud.stream.bindings.consume-in-0.destination</code>	Specifies the input destination event hub, which for this tutorial is the hub you created earlier.
<code>spring.cloud.stream.bindings.consume-in-0.group</code>	Specifies a Consumer Group from Azure Event Hubs, which you can set to <code>\$Default</code> in order to use the basic consumer group that was created when you created your Azure Event Hubs instance.
<code>spring.cloud.stream.bindings.supply-out-0.destination</code>	Specifies the output destination event hub, which for this tutorial is the same as the input destination.

❗ Note

If you enable automatic topic creation, be sure to add the configuration item `spring.cloud.stream.kafka.binder.replicationFactor`, with the value set to at least 1. For more information, see [Spring Cloud Stream Kafka Binder Reference Guide](#).

2. Edit the startup class file to show the following content.

Java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.messaging.Message;
import org.springframework.messaging.support.GenericMessage;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Sinks;
import java.util.function.Consumer;
import java.util.function.Supplier;

@SpringBootApplication
public class EventHubKafkaBinderApplication implements
CommandLineRunner {

    private static final Logger LOGGER =
LoggerFactory.getLogger(EventHubKafkaBinderApplication.class);

    private static final Sinks.Many<Message<String>> many =
Sinks.many().unicast().onBackpressureBuffer();

    public static void main(String[] args) {
        SpringApplication.run(EventHubKafkaBinderApplication.class,
args);
    }

    @Bean
    public Supplier<Flux<Message<String>>> supply() {
        return ()->many.asFlux()
            .doOnNext(m->LOGGER.info("Manually sending
message {}", m))
            .doOnError(t->LOGGER.error("Error encountered",
t));
    }

    @Bean
    public Consumer<Message<String>> consume() {
        return message->LOGGER.info("New message received: '{}'",
message.getPayload());
    }

    @Override
    public void run(String... args) {
        many.emitNext(new GenericMessage<>("Hello World"),
Sinks.EmitFailureHandler.FAIL_FAST);
    }
}
```

```
}
```

💡 Tip

In this tutorial, there are no authentication operations in the configurations or the code. However, connecting to Azure services requires authentication. To complete the authentication, you need to use Azure Identity. Spring Cloud Azure uses `DefaultAzureCredential`, which the Azure Identity library provides to help you get credentials without any code changes.

`DefaultAzureCredential` supports multiple authentication methods and determines which method to use at runtime. This approach enables your app to use different authentication methods in different environments (such as local and production environments) without implementing environment-specific code. For more information, see [DefaultAzureCredential](#).

To complete the authentication in local development environments, you can use Azure CLI, Visual Studio Code, PowerShell, or other methods. For more information, see [Azure authentication in Java development environments](#). To complete the authentication in Azure hosting environments, we recommend using user-assigned managed identity. For more information, see [What are managed identities for Azure resources?](#)

3. Start the application. Messages like the following example will be posted in your application log:

Output

```
Kafka version: 3.0.1
Kafka commitId: 62abe01bee039651
Kafka startTimeMs: 1622616433956
New message received: 'Hello World'
```

Deploy to Azure Spring Apps

Now that you have the Spring Boot application running locally, it's time to move it to production. [Azure Spring Apps](#) makes it easy to deploy Spring Boot applications to Azure without any code changes. 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. To deploy your application to Azure Spring Apps, see [Deploy your first application to Azure Spring Apps](#).

Next steps

[Azure for Spring developers](#)

[Spring Cloud Azure Stream Binder Event Hubs Kafka Samples](#)

Quickstart: Azure Cosmos DB for NoSQL library for Java

Article • 01/08/2024

APPLIES TO:  NoSQL

Get started with the Azure Cosmos DB for NoSQL client library for Java to query data in your containers and perform common operations on individual items. Follow these steps to deploy a minimal solution to your environment using the Azure Developer CLI.

[API reference documentation](#) | [Library source code](#) | [Package \(Maven\)](#) | [Azure Developer CLI](#)

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [GitHub account](#)

Setting up

Deploy this project's development container to your environment. Then, use the Azure Developer CLI (`azd`) to create an Azure Cosmos DB for NoSQL account and deploy a containerized sample application. The sample application uses the client library to manage, create, read, and query sample data.

 GITHUB CODESPACES 

Important

GitHub accounts include an entitlement of storage and core hours at no cost. For more information, see [included storage and core hours for GitHub accounts](#).

1. Open a terminal in the root directory of the project.
2. Authenticate to the Azure Developer CLI using `azd auth login`. Follow the steps specified by the tool to authenticate to the CLI using your preferred Azure credentials.

Azure CLI

```
azd auth login
```

3. Use `azd init` to initialize the project.

```
Azure CLI
```

```
azd init
```

4. During initialization, configure a unique environment name.

 **Tip**

The environment name will also be used as the target resource group name. For this quickstart, consider using `msdocs-cosmos-db-nosql`.

5. Deploy the Azure Cosmos DB for NoSQL account using `azd up`. The Bicep templates also deploy a sample web application.

```
Azure CLI
```

```
azd up
```

6. During the provisioning process, select your subscription and desired location. Wait for the provisioning process to complete. The process can take **approximately five minutes**.
7. Once the provisioning of your Azure resources is done, a URL to the running web application is included in the output.

```
Output
```

```
Deploying services (azd deploy)
```

```
(✓) Done: Deploying service web
- Endpoint: <https://[container-app-sub-domain].azurecontainerapps.io>
```

```
SUCCESS: Your application was provisioned and deployed to Azure in 5
minutes 0 seconds.
```

8. Use the URL in the console to navigate to your web application in the browser. Observe the output of the running app.

Azure Cosmos DB for NoSQL | Go Quickstart

```
Current Status: Starting...
Get database: cosmicworks
Get container: products
Upserter item: {70b63682-b93a-4c77-aad2-65501347265f gear-surf-surfboards Yamba Surfboard 12 850 false}
Status code: 201
Request charge: 7.24
Upserter item: {25a68543-b90c-439d-8332-7ef41e06a0e0 gear-surf-surfboards Kiama Classic Surfboard 25 790 true}
Status code: 201
Request charge: 2.04
Read item id: 70b63682-b93a-4c77-aad2-65501347265f
Read item: {70b63682-b93a-4c77-aad2-65501347265f gear-surf-surfboards Yamba Surfboard 12 850 false}
Status code: 200
Request charge: 1.00
```

Install the client library

The client library is available through Maven, as the `azure-spring-data-cosmos` package.

1. Navigate to the `/src/web` folder and open the `pom.xml` file.
2. If it doesn't already exist, add an entry for the `azure-spring-data-cosmos` package.

XML

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-spring-data-cosmos</artifactId>
</dependency>
```

3. Also, add another dependency for the `azure-identity` package if it doesn't already exist.

XML

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-identity</artifactId>
</dependency>
```

Object model

[] Expand table

Name	Description
<code>EnableCosmosRepositories</code>	This type is a method decorator used to configure a repository to access Azure Cosmos DB for NoSQL.
<code>CosmosRepository</code>	This class is the primary client class and is used to manage data

Name	Description
	within a container.
<code>CosmosClientBuilder</code>	This class is a factory used to create a client used by the repository.
<code>Query</code>	This type is a method decorator used to specify the query that the repository executes.

Code examples

- [Authenticate the client](#)
- [Get a database](#)
- [Get a container](#)
- [Create an item](#)
- [Get an item](#)
- [Query items](#)

The sample code in the template uses a database named `cosmicworks` and container named `products`. The `products` container contains details such as name, category, quantity, a unique identifier, and a sale flag for each product. The container uses the `/category` property as a logical partition key.

Authenticate the client

Application requests to most Azure services must be authorized. Use the `DefaultAzureCredential` type as the preferred way to implement a passwordless connection between your applications and Azure Cosmos DB for NoSQL. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime.

Important

You can also authorize requests to Azure services using passwords, connection strings, or other credentials directly. However, this approach should be used with caution. Developers must be diligent to never expose these secrets in an unsecure location. Anyone who gains access to the password or secret key is able to authenticate to the database service. `DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication without the risk of storing keys.

First, this sample creates a new class that inherits from `AbstractCosmosConfiguration` to configure the connection to Azure Cosmos DB for NoSQL.

Java

```
@Configuration  
@EnableCosmosRepositories  
public class CosmosConfiguration extends AbstractCosmosConfiguration {
```

Within the configuration class, this sample creates a new instance of the `CosmosClientBuilder` class and configures authentication using a `DefaultAzureCredential` instance.

Java

```
@Bean  
public CosmosClientBuilder getCosmosClientBuilder() {  
    DefaultAzureCredential azureTokenCredential = new  
DefaultAzureCredentialBuilder()  
    .build();  
  
    return new CosmosClientBuilder()  
        .endpoint(uri)  
        .credential(azureTokenCredential);  
}
```

Get a database

In the configuration class, the sample implements a method to return the name of the existing database named `cosmicworks`.

Java

```
@Override  
protected String getDatabaseName() {  
    return "cosmicworks";  
}
```

Get a container

Use the `Container` method decorator to configure a class to represent items in a container. Author the class to include all of the members you want to serialize into JSON. In this example, the type has a unique identifier, and fields for category, name, quantity, price, and clearance.

Java

```
@Container(containerName = "products", autoCreateContainer = false)
public class Item {
    private String id;
    private String name;
    private Integer quantity;
    private Boolean sale;

    @PartitionKey
    private String category;
```

Create an item

Create an item in the container using `repository.save`.

Java

```
Item item = new Item(
    "70b63682-b93a-4c77-aad2-65501347265f",
    "gear-surf-surfboards",
    "Yamba Surfboard",
    12,
    false
);
Item created_item = repository.save(item);
```

Read an item

Perform a point read operation by using both the unique identifier (`id`) and partition key fields. Use `repository.findById` to efficiently retrieve the specific item.

Java

```
PartitionKey partitionKey = new PartitionKey("gear-surf-surfboards");
Optional<Item> existing_item = repository.findById("70b63682-b93a-4c77-aad2-
65501347265f", partitionKey);
if (existing_item.isPresent()) {
    // Do something
}
```

Query items

Perform a query over multiple items in a container by defining a query in the repository's interface. This sample uses the `Query` method decorator to define a method

that executes this parameterized query:

nosql

```
SELECT * FROM products p WHERE p.category = @category
```

Java

```
@Repository
public interface ItemRepository extends CosmosRepository<Item, String> {
    @Query("SELECT * FROM products p WHERE p.category = @category")
    List<Item> getItemsByCategory(@Param("category") String category);
}
```

Fetch all of the results of the query using `repository.getItemsByCategory`. Loop through the results of the query.

Java

```
List<Item> items = repository.getItemsByCategory("gear-surf-surfboards");
for (Item item : items) {
    // Do something
}
```

Related content

- [.NET Quickstart](#)
- [JavaScript/Node.js Quickstart](#)
- [java Quickstart](#)
- [Go Quickstart](#)

Next step

[Tutorial: Build a Java web app](#)

Use Java to send events to or receive events from Azure Event Hubs

Article • 02/16/2024

This quickstart shows how to send events to and receive events from an event hub using the `azure-messaging-eventhubs` Java package.

Tip

If you're working with Azure Event Hubs resources in a Spring application, we recommend that you consider [Spring Cloud Azure](#) as an alternative. Spring Cloud Azure is an open-source project that provides seamless Spring integration with Azure services. To learn more about Spring Cloud Azure, and to see an example using Event Hubs, see [Spring Cloud Stream with Azure Event Hubs](#).

Prerequisites

If you're new to Azure Event Hubs, see [Event Hubs overview](#) before you do this quickstart.

To complete this quickstart, you need the following prerequisites:

- **Microsoft Azure subscription.** To use Azure services, including Azure Event Hubs, you need a subscription. If you don't have an existing Azure account, you can sign up for a [free trial](#) or use your MSDN subscriber benefits when you [create an account](#).
- A Java development environment. This quickstart uses [Eclipse](#). Java Development Kit (JDK) with version 8 or above is required.
- **Create an Event Hubs namespace and an event hub.** The first step is to use the [Azure portal](#) to create a namespace of type Event Hubs, and obtain the management credentials your application needs to communicate with the event hub. To create a namespace and an event hub, follow the procedure in [this article](#). Then, get the **connection string for the Event Hubs namespace** by following instructions from the article: [Get connection string](#). You use the connection string later in this quickstart.

Send events

This section shows you how to create a Java application to send events an event hub.

Add reference to Azure Event Hubs library

First, create a new **Maven** project for a console/shell application in your favorite Java development environment. Update the `pom.xml` file as follows. The Java client library for Event Hubs is available in the [Maven Central Repository](#).

Passwordless (Recommended)

XML

```
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-messaging-eventhubs</artifactId>
    <version>5.18.0</version>
</dependency>
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-identity</artifactId>
    <version>1.11.2</version>
    <scope>compile</scope>
</dependency>
```

ⓘ Note

Update the version to the latest version published to the Maven repository.

Authenticate the app to Azure

This quickstart shows you two ways of connecting to Azure Event Hubs: passwordless and connection string. The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to an Event Hubs namespace. You don't need to worry about having hard-coded connection strings in your code or in a configuration file or in a secure storage like Azure Key Vault. The second option shows you how to use a connection string to connect to an Event Hubs namespace. If you're new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Event Hubs has the correct permissions. You'll need the [Azure Event Hubs Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Event Hubs Data Owner` role to your user account, which provides full access to Azure Event Hubs resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

Azure built-in roles for Azure Event Hubs

For Azure Event Hubs, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to an Event Hubs namespace:

- [Azure Event Hubs Data Owner](#): Enables data access to Event Hubs namespace and its entities (queues, topics, subscriptions, and filters)
- [Azure Event Hubs Data Sender](#): Use this role to give the sender access to Event Hubs namespace and its entities.
- [Azure Event Hubs Data Receiver](#): Use this role to give the receiver access to Event Hubs namespace and its entities.

If you want to create a custom role, see [Rights required for Event Hubs operations](#).

Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. In the Azure portal, locate your Event Hubs namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the Azure portal interface for managing access control. The top navigation bar includes a search bar, a 'Download role assignments' button (labeled 2), and other standard UI elements like 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. Below the search bar, the left sidebar lists various service management options: Overview, Activity log, Access control (IAM) (which is highlighted with a red box and labeled 1), Tags, Diagnose and solve problems, Settings (with sub-options like Shared access policies, Geo-Recovery, Migrate to premium, Encryption, Configuration, Properties, and Locks), and Entities (Queues and Topics). The main content area is titled 'spsbusns11102 | Access control (IAM)' and 'Service Bus Namespace'. It features a 'My access' section with a 'Check access' button. The 'Role assignments' tab is selected, showing a table with columns: Role, Principal, and Last updated. A large callout box on the right provides instructions for users who do not have access to certain options. At the bottom of the page, there are two main sections: 'Grant access to this resource' (with a 'Add role assignment' button) and 'View access to this resource' (with a 'View' button).

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Event Hubs Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Write code to send messages to the event hub

Passwordless (Recommended)

Add a class named `Sender`, and add the following code to the class:

ⓘ Important

- Update `<NAMESPACE NAME>` with the name of your Event Hubs namespace.
- Update `<EVENT HUB NAME>` with the name of your event hub.

Java

```
package ehubquickstart;

import com.azure.messaging.eventhubs.*;
import java.util.Arrays;
import java.util.List;

import com.azure.identity.*;

public class SenderAAD {

    // replace <NAMESPACE NAME> with the name of your Event Hubs
    // namespace.
    // Example: private static final String namespaceName =
    "contosons.servicebus.windows.net";
    private static final String namespaceName = "<NAMESPACE
NAME>.servicebus.windows.net";

    // Replace <EVENT HUB NAME> with the name of your event hub.
    // Example: private static final String eventHubName = "ordersehub";
    private static final String eventHubName = "<EVENT HUB NAME>";

    public static void main(String[] args) {
        publishEvents();
    }
    /**
     * Code sample for publishing events.
     * @throws IllegalArgumentException if the EventData is bigger than
     the max batch size.
     */
    public static void publishEvents() {
        // create a token using the default Azure credential
        DefaultAzureCredential credential = new
DefaultAzureCredentialBuilder()
            .authorityHost(AzureAuthorityHosts.AZURE_PUBLIC_CLOUD)
            .build();
```

```

    // create a producer client
    EventHubProducerClient producer = new EventHubClientBuilder()
        .fullyQualifiedNamespace(namespaceName)
        .eventHubName(eventHubName)
        .credential(credential)
        .buildProducerClient();

    // sample events in an array
    List<EventData> allEvents = Arrays.asList(new EventData("Foo"),
new EventData("Bar"));

    // create a batch
    EventDataBatch eventDataBatch = producer.createBatch();

    for (EventData eventData : allEvents) {
        // try to add the event from the array to the batch
        if (!eventDataBatch.tryAdd(eventData)) {
            // if the batch is full, send it and then create a new
batch
            producer.send(eventDataBatch);
            eventDataBatch = producer.createBatch();

            // Try to add that event that couldn't fit before.
            if (!eventDataBatch.tryAdd(eventData)) {
                throw new IllegalArgumentException("Event is too
large for an empty batch. Max size: "
+ eventDataBatch.getMaxSizeInBytes());
            }
        }
    }
    // send the last batch of remaining events
    if (eventDataBatch.getCount() > 0) {
        producer.send(eventDataBatch);
    }
    producer.close();
}
}

```

Build the program, and ensure that there are no errors. You'll run this program after you run the receiver program.

Receive events

The code in this tutorial is based on the [EventProcessorClient sample on GitHub](#), which you can examine to see the full working application.

Follow these recommendations when using Azure Blob Storage as a checkpoint store:

- Use a separate container for each consumer group. You can use the same storage account, but use one container per each group.
- Don't use the container for anything else, and don't use the storage account for anything else.
- Storage account should be in the same region as the deployed application is located in. If the application is on-premises, try to choose the closest region possible.

On the **Storage account** page in the Azure portal, in the **Blob service** section, ensure that the following settings are disabled.

- Hierarchical namespace
- Blob soft delete
- Versioning

Create an Azure Storage and a blob container

In this quickstart, you use Azure Storage (specifically, Blob Storage) as the checkpoint store. Checkpointing is a process by which an event processor marks or commits the position of the last successfully processed event within a partition. Marking a checkpoint is typically done within the function that processes the events. To learn more about checkpointing, see [Event processor](#).

Follow these steps to create an Azure Storage account.

1. [Create an Azure Storage account](#)
2. [Create a blob container](#)
3. Authenticate to the blob container

Passwordless (Recommended)

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users

only the minimum permissions needed and creates more secure production environments.

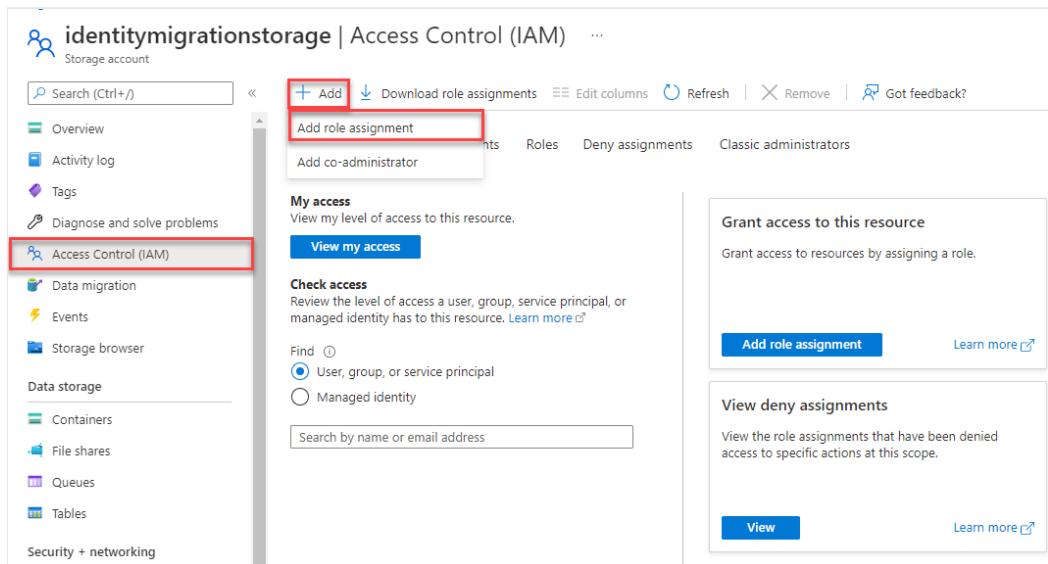
The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

ⓘ Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



The screenshot shows the Azure portal interface for managing access control. The left sidebar lists various storage account management options, with 'Access Control (IAM)' highlighted by a red box. The main content area shows the 'Role assignments' tab selected. At the top, there's a search bar and several action buttons: '+ Add', 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. A red box highlights the '+ Add' button. Below these buttons, a dropdown menu is open, with 'Add role assignment' also highlighted by a red box. To the right of the main content area, there are three sections: 'My access' (View my level of access to this resource), 'Grant access to this resource' (Grant access to resources by assigning a role), and 'View deny assignments' (View the role assignments that have been denied access to specific actions at this scope). Each section has its own 'Add role assignment' button.

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the

matching result and then choose **Next**.

6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Add Event Hubs libraries to your Java project

Passwordless (Recommended)

Add the following dependencies in the pom.xml file.

- [azure-messaging-eventhubs](#)
- [azure-messaging-eventhubs-checkpointstore-blob](#)
- [azure-identity](#)

XML

```
<dependencies>
    <dependency>
        <groupId>com.azure</groupId>
        <artifactId>azure-messaging-eventhubs</artifactId>
        <version>5.15.0</version>
    </dependency>
    <dependency>
        <groupId>com.azure</groupId>
        <artifactId>azure-messaging-eventhubs-checkpointstore-
blob</artifactId>
        <version>1.16.1</version>
    </dependency>
    <dependency>
        <groupId>com.azure</groupId>
        <artifactId>azure-identity</artifactId>
        <version>1.8.0</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Passwordless (Recommended)

1. Add the following `import` statements at the top of the Java file.

Java

```
import com.azure.messaging.eventhubs.*;
import com.azure.messaging.eventhubs.checkpointstore.blob.BlobCheckpointStore;
import com.azure.messaging.eventhubs.models.*;
import com.azure.storage.blob.*;
import java.util.function.Consumer;

import com.azure.identity.*;
```

2. Create a class named `Receiver`, and add the following string variables to the class. Replace the placeholders with the correct values.

ⓘ Important

Replace the placeholders with the correct values.

- `<NAMESPACE NAME>` with the name of your Event Hubs namespace.
- `<EVENT HUB NAME>` with the name of your event hub in the namespace.

Java

```
private static final String namespaceName = "<NAMESPACE
NAME>.servicebus.windows.net";
private static final String eventHubName = "<EVENT HUB NAME>";
```

3. Add the following `main` method to the class.

ⓘ Important

Replace the placeholders with the correct values.

- `<STORAGE ACCOUNT NAME>` with the name of your Azure Storage account.

- <CONTAINER NAME> with the name of the blob container in the storage account

Java

```
// create a token using the default Azure credential
DefaultAzureCredential credential = new
DefaultAzureCredentialBuilder()
    .authorityHost(AzureAuthorityHosts.AZURE_PUBLIC_CLOUD)
    .build();

// Create a blob container client that you use later to build an
// event processor client to receive and process events
BlobContainerAsyncClient blobContainerAsyncClient = new
BlobContainerClientBuilder()
    .credential(credential)
    .endpoint("https://<STORAGE ACCOUNT
NAME>.blob.core.windows.net")
    .containerName("<CONTAINER NAME>")
    .buildAsyncClient();

// Create an event processor client to receive and process events
// and errors.
EventProcessorClient eventProcessorClient = new
EventProcessorClientBuilder()
    .fullyQualifiedNamespace(namespaceName)
    .eventHubName(eventHubName)

    .consumerGroup(EventHubClientBuilder.DEFAULT_CONSUMER_GROUP_NAME)
    .processEvent(PARTITION_PROCESSOR)
    .processError(ERROR_HANDLER)
    .checkpointStore(new
BlobCheckpointStore(blobContainerAsyncClient))
    .credential(credential)
    .buildEventProcessorClient();

System.out.println("Starting event processor");
eventProcessorClient.start();

System.out.println("Press enter to stop.");
System.in.read();

System.out.println("Stopping event processor");
eventProcessorClient.stop();
System.out.println("Event processor stopped.");

System.out.println("Exiting process");
```

4. Add the two helper methods (`PARTITION_PROCESSOR` and `ERROR_HANDLER`) that process events and errors to the `Receiver` class.

```
Java

public static final Consumer<EventContext> PARTITION_PROCESSOR =
eventContext -> {
    PartitionContext partitionContext =
eventContext.getPartitionContext();
    EventData eventData = eventContext.getEventData();

    System.out.printf("Processing event from partition %s with sequence
number %d with body: %s%n",
        partitionContext.getPartitionId(),
eventData.getSequenceNumber(), eventData.getBodyAsString());

    // Every 10 events received, it will update the checkpoint stored
in Azure Blob Storage.
    if (eventData.getSequenceNumber() % 10 == 0) {
        eventContext.updateCheckpoint();
    }
};

public static final Consumer<ErrorContext> ERROR_HANDLER = errorContext
-> {
    System.out.printf("Error occurred in partition processor for
partition %s, %s.%n",
        errorContext.getPartitionContext().getPartitionId(),
        errorContext.getThrowable());
};
```

5. Build the program, and ensure that there are no errors.

Run the applications

1. Run the `Receiver` application first.
2. Then, run the `Sender` application.
3. In the `Receiver` application window, confirm that you see the events that were published by the `Sender` application.

```
Windows Command Prompt

Starting event processor
Press enter to stop.
Processing event from partition 0 with sequence number 331 with body:
Foo
```

```
Processing event from partition 0 with sequence number 332 with body:  
Bar
```

4. Press **ENTER** in the receiver application window to stop the application.

Windows Command Prompt

```
Starting event processor  
Press enter to stop.  
Processing event from partition 0 with sequence number 331 with body:  
Foo  
Processing event from partition 0 with sequence number 332 with body:  
Bar  
  
Stopping event processor  
Event processor stopped.  
Exiting process
```

Next steps

See the following samples on GitHub:

- [azure-messaging-eventhubs samples ↗](#)
- [azure-messaging-eventhubs-checkpointstore-blob samples ↗](#).

Quickstart: Stream data with Azure Event Hubs and Apache Kafka

Article • 02/16/2024

This quickstart shows you how to stream data into and from Azure Event Hubs using the Apache Kafka protocol. You'll not change any code in the sample Kafka producer or consumer apps. You just update the configurations that the clients use to point to an Event Hubs namespace, which exposes a Kafka endpoint. You also don't build and use a Kafka cluster on your own. Instead, you use the Event Hubs namespace with the Kafka endpoint.

ⓘ Note

This sample is available on [GitHub](#) ↗

Prerequisites

To complete this quickstart, make sure you have the following prerequisites:

- Read through the [Event Hubs for Apache Kafka](#) article.
- An Azure subscription. If you don't have one, [create a free account](#) ↗ before you begin.
- Create a Windows virtual machine and install the following components:
 - [Java Development Kit \(JDK\) 1.7+](#).
 - [Download](#) ↗ and [install](#) ↗ a Maven binary archive.
 - [Git](#) ↗

Create an Azure Event Hubs namespace

When you create an Event Hubs namespace, the Kafka endpoint for the namespace is automatically enabled. You can stream events from your applications that use the Kafka protocol into event hubs. Follow step-by-step instructions in the [Create an event hub using Azure portal](#) to create an Event Hubs namespace. If you're using a dedicated cluster, see [Create a namespace and event hub in a dedicated cluster](#).

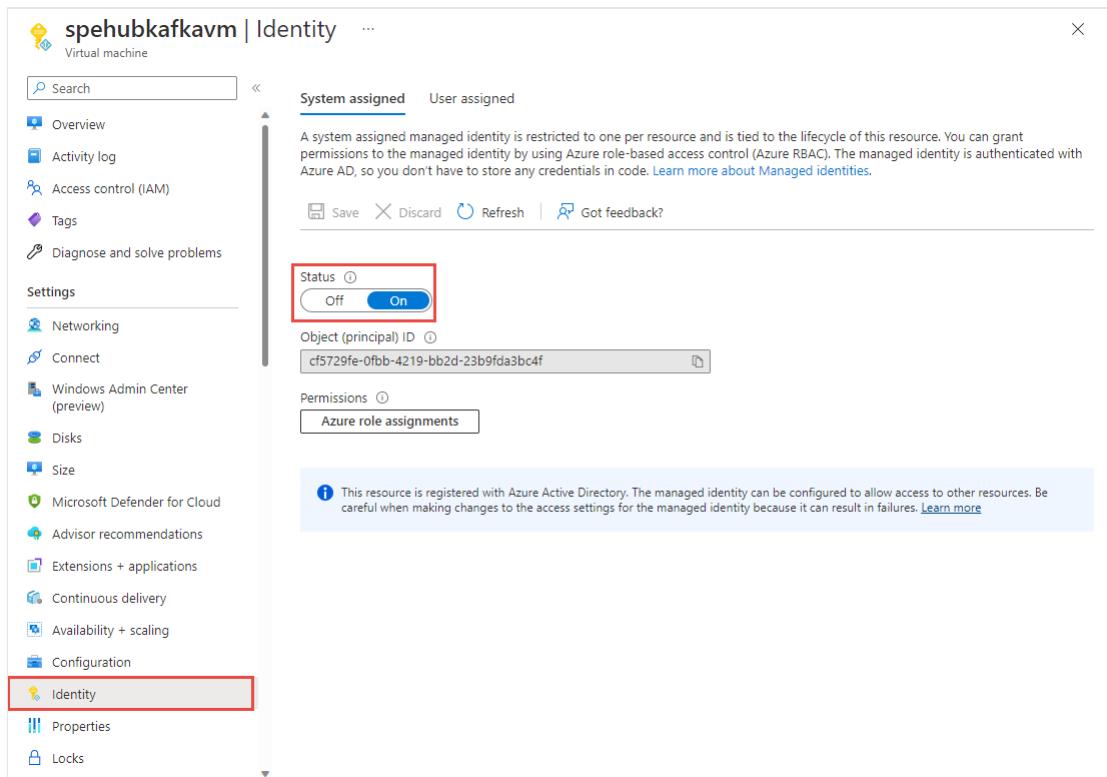
ⓘ Note

Event Hubs for Kafka isn't supported in the **basic** tier.

Send and receive messages with Kafka in Event Hubs

Passwordless (Recommended)

1. Enable a system-assigned managed identity for the virtual machine. For more information about configuring managed identity on a VM, see [Configure managed identities for Azure resources on a VM using the Azure portal](#).
Managed identities for Azure resources provide Azure services with an automatically managed identity in Microsoft Entra ID. You can use this identity to authenticate to any service that supports Microsoft Entra authentication, without having credentials in your code.



2. Using the **Access control** page of the Event Hubs namespace you created, assign **Azure Event Hubs Data Owner** role to the VM's managed identity.
Azure Event Hubs supports using Microsoft Entra ID to authorize requests to Event Hubs resources. With Microsoft Entra ID, you can use Azure role-based access control (Azure RBAC) to grant permissions to a security principal, which may be a user, or an application service principal.

a. In the Azure portal, navigate to your Event Hubs namespace. Go to "Access Control (IAM)" in the left navigation.

b. Select + Add and select **Add role assignment**.

The screenshot shows the 'Access control (IAM)' blade for an Event Hubs namespace named 'spkafkaehubns'. The left sidebar lists various management options like Tags, Diagnose and solve problems, Events, Settings, Entities, and Monitoring. The 'Access control (IAM)' item is highlighted with a red box and a blue step 1 indicator. At the top, there's a search bar, a '+ Add' button (highlighted with a red box and step 2), and a 'Download role assignments' link. A dropdown menu is open at the '+ Add' button, with 'Add role assignment' highlighted with a blue step 3 indicator. Other options in the dropdown include 'Add co-administrator', 'My access', 'Check access', 'Grant access to this resource', 'View access to this resource', 'View deny assignments', and 'Check access'.

c. In the Role tab, select **Azure Event Hubs Data Owner**, and select the **Next** button.

The screenshot shows the 'Add role assignment' dialog. The 'Role' tab is selected, showing a list of available roles. The 'Azure Event Hubs Data Owner' role is highlighted with a red box. The table lists the following roles:

Name	Description	Type	Category	Details
Owner	Grants full access to manage all resources, including th...	BuiltinRole	General	View
Contributor	Grants full access to manage all resources, but does no...	BuiltinRole	General	View
Reader	View all resources, but does not allow you to make any...	BuiltinRole	General	View
Azure Event Hubs Data Owner	Allows for full access to Azure Event Hubs resources.	BuiltinRole	Analytics	View
Azure Event Hubs Data Receiver	Allows receive access to Azure Event Hubs resources.	BuiltinRole	Analytics	View
Azure Event Hubs Data Sender	Allows send access to Azure Event Hubs resources.	BuiltinRole	Analytics	View
Log Analytics Contributor	Log Analytics Contributor can read all monitoring data ...	BuiltinRole	Analytics	View
Log Analytics Reader	Log Analytics Reader can view and search all monitorin...	BuiltinRole	Analytics	View
Managed Application Contributor Role	Allows for creating managed application resources.	BuiltinRole	Management + Gover...	View
Managed Application Operator Role	Lets you read and perform actions on Managed Appli...	BuiltinRole	Management + Gover...	View
Managed Applications Reader	Lets you read resources in a managed app and request...	BuiltinRole	Management + Gover...	View
Monitoring Contributor	Can read all monitoring data and update monitoring se...	BuiltinRole	Monitor	View
Monitoring Metrics Publisher	Enables publishing metrics against Azure resources	BuiltinRole	Monitor	View

At the bottom of the dialog, there are buttons for 'Review + assign', 'Previous', and 'Next'. The 'Next' button is highlighted with a red box.

d. In the **Members** tab, select the **Managed Identity** in the **Assign access to** section.

e. Select the **+Select members** link.

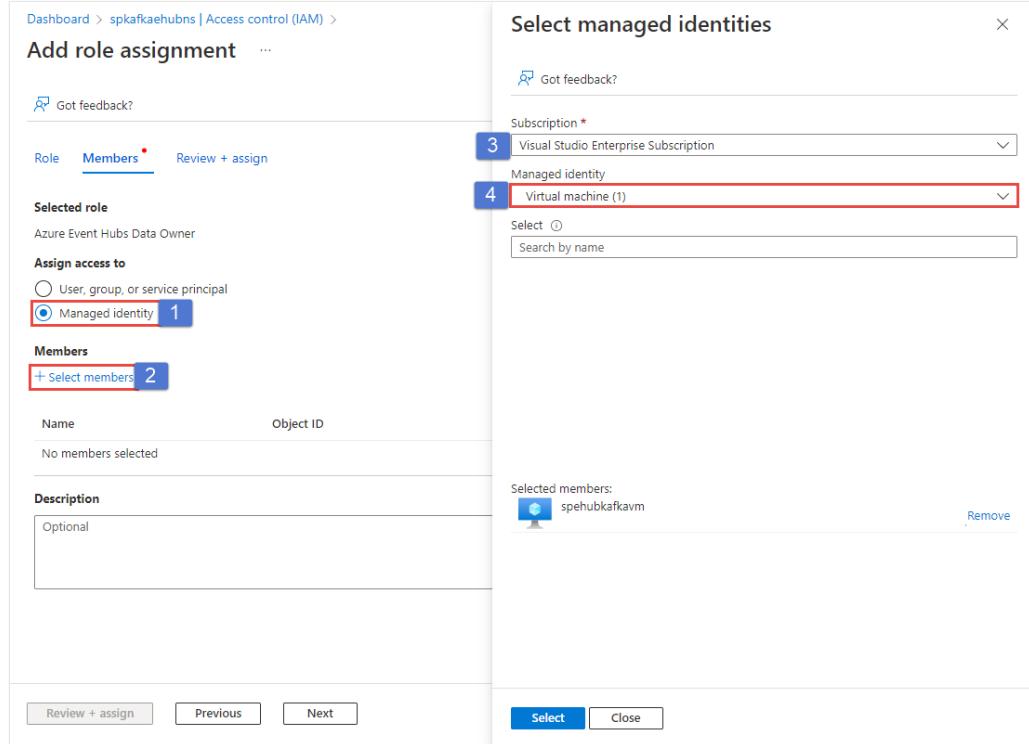
f. On the **Select managed identities** page, follow these steps:

i. Select the **Azure subscription** that has the VM.

ii. For **Managed identity**, select **Virtual machine**

iii. Select your virtual machine's managed identity.

iv. Select **Select** at the bottom of the page.



g. Select **Review + Assign**.

[Got feedback?](#)

[Role](#) [Members](#) [Review + assign](#)

Selected role
Azure Event Hubs Data Owner

Assign access to
 User, group, or service principal
 Managed identity

Members
[+ Select members](#)

Name	Object ID	Type
spehubkafkavm	cf5729fe-0fbb-4219-bb2d-23b9fda3bc4f	Virtual machine ⓘ

Description
Optional

[Review + assign](#) [Previous](#) [Next](#)

3. Restart the VM and sign in back to the VM for which you configured the managed identity.
4. Clone the [Azure Event Hubs for Kafka repository ↗](#).
5. Navigate to `azure-event-hubs-for-kafka/tutorials/oauth/java/managedidentity/consumer`.
6. Switch to the `src/main/resources/` folder, and open `consumer.config`. Replace `namespacename` with the name of your Event Hubs namespace.

XML

```
bootstrap.servers=NAMESPACENAME.servicebus.windows.net:9093
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuth
BearerLoginModule required;
sasl.login.callback.handler.class=CustomAuthenticateCallbackHandler
;
```

 **Note**

You can find all the OAuth samples for Event Hubs for Kafka [here](#).

7. Switch back to the **Consumer** folder where the pom.xml file is and, and run the consumer code and process events from event hub using your Kafka clients:

Java

```
mvn clean package  
mvn exec:java -Dexec.mainClass="TestConsumer"
```

8. Launch another command prompt window, and navigate to `azure-event-hubs-for-kafka/tutorials/oauth/java/managedidentity/producer`.
9. Switch to the `src/main/resources/` folder, and open `producer.config`. Replace `mynamespace` with the name of your Event Hubs namespace.
10. Switch back to the **Producer** folder where the `pom.xml` file is and, run the producer code and stream events into Event Hubs:

shell

```
mvn clean package  
mvn exec:java -Dexec.mainClass="TestProducer"
```

You should see messages about events sent in the producer window. Now, check the consumer app window to see the messages that it receives from the event hub.

The image shows two side-by-side terminal windows. The left window, titled 'Administrator: Command Prompt - mvn exec:java -D...', displays a series of 'Test Data' messages from thread #20, starting with '#51' and ending with '#99'. The right window, also titled 'Administrator: Command Prompt - mvn exec:java -Dexec.mainC...', shows 'Consumer Record' entries, each consisting of a timestamp and a 'Test Data' value. Both windows have scroll bars at the bottom.

```
Administrator: Command Prompt - mvn exec:java -D... Test Data #51 from thread #20
Test Data #52 from thread #20
Test Data #53 from thread #20
Test Data #54 from thread #20
Test Data #55 from thread #20
Test Data #56 from thread #20
Test Data #57 from thread #20
Test Data #58 from thread #20
Test Data #59 from thread #20
Test Data #60 from thread #20
Test Data #61 from thread #20
Test Data #62 from thread #20
Test Data #63 from thread #20
Test Data #64 from thread #20
Test Data #65 from thread #20
Test Data #66 from thread #20
Test Data #67 from thread #20
Test Data #68 from thread #20
Test Data #69 from thread #20
Test Data #70 from thread #20
Test Data #71 from thread #20
Test Data #72 from thread #20
Test Data #73 from thread #20
Test Data #74 from thread #20
Test Data #75 from thread #20
Test Data #76 from thread #20
Test Data #77 from thread #20
Test Data #78 from thread #20
Test Data #79 from thread #20
Test Data #80 from thread #20
Test Data #81 from thread #20
Test Data #82 from thread #20
Test Data #83 from thread #20
Test Data #84 from thread #20
Test Data #85 from thread #20
Test Data #86 from thread #20
Test Data #87 from thread #20
Test Data #88 from thread #20
Test Data #89 from thread #20
Test Data #90 from thread #20
Test Data #91 from thread #20
Test Data #92 from thread #20
Test Data #93 from thread #20
Test Data #94 from thread #20
Test Data #95 from thread #20
Test Data #96 from thread #20
Test Data #97 from thread #20
Test Data #98 from thread #20
Test Data #99 from thread #20
Finished sending 100 messages from thread #20!
Administrator: Command Prompt - mvn exec:java -Dexec.mainC... Consumer Record:(1664306444219, Test Data #57, 0, 53)
Consumer Record:(1664306444219, Test Data #58, 0, 54)
Consumer Record:(1664306444221, Test Data #59, 0, 55)
Consumer Record:(1664306444222, Test Data #60, 0, 56)
Consumer Record:(1664306444223, Test Data #61, 0, 57)
Consumer Record:(1664306444223, Test Data #62, 0, 58)
Consumer Record:(1664306444224, Test Data #63, 0, 59)
Consumer Record:(1664306444227, Test Data #64, 0, 60)
Consumer Record:(1664306444228, Test Data #65, 0, 61)
Consumer Record:(1664306444230, Test Data #66, 0, 62)
Consumer Record:(1664306444231, Test Data #67, 0, 63)
Consumer Record:(1664306444232, Test Data #68, 0, 64)
Consumer Record:(1664306444233, Test Data #69, 0, 65)
Consumer Record:(1664306444234, Test Data #70, 0, 66)
Consumer Record:(1664306444235, Test Data #71, 0, 67)
Consumer Record:(1664306444237, Test Data #72, 0, 68)
Consumer Record:(1664306444239, Test Data #73, 0, 69)
Consumer Record:(1664306444240, Test Data #74, 0, 70)
Consumer Record:(1664306444240, Test Data #75, 0, 71)
Consumer Record:(1664306444241, Test Data #76, 0, 72)
Consumer Record:(1664306444242, Test Data #77, 0, 73)
Consumer Record:(1664306444243, Test Data #78, 0, 74)
Consumer Record:(1664306444243, Test Data #79, 0, 75)
Consumer Record:(1664306444244, Test Data #80, 0, 76)
Consumer Record:(1664306444245, Test Data #81, 0, 77)
Consumer Record:(1664306444246, Test Data #82, 0, 78)
Consumer Record:(1664306444246, Test Data #83, 0, 79)
Consumer Record:(1664306444249, Test Data #84, 0, 80)
Consumer Record:(1664306444250, Test Data #85, 0, 81)
Consumer Record:(1664306444250, Test Data #86, 0, 82)
Consumer Record:(1664306444252, Test Data #87, 0, 83)
Consumer Record:(1664306444253, Test Data #88, 0, 84)
Consumer Record:(1664306444254, Test Data #89, 0, 85)
Consumer Record:(1664306444254, Test Data #90, 0, 86)
Consumer Record:(1664306444255, Test Data #91, 0, 87)
Consumer Record:(1664306444256, Test Data #92, 0, 88)
Consumer Record:(1664306444257, Test Data #93, 0, 89)
Consumer Record:(1664306444257, Test Data #94, 0, 90)
Consumer Record:(1664306444260, Test Data #95, 0, 91)
Consumer Record:(1664306444260, Test Data #96, 0, 92)
Consumer Record:(1664306444261, Test Data #97, 0, 93)
Consumer Record:(1664306444263, Test Data #98, 0, 94)
Consumer Record:(1664306444272, Test Data #99, 0, 95)
Consumer Record:(1664306444137, Test Data #6, 0, 96)
Consumer Record:(1664306444086, Test Data #4, 0, 97)
Consumer Record:(1664306444141, Test Data #7, 0, 98)
Consumer Record:(1664306444120, Test Data #5, 0, 99)
```

Schema validation for Kafka with Schema Registry

You can use Azure Schema Registry to perform schema validation when you stream data with your Kafka applications using Event Hubs. Azure Schema Registry of Event Hubs provides a centralized repository for managing schemas and you can seamlessly connect your new or existing Kafka applications with Schema Registry.

To learn more, see [Validate schemas for Apache Kafka applications using Avro](#).

Next steps

In this article, you learned how to stream into Event Hubs without changing your protocol clients or running your own clusters. To learn more, see [Apache Kafka developer guide for Azure Event Hubs](#).

Quickstart: Azure Key Vault Certificate client library for Java (Certificates)

Article • 04/07/2024

Get started with the Azure Key Vault Certificate client library for Java. Follow the steps below to install the package and try out example code for basic tasks.

Tip

If you're working with Azure Key Vault Certificates resources in a Spring application, we recommend that you consider [Spring Cloud Azure](#) as an alternative. Spring Cloud Azure is an open-source project that provides seamless Spring integration with Azure services. To learn more about Spring Cloud Azure, and to see an example using Key Vault Certificates, see [Enable HTTPS in Spring Boot with Azure Key Vault certificates](#).

Additional resources:

- [Source code ↗](#)
- [API reference documentation ↗](#)
- [Product documentation](#)
- [Samples ↗](#)

Prerequisites

- An Azure subscription - [create one for free ↗](#).
- [Java Development Kit \(JDK\)](#) version 8 or above
- [Apache Maven ↗](#)
- [Azure CLI](#)

This quickstart assumes you are running [Azure CLI](#) and [Apache Maven ↗](#) in a Linux terminal window.

Setting up

This quickstart is using the Azure Identity library with Azure CLI to authenticate user to Azure Services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls, for more information, see [Authenticate the client with Azure Identity client library](#).

Sign in to Azure

1. Run the `login` command.

```
Azure CLI
```

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

Create a new Java console app

In a console window, use the `mvn` command to create a new Java console app with the name `akv-certificates-java`.

```
Console
```

```
mvn archetype:generate -DgroupId=com.keyvault.certificates.quickstart  
-DartifactId=akv-certificates-java  
-DarchetypeArtifactId=maven-archetype-quickstart  
-DarchetypeVersion=1.4  
-DinteractiveMode=false
```

The output from generating the project will look something like this:

```
Console
```

```
[INFO] -----  
-----  
[INFO] Using following parameters for creating project from Archetype:  
maven-archetype-quickstart:1.4  
[INFO] -----  
-----  
[INFO] Parameter: groupId, Value: com.keyvault.certificates.quickstart  
[INFO] Parameter: artifactId, Value: akv-certificates-java  
[INFO] Parameter: version, Value: 1.0-SNAPSHOT  
[INFO] Parameter: package, Value: com.keyvault.certificates.quickstart  
[INFO] Parameter: packageInPathFormat, Value: com/keyvault/quickstart  
[INFO] Parameter: package, Value: com.keyvault.certificates.quickstart  
[INFO] Parameter: groupId, Value: com.keyvault.certificates.quickstart  
[INFO] Parameter: artifactId, Value: akv-certificates-java
```

```
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Project created from Archetype in dir: /home/user/quickstarts/akv-
certificates-java
[INFO] -----
---
[INFO] BUILD SUCCESS
[INFO] -----
---
[INFO] Total time:  38.124 s
[INFO] Finished at: 2019-11-15T13:19:06-08:00
[INFO] -----
---
```

Change your directory to the newly created `akv-certificates-java/` folder.

Console

```
cd akv-certificates-java
```

Install the package

Open the `pom.xml` file in your text editor. Add the following dependency elements to the group of dependencies.

XML

```
<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-security-keyvault-certificates</artifactId>
  <version>4.1.3</version>
</dependency>

<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-identity</artifactId>
  <version>1.2.0</version>
</dependency>
```

Create a resource group and key vault

This quickstart uses a pre-created Azure key vault. You can create a key vault by following the steps in the [Azure CLI quickstart](#), [Azure PowerShell quickstart](#), or [Azure portal quickstart](#).

Alternatively, you can simply run the Azure CLI or Azure PowerShell commands below.

ⓘ Important

Each key vault must have a unique name. Replace <your-unique-keyvault-name> with the name of your key vault in the following examples.

Azure CLI

Azure CLI

```
az group create --name "myResourceGroup" -l "EastUS"  
az keyvault create --name "<your-unique-keyvault-name>" -g  
"myResourceGroup"
```

Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

Set environment variables

This application is using your key vault name as an environment variable called `KEY_VAULT_NAME`.

Windows

Windows Command Prompt

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

Windows PowerShell

PowerShell

```
$Env:KEY_VAULT_NAME=""
```

macOS or Linux

Windows Command Prompt

```
export KEY_VAULT_NAME=<your-key-vault-name>
```

Object model

The Azure Key Vault Certificate client library for Java allows you to manage certificates. The [Code examples](#) section shows how to create a client, create a certificate, retrieve a certificate, and delete a certificate.

The entire console app is [below](#).

Code examples

Add directives

Add the following directives to the top of your code:

Java

```
import com.azure.core.util.polling.SyncPoller;
import com.azure.identity.DefaultAzureCredentialBuilder;

import com.azure.security.keyvault.certificates.CertificateClient;
import com.azure.security.keyvault.certificates.CertificateClientBuilder;
import com.azure.security.keyvault.certificates.models.CertificateOperation;
import com.azure.security.keyvault.certificates.models.CertificatePolicy;
import com.azure.security.keyvault.certificates.models.DeletedCertificate;
import com.azure.security.keyvault.certificates.models.KeyVaultCertificate;
import
```

```
com.azure.security.keyvault.certificates.models.KeyVaultCertificateWithPolicy;
```

Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this example, the name of your key vault is expanded to the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

Java

```
String keyVaultName = System.getenv("KEY_VAULT_NAME");
String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";

CertificateClient certificateClient = new CertificateClientBuilder()
    .vaultUrl(keyVaultUri)
    .credential(new DefaultAzureCredentialBuilder().build())
    .buildClient();
```

Save a secret

Now that your application is authenticated, you can create a certificate in your key vault using the `certificateClient.beginCreateCertificate` method. This requires a name for the certificate and a certificate policy -- we've assigned the value "myCertificate" to the `certificateName` variable in this sample and use a default policy.

Certificate creation is a long running operation, for which you can poll its progress or wait for it to complete.

Java

```
SyncPoller<CertificateOperation, KeyVaultCertificateWithPolicy>
certificatePoller =
    certificateClient.beginCreateCertificate(certificateName,
CertificatePolicy.getDefault());
certificatePoller.waitForCompletion();
```

You can obtain the certificate once creation has completed with via the following call:

Java

```
KeyVaultCertificate createdCertificate = certificatePoller.getFinalResult();
```

Retrieve a certificate

You can now retrieve the previously created certificate with the

`certificateClient.getCertificate` method.

Java

```
KeyVaultCertificate retrievedCertificate =
certificateClient.getCertificate(certificateName);
```

You can now access the details of the retrieved certificate with operations like

`retrievedCertificate.getName`, `retrievedCertificate.getProperties`, etc. As well as its contents `retrievedCertificate.getConten`.

Delete a certificate

Finally, let's delete the certificate from your key vault with the

`certificateClient.beginDeleteCertificate` method, which is also a long running operation.

Java

```
SyncPoller<DeletedCertificate, Void> deletionPoller =
certificateClient.beginDeleteCertificate(certificateName);
deletionPoller.waitForCompletion();
```

Clean up resources

When no longer needed, you can use the Azure CLI or Azure PowerShell to remove your key vault and the corresponding resource group.

Azure CLI

```
az group delete -g "myResourceGroup"
```

Azure PowerShell

```
Remove-AzResourceGroup -Name "myResourceGroup"
```

Sample code

Java

```
package com.keyvault.certificates.quickstart;

import com.azure.core.util.polling.SyncPoller;
import com.azure.identity.DefaultAzureCredentialBuilder;

import com.azure.security.keyvault.certificates.CertificateClient;
import com.azure.security.keyvault.certificates.CertificateClientBuilder;
import com.azure.security.keyvault.certificates.models.CertificateOperation;
import com.azure.security.keyvault.certificates.models.CertificatePolicy;
import com.azure.security.keyvault.certificates.models.DeletedCertificate;
import com.azure.security.keyvault.certificates.models.KeyVaultCertificate;
import
com.azure.security.keyvault.certificates.models.KeyVaultCertificateWithPolic
y;

public class App {
    public static void main(String[] args) throws InterruptedException,
IllegalArgumentException {
        String keyVaultName = System.getenv("KEY_VAULT_NAME");
        String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";

        System.out.printf("key vault name = %s and kv uri = %s \n",
keyVaultName, keyVaultUri);

        CertificateClient certificateClient = new CertificateClientBuilder()
            .vaultUrl(keyVaultUri)
            .credential(new DefaultAzureCredentialBuilder().build())
            .buildClient();

        String certificateName = "myCertificate";

        System.out.print("Creating a certificate in " + keyVaultName + "
called '" + certificateName + " ... ");
    }
}
```

```

        SyncPoller<CertificateOperation, KeyVaultCertificateWithPolicy>
certificatePoller =
    certificateClient.beginCreateCertificate(certificateName,
CertificatePolicy.getDefault());
certificatePoller.waitForCompletion();

System.out.print("done.");
System.out.println("Retrieving certificate from " + keyVaultName +
".");

KeyVaultCertificate retrievedCertificate =
certificateClient.getCertificate(certificateName);

System.out.println("Your certificate's ID is '" +
retrievedCertificate.getId() + "'.");

System.out.println("Deleting your certificate from " + keyVaultName
+ " ...");

SyncPoller<DeletedCertificate, Void> deletionPoller =
certificateClient.beginDeleteCertificate(certificateName);
deletionPoller.waitForCompletion();

System.out.print("done.");
}
}

```

Next steps

In this quickstart you created a key vault, created a certificate, retrieved it, and then deleted it. To learn more about Key Vault and how to integrate it with your applications, continue on to the articles below.

- Read an [Overview of Azure Key Vault](#)
- See the [Azure Key Vault developer's guide](#)
- How to [Secure access to a key vault](#)

Quickstart: Azure Key Vault Key client library for Java

Article • 04/07/2024

Get started with the Azure Key Vault Key client library for Java. Follow these steps to install the package and try out example code for basic tasks.

Additional resources:

- [Source code ↗](#)
- [API reference documentation ↗](#)
- [Product documentation](#)
- [Samples ↗](#)

Prerequisites

- An Azure subscription - [create one for free ↗](#).
- [Java Development Kit \(JDK\)](#) version 8 or above
- [Apache Maven ↗](#)
- [Azure CLI](#)

This quickstart assumes you're running [Azure CLI](#) and [Apache Maven ↗](#) in a Linux terminal window.

Setting up

This quickstart is using the Azure Identity library with Azure CLI to authenticate user to Azure Services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls, for more information, see [Authenticate the client with Azure Identity client library](#).

Sign in to Azure

1. Run the `login` command.

```
Azure CLI
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

Create a new Java console app

In a console window, use the `mvn` command to create a new Java console app with the name `akv-keys-java`.

```
Console

mvn archetype:generate -DgroupId=com.keyvault.keys.quickstart
-DartifactId=akv-keys-java
-DarchetypeArtifactId=maven-archetype-quickstart
-DarchetypeVersion=1.4
-DinteractiveMode=false
```

The output from generating the project will look something like this:

```
Console

[INFO] -----
-----
[INFO] Using following parameters for creating project from Archetype:
maven-archetype-quickstart:1.4
[INFO] -----
-----
[INFO] Parameter: groupId, Value: com.keyvault.keys.quickstart
[INFO] Parameter: artifactId, Value: akv-keys-java
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.keyvault.keys.quickstart
[INFO] Parameter: packageInPathFormat, Value: com/keyvault/quickstart
[INFO] Parameter: package, Value: com.keyvault.keys.quickstart
[INFO] Parameter: groupId, Value: com.keyvault.keys.quickstart
[INFO] Parameter: artifactId, Value: akv-keys-java
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Project created from Archetype in dir: /home/user/quickstarts/akv-
keys-java
[INFO] -----
---
[INFO] BUILD SUCCESS
[INFO] -----
---
[INFO] Total time: 38.124 s
[INFO] Finished at: 2019-11-15T13:19:06-08:00
```

[INFO] -----

Change your directory to the newly created `akv-keys-java/` folder.

Console

```
cd akv-keys-java
```

Install the package

Open the `pom.xml` file in your text editor. Add the following dependency elements to the group of dependencies.

XML

```
<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-security-keyvault-keys</artifactId>
  <version>4.2.3</version>
</dependency>

<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-identity</artifactId>
  <version>1.2.0</version>
</dependency>
```

Create a resource group and key vault

This quickstart uses a pre-created Azure key vault. You can create a key vault by following the steps in the [Azure CLI quickstart](#), [Azure PowerShell quickstart](#), or [Azure portal quickstart](#).

Alternatively, you can simply run the Azure CLI or Azure PowerShell commands below.

Important

Each key vault must have a unique name. Replace `<your-unique-keyvault-name>` with the name of your key vault in the following examples.

Azure CLI

Azure CLI

```
az group create --name "myResourceGroup" -l "EastUS"  
az keyvault create --name "<your-unique-keyvault-name>" -g  
"myResourceGroup"
```

Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

Set environment variables

This application is using your key vault name as an environment variable called `KEY_VAULT_NAME`.

Windows

Windows Command Prompt

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

Windows PowerShell

PowerShell

```
$Env:KEY_VAULT_NAME=<your-key-vault-name>"
```

macOS or Linux

Windows Command Prompt

```
export KEY_VAULT_NAME=<your-key-vault-name>
```

Object model

The Azure Key Vault Key client library for Java allows you to manage keys. The [Code examples](#) section shows how to create a client, create a key, retrieve a key, and delete a key.

The entire console app is supplied in [Sample code](#).

Code examples

Add directives

Add the following directives to the top of your code:

Java

```
import com.azure.core.util.polling.SyncPoller;
import com.azure.identity.DefaultAzureCredentialBuilder;

import com.azure.security.keyvault.keys.KeyClient;
import com.azure.security.keyvault.keys.KeyClientBuilder;
import com.azure.security.keyvault.keys.models.DeletedKey;
import com.azure.security.keyvault.keys.models.KeyType;
import com.azure.security.keyvault.keys.models.KeyVaultKey;
```

Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) class is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication

methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this example, the name of your key vault is expanded to the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

Java

```
String keyVaultName = System.getenv("KEY_VAULT_NAME");
String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";

KeyClient keyClient = new KeyClientBuilder()
    .vaultUrl(keyVaultUri)
    .credential(new DefaultAzureCredentialBuilder().build())
    .buildClient();
```

Create a key

Now that your application is authenticated, you can create a key in your key vault using the `keyClient.createKey` method. This requires a name for the key and a key type. We've assigned the value "myKey" to the `keyName` variable and use a RSA `KeyType` in this sample.

Java

```
keyClient.createKey(keyName, KeyType.RSA);
```

You can verify that the key has been set with the `az keyvault key show` command:

Azure CLI

```
az keyvault key show --vault-name <your-unique-key-vault-name> --name myKey
```

Retrieve a key

You can now retrieve the previously created key with the `keyClient.getKey` method.

Java

```
KeyVaultKey retrievedKey = keyClient.getKey(keyName);
```

You can now access the details of the retrieved key with operations like

`retrievedKey.getProperties`, `retrievedKey.getKeyOperations`, etc.

Delete a key

Finally, let's delete the key from your key vault with the `keyClient.beginDeleteKey` method.

Key deletion is a long running operation, for which you can poll its progress or wait for it to complete.

Java

```
SyncPoller<DeletedKey, Void> deletionPoller =
keyClient.beginDeleteKey(keyName);
deletionPoller.waitForCompletion();
```

You can verify that the key has been deleted with the [az keyvault key show](#) command:

Azure CLI

```
az keyvault key show --vault-name <your-unique-key-vault-name> --name myKey
```

Clean up resources

When no longer needed, you can use the Azure CLI or Azure PowerShell to remove your key vault and the corresponding resource group.

Azure CLI

```
az group delete -g "myResourceGroup"
```

Azure PowerShell

```
Remove-AzResourceGroup -Name "myResourceGroup"
```

Sample code

Java

```
package com.keyvault.keys.quickstart;

import com.azure.core.util.polling.SyncPoller;
import com.azure.identity.DefaultAzureCredentialBuilder;

import com.azure.security.keyvault.keys.KeyClient;
import com.azure.security.keyvault.keys.KeyClientBuilder;
import com.azure.security.keyvault.keys.models.DeletedKey;
import com.azure.security.keyvault.keys.models.KeyType;
import com.azure.security.keyvault.keys.models.KeyVaultKey;

public class App {
    public static void main(String[] args) throws InterruptedException,
IllegalArgumentException {
        String keyVaultName = System.getenv("KEY_VAULT_NAME");
        String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";

        System.out.printf("key vault name = %s and key vault URI = %s \n",
keyVaultName, keyVaultUri);

        KeyClient keyClient = new KeyClientBuilder()
            .vaultUrl(keyVaultUri)
            .credential(new DefaultAzureCredentialBuilder().build())
            .buildClient();

        String keyName = "myKey";

        System.out.print("Creating a key in " + keyVaultName + " called '" +
keyName + " ... ");

        keyClient.createKey(keyName, KeyType.RSA);

        System.out.print("done.");
        System.out.println("Retrieving key from " + keyVaultName + ".");

        KeyVaultKey retrievedKey = keyClient.getKey(keyName);

        System.out.println("Your key's ID is '" + retrievedKey.getId() +
".'");
        System.out.println("Deleting your key from " + keyVaultName + " ...
");

        SyncPoller<DeletedKey, Void> deletionPoller =
keyClient.beginDeleteKey(keyName);
        deletionPoller.waitForCompletion();

        System.out.print("done.");
    }
}
```

```
    }  
}
```

Next steps

In this quickstart, you created a key vault, created a key, retrieved it, and then deleted it. To learn more about Key Vault and how to integrate it with your applications, continue on to these articles.

- Read an [Overview of Azure Key Vault](#)
- Read the [Key Vault security overview](#)
- See the [Azure Key Vault developer's guide](#)
- How to [Secure access to a key vault](#)

Quickstart: Azure Key Vault Secret client library for Java

Article • 04/07/2024

Get started with the Azure Key Vault Secret client library for Java. Follow these steps to install the package and try out example code for basic tasks.

Tip

If you're working with Azure Key Vault Secrets resources in a Spring application, we recommend that you consider [Spring Cloud Azure](#) as an alternative. Spring Cloud Azure is an open-source project that provides seamless Spring integration with Azure services. To learn more about Spring Cloud Azure, and to see an example using Key Vault Secrets, see [Load a secret from Azure Key Vault in a Spring Boot application](#).

Additional resources:

- [Source code ↗](#)
- [API reference documentation ↗](#)
- [Product documentation](#)
- [Samples ↗](#)

Prerequisites

- An Azure subscription - [create one for free ↗](#).
- [Java Development Kit \(JDK\)](#) version 8 or above
- [Apache Maven ↗](#)
- [Azure CLI](#)

This quickstart assumes you're running [Azure CLI](#) and [Apache Maven ↗](#) in a Linux terminal window.

Setting up

This quickstart is using the Azure Identity library with Azure CLI to authenticate user to Azure Services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls, for more information, see [Authenticate the client with Azure Identity client library](#).

Sign in to Azure

1. Run the `login` command.

```
Azure CLI  
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

Create a new Java console app

In a console window, use the `mvn` command to create a new Java console app with the name `akv-secrets-java`.

```
Console  
  
mvn archetype:generate -DgroupId=com.keyvault.secrets.quickstart  
-DartifactId=akv-secrets-java  
-DarchetypeArtifactId=maven-archetype-quickstart  
-DarchetypeVersion=1.4  
-DinteractiveMode=false
```

The output from generating the project will look something like this:

```
Console  
  
[INFO] -----  
-----  
[INFO] Using following parameters for creating project from Archetype:  
maven-archetype-quickstart:1.4  
[INFO] -----  
-----  
[INFO] Parameter: groupId, Value: com.keyvault.secrets.quickstart  
[INFO] Parameter: artifactId, Value: akv-secrets-java  
[INFO] Parameter: version, Value: 1.0-SNAPSHOT  
[INFO] Parameter: package, Value: com.keyvault.secrets.quickstart  
[INFO] Parameter: packageInPathFormat, Value: com/keyvault/quickstart  
[INFO] Parameter: package, Value: com.keyvault.secrets.quickstart  
[INFO] Parameter: groupId, Value: com.keyvault.secrets.quickstart  
[INFO] Parameter: artifactId, Value: akv-secrets-java
```

```
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Project created from Archetype in dir: /home/user/quickstarts/akv-
secrets-java
[INFO] -----
---
[INFO] BUILD SUCCESS
[INFO] -----
---
[INFO] Total time: 38.124 s
[INFO] Finished at: 2019-11-15T13:19:06-08:00
[INFO] -----
---
```

Change your directory to the newly created `akv-secrets-java/` folder.

Azure CLI

```
cd akv-secrets-java
```

Install the package

Open the `pom.xml` file in your text editor. Add the following dependency elements to the group of dependencies.

XML

```
<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-security-keyvault-secrets</artifactId>
  <version>4.2.3</version>
</dependency>

<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-identity</artifactId>
  <version>1.2.0</version>
</dependency>
```

Create a resource group and key vault

This quickstart uses a pre-created Azure key vault. You can create a key vault by following the steps in the [Azure CLI quickstart](#), [Azure PowerShell quickstart](#), or [Azure portal quickstart](#).

Alternatively, you can simply run the Azure CLI or Azure PowerShell commands below.

ⓘ Important

Each key vault must have a unique name. Replace <your-unique-keyvault-name> with the name of your key vault in the following examples.

Azure CLI

Azure CLI

```
az group create --name "myResourceGroup" -l "EastUS"  
az keyvault create --name "<your-unique-keyvault-name>" -g  
"myResourceGroup"
```

Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

Set environment variables

This application is using your key vault name as an environment variable called `KEY_VAULT_NAME`.

Windows

Windows Command Prompt

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

Windows PowerShell

PowerShell

```
$Env:KEY_VAULT_NAME=""
```

macOS or Linux

Windows Command Prompt

```
export KEY_VAULT_NAME=<your-key-vault-name>
```

Object model

The Azure Key Vault Secret client library for Java allows you to manage secrets. The [Code examples](#) section shows how to create a client, set a secret, retrieve a secret, and delete a secret.

Code examples

Add directives

Add the following directives to the top of your code:

Java

```
import com.azure.core.util.polling.SyncPoller;
import com.azure.identity.DefaultAzureCredentialBuilder;

import com.azure.security.keyvault.secrets.SecretClient;
import com.azure.security.keyvault.secrets.SecretClientBuilder;
import com.azure.security.keyvault.secrets.models.DeletedSecret;
import com.azure.security.keyvault.secrets.models.KeyVaultSecret;
```

Authenticate and create a client

Application requests to most Azure services must be authorized. Using the `DefaultAzureCredential` class is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this example, the name of your key vault is expanded to the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

Java

```
String keyVaultName = System.getenv("KEY_VAULT_NAME");
String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";

SecretClient secretClient = new SecretClientBuilder()
    .vaultUrl(keyVaultUri)
    .credential(new DefaultAzureCredentialBuilder().build())
    .buildClient();
```

Save a secret

Now that your application is authenticated, you can put a secret into your key vault using the `secretClient.setSecret` method. This requires a name for the secret—we've assigned the value "mySecret" to the `secretName` variable in this sample.

Java

```
secretClient.setSecret(new KeyVaultSecret(secretName, secretValue));
```

You can verify that the secret has been set with the `az keyvault secret show` command:

Azure CLI

```
az keyvault secret show --vault-name <your-unique-key-vault-name> --name  
mySecret
```

Retrieve a secret

You can now retrieve the previously set secret with the `secretClient.getSecret` method.

Java

```
KeyVaultSecret retrievedSecret = secretClient.getSecret(secretName);
```

You can now access the value of the retrieved secret with `retrievedSecret.getValue()`.

Delete a secret

Finally, let's delete the secret from your key vault with the `secretClient.beginDeleteSecret` method.

Secret deletion is a long running operation, for which you can poll its progress or wait for it to complete.

Java

```
SyncPoller<DeletedSecret, Void> deletionPoller =  
secretClient.beginDeleteSecret(secretName);  
deletionPoller.waitForCompletion();
```

You can verify that the secret has been deleted with the `az keyvault secret show` command:

Azure CLI

```
az keyvault secret show --vault-name <your-unique-key-vault-name> --name  
mySecret
```

Clean up resources

When no longer needed, you can use the Azure CLI or Azure PowerShell to remove your key vault and the corresponding resource group.

Azure CLI

```
az group delete -g "myResourceGroup"
```

Azure PowerShell

```
Remove-AzResourceGroup -Name "myResourceGroup"
```

Sample code

Java

```
package com.keyvault.secrets.quickstart;

import java.io.Console;

import com.azure.core.util.polling.SyncPoller;
import com.azure.identity.DefaultAzureCredentialBuilder;

import com.azure.security.keyvault.secrets.SecretClient;
import com.azure.security.keyvault.secrets.SecretClientBuilder;
import com.azure.security.keyvault.secrets.models.DeletedSecret;
import com.azure.security.keyvault.secrets.models.KeyVaultSecret;

public class App {
    public static void main(String[] args) throws InterruptedException,
IllegalArgumentException {
        String keyVaultName = System.getenv("KEY_VAULT_NAME");
        String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";

        System.out.printf("key vault name = %s and key vault URI = %s \n",
keyVaultName, keyVaultUri);

        SecretClient secretClient = new SecretClientBuilder()
            .vaultUrl(keyVaultUri)
            .credential(new DefaultAzureCredentialBuilder().build())
            .buildClient();

        Console con = System.console();

        String secretName = "mySecret";

        System.out.println("Please provide the value of your secret > ");

        String secretValue = con.readLine();

        System.out.print("Creating a secret in " + keyVaultName + " called
" + secretName + "' with value '" + secretValue + "' ... ");

        secretClient.setSecret(new KeyVaultSecret(secretName, secretValue));
    }
}
```

```
        System.out.println("done.");
        System.out.println("Forgetting your secret.");

        secretValue = "";
        System.out.println("Your secret's value is " + secretValue + ".");
```

```
        System.out.println("Retrieving your secret from " + keyVaultName +
".");

        KeyVaultSecret retrievedSecret = secretClient.getSecret(secretName);

        System.out.println("Your secret's value is " +
retrievedSecret.getValue() + ".");
        System.out.print("Deleting your secret from " + keyVaultName + " ...
");

        SyncPoller<DeletedSecret, Void> deletionPoller =
secretClient.beginDeleteSecret(secretName);
        deletionPoller.waitForCompletion();

        System.out.println("done.");
    }
}
```

Next steps

In this quickstart, you created a key vault, stored a secret, retrieved it, and then deleted it. To learn more about Key Vault and how to integrate it with your applications, continue on to these articles.

- Read an [Overview of Azure Key Vault](#)
- See the [Azure Key Vault developer's guide](#)
- How to [Secure access to a key vault](#)

Quickstart: Use Java and JDBC with Azure Database for MySQL

Article • 05/04/2023

APPLIES TO:  Azure Database for MySQL - Single Server

Important

Azure Database for MySQL - Single Server is on the retirement path. We strongly recommend for you to upgrade to Azure Database for MySQL - Flexible Server. For more information about migrating to Azure Database for MySQL - Flexible Server, see [What's happening to Azure Database for MySQL Single Server?](#)

This article demonstrates creating a sample application that uses Java and [JDBC](#) to store and retrieve information in [Azure Database for MySQL](#).

JDBC is the standard Java API to connect to traditional relational databases.

In this article, we'll include two authentication methods: Microsoft Entra authentication and MySQL authentication. The **Passwordless** tab shows the Microsoft Entra authentication and the **Password** tab shows the MySQL authentication.

Microsoft Entra authentication is a mechanism for connecting to Azure Database for MySQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

MySQL authentication uses accounts stored in MySQL. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `user` table. Because these passwords are stored in MySQL, you'll need to manage the rotation of the passwords by yourself.

Prerequisites

- An Azure account. If you don't have one, [get a free trial](#).
- [Azure Cloud Shell](#) or [Azure CLI](#). We recommend Azure Cloud Shell so you'll be logged in automatically and have access to all the tools you'll need.
- A supported [Java Development Kit](#), version 8 (included in Azure Cloud Shell).
- The [Apache Maven](#) build tool.

- MySQL command line client. You can connect to your server using the [mysql.exe](#) command-line tool with Azure Cloud Shell. Alternatively, you can use the `mysql` command line in your local environment.

Prepare the working environment

First, set up some environment variables. In [Azure Cloud Shell](#), run the following commands:

Passwordless (Recommended)

Bash

```
export AZ_RESOURCE_GROUP=database-workshop
export AZ_DATABASE_SERVER_NAME=<YOUR_DATABASE_SERVER_NAME>
export AZ_DATABASE_NAME=demo
export AZ_LOCATION=<YOUR_AZURE_REGION>
export AZ_MYSQL_AD_NON_ADMIN_USERNAME=demo-non-admin
export AZ_LOCAL_IP_ADDRESS=<YOUR_LOCAL_IP_ADDRESS>
export CURRENT_USERNAME=$(az ad signed-in-user show --query userPrincipalName -o tsv)
export CURRENT_USER_OBJECTID=$(az ad signed-in-user show --query id -o tsv)
```

Replace the placeholders with the following values, which are used throughout this article:

- <YOUR_DATABASE_SERVER_NAME>: The name of your MySQL server, which should be unique across Azure.
- <YOUR_AZURE_REGION>: The Azure region you'll use. You can use `eastus` by default, but we recommend that you configure a region closer to where you live. You can see the full list of available regions by entering `az account list-locations`.
- <YOUR_LOCAL_IP_ADDRESS>: The IP address of your local computer, from which you'll run your application. One convenient way to find it is to open [whatismyip.akamai.com](#).

Next, create a resource group by using the following command:

Azure CLI

```
az group create \
--name $AZ_RESOURCE_GROUP \
```

```
--location $AZ_LOCATION \
--output tsv
```

Create an Azure Database for MySQL instance

Create a MySQL server and set up admin user

The first thing you'll create is a managed MySQL server.

ⓘ Note

You can read more detailed information about creating MySQL servers in [Quickstart: Create an Azure Database for MySQL server by using the Azure portal](#).

Passwordless (Recommended)

If you're using Azure CLI, run the following command to make sure it has sufficient permission:

Azure CLI

```
az login --scope https://graph.microsoft.com/.default
```

Then, run the following command to create the server:

Azure CLI

```
az mysql server create \
--resource-group $AZ_RESOURCE_GROUP \
--name $AZ_DATABASE_SERVER_NAME \
--location $AZ_LOCATION \
--sku-name B_Gen5_1 \
--storage-size 5120 \
--output tsv
```

Next, run the following command to set the Microsoft Entra admin user:

Azure CLI

```
az mysql server ad-admin create \
--resource-group $AZ_RESOURCE_GROUP \
--server-name $AZ_DATABASE_SERVER_NAME \
```

```
--display-name $CURRENT_USERNAME \
--object-id $CURRENT_USER_OBJECTID
```

ⓘ Important

When setting the administrator, a new user is added to the Azure Database for MySQL server with full administrator permissions. You can only create one Microsoft Entra admin per MySQL server. Selection of another user will overwrite the existing Microsoft Entra admin configured for the server.

This command creates a small MySQL server and sets the Active Directory admin to the signed-in user.

Configure a firewall rule for your MySQL server

Azure Databases for MySQL instances are secured by default. These instances have a firewall that doesn't allow any incoming connection. To be able to use your database, you need to add a firewall rule that will allow the local IP address to access the database server.

Because you configured your local IP address at the beginning of this article, you can open the server's firewall by running the following command:

Azure CLI

```
az mysql server firewall-rule create \
  --resource-group $AZ_RESOURCE_GROUP \
  --name $AZ_DATABASE_SERVER_NAME-database-allow-local-ip \
  --server $AZ_DATABASE_SERVER_NAME \
  --start-ip-address $AZ_LOCAL_IP_ADDRESS \
  --end-ip-address $AZ_LOCAL_IP_ADDRESS \
  --output tsv
```

If you're connecting to your MySQL server from Windows Subsystem for Linux (WSL) on a Windows computer, you'll need to add the WSL host ID to your firewall.

Obtain the IP address of your host machine by running the following command in WSL:

Bash

```
cat /etc/resolv.conf
```

Copy the IP address following the term `nameserver`, then use the following command to set an environment variable for the WSL IP Address:

Bash

```
AZ_WSL_IP_ADDRESS=<the-copied-IP-address>
```

Then, use the following command to open the server's firewall to your WSL-based app:

Azure CLI

```
az mysql server firewall-rule create \
    --resource-group $AZ_RESOURCE_GROUP \
    --name $AZ_DATABASE_SERVER_NAME-database-allow-local-ip-wsl \
    --server $AZ_DATABASE_SERVER_NAME \
    --start-ip-address $AZ_WSL_IP_ADDRESS \
    --end-ip-address $AZ_WSL_IP_ADDRESS \
    --output tsv
```

Configure a MySQL database

The MySQL server that you created earlier is empty. Use the following command to create a new database.

Azure CLI

```
az mysql db create \
    --resource-group $AZ_RESOURCE_GROUP \
    --name $AZ_DATABASE_NAME \
    --server-name $AZ_DATABASE_SERVER_NAME \
    --output tsv
```

Create a MySQL non-admin user and grant permission

Next, create a non-admin user and grant all permissions to the database.

 **Note**

You can read more detailed information about creating MySQL users in [Create users in Azure Database for MySQL](#).

>Passwordless (Recommended)

Create a SQL script called *create_ad_user.sql* for creating a non-admin user. Add the following contents and save it locally:

Bash

```
export AZ_MYSQL_AD_NON_ADMIN_USERID=$CURRENT_USER_OBJECTID

cat << EOF > create_ad_user.sql
SET aad_auth_validate_oids_in_tenant = OFF;

CREATE AADUSER '$AZ_MYSQL_AD_NON_ADMIN_USERNAME' IDENTIFIED BY
'$AZ_MYSQL_AD_NON_ADMIN_USERNAME';

GRANT ALL PRIVILEGES ON $AZ_DATABASE_NAME.* TO
'$AZ_MYSQL_AD_NON_ADMIN_USERNAME'@'%';

FLUSH privileges;

EOF
```

Then, use the following command to run the SQL script to create the Microsoft Entra non-admin user:

Bash

```
mysql -h $AZ_DATABASE_SERVER_NAME.mysql.database.azure.com --user
$CURRENT_USERNAME@$AZ_DATABASE_SERVER_NAME --enable-cleartext-plugin --
password=$(az account get-access-token --resource-type oss-rdbms --
output tsv --query accessToken) < create_ad_user.sql
```

Now use the following command to remove the temporary SQL script file:

Bash

```
rm create_ad_user.sql
```

Create a new Java project

Using your favorite IDE, create a new Java project using Java 8 or above. Create a *pom.xml* file in its root directory and add the following contents:

Passwordless (Recommended)

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>

  <properties>
    <java.version>1.8</java.version>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.30</version>
    </dependency>
    <dependency>
      <groupId>com.azure</groupId>
      <artifactId>azure-identity-extensions</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>
</project>
```

This file is an [Apache Maven](#) file that configures your project to use Java 8 and a recent MySQL driver for Java.

Prepare a configuration file to connect to Azure Database for MySQL

Run the following script in the project root directory to create a `src/main/resources/database.properties` file and add configuration details:

Passwordless (Recommended)

Bash

```
mkdir -p src/main/resources && touch
src/main/resources/database.properties

cat << EOF > src/main/resources/database.properties
```

```
url=jdbc:mysql://${AZ_DATABASE_SERVER_NAME}.mysql.database.azure.com:3306/${AZ_DATABASE_NAME}?
sslMode=REQUIRED&serverTimezone=UTC&defaultAuthenticationPlugin=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin&authenticationPlugins=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin
user=${AZ_MYSQL_AD_NON_ADMIN_USERNAME}@${AZ_DATABASE_SERVER_NAME}
EOF
```

ⓘ Note

If you are using `MysqlConnectionPoolDataSource` class as the datasource in your application, please remove
"`defaultAuthenticationPlugin=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin`" in the url.

Bash

```
mkdir -p src/main/resources && touch
src/main/resources/database.properties

cat << EOF > src/main/resources/database.properties
url=jdbc:mysql://${AZ_DATABASE_SERVER_NAME}.mysql.database.azure.com:3306/${AZ_DATABASE_NAME}?
sslMode=REQUIRED&serverTimezone=UTC&authenticationPlugins=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin
user=${AZ_MYSQL_AD_NON_ADMIN_USERNAME}@${AZ_DATABASE_SERVER_NAME}
EOF
```

ⓘ Note

The configuration property `url` has `?serverTimezone=UTC` appended to tell the JDBC driver to use the UTC date format (or Coordinated Universal Time) when connecting to the database. Otherwise, your Java server would not use the same date format as the database, which would result in an error.

Create an SQL file to generate the database schema

Next, you'll use a `src/main/resources/schema.sql` file to create a database schema. Create that file, then add the following contents:

Bash

```
touch src/main/resources/schema.sql

cat << EOF > src/main/resources/schema.sql
DROP TABLE IF EXISTS todo;
CREATE TABLE todo (id SERIAL PRIMARY KEY, description VARCHAR(255), details
VARCHAR(4096), done BOOLEAN);
EOF
```

Code the application

Connect to the database

Next, add the Java code that will use JDBC to store and retrieve data from your MySQL server.

Create a *src/main/java/DemoApplication.java* file and add the following contents:

Java

```
package com.example.demo;

import com.mysql.cj.jdbc.AbandonedConnectionCleanupThread;

import java.sql.*;
import java.util.*;
import java.util.logging.Logger;

public class DemoApplication {

    private static final Logger log;

    static {
        System.setProperty("java.util.logging.SimpleFormatter.format", "%4$-7s %5$s %n");
        log =Logger.getLogger(DemoApplication.class.getName());
    }

    public static void main(String[] args) throws Exception {
        log.info("Loading application properties");
        Properties properties = new Properties();

        properties.load(DemoApplication.class.getClassLoader().getResourceAsStream("database.properties"));

        log.info("Connecting to the database");
        Connection connection =
DriverManager.getConnection(properties.getProperty("url"), properties);
        log.info("Database connection test: " + connection.getCatalog());
```

```

        log.info("Create database schema");
        Scanner scanner = new
Scanner(DemoApplication.class.getClassLoader().getResourceAsStream("schema.s
ql"));
        Statement statement = connection.createStatement();
        while (scanner.hasNextLine()) {
            statement.execute(scanner.nextLine());
        }

        /* Prepare to store and retrieve data from the MySQL server.
        Todo todo = new Todo(1L, "configuration", "congratulations, you have
set up JDBC correctly!", true);
        insertData(todo, connection);
        todo = readData(connection);
        todo.setDetails("congratulations, you have updated data!");
        updateData(todo, connection);
        deleteData(todo, connection);
        */

        log.info("Closing database connection");
        connection.close();
        AbandonedConnectionCleanupThread.uncheckedShutdown();
    }
}

```

This Java code will use the *database.properties* and the *schema.sql* files that you created earlier. After connecting to the MySQL server, you can create a schema to store your data.

In this file, you can see that we commented methods to insert, read, update and delete data. You'll implement those methods in the rest of this article, and you'll be able to uncomment them one after each other.

Note

The database credentials are stored in the *user* and *password* properties of the *database.properties* file. Those credentials are used when executing

`DriverManager.getConnection(properties.getProperty("url"), properties);`, as the properties file is passed as an argument.

Note

The `AbandonedConnectionCleanupThread.uncheckedShutdown();` line at the end is a MySQL driver command to destroy an internal thread when shutting down the application. You can safely ignore this line.

You can now execute this main class with your favorite tool:

- Using your IDE, you should be able to right-click on the *DemoApplication* class and execute it.
- Using Maven, you can run the application with the following command: `mvn exec:java -Dexec.mainClass="com.example.demo.DemoApplication"`.

The application should connect to the Azure Database for MySQL, create a database schema, and then close the connection. You should see output similar to the following example in the console logs:

```
Output

[INFO    ] Loading application properties
[INFO    ] Connecting to the database
[INFO    ] Database connection test: demo
[INFO    ] Create database schema
[INFO    ] Closing database connection
```

Create a domain class

Create a new `Todo` Java class, next to the `DemoApplication` class, and add the following code:

```
Java

package com.example.demo;

public class Todo {

    private Long id;
    private String description;
    private String details;
    private boolean done;

    public Todo() {
    }

    public Todo(Long id, String description, String details, boolean done) {
        this.id = id;
        this.description = description;
        this.details = details;
        this.done = done;
    }

    public Long getId() {
        return id;
    }
}
```

```

public void setId(Long id) {
    this.id = id;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

public String getDetails() {
    return details;
}

public void setDetails(String details) {
    this.details = details;
}

public boolean isDone() {
    return done;
}

public void setDone(boolean done) {
    this.done = done;
}

@Override
public String toString() {
    return "Todo{" +
        "id=" + id +
        ", description='" + description + '\'' +
        ", details='" + details + '\'' +
        ", done=" + done +
        '}';
}
}

```

This class is a domain model mapped on the `todo` table that you created when executing the `schema.sql` script.

Insert data into Azure Database for MySQL

In the `src/main/java/DemoApplication.java` file, after the `main` method, add the following method to insert data into the database:

Java

```
private static void insertData(Todo todo, Connection connection) throws
SQLException {
    log.info("Insert data");
    PreparedStatement insertStatement = connection
        .prepareStatement("INSERT INTO todo (id, description, details,
done) VALUES (?, ?, ?, ?, ?);");

    insertStatement.setLong(1, todo.getId());
    insertStatement.setString(2, todo.getDescription());
    insertStatement.setString(3, todo.getDetails());
    insertStatement.setBoolean(4, todo.isDone());
    insertStatement.executeUpdate();
}
```

You can now uncomment the two following lines in the `main` method:

Java

```
Todo todo = new Todo(1L, "configuration", "congratulations, you have set up
JDBC correctly!", true);
insertData(todo, connection);
```

Executing the main class should now produce the following output:

Output

```
[INFO] Loading application properties
[INFO] Connecting to the database
[INFO] Database connection test: demo
[INFO] Create database schema
[INFO] Insert data
[INFO] Closing database connection
```

Reading data from Azure Database for MySQL

Next, read the data previously inserted to validate that your code works correctly.

In the `src/main/java/DemoApplication.java` file, after the `insertData` method, add the following method to read data from the database:

Java

```
private static Todo readData(Connection connection) throws SQLException {
    log.info("Read data");
    PreparedStatement readStatement = connection.prepareStatement("SELECT *
FROM todo;");
    ResultSet resultSet = readStatement.executeQuery();
```

```
if (!resultSet.next()) {
    log.info("There is no data in the database!");
    return null;
}
Todo todo = new Todo();
todo.setId(resultSet.getLong("id"));
todo.setDescription(resultSet.getString("description"));
todo.setDetails(resultSet.getString("details"));
todo.setDone(resultSet.getBoolean("done"));
log.info("Data read from the database: " + todo.toString());
return todo;
}
```

You can now uncomment the following line in the `main` method:

Java

```
todo = readData(connection);
```

Executing the main class should now produce the following output:

Output

```
[INFO    ] Loading application properties
[INFO    ] Connecting to the database
[INFO    ] Database connection test: demo
[INFO    ] Create database schema
[INFO    ] Insert data
[INFO    ] Read data
[INFO    ] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have set up JDBC
correctly!', done=true}
[INFO    ] Closing database connection
```

Updating data in Azure Database for MySQL

Next, update the data you previously inserted.

Still in the `src/main/java/DemoApplication.java` file, after the `readData` method, add the following method to update data inside the database:

Java

```
private static void updateData(Todo todo, Connection connection) throws
SQLException {
    log.info("Update data");
    PreparedStatement updateStatement = connection
        .prepareStatement("UPDATE todo SET description = ?, details = ?,"
```

```
done = ? WHERE id = ?;");

updateStatement.setString(1, todo.getDescription());
updateStatement.setString(2, todo.getDetails());
updateStatement.setBoolean(3, todo.isDone());
updateStatement.setLong(4, todo.getId());
updateStatement.executeUpdate();
readData(connection);
}
```

You can now uncomment the two following lines in the `main` method:

Java

```
todo.setDetails("congratulations, you have updated data!");
updateData(todo, connection);
```

Executing the main class should now produce the following output:

Output

```
[INFO    ] Loading application properties
[INFO    ] Connecting to the database
[INFO    ] Database connection test: demo
[INFO    ] Create database schema
[INFO    ] Insert data
[INFO    ] Read data
[INFO    ] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have set up JDBC
correctly!', done=true}
[INFO    ] Update data
[INFO    ] Read data
[INFO    ] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have updated
data!', done=true}
[INFO    ] Closing database connection
```

Deleting data in Azure Database for MySQL

Finally, delete the data you previously inserted.

Still in the `src/main/java/DemoApplication.java` file, after the `updateData` method, add the following method to delete data inside the database:

Java

```
private static void deleteData(Todo todo, Connection connection) throws
SQLException {
```

```
    log.info("Delete data");
    PreparedStatement deleteStatement = connection.prepareStatement("DELETE
FROM todo WHERE id = ?;");
    deleteStatement.setLong(1, todo.getId());
    deleteStatement.executeUpdate();
    readData(connection);
}
```

You can now uncomment the following line in the `main` method:

Java

```
deleteData(todo, connection);
```

Executing the main class should now produce the following output:

Output

```
[INFO    ] Loading application properties
[INFO    ] Connecting to the database
[INFO    ] Database connection test: demo
[INFO    ] Create database schema
[INFO    ] Insert data
[INFO    ] Read data
[INFO    ] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have set up JDBC
correctly!', done=true}
[INFO    ] Update data
[INFO    ] Read data
[INFO    ] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have updated
data!', done=true}
[INFO    ] Delete data
[INFO    ] Read data
[INFO    ] There is no data in the database!
[INFO    ] Closing database connection
```

Clean up resources

Congratulations! You've created a Java application that uses JDBC to store and retrieve data from Azure Database for MySQL.

To clean up all resources used during this quickstart, delete the resource group using the following command:

Azure CLI

```
az group delete \
--name $AZ_RESOURCE_GROUP \
--yes
```

Next steps

Migrate your MySQL database to Azure Database for MySQL using dump and restore

Quickstart: Use Java and JDBC with Azure Database for PostgreSQL

Article • 10/12/2023

APPLIES TO:  Azure Database for PostgreSQL - Single Server

Important

Azure Database for PostgreSQL - Single Server is on the retirement path. We strongly recommend for you to upgrade to Azure Database for PostgreSQL - Flexible Server. For more information about migrating to Azure Database for PostgreSQL - Flexible Server, see [What's happening to Azure Database for PostgreSQL Single Server?](#)

This article demonstrates how to create a sample application that uses Java and [JDBC](#) to store and retrieve information in [Azure Database for PostgreSQL](#).

JDBC is the standard Java API to connect to traditional relational databases.

In this article, we'll include two authentication methods: Microsoft Entra authentication and PostgreSQL authentication. The **Passwordless** tab shows the Microsoft Entra authentication and the **Password** tab shows the PostgreSQL authentication.

Microsoft Entra authentication is a mechanism for connecting to Azure Database for PostgreSQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

PostgreSQL authentication uses accounts stored in PostgreSQL. If you choose to use passwords as credentials for the accounts, these credentials will be stored in the `user` table. Because these passwords are stored in PostgreSQL, you'll need to manage the rotation of the passwords by yourself.

Prerequisites

- An Azure account. If you don't have one, [get a free trial](#).
- [Azure Cloud Shell](#) or [Azure CLI](#) 2.37.0 or above required. We recommend Azure Cloud Shell so you'll be logged in automatically and have access to all the tools you'll need.
- A supported [Java Development Kit](#), version 8 (included in Azure Cloud Shell).

- The [Apache Maven](#) build tool.

Prepare the working environment

First, set up some environment variables. In [Azure Cloud Shell](#), run the following commands:

Passwordless (Recommended)

Bash

```
export AZ_RESOURCE_GROUP=database-workshop
export AZ_DATABASE_SERVER_NAME=<YOUR_DATABASE_SERVER_NAME>
export AZ_DATABASE_NAME=demo
export AZ_LOCATION=<YOUR_AZURE_REGION>
export AZ_POSTGRESQL_AD_NON_ADMIN_USERNAME=
<YOUR_POSTGRESQL_AD_NON_ADMIN_USERNAME>
export AZ_LOCAL_IP_ADDRESS=<YOUR_LOCAL_IP_ADDRESS>
export CURRENT_USERNAME=$(az ad signed-in-user show --query
userPrincipalName -o tsv)
export CURRENT_USER_OBJECTID=$(az ad signed-in-user show --query id -o
tsv)
```

Replace the placeholders with the following values, which are used throughout this article:

- <YOUR_DATABASE_SERVER_NAME>: The name of your PostgreSQL server, which should be unique across Azure.
- <YOUR_AZURE_REGION>: The Azure region you'll use. You can use `eastus` by default, but we recommend that you configure a region closer to where you live. You can see the full list of available regions by entering `az account list-locations`.
- <YOUR_POSTGRESQL_AD_NON_ADMIN_USERNAME>: The username of your PostgreSQL database server. Make sure the username is a valid user in your Microsoft Entra tenant.
- <YOUR_LOCAL_IP_ADDRESS>: The IP address of your local computer, from which you'll run your Spring Boot application. One convenient way to find it is to open [whatismyip.akamai.com](#).

ⓘ Important

When setting <YOUR_POSTGRESQL_AD_NON_ADMIN_USERNAME>, the username must already exist in your Microsoft Entra tenant or you will be

unable to create a Microsoft Entra user in your database.

Next, create a resource group by using the following command:

Azure CLI

```
az group create \
--name $AZ_RESOURCE_GROUP \
--location $AZ_LOCATION \
--output tsv
```

Create an Azure Database for PostgreSQL instance

The following sections describe how to create and configure your database instance.

Create a PostgreSQL server and set up admin user

The first thing you'll create is a managed PostgreSQL server with an admin user.

ⓘ Note

You can read more detailed information about creating PostgreSQL servers in [Create an Azure Database for PostgreSQL server by using the Azure portal](#).

Passwordless (Recommended)

If you're using Azure CLI, run the following command to make sure it has sufficient permission:

Azure CLI

```
az login --scope https://graph.microsoft.com/.default
```

Then run following command to create the server:

Azure CLI

```
az postgres server create \
--resource-group $AZ_RESOURCE_GROUP \
--name $AZ_DATABASE_SERVER_NAME \
```

```
--location $AZ_LOCATION \
--sku-name B_Gen5_1 \
--storage-size 5120 \
--output tsv
```

Now run the following command to set the Microsoft Entra admin user:

Azure CLI

```
az postgres server ad-admin create \
--resource-group $AZ_RESOURCE_GROUP \
--server-name $AZ_DATABASE_SERVER_NAME \
--display-name $CURRENT_USERNAME \
--object-id $CURRENT_USER_OBJECTID
```

ⓘ Important

When setting the administrator, a new user is added to the Azure Database for PostgreSQL server with full administrator permissions. Only one Microsoft Entra admin can be created per PostgreSQL server and selection of another one will overwrite the existing Microsoft Entra admin configured for the server.

This command creates a small PostgreSQL server and sets the Active Directory admin to the signed-in user.

Configure a firewall rule for your PostgreSQL server

Azure Database for PostgreSQL instances are secured by default. They have a firewall that doesn't allow any incoming connection. To be able to use your database, you need to add a firewall rule that will allow the local IP address to access the database server.

Because you configured your local IP address at the beginning of this article, you can open the server's firewall by running the following command:

Azure CLI

```
az postgres server firewall-rule create \
--resource-group $AZ_RESOURCE_GROUP \
--name $AZ_DATABASE_SERVER_NAME-database-allow-local-ip \
--server $AZ_DATABASE_SERVER_NAME \
--start-ip-address $AZ_LOCAL_IP_ADDRESS \
--end-ip-address $AZ_LOCAL_IP_ADDRESS \
--output tsv
```

If you're connecting to your PostgreSQL server from Windows Subsystem for Linux (WSL) on a Windows computer, you'll need to add the WSL host ID to your firewall.

Obtain the IP address of your host machine by running the following command in WSL:

```
Bash
```

```
cat /etc/resolv.conf
```

Copy the IP address following the term `nameserver`, then use the following command to set an environment variable for the WSL IP Address:

```
Bash
```

```
AZ_WSL_IP_ADDRESS=<the-copied-IP-address>
```

Then, use the following command to open the server's firewall to your WSL-based app:

```
Azure CLI
```

```
az postgres server firewall-rule create \
    --resource-group $AZ_RESOURCE_GROUP \
    --name $AZ_DATABASE_SERVER_NAME-database-allow-local-ip \
    --server $AZ_DATABASE_SERVER_NAME \
    --start-ip-address $AZ_WSL_IP_ADDRESS \
    --end-ip-address $AZ_WSL_IP_ADDRESS \
    --output tsv
```

Configure a PostgreSQL database

The PostgreSQL server that you created earlier is empty. Use the following command to create a new database.

```
Azure CLI
```

```
az postgres db create \
    --resource-group $AZ_RESOURCE_GROUP \
    --name $AZ_DATABASE_NAME \
    --server-name $AZ_DATABASE_SERVER_NAME \
    --output tsv
```

Create a PostgreSQL non-admin user and grant permission

Next, create a non-admin user and grant all permissions to the database.

ⓘ Note

You can read more detailed information about creating PostgreSQL users in [Create users in Azure Database for PostgreSQL](#).

Passwordless (Recommended)

Create a SQL script called *create_ad_user.sql* for creating a non-admin user. Add the following contents and save it locally:

Bash

```
cat << EOF > create_ad_user.sql
SET aad_validate_oids_in_tenant = off;
CREATE ROLE "$AZ_POSTGRESQL_AD_NON_ADMIN_USERNAME" WITH LOGIN IN ROLE
azure_ad_user;
GRANT ALL PRIVILEGES ON DATABASE $AZ_DATABASE_NAME TO
"$AZ_POSTGRESQL_AD_NON_ADMIN_USERNAME";
EOF
```

Then, use the following command to run the SQL script to create the Microsoft Entra non-admin user:

Bash

```
psql "host=$AZ_DATABASE_SERVER_NAME.postgres.database.azure.com
user=$CURRENT_USERNAME@$AZ_DATABASE_SERVER_NAME dbname=$AZ_DATABASE_NAME
port=5432 password=$(az account get-access-token --resource-type oss-
rdbms --output tsv --query accessToken) sslmode=require" <
create_ad_user.sql
```

Now use the following command to remove the temporary SQL script file:

Bash

```
rm create_ad_user.sql
```

Create a new Java project

Using your favorite IDE, create a new Java project using Java 8 or above, and add a *pom.xml* file in its root directory with the following contents:

Passwordless (Recommended)

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>

  <properties>
    <java.version>1.8</java.version>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.3.6</version>
    </dependency>
    <dependency>
      <groupId>com.azure</groupId>
      <artifactId>azure-identity-extensions</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>
</project>
```

This file is an [Apache Maven](#) file that configures your project to use Java 8 and a recent PostgreSQL driver for Java.

Prepare a configuration file to connect to Azure Database for PostgreSQL

Create a `src/main/resources/application.properties` file, then add the following contents:

Passwordless (Recommended)

Bash

```
cat << EOF > src/main/resources/application.properties
url=jdbc:postgresql://${AZ_DATABASE_SERVER_NAME}.postgres.database.azure
.com:5432/${AZ_DATABASE_NAME}?
sslmode=require&authenticationPluginClassName=com.azure.identity.extensions.jdbc.postgresql.AzurePostgresqlAuthenticationPlugin
user=${AZ_POSTGRESQL_AD_NON_ADMIN_USERNAME}@${AZ_DATABASE_SERVER_NAME}
EOF
```

ⓘ Note

The configuration property `url` has `?sslmode=require` appended to tell the JDBC driver to use TLS ([Transport Layer Security](#)) when connecting to the database. Using TLS is mandatory with Azure Database for PostgreSQL, and it's a good security practice.

Create an SQL file to generate the database schema

You'll use a `src/main/resources/schema.sql` file to create a database schema. Create that file, then add the following contents:

Bash

```
cat << EOF > src/main/resources/schema.sql
DROP TABLE IF EXISTS todo;
CREATE TABLE todo (id SERIAL PRIMARY KEY, description VARCHAR(255), details
VARCHAR(4096), done BOOLEAN);
EOF
```

Code the application

Connect to the database

Next, add the Java code that will use JDBC to store and retrieve data from your PostgreSQL server.

Create a `src/main/java/DemoApplication.java` file, then add the following contents:

Java

```
package com.example.demo;
```

```

import java.sql.*;
import java.util.*;
import java.util.logging.Logger;

public class DemoApplication {

    private static final Logger log;

    static {
        System.setProperty("java.util.logging.SimpleFormatter.format", "[%4$-7s] %5$s %n");
        log =Logger.getLogger(DemoApplication.class.getName());
    }

    public static void main(String[] args) throws Exception {
        log.info("Loading application properties");
        Properties properties = new Properties();

        properties.load(DemoApplication.class.getClassLoader().getResourceAsStream("application.properties"));

        log.info("Connecting to the database");
        Connection connection =
DriverManager.getConnection(properties.getProperty("url"), properties);
        log.info("Database connection test: " + connection.getCatalog());

        log.info("Create database schema");
        Scanner scanner = new
Scanner(DemoApplication.class.getClassLoader().getResourceAsStream("schema.sql"));
        Statement statement = connection.createStatement();
        while (scanner.hasNextLine()) {
            statement.execute(scanner.nextLine());
        }

        /* Prepare for data processing in the PostgreSQL server.
        Todo todo = new Todo(1L, "configuration", "congratulations, you have
set up JDBC correctly!", true);
        insertData(todo, connection);
        todo = readData(connection);
        todo.setDetails("congratulations, you have updated data!");
        updateData(todo, connection);
        deleteData(todo, connection);
        */

        log.info("Closing database connection");
        connection.close();
    }
}

```

This Java code will use the *application.properties* and the *schema.sql* files that you created earlier in order to connect to the PostgreSQL server and create a schema that will store your data.

In this file, you can see that we commented methods to insert, read, update and delete data. You'll code those methods in the rest of this article, and you'll be able to uncomment them one after another.

ⓘ Note

The database credentials are stored in the `user` and `password` properties of the `application.properties` file. Those credentials are used when executing `DriverManager.getConnection(properties.getProperty("url"), properties);`, as the properties file is passed as an argument.

You can now execute this main class with your favorite tool:

- Using your IDE, you should be able to right-click on the `DemoApplication` class and execute it.
- Using Maven, you can run the application by using the following command: `mvn exec:java -Dexec.mainClass="com.example.demo.DemoApplication"`.

The application should connect to the Azure Database for PostgreSQL, create a database schema, and then close the connection, as you should see in the console logs:

Output

```
[INFO    ] Loading application properties
[INFO    ] Connecting to the database
[INFO    ] Database connection test: demo
[INFO    ] Create database schema
[INFO    ] Closing database connection
```

Create a domain class

Create a new `Todo` Java class, next to the `DemoApplication` class, and add the following code:

Java

```
package com.example.demo;

public class Todo {

    private Long id;
    private String description;
    private String details;
    private boolean done;
```

```
public Todo() {
}

public Todo(Long id, String description, String details, boolean done) {
    this.id = id;
    this.description = description;
    this.details = details;
    this.done = done;
}

public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

public String getDetails() {
    return details;
}

public void setDetails(String details) {
    this.details = details;
}

public boolean isDone() {
    return done;
}

public void setDone(boolean done) {
    this.done = done;
}

@Override
public String toString() {
    return "Todo{" +
        "id=" + id +
        ", description='" + description + '\'' +
        ", details='" + details + '\'' +
        ", done=" + done +
        '}';
}
}
```

This class is a domain model mapped on the `todo` table that you created when executing the `schema.sql` script.

Insert data into Azure Database for PostgreSQL

In the `src/main/java/DemoApplication.java` file, after the main method, add the following method to insert data into the database:

Java

```
private static void insertData(Todo todo, Connection connection) throws
SQLException {
    log.info("Insert data");
    PreparedStatement insertStatement = connection
        .prepareStatement("INSERT INTO todo (id, description, details,
done) VALUES (?, ?, ?, ?);");

    insertStatement.setLong(1, todo.getId());
    insertStatement.setString(2, todo.getDescription());
    insertStatement.setString(3, todo.getDetails());
    insertStatement.setBoolean(4, todo.isDone());
    insertStatement.executeUpdate();
}
```

You can now uncomment the two following lines in the `main` method:

Java

```
Todo todo = new Todo(1L, "configuration", "congratulations, you have set up
JDBC correctly!", true);
insertData(todo, connection);
```

Executing the main class should now produce the following output:

Output

```
[INFO    ] Loading application properties
[INFO    ] Connecting to the database
[INFO    ] Database connection test: demo
[INFO    ] Create database schema
[INFO    ] Insert data
[INFO    ] Closing database connection
```

Reading data from Azure Database for PostgreSQL

To validate that your code works correctly, read the data that you previously inserted.

In the `src/main/java/DemoApplication.java` file, after the `insertData` method, add the following method to read data from the database:

```
Java

private static Todo readData(Connection connection) throws SQLException {
    log.info("Read data");
    PreparedStatement readStatement = connection.prepareStatement("SELECT * FROM todo;");
    ResultSet resultSet = readStatement.executeQuery();
    if (!resultSet.next()) {
        log.info("There is no data in the database!");
        return null;
    }
    Todo todo = new Todo();
    todo.setId(resultSet.getLong("id"));
    todo.setDescription(resultSet.getString("description"));
    todo.setDetails(resultSet.getString("details"));
    todo.setDone(resultSet.getBoolean("done"));
    log.info("Data read from the database: " + todo.toString());
    return todo;
}
```

You can now uncomment the following line in the `main` method:

```
Java

todo = readData(connection);
```

Executing the main class should now produce the following output:

```
Output

[INFO    ] Loading application properties
[INFO    ] Connecting to the database
[INFO    ] Database connection test: demo
[INFO    ] Create database schema
[INFO    ] Insert data
[INFO    ] Read data
[INFO    ] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have set up JDBC correctly!', done=true}
[INFO    ] Closing database connection
```

Updating data in Azure Database for PostgreSQL

Next, update the data you previously inserted.

Still in the `src/main/java/DemoApplication.java` file, after the `readData` method, add the following method to update data inside the database:

Java

```
private static void updateData(Todo todo, Connection connection) throws
SQLException {
    log.info("Update data");
    PreparedStatement updateStatement = connection
        .prepareStatement("UPDATE todo SET description = ?, details = ?,"
        "done = ? WHERE id = ?;");
    updateStatement.setString(1, todo.getDescription());
    updateStatement.setString(2, todo.getDetails());
    updateStatement.setBoolean(3, todo.isDone());
    updateStatement.setLong(4, todo.getId());
    updateStatement.executeUpdate();
    readData(connection);
}
```

You can now uncomment the two following lines in the `main` method:

Java

```
todo.setDetails("congratulations, you have updated data!");
updateData(todo, connection);
```

Executing the main class should now produce the following output:

Output

```
[INFO] Loading application properties
[INFO] Connecting to the database
[INFO] Database connection test: demo
[INFO] Create database schema
[INFO] Insert data
[INFO] Read data
[INFO] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have set up JDBC
correctly!', done=true}
[INFO] Update data
[INFO] Read data
[INFO] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have updated
data!', done=true}
[INFO] Closing database connection
```

Deleting data in Azure Database for PostgreSQL

Finally, delete the data you previously inserted.

Still in the `src/main/java/DemoApplication.java` file, after the `updateData` method, add the following method to delete data inside the database:

Java

```
private static void deleteData(Todo todo, Connection connection) throws
SQLException {
    log.info("Delete data");
    PreparedStatement deleteStatement = connection.prepareStatement("DELETE
FROM todo WHERE id = ?;");
    deleteStatement.setLong(1, todo.getId());
    deleteStatement.executeUpdate();
    readData(connection);
}
```

You can now uncomment the following line in the `main` method:

Java

```
deleteData(todo, connection);
```

Executing the main class should now produce the following output:

Output

```
[INFO] Loading application properties
[INFO] Connecting to the database
[INFO] Database connection test: demo
[INFO] Create database schema
[INFO] Insert data
[INFO] Read data
[INFO] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have set up JDBC
correctly!', done=true}
[INFO] Update data
[INFO] Read data
[INFO] Data read from the database: Todo{id=1,
description='configuration', details='congratulations, you have updated
data!', done=true}
[INFO] Delete data
[INFO] Read data
[INFO] There is no data in the database!
[INFO] Closing database connection
```

Clean up resources

Congratulations! You've created a Java application that uses JDBC to store and retrieve data from Azure Database for PostgreSQL.

To clean up all resources used during this quickstart, delete the resource group using the following command:

Azure CLI

```
az group delete \
--name $AZ_RESOURCE_GROUP \
--yes
```

Next steps

[Migrate your database using Export and Import](#)

Send messages to and receive messages from Azure Service Bus queues (Java)

Article • 02/28/2024

In this quickstart, you create a Java app to send messages to and receive messages from an Azure Service Bus queue.

ⓘ Note

This quick start provides step-by-step instructions for a simple scenario of sending messages to a Service Bus queue and receiving them. You can find pre-built Java samples for Azure Service Bus in the [Azure SDK for Java repository on GitHub](#).

💡 Tip

If you're working with Azure Service Bus resources in a Spring application, we recommend that you consider [Spring Cloud Azure](#) as an alternative. Spring Cloud Azure is an open-source project that provides seamless Spring integration with Azure services. To learn more about Spring Cloud Azure, and to see an example using Service Bus, see [Spring Cloud Stream with Azure Service Bus](#).

Prerequisites

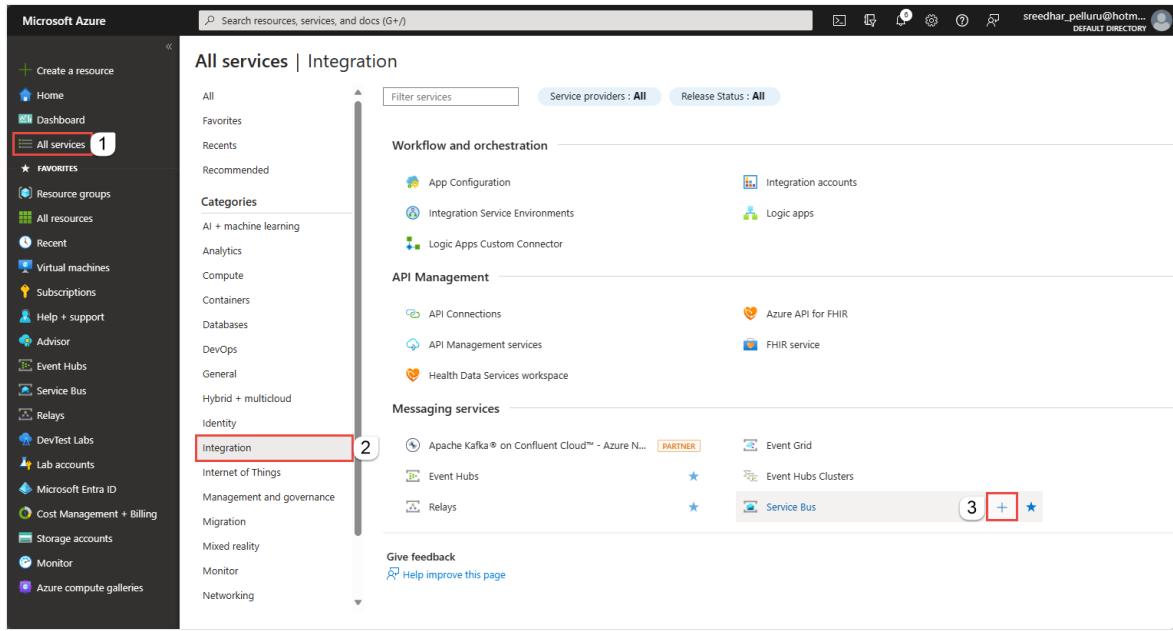
- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
- Install [Azure SDK for Java](#). If you're using Eclipse, you can install the [Azure Toolkit for Eclipse](#) that includes the Azure SDK for Java. You can then add the [Microsoft Azure Libraries for Java](#) to your project. If you're using IntelliJ, see [Install the Azure Toolkit for IntelliJ](#).

Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.

To create a namespace:

1. Sign in to the [Azure portal](#).
2. In the left navigation pane of the portal, select **All services**, select **Integration** from the list of categories, hover the mouse over **Service Bus**, and then select + button on the Service Bus tile.



3. In the **Basics** tag of the [Create namespace](#) page, follow these steps:
 - a. For **Subscription**, choose an Azure subscription in which to create the namespace.
 - b. For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
 - c. Enter a **name for the namespace**. The namespace name should adhere to the following naming conventions:
 - The name must be unique across Azure. The system immediately checks to see if the name is available.
 - The name length is at least 6 and at most 50 characters.
 - The name can contain only letters, numbers, hyphens “-”.
 - The name must start with a letter and end with a letter or number.
 - The name doesn't end with “-sb” or “-mgmt”.
 - d. For **Location**, choose the region in which your namespace should be hosted.
 - e. For **Pricing tier**, select the pricing tier (Basic, Standard, or Premium) for the namespace. For this quickstart, select **Standard**.

Important

If you want to use [topics and subscriptions](#), choose either Standard or Premium. Topics/subscriptions aren't supported in the Basic pricing tier.

If you selected the **Premium** pricing tier, specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, 4, 8 or 16 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

f. Select **Review + create** at the bottom of the page.

The screenshot shows the 'Create namespace' wizard in the Azure portal. The title bar says 'Create namespace' with a 'Service Bus' icon. Below it, there are tabs: 'Basics' (which is selected), 'Advanced', 'Networking', 'Tags', and 'Review + create'. The main area is titled 'Project Details' with a note about managing resources via a subscription and resource groups. It shows a 'Subscription' dropdown set to 'Visual Studio Enterprise Subscription' and a 'Resource group' dropdown showing '(New) spsbusrg' with a 'Create new' link. The 'Instance Details' section contains fields for 'Namespace name' (set to 'contosoordersns'), 'Location' (set to 'East US'), and 'Pricing tier' (set to 'Standard'). Below these fields is a link to 'Browse the available plans and their features'. At the bottom of the screen are three buttons: 'Review + create' (highlighted in blue), '< Previous', and 'Next: Advanced >'.

g. On the **Review + create** page, review settings, and select **Create**.

4. Once the deployment of the resource is successful, select **Go to resource** on the deployment page.

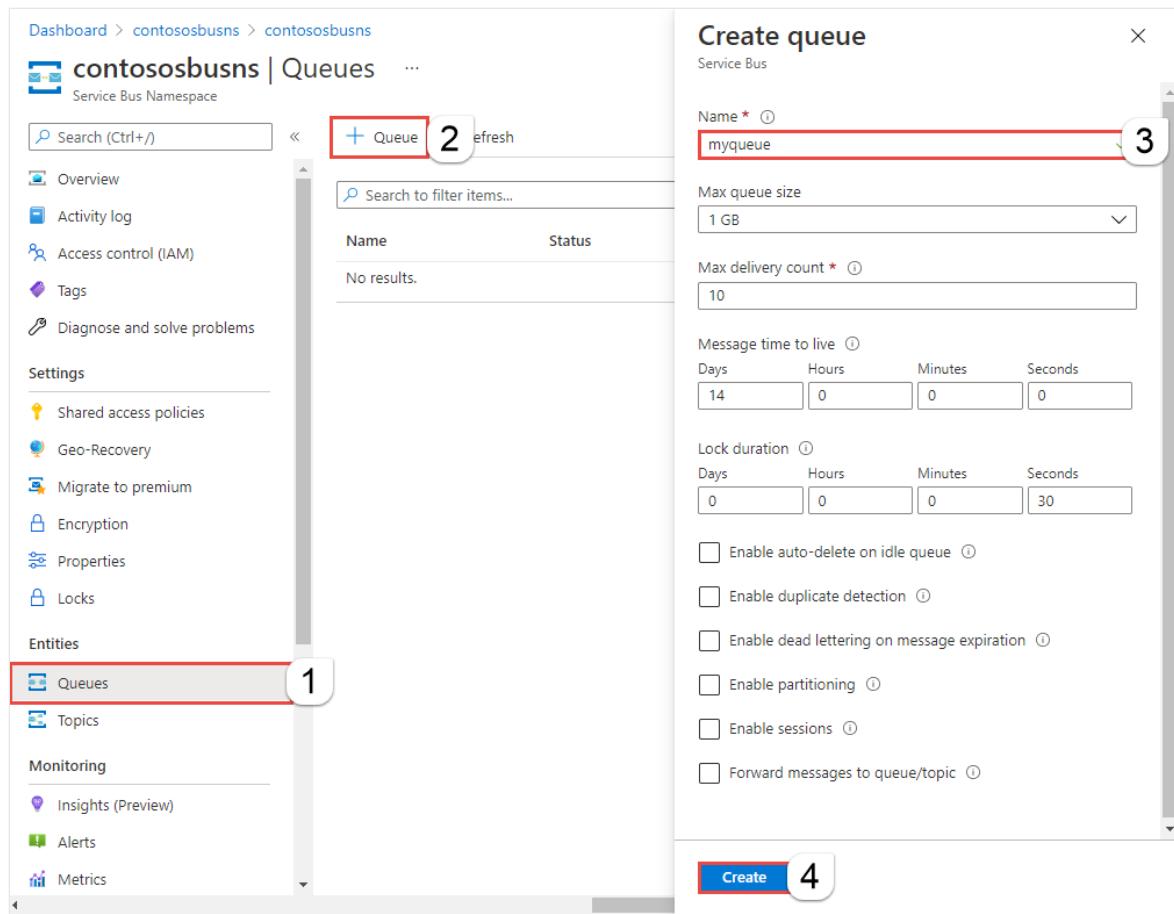
The screenshot shows the Azure portal's deployment overview for the service bus namespace 'contosoordersns'. The main message is 'Your deployment is complete'. Deployment details include the name 'contosoordersns', subscription 'Visual Studio Enterprise Subscription', and resource group 'spsbusrg'. The start time was 10/20/2022, 4:45:03 PM, and the correlation ID is a453ace1-bab9-4c4a-81ad-a1c5366460ea. A 'Go to resource' button is highlighted in red.

5. You see the home page for your service bus namespace.

The screenshot shows the Azure portal's Service Bus Namespace overview for 'spsbusns1128'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Geo-Recovery, Migrate to premium, Encryption, Configuration, Properties, Locks), Entities (Queues, Topics), Monitoring (Insights (Preview), Alerts), and Queues (0) / Topics (0). The main area displays 'Essentials' information like Resource group (spsbusrg), Status (Active), Location (East US), and Pricing tier (Standard). It also shows 'Requests' and 'Messages' over time, with a chart showing 0 incoming and outgoing messages at 5:15 PM UTC-05:00.

Create a queue in the Azure portal

1. On the Service Bus Namespace page, select **Queues** in the left navigational menu.
2. On the **Queues** page, select **+ Queue** on the toolbar.
3. Enter a name for the queue, and leave the other values with their defaults.
4. Now, select **Create**.



Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Service Bus: **passwordless** and **connection string**.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace. You don't need to worry about having hard-coded connection string in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a connection string to connect to a Service Bus namespace. If you are new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Service Bus has the correct permissions. You'll need the [Azure Service Bus Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Service Bus Data Owner` role to your user account, which provides full access to Azure Service Bus resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

Azure built-in roles for Azure Service Bus

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters). A member of this role can send and receive messages from queues or topics/subscriptions.
- [Azure Service Bus Data Sender](#): Use this role to give the send access to Service Bus namespace and its entities.
- [Azure Service Bus Data Receiver](#): Use this role to give the receive access to Service Bus namespace and its entities.

If you want to create a custom role, see [Rights required for Service Bus operations](#).

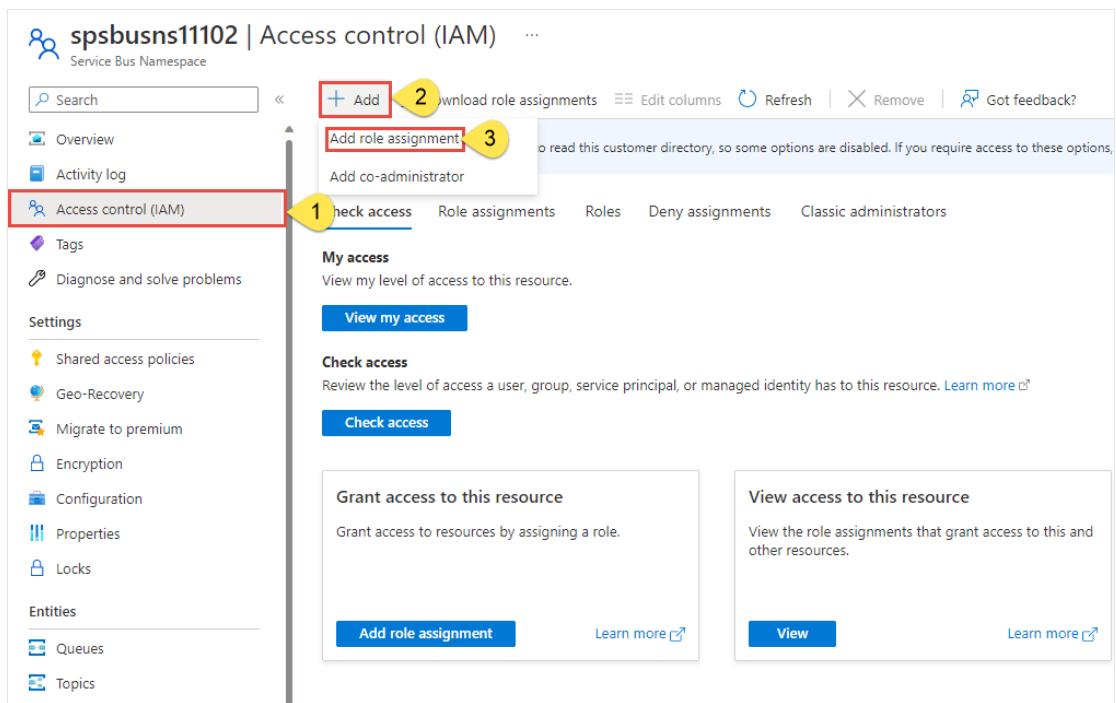
Add Microsoft Entra user to Azure Service Bus Owner role

Add your Microsoft Entra user name to the [Azure Service Bus Data Owner](#) role at the Service Bus namespace level. It will allow an app running in the context of your user account to send messages to a queue or a topic, and receive messages from a queue or a topic's subscription.

 **Important**

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. If you don't have the Service Bus Namespace page open in the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



5. Use the search box to filter the results to the desired role. For this example, search for **Azure Service Bus Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.

8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Send messages to a queue

In this section, you'll create a Java console project, and add code to send messages to the queue that you created earlier.

Create a Java console project

Create a Java project using Eclipse or a tool of your choice.

Configure your application to use Service Bus

Add references to Azure Core and Azure Service Bus libraries.

If you're using Eclipse and created a Java console application, convert your Java project to a Maven: right-click the project in the **Package Explorer** window, select **Configure -> Convert to Maven project**. Then, add dependencies to these two libraries as shown in the following example.

Passwordless (Recommended)

Update the `pom.xml` file to add dependencies to Azure Service Bus and Azure Identity packages.

XML

```
<dependencies>
    <dependency>
        <groupId>com.azure</groupId>
        <artifactId>azure-messaging-servicebus</artifactId>
        <version>7.13.3</version>
    </dependency>
    <dependency>
        <groupId>com.azure</groupId>
        <artifactId>azure-identity</artifactId>
        <version>1.8.0</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Add code to send messages to the queue

1. Add the following `import` statements at the topic of the Java file.

Passwordless (Recommended)

Java

```
import com.azure.messaging.servicebus.*;
import com.azure.identity.*;

import java.util.concurrent.TimeUnit;
import java.util.Arrays;
import java.util.List;
```

2. In the class, define variables to hold connection string and queue name.

Passwordless (Recommended)

Java

```
static String queueName = "<QUEUE NAME>";
```

ⓘ Important

Replace `<QUEUE NAME>` with the name of the queue.

3. Add a method named `sendMessage` in the class to send one message to the queue.

Passwordless (Recommended)

ⓘ Important

Replace `NAMESPACENAME` with the name of your Service Bus namespace.

Java

```
static void sendMessage()
{
    // create a token using the default Azure credential
    DefaultAzureCredential credential = new
```

```
DefaultAzureCredentialBuilder()
    .build();

    ServiceBusSenderClient senderClient = new
ServiceBusClientBuilder()

    .fullyQualifiedNamespace("NAMESPACENAME.servicebus.windows.net")
        .credential(credential)
        .sender()
        .queueName(queueName)
        .buildClient();

    // send one message to the queue
    senderClient.sendMessage(new ServiceBusMessage("Hello,
World!"));
    System.out.println("Sent a single message to the queue: " +
queueName);
}
```

4. Add a method named `createMessages` in the class to create a list of messages.

Typically, you get these messages from different parts of your application. Here, we create a list of sample messages.

Java

```
static List<ServiceBusMessage> createMessages()
{
    // create a list of messages and return it to the caller
    ServiceBusMessage[] messages = {
        new ServiceBusMessage("First message"),
        new ServiceBusMessage("Second message"),
        new ServiceBusMessage("Third message")
    };
    return Arrays.asList(messages);
}
```

5. Add a method named `sendMessageBatch` method to send messages to the queue you created. This method creates a `ServiceBusSenderClient` for the queue, invokes the `createMessages` method to get the list of messages, prepares one or more batches, and sends the batches to the queue.

Passwordless (Recommended)

 **Important**

Replace `NAMESPACENAME` with the name of your Service Bus namespace.

Java

```
static void sendMessageBatch()
{
    // create a token using the default Azure credential
    DefaultAzureCredential credential = new
DefaultAzureCredentialBuilder()
    .build();

    ServiceBusSenderClient senderClient = new
ServiceBusClientBuilder()

    .fullyQualifiedNamespace("NAMESPACENAME.servicebus.windows.net")
        .credential(credential)
        .sender()
        .queueName(queueName)
        .buildClient();

    // Creates an ServiceBusMessageBatch where the ServiceBus.
    ServiceBusMessageBatch messageBatch =
senderClient.createMessageBatch();

    // create a list of messages
    List<ServiceBusMessage> listOfMessages = createMessages();

    // We try to add as many messages as a batch can fit based on
    the maximum size and send to Service Bus when
    // the batch can hold no more messages. Create a new batch for
    next set of messages and repeat until all
    // messages are sent.
    for (ServiceBusMessage message : listOfMessages) {
        if (messageBatch.tryAddMessage(message)) {
            continue;
        }

        // The batch is full, so we create a new batch and send the
        batch.
        senderClient.sendMessages(messageBatch);
        System.out.println("Sent a batch of messages to the queue:
" + queueName);

        // create a new batch
        messageBatch = senderClient.createMessageBatch();

        // Add that message that we couldn't before.
        if (!messageBatch.tryAddMessage(message)) {
            System.err.printf("Message is too large for an empty
batch. Skipping. Max size: %s.", messageBatch.getMaxSizeInBytes());
        }
    }
}
```

```
        if (messageBatch.getCount() > 0) {
            senderClient.sendMessages(messageBatch);
            System.out.println("Sent a batch of messages to the queue:
" + queueName);
        }

        //close the client
        senderClient.close();
    }
```

Receive messages from a queue

In this section, you add code to retrieve messages from the queue.

1. Add a method named `receiveMessages` to receive messages from the queue. This method creates a `ServiceBusProcessorClient` for the queue by specifying a handler for processing messages and another one for handling errors. Then, it starts the processor, waits for few seconds, prints the messages that are received, and then stops and closes the processor.

Passwordless (Recommended)

ⓘ Important

- Replace `NAMESPACENAME` with the name of your Service Bus namespace.
- Replace `QueueTest` in `QueueTest::processMessage` in the code with the name of your class.

Java

```
// handles received messages
static void receiveMessages() throws InterruptedException
{
    DefaultAzureCredential credential = new
DefaultAzureCredentialBuilder()
    .build();

    ServiceBusProcessorClient processorClient = new
ServiceBusClientBuilder()

    .fullyQualifiedNamespace("NAMESPACENAME.servicebus.windows.net")
        .credential(credential)
        .processor()
```

```

        .queueName(queueName)
        .processMessage(context -> processMessage(context))
        .processError(context -> processError(context))
        .buildProcessorClient();

    System.out.println("Starting the processor");
    processorClient.start();

    TimeUnit.SECONDS.sleep(10);
    System.out.println("Stopping and closing the processor");
    processorClient.close();
}

```

2. Add the `processMessage` method to process a message received from the Service Bus subscription.

Java

```

private static void processMessage(ServiceBusReceivedMessageContext
context) {
    ServiceBusReceivedMessage message = context.getMessage();
    System.out.printf("Processing message. Session: %s, Sequence #: %s.
Contents: %s%n", message.getMessageId(),
    message.getSequenceNumber(), message.getBody());
}

```

3. Add the `processError` method to handle error messages.

Java

```

private static void processError(ServiceBusErrorHandlerContext context) {
    System.out.printf("Error when receiving messages from namespace:
'%s'. Entity: '%s'%n",
    context.getFullyQualifiedNamespace(), context.getEntityPath());

    if (!(context.getException() instanceof ServiceBusException)) {
        System.out.printf("Non-ServiceBusException occurred: %s%n",
        context.getException());
        return;
    }

    ServiceBusException exception = (ServiceBusException)
    context.getException();
    ServiceBusFailureReason reason = exception.getReason();

    if (reason == ServiceBusFailureReason.MESSAGING_ENTITY_DISABLED
    || reason == ServiceBusFailureReason.MESSAGING_ENTITY_NOT_FOUND
    || reason == ServiceBusFailureReason.UNAUTHORIZED) {
        System.out.printf("An unrecoverable error occurred. Stopping
processing with reason %s: %s%n",

```

```
        reason, exception.getMessage());
    } else if (reason == ServiceBusFailureReason.MESSAGE_LOCK_LOST) {
        System.out.printf("Message lock lost for message: %s%n",
context.getException());
    } else if (reason == ServiceBusFailureReason.SERVICE_BUSY) {
        try {
            // Choosing an arbitrary amount of time to wait until
            // trying again.
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            System.err.println("Unable to sleep for period of time");
        }
    } else {
        System.out.printf("Error source %s, reason %s, message: %s%n",
context.getErrorSource(),
reason, context.getException());
    }
}
```

4. Update the `main` method to invoke `sendMessage`, `sendMessageBatch`, and `receiveMessages` methods and to throw `InterruptedException`.

Java

```
public static void main(String[] args) throws InterruptedException {
    sendMessage();
    sendMessageBatch();
    receiveMessages();
}
```

Run the app

Passwordless (Recommended)

1. If you're using Eclipse, right-click the project, select **Export**, expand **Java**, select **Runnable JAR file**, and follow the steps to create a runnable JAR file.
2. If you're signed into the machine using a user account that's different from the user account added to the **Azure Service Bus Data Owner** role, follow these steps. Otherwise, skip this step and move on to run the Jar file in the next step.
 - a. [Install Azure CLI](#) on your machine.
 - b. Run the following CLI command to sign in to Azure. Use the same user account that you added to the **Azure Service Bus Data Owner** role.

Azure CLI

```
az login
```

3. Run the Jar file using the following command.

Java

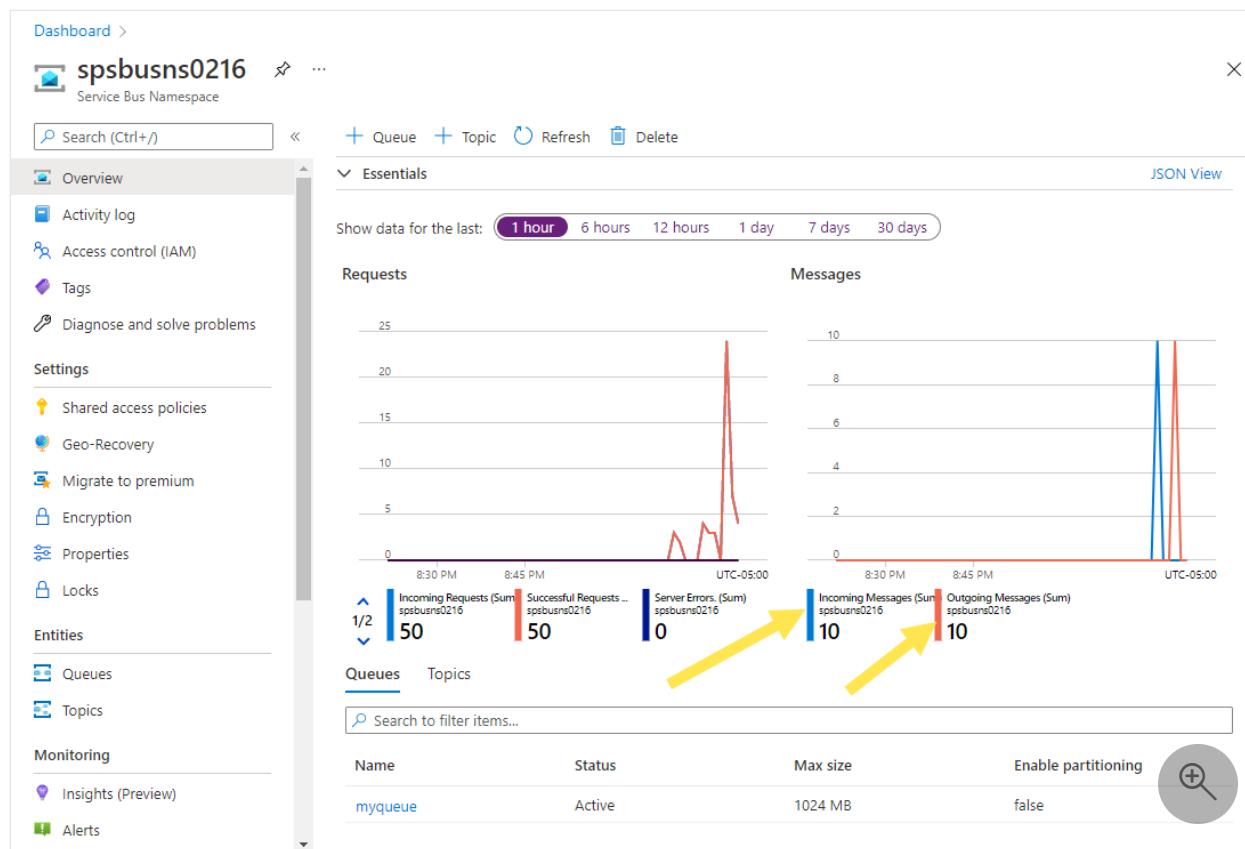
```
java -jar <JAR FILE NAME>
```

4. You see the following output in the console window.

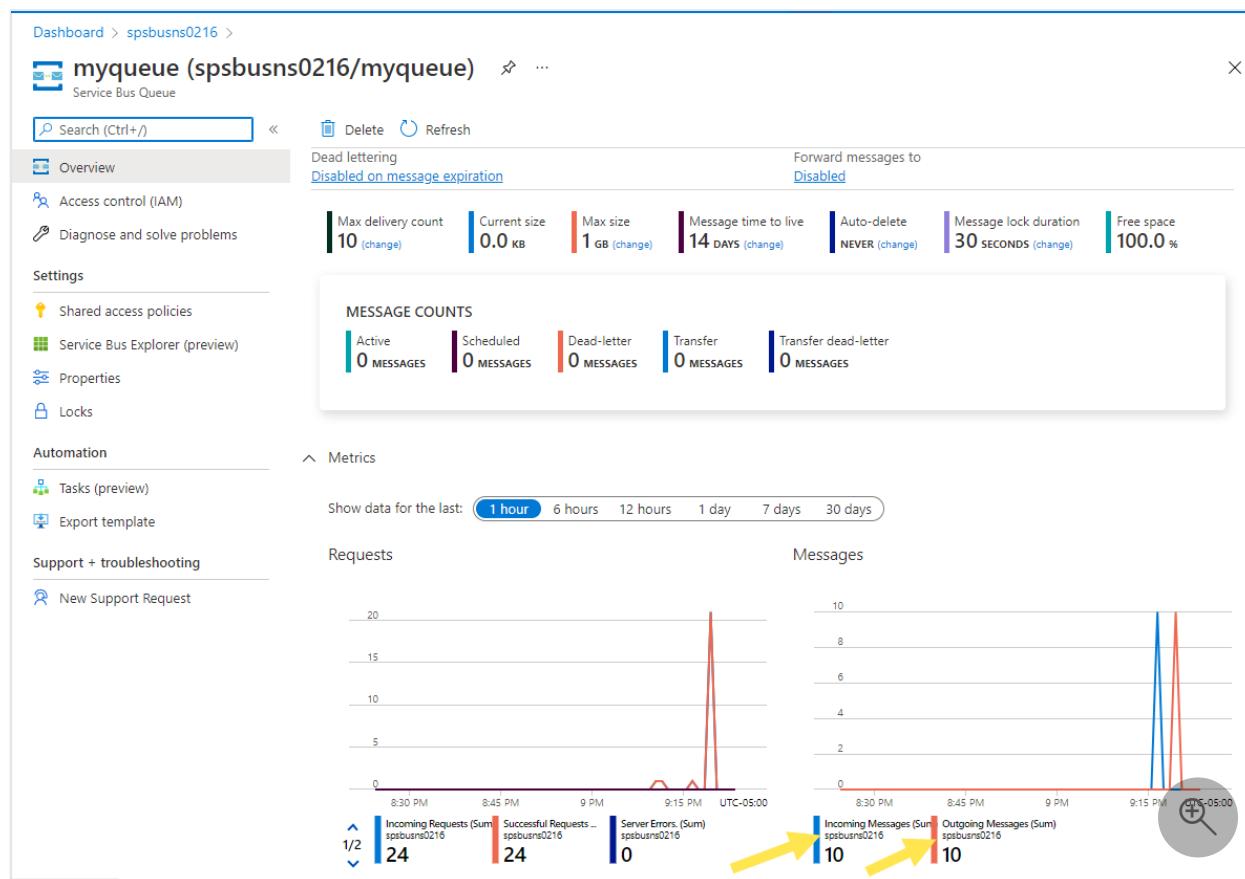
Console

```
Sent a single message to the queue: myqueue
Sent a batch of messages to the queue: myqueue
Starting the processor
Processing message. Session: 88d961dd801f449e9c3e0f8a5393a527,
Sequence #: 1. Contents: Hello, World!
Processing message. Session: e90c8d9039ce403bbe1d0ec7038033a0,
Sequence #: 2. Contents: First message
Processing message. Session: 311a216a560c47d184f9831984e6ac1d,
Sequence #: 3. Contents: Second message
Processing message. Session: f9a871be07414baf9505f2c3d466c4ab,
Sequence #: 4. Contents: Third message
Stopping and closing the processor
```

On the **Overview** page for the Service Bus namespace in the Azure portal, you can see **incoming** and **outgoing** message count. Wait for a minute or so and then refresh the page to see the latest values.



Select the queue on this **Overview** page to navigate to the **Service Bus Queue** page. You see the **incoming** and **outgoing** message count on this page too. You also see other information such as the **current size** of the queue, **maximum size**, **active message count**, and so on.



Next Steps

See the following documentation and samples:

- [Azure Service Bus client library for Java - Readme ↗](#)
- [Samples on GitHub](#)
- [Java API reference ↗](#)

Send messages to an Azure Service Bus topic and receive messages from subscriptions to the topic (Java)

Article • 02/28/2024

In this quickstart, you write Java code using the `azure-messaging-servicebus` package to send messages to an Azure Service Bus topic and then receive messages from subscriptions to that topic.

ⓘ Note

This quick start provides step-by-step instructions for a simple scenario of sending a batch of messages to a Service Bus topic and receiving those messages from a subscription of the topic. You can find pre-built Java samples for Azure Service Bus in the [Azure SDK for Java repository on GitHub](#).

💡 Tip

If you're working with Azure Service Bus resources in a Spring application, we recommend that you consider [Spring Cloud Azure](#) as an alternative. Spring Cloud Azure is an open-source project that provides seamless Spring integration with Azure services. To learn more about Spring Cloud Azure, and to see an example using Service Bus, see [Spring Cloud Stream with Azure Service Bus](#).

Prerequisites

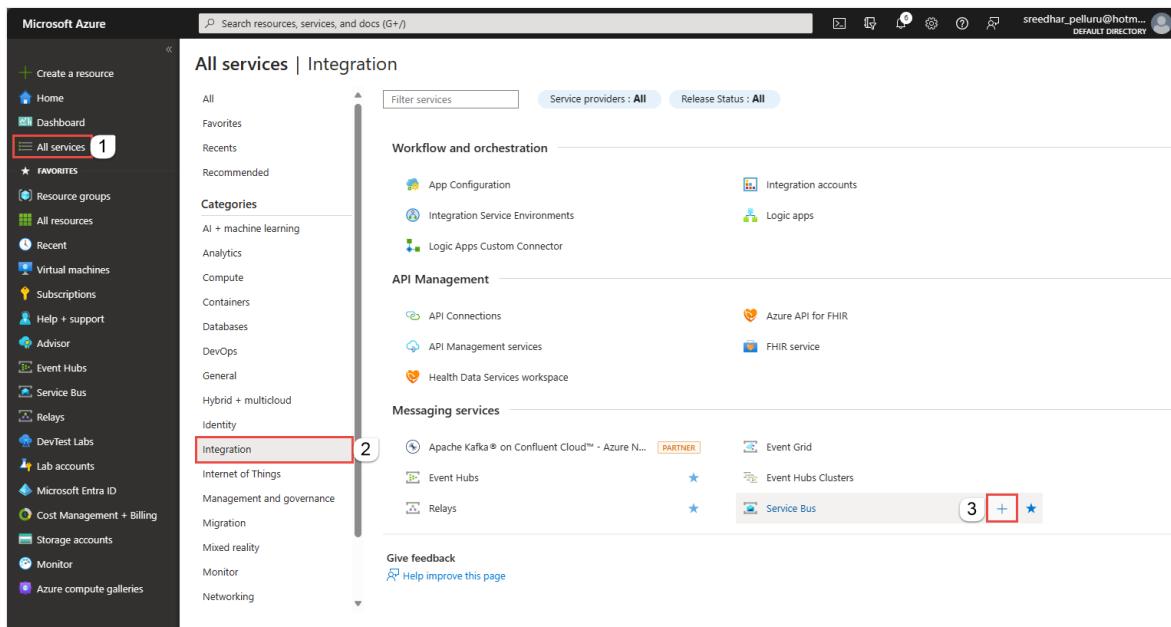
- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [Visual Studio or MSDN subscriber benefits](#) or sign-up for a [free account](#).
- Install [Azure SDK for Java](#). If you're using Eclipse, you can install the [Azure Toolkit for Eclipse](#) that includes the Azure SDK for Java. You can then add the [Microsoft Azure Libraries for Java](#) to your project. If you're using IntelliJ, see [Install the Azure Toolkit for IntelliJ](#).

Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.

To create a namespace:

1. Sign in to the [Azure portal](#).
2. In the left navigation pane of the portal, select **All services**, select **Integration** from the list of categories, hover the mouse over **Service Bus**, and then select + button on the Service Bus tile.



3. In the **Basics** tag of the [Create namespace](#) page, follow these steps:
 - a. For **Subscription**, choose an Azure subscription in which to create the namespace.
 - b. For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
 - c. Enter a **name for the namespace**. The namespace name should adhere to the following naming conventions:
 - The name must be unique across Azure. The system immediately checks to see if the name is available.
 - The name length is at least 6 and at most 50 characters.
 - The name can contain only letters, numbers, hyphens “-”.
 - The name must start with a letter and end with a letter or number.
 - The name doesn't end with “-sb” or “-mgmt”.
 - d. For **Location**, choose the region in which your namespace should be hosted.

- e. For **Pricing tier**, select the pricing tier (Basic, Standard, or Premium) for the namespace. For this quickstart, select **Standard**.

ⓘ Important

If you want to use [topics and subscriptions](#), choose either Standard or Premium. Topics/subscriptions aren't supported in the Basic pricing tier.

If you selected the **Premium** pricing tier, specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, 4, 8 or 16 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

- f. Select **Review + create** at the bottom of the page.

The screenshot shows the 'Create namespace' dialog box. The 'Basics' tab is selected. In the 'Project Details' section, the 'Subscription' dropdown is set to 'Visual Studio Enterprise Subscription'. Below it, the 'Resource group' dropdown shows '(New) spsbusrg' with a 'Create new' link. In the 'Instance Details' section, the 'Namespace name' input field contains 'contosoordersns' with a green checkmark and '.servicebus.windows.net' suffix. The 'Location' dropdown is set to 'East US'. The 'Pricing tier' dropdown is set to 'Standard' with a link to 'Browse the available plans and their features'. At the bottom, there are three buttons: 'Review + create' (highlighted in blue), '< Previous', and 'Next: Advanced >'.

- g. On the **Review + create** page, review settings, and select **Create**.
4. Once the deployment of the resource is successful, select **Go to resource** on the deployment page.

The screenshot shows the 'Deployment' blade for a deployment named 'contosoordersns'. The status is 'Your deployment is complete'. Deployment details include a name 'contosoordersns', a subscription 'Visual Studio Enterprise Subscription', and a resource group 'spsbusrg'. The start time was 10/20/2022, 4:45:03 PM, and the correlation ID is a453ace1-bab9-4c4a-81ad-a1c5366460ea. A 'Go to resource' button is highlighted in red.

5. You see the home page for your service bus namespace.

The screenshot shows the 'Overview' page for a Service Bus Namespace named 'spsbusns1128'. The left sidebar includes options like 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings' (with 'Shared access policies' and 'Geo-Recovery'), 'Migrate to premium', 'Encryption', 'Configuration', 'Properties', 'Locks', 'Entities' (Queues and Topics), 'Monitoring' (with 'Insights (Preview)' and 'Alerts'), and 'Queues (0)' and 'Topics (0)' counts at the bottom. The main area displays 'Essentials' information such as Resource group (spbusrg), Status (Active), Location (East US), Subscription (Visual Studio Enterprise Subscription), Subscription ID, Tags, and various metrics for Requests and Messages over the last hour.

Create a topic using the Azure portal

1. On the Service Bus Namespace page, select **Topics** on the left menu.
2. Select **+ Topic** on the toolbar.
3. Enter a name for the topic. Leave the other options with their default values.
4. Select **Create**.

All services > Resource groups > spsbusrg > spsbusns1128

spsbusns1128 | Topics

Service Bus Namespace

Search 2 + Topic Refresh Give feedback

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings Shared access policies Geo-Recovery Migrate to premium Encryption Configuration Properties Locks

Entities Queues **Topics** 1 Topics

Monitoring Insights (Preview) Alerts

Create 4 Give feedback

Create a subscription to the topic

1. Select the **topic** that you created in the previous section.

spsbusns1128 | Topics

Service Bus Namespace

Search + Topic Refresh Give feedback

Properties Locks

Entities Queues Topics

Monitoring Insights (Preview) Alerts Metrics Diagnostic settings Logs Workbooks

Name	Status	Scheduled messages	Max size	Subscription count	Enable partitioning
mytopic	Active	0	1024 MB	0	false

2. On the **Service Bus Topic** page, select **+ Subscription** on the toolbar.

3. On the **Create subscription** page, follow these steps:

- a. Enter **S1** for **name** of the subscription.
- b. Enter **3** for **Max delivery count**.
- c. Then, select **Create** to create the subscription.

Create subscription

Service Bus

Name * ⓘ

S1



Max delivery count * ⓘ

10

Auto-delete after idle for ⓘ

Days

Hours

Minutes

Seconds

14

0

0

0

Never auto-delete

Forward messages to queue/topic ⓘ

MESSAGE SESSIONS

Service bus sessions allow ordered handling of unbounded sequences of related messages. With sessions enabled a subscription can guarantee first-in-first-out delivery of messages. [Learn more.](#)

Enable sessions

MESSAGE TIME TO LIVE AND DEAD-LETTERING

Message time to live (default) ⓘ

Days

Hours

Minutes

Seconds

14

0

0

0

Enable dead lettering on message expiration

Move messages that cause filter evaluation exceptions to the dead-letter subqueue

MESSAGE LOCK DURATION

Lock duration ⓘ

Days

Hours

Minutes

Seconds

0

0

1

0

Create

Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Service Bus: **passwordless** and **connection string**.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace. You don't

need to worry about having hard-coded connection string in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a connection string to connect to a Service Bus namespace. If you are new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Service Bus has the correct permissions. You'll need the [Azure Service Bus Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Service Bus Data Owner` role to your user account, which provides full access to Azure Service Bus resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

Azure built-in roles for Azure Service Bus

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters). A member of this role can send and receive messages from queues or topics/subscriptions.
- [Azure Service Bus Data Sender](#): Use this role to give the send access to Service Bus namespace and its entities.

- **Azure Service Bus Data Receiver:** Use this role to give the receive access to Service Bus namespace and its entities.

If you want to create a custom role, see [Rights required for Service Bus operations](#).

Add Microsoft Entra user to Azure Service Bus Owner role

Add your Microsoft Entra user name to the **Azure Service Bus Data Owner** role at the Service Bus namespace level. It will allow an app running in the context of your user account to send messages to a queue or a topic, and receive messages from a queue or a topic's subscription.

Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. If you don't have the Service Bus Namespace page open in the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'Access control (IAM)' page for a Service Bus Namespace named 'spsbusns11102'. A red box highlights the 'Access control (IAM)' menu item. Step 1 is indicated by a yellow circle with '1' on the 'Check access' button. Step 2 is indicated by a yellow circle with '2' on the 'Add role assignment' button. Step 3 is indicated by a yellow circle with '3' on the 'Add role assignment' button. The interface includes sections for 'My access', 'Check access', 'Grant access to this resource', and 'View access to this resource'.

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Service Bus Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Send messages to a topic

In this section, you create a Java console project, and add code to send messages to the topic you created.

Create a Java console project

Create a Java project using Eclipse or a tool of your choice.

Configure your application to use Service Bus

Add references to Azure Core and Azure Service Bus libraries.

If you're using Eclipse and created a Java console application, convert your Java project to a Maven: right-click the project in the **Package Explorer** window, select **Configure -> Convert to Maven project**. Then, add dependencies to these two libraries as shown in the following example.

Passwordless (Recommended)

Update the `pom.xml` file to add dependencies to Azure Service Bus and Azure Identity packages.

XML

```
<dependencies>
    <dependency>
        <groupId>com.azure</groupId>
        <artifactId>azure-messaging-servicebus</artifactId>
        <version>7.13.3</version>
    </dependency>
    <dependency>
        <groupId>com.azure</groupId>
        <artifactId>azure-identity</artifactId>
        <version>1.8.0</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Add code to send messages to the topic

1. Add the following `import` statements at the topic of the Java file.

Passwordless (Recommended)

Java

```
import com.azure.messaging.servicebus.*;
import com.azure.identity.*;

import java.util.concurrent.TimeUnit;
import java.util.Arrays;
import java.util.List;
```

2. In the class, define variables to hold connection string (not needed for passwordless scenario), topic name, and subscription name.

Passwordless (Recommended)

Java

```
static String topicName = "<TOPIC NAME>";  
static String subName = "<SUBSCRIPTION NAME>";
```

ⓘ Important

Replace `<TOPIC NAME>` with the name of the topic, and `<SUBSCRIPTION NAME>` with the name of the topic's subscription.

3. Add a method named `sendMessage` in the class to send one message to the topic.

Passwordless (Recommended)

ⓘ Important

Replace `NAMESPACENAME` with the name of your Service Bus namespace.

Java

```
static void sendMessage()  
{  
    // create a token using the default Azure credential  
    DefaultAzureCredential credential = new  
    DefaultAzureCredentialBuilder()  
        .build();  
  
    ServiceBusSenderClient senderClient = new  
    ServiceBusClientBuilder()  
  
        .fullyQualifiedNamespace("NAMESPACENAME.servicebus.windows.net")  
            .credential(credential)  
            .sender()  
            .topicName(topicName)  
            .buildClient();  
  
    // send one message to the topic  
    senderClient.sendMessage(new ServiceBusMessage("Hello,  
    World!"));  
}
```

```
        System.out.println("Sent a single message to the topic: " +  
topicName);  
    }  
  
}
```

4. Add a method named `createMessages` in the class to create a list of messages.

Typically, you get these messages from different parts of your application. Here, we create a list of sample messages.

Java

```
static List<ServiceBusMessage> createMessages()  
{  
    // create a list of messages and return it to the caller  
    ServiceBusMessage[] messages = {  
        new ServiceBusMessage("First message"),  
        new ServiceBusMessage("Second message"),  
        new ServiceBusMessage("Third message")  
    };  
    return Arrays.asList(messages);  
}
```

5. Add a method named `sendMessageBatch` method to send messages to the topic you created. This method creates a `ServiceBusSenderClient` for the topic, invokes the `createMessages` method to get the list of messages, prepares one or more batches, and sends the batches to the topic.

Passwordless (Recommended)

 **Important**

Replace `NAMESPACE NAME` with the name of your Service Bus namespace.

Java

```
static void sendMessageBatch()  
{  
    // create a token using the default Azure credential  
    DefaultAzureCredential credential = new  
    DefaultAzureCredentialBuilder()  
        .build();  
  
    ServiceBusSenderClient senderClient = new  
    ServiceBusClientBuilder()
```

```

        .fullyQualifiedNamespace("NAMESPACENAME.servicebus.windows.net")
            .credential(credential)
            .sender()
            .topicName(topicName)
            .buildClient();

        // Creates an ServiceBusMessageBatch where the ServiceBus.
        ServiceBusMessageBatch messageBatch =
        senderClient.createMessageBatch();

        // create a list of messages
        List<ServiceBusMessage> listOfMessages = createMessages();

        // We try to add as many messages as a batch can fit based on
        the maximum size and send to Service Bus when
        // the batch can hold no more messages. Create a new batch for
        next set of messages and repeat until all
        // messages are sent.
        for (ServiceBusMessage message : listOfMessages) {
            if (messageBatch.tryAddMessage(message)) {
                continue;
            }

            // The batch is full, so we create a new batch and send the
            batch.
            senderClient.sendMessages(messageBatch);
            System.out.println("Sent a batch of messages to the topic:
" + topicName);

            // create a new batch
            messageBatch = senderClient.createMessageBatch();

            // Add that message that we couldn't before.
            if (!messageBatch.tryAddMessage(message)) {
                System.err.printf("Message is too large for an empty
batch. Skipping. Max size: %s.", messageBatch.getMaxSizeInBytes());
            }
        }

        if (messageBatch.getCount() > 0) {
            senderClient.sendMessages(messageBatch);
            System.out.println("Sent a batch of messages to the topic:
" + topicName);
        }

        //close the client
        senderClient.close();
    }
}

```

Receive messages from a subscription

In this section, you add code to retrieve messages from a subscription to the topic.

1. Add a method named `receiveMessages` to receive messages from the subscription. This method creates a `ServiceBusProcessorClient` for the subscription by specifying a handler for processing messages and another one for handling errors. Then, it starts the processor, waits for few seconds, prints the messages that are received, and then stops and closes the processor.

Passwordless (Recommended)

ⓘ Important

- Replace `NAMESPACENAME` with the name of your Service Bus namespace.
- Replace `ServiceBusTopicTest` in `ServiceBusTopicTest::processMessage` in the code with the name of your class.

Java

```
// handles received messages
static void receiveMessages() throws InterruptedException
{
    DefaultAzureCredential credential = new
DefaultAzureCredentialBuilder()
        .build();

    // Create an instance of the processor through the
    ServiceBusClientBuilder
    ServiceBusProcessorClient processorClient = new
ServiceBusClientBuilder()

    .fullyQualifiedNamespace("NAMESPACENAME.servicebus.windows.net")
        .credential(credential)
        .processor()
        .topicName(topicName)
        .subscriptionName(subName)
        .processMessage(context -> processMessage(context))
        .processError(context -> processError(context))
        .buildProcessorClient();

    System.out.println("Starting the processor");
    processorClient.start();

    TimeUnit.SECONDS.sleep(10);
    System.out.println("Stopping and closing the processor");
```

```
    processorClient.close();
}
```

2. Add the `processMessage` method to process a message received from the Service Bus subscription.

Java

```
private static void processMessage(ServiceBusReceivedMessageContext context) {
    ServiceBusReceivedMessage message = context.getMessage();
    System.out.printf("Processing message. Session: %s, Sequence #: %s.
Contents: %s%n", message.getMessageId(),
    message.getSequenceNumber(), message.getBody());
}
```

3. Add the `processError` method to handle error messages.

Java

```
private static void processError(ServiceBusErrorContext context) {
    System.out.printf("Error when receiving messages from namespace:
'%s'. Entity: '%s'%n",
    context.getFullyQualifiedNamespace(), context.getEntityPath());

    if (!(context.getException() instanceof ServiceBusException)) {
        System.out.printf("Non-ServiceBusException occurred: %s%n",
    context.getException());
        return;
    }

    ServiceBusException exception = (ServiceBusException)
context.getException();
    ServiceBusFailureReason reason = exception.getReason();

    if (reason == ServiceBusFailureReason.MESSAGING_ENTITY_DISABLED
        || reason == ServiceBusFailureReason.MESSAGING_ENTITY_NOT_FOUND
        || reason == ServiceBusFailureReason.UNAUTHORIZED) {
        System.out.printf("An unrecoverable error occurred. Stopping
processing with reason %s: %s%n",
        reason, exception.getMessage());
    } else if (reason == ServiceBusFailureReason.MESSAGE_LOCK_LOST) {
        System.out.printf("Message lock lost for message: %s%n",
    context.getException());
    } else if (reason == ServiceBusFailureReason.SERVICE_BUSY) {
        try {
            // Choosing an arbitrary amount of time to wait until
            // trying again.
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
```

```
        System.err.println("Unable to sleep for period of time");
    }
} else {
    System.out.printf("Error source %s, reason %s, message: %s%n",
context.getErrorSource(),
reason, context.getException());
}
}
```

4. Update the `main` method to invoke `sendMessage`, `sendMessageBatch`, and `receiveMessages` methods and to throw `InterruptedException`.

Java

```
public static void main(String[] args) throws InterruptedException {
    sendMessage();
    sendMessageBatch();
    receiveMessages();
}
```

Run the app

Run the program to see the output similar to the following output:

Passwordless (Recommended)

1. If you're using Eclipse, right-click the project, select **Export**, expand **Java**, select **Runnable JAR file**, and follow the steps to create a runnable JAR file.
2. If you're signed into the machine using a user account that's different from the user account added to the **Azure Service Bus Data Owner** role, follow these steps. Otherwise, skip this step and move on to run the Jar file in the next step.
 - a. [Install Azure CLI](#) on your machine.
 - b. Run the following CLI command to sign in to Azure. Use the same user account that you added to the **Azure Service Bus Data Owner** role.

Azure CLI

```
az login
```

3. Run the Jar file using the following command.

Java

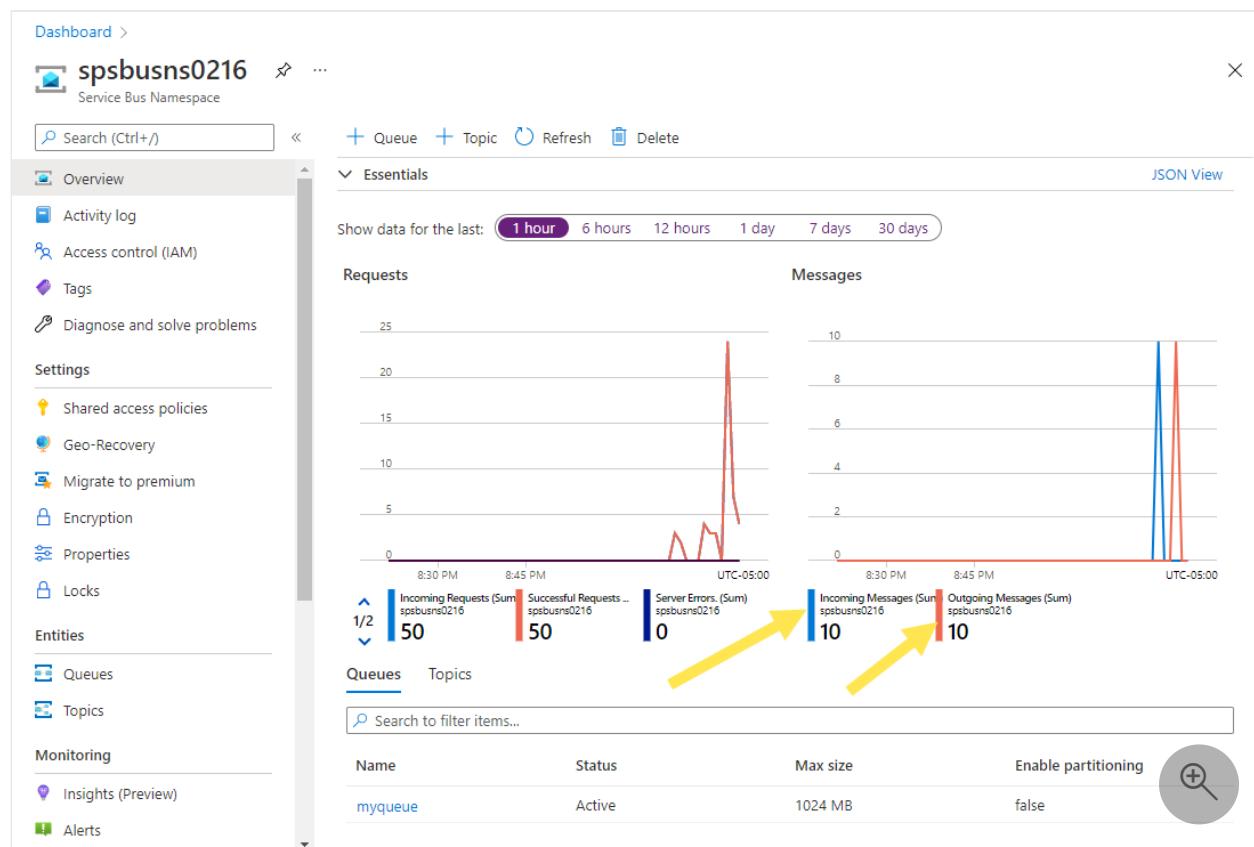
```
java -jar <JAR FILE NAME>
```

4. You see the following output in the console window.

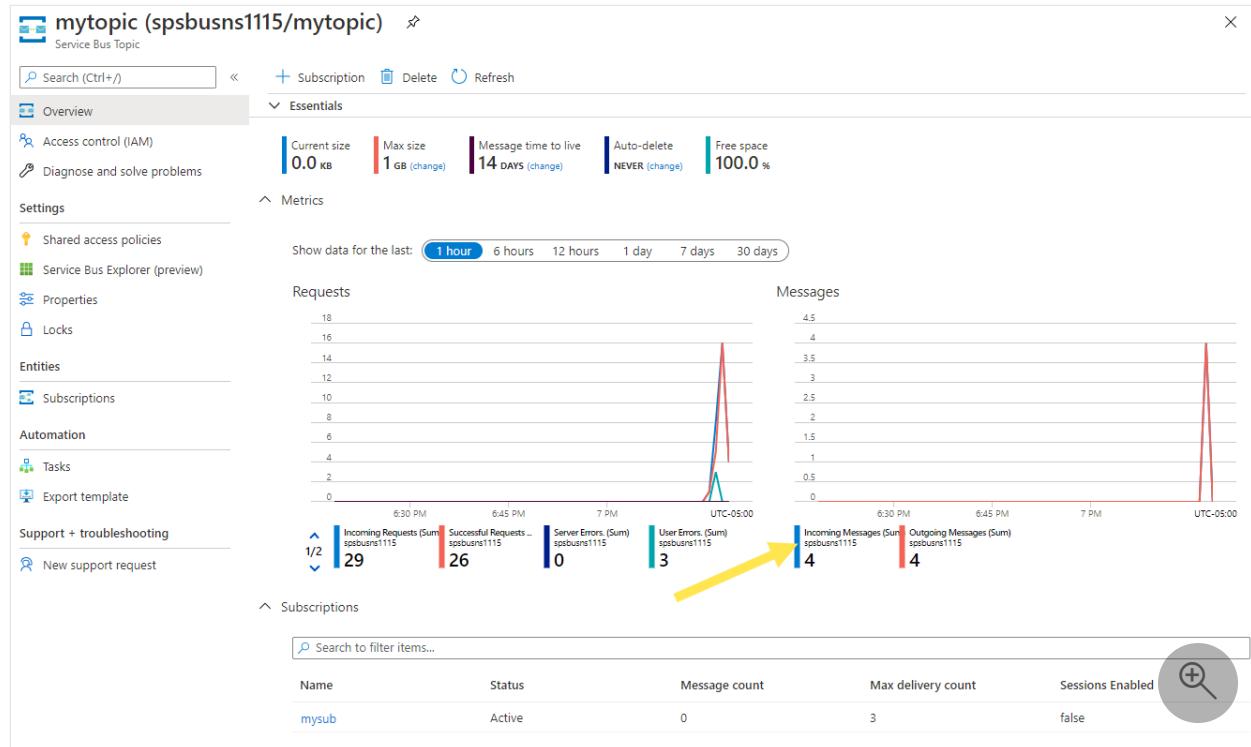
Console

```
Sent a single message to the topic: mytopic
Sent a batch of messages to the topic: mytopic
Starting the processor
Processing message. Session: e0102f5fbaf646988a2f4b65f7d32385,
Sequence #: 1. Contents: Hello, World!
Processing message. Session: 3e991e232ca248f2bc332caa8034bed9,
Sequence #: 2. Contents: First message
Processing message. Session: 56d3a9ea7df446f8a2944ee72cca4ea0,
Sequence #: 3. Contents: Second message
Processing message. Session: 7bd3bd3e966a40ebbc9b29b082da14bb,
Sequence #: 4. Contents: Third message
```

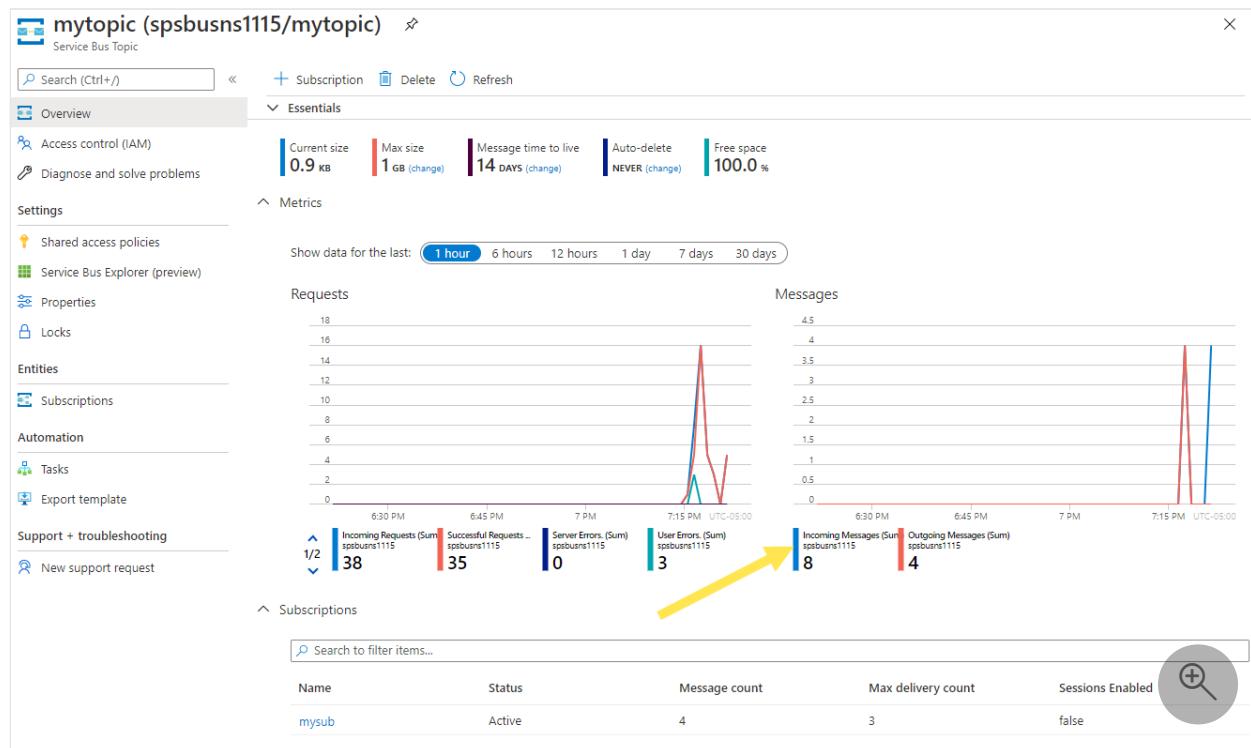
On the **Overview** page for the Service Bus namespace in the Azure portal, you can see **incoming** and **outgoing** message count. Wait for a minute or so and then refresh the page to see the latest values.



Switch to the **Topics** tab in the middle-bottom pane, and select the topic to see the **Service Bus Topic** page for your topic. On this page, you should see four incoming and four outgoing messages in the **Messages** chart.



If you comment out the `receiveMessages` call in the `main` method and run the app again, on the **Service Bus Topic** page, you see 8 incoming messages (4 new) but four outgoing messages.



On this page, if you select a subscription, you get to the **Service Bus Subscription** page. You can see the active message count, dead-letter message count, and more on this

page. In this example, there are four active messages that the receiver hasn't received yet.

The screenshot shows the Azure Service Bus Subscription 'mysub' (spsbusns1115/mytopic/mysub) overview. A yellow arrow points to the 'Active message count' section, which displays '4 MESSAGES'. Other metrics shown include Max delivery count (3), Message time to live (14 DAYS), Message lock duration (30 SECONDS), Auto-delete after idle for (14 DAYS), Dead-letter message count (0 MESSAGES), Transfer message count (0 MESSAGES), Scheduled message count (0 MESSAGES), and Transfer message count (0 MESSAGES). The 'FILTERS' section shows a single filter named '\$Default' of type SqlFilter. A search bar and refresh button are also visible at the top.

Next steps

See the following documentation and samples:

- [Azure Service Bus client library for Java - Readme ↗](#)
- [Samples on GitHub](#)
- [Java API reference ↗](#)

Quickstart: Azure Blob Storage client library for Java

Article • 03/04/2024

ⓘ Note

The **Build from scratch** option walks you step by step through the process of creating a new project, installing packages, writing the code, and running a basic console app. This approach is recommended if you want to understand all the details involved in creating an app that connects to Azure Blob Storage. If you prefer to automate deployment tasks and start with a completed project, choose [Start with a template](#).

Get started with the Azure Blob Storage client library for Java to manage blobs and containers.

In this article, you follow steps to install the package and try out example code for basic tasks.

💡 Tip

If you're working with Azure Storage resources in a Spring application, we recommend that you consider [Spring Cloud Azure](#) as an alternative. Spring Cloud Azure is an open-source project that provides seamless Spring integration with Azure services. To learn more about Spring Cloud Azure, and to see an example using Blob Storage, see [Upload a file to an Azure Storage Blob](#).

[API reference documentation](#) | [Library source code](#) | [Package \(Maven\)](#) | [Samples](#)

Prerequisites

- Azure account with an active subscription - [create an account for free](#)
- Azure Storage account - [create a storage account](#).
- [Java Development Kit \(JDK\)](#) version 8 or above
- [Apache Maven](#)

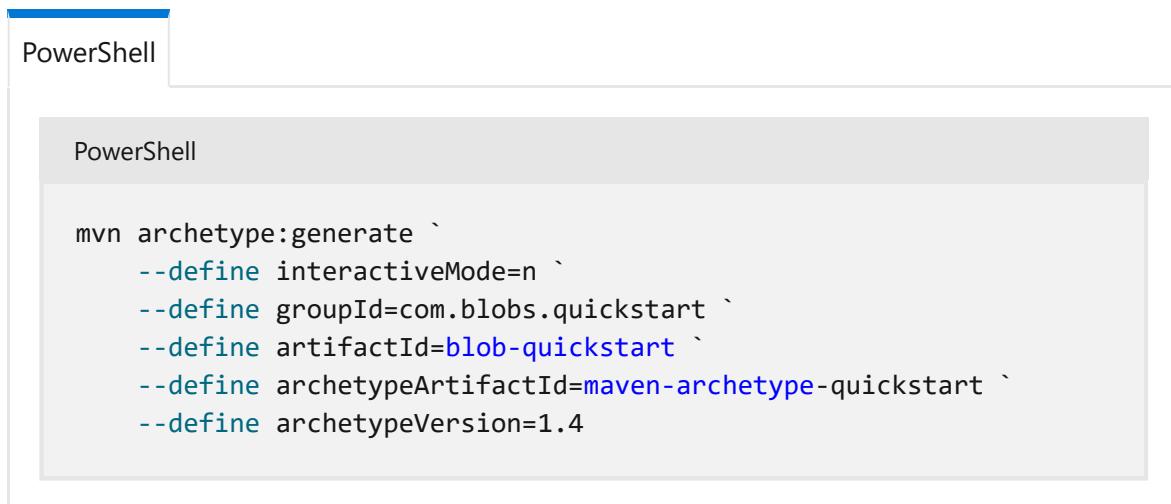
Setting up

This section walks you through preparing a project to work with the Azure Blob Storage client library for Java.

Create the project

Create a Java application named *blob-quickstart*.

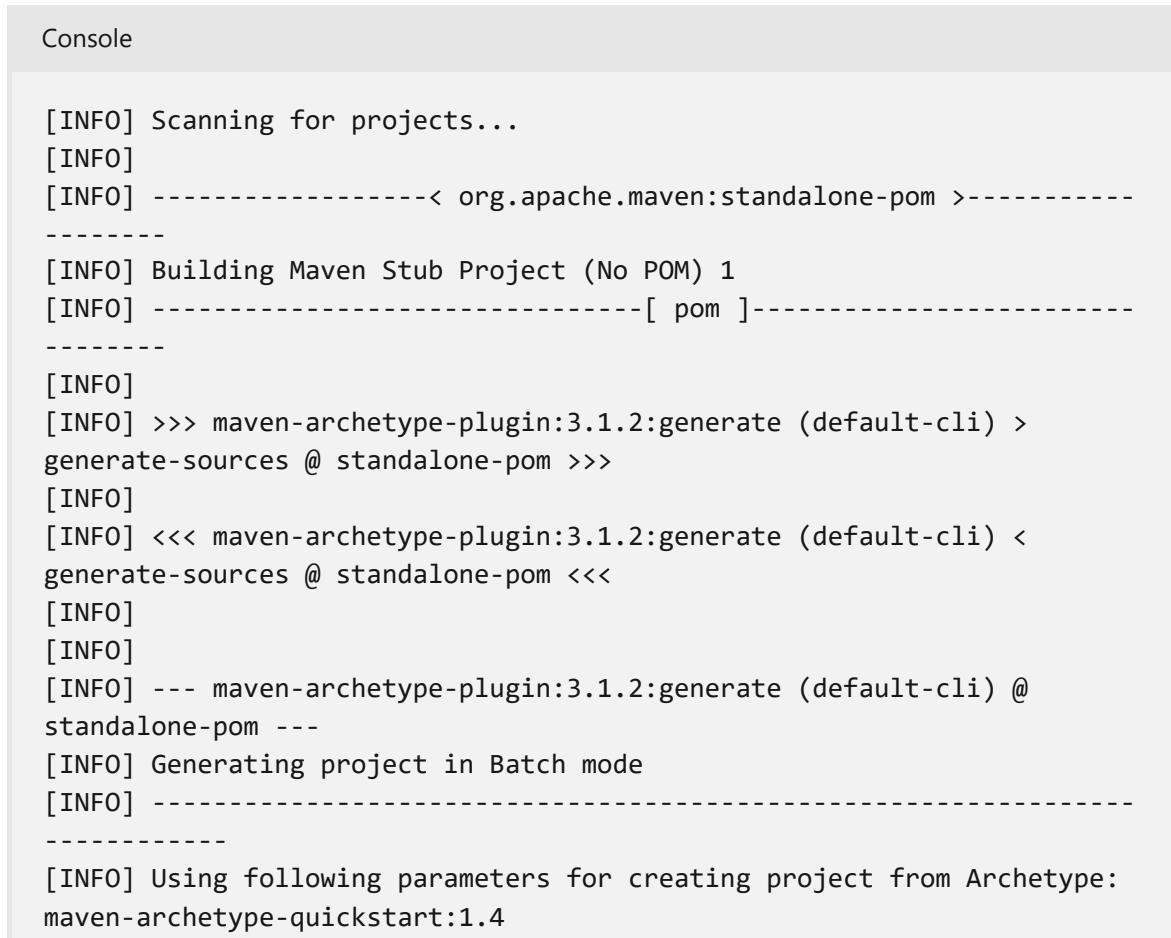
1. In a console window (such as PowerShell or Bash), use Maven to create a new console app with the name *blob-quickstart*. Type the following **mvn** command to create a "Hello world!" Java project.



```
PowerShell

mvn archetype:generate \
  --define interactiveMode=n \
  --define groupId=com.blobs.quickstart \
  --define artifactId=blob-quickstart \
  --define archetypeArtifactId=maven-archetype-quickstart \
  --define archetypeVersion=1.4
```

2. The output from generating the project should look something like this:



```
Console

[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.apache.maven:standalone-pom >-----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----[ pom ]-----
[INFO]
[INFO] >>> maven-archetype-plugin:3.1.2:generate (default-cli) >
generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:3.1.2:generate (default-cli) <
generate-sources @ standalone-pom <<<
[INFO]
[INFO]
[INFO] --- maven-archetype-plugin:3.1.2:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] -----
[INFO] Using following parameters for creating project from Archetype:
maven-archetype-quickstart:1.4
```

```
[INFO] -----
[INFO] Parameter: groupId, Value: com.blobs.quickstart
[INFO] Parameter: artifactId, Value: blob-quickstart
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.blobs.quickstart
[INFO] Parameter: packageInPathFormat, Value: com/blobs/quickstart
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.blobs.quickstart
[INFO] Parameter: groupId, Value: com.blobs.quickstart
[INFO] Parameter: artifactId, Value: blob-quickstart
[INFO] Project created from Archetype in dir: C:\QuickStarts\blob-quickstart
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.056 s
[INFO] Finished at: 2019-10-23T11:09:21-07:00
[INFO] -----
```
``
```

3. Switch to the newly created *blob-quickstart* folder.

```
Console
```

```
cd blob-quickstart
```

4. Inside the *blob-quickstart* directory, create another directory called *data*. This folder is where the blob data files will be created and stored.

```
Console
```

```
mkdir data
```

## Install the packages

Open the `pom.xml` file in your text editor.

Add `azure-sdk-bom` to take a dependency on the latest version of the library. In the following snippet, replace the `{bom_version_to_target}` placeholder with the version number. Using `azure-sdk-bom` keeps you from having to specify the version of each individual dependency. To learn more about the BOM, see the [Azure SDK BOM README](#).

XML

```
<dependencyManagement>
 <dependencies>
 <dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-sdk-bom</artifactId>
 <version>{bom_version_to_target}</version>
 <type>pom</type>
 <scope>import</scope>
 </dependency>
 </dependencies>
</dependencyManagement>
```

Then add the following dependency elements to the group of dependencies. The **azure-identity** dependency is needed for passwordless connections to Azure services.

XML

```
<dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-storage-blob</artifactId>
</dependency>
<dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-identity</artifactId>
</dependency>
```

## Set up the app framework

From the project directory, follow steps to create the basic structure of the app:

1. Navigate to the `/src/main/java/com/blobs/quickstart` directory
2. Open the `App.java` file in your editor
3. Delete the line `System.out.println("Hello world!");`
4. Add the necessary `import` directives

The code should resemble this framework:

Java

```
package com.blobs.quickstart;

/**
 * Azure Blob Storage quickstart
 */
import com.azure.identity.*;
import com.azure.storage.blob.*;
```

```

import com.azure.storage.blob.models.*;
import java.io.*;

public class App
{
 public static void main(String[] args) throws IOException
 {
 // Quickstart code goes here
 }
}

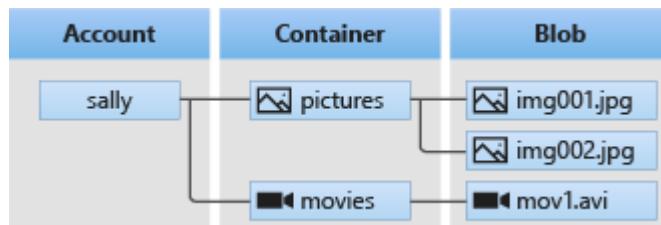
```

## Object model

Azure Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data doesn't adhere to a particular data model or definition, such as text or binary data. Blob storage offers three types of resources:

- The storage account
- A container in the storage account
- A blob in the container

The following diagram shows the relationship between these resources.



Use the following Java classes to interact with these resources:

- **BlobServiceClient**: The `BlobServiceClient` class allows you to manipulate Azure Storage resources and blob containers. The storage account provides the top-level namespace for the Blob service.
- **BlobServiceClientBuilder**: The `BlobServiceClientBuilder` class provides a fluent builder API to help aid the configuration and instantiation of `BlobServiceClient` objects.
- **BlobContainerClient**: The `BlobContainerClient` class allows you to manipulate Azure Storage containers and their blobs.
- **BlobClient**: The `BlobClient` class allows you to manipulate Azure Storage blobs.
- **BlobItem**: The `BlobItem` class represents individual blobs returned from a call to `listBlobs`.

# Code examples

These example code snippets show you how to perform the following actions with the Azure Blob Storage client library for Java:

- [Authenticate to Azure and authorize access to blob data](#)
- [Create a container](#)
- [Upload blobs to a container](#)
- [List the blobs in a container](#)
- [Download blobs](#)
- [Delete a container](#)

## ⓘ Important

Make sure you have the correct dependencies in pom.xml and the necessary directives for the code samples to work, as described in the [setting up](#) section.

## Authenticate to Azure and authorize access to blob data

Application requests to Azure Blob Storage must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code, including Blob Storage.

You can also authorize requests to Azure Blob Storage by using the account access key. However, this approach should be used with caution. Developers must be diligent to never expose the access key in an unsecure location. Anyone who has the access key is able to authorize requests against the storage account, and effectively has access to all the data. `DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

### Passwordless (Recommended)

`DefaultAzureCredential` is a class provided by the Azure Identity client library for Java. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` looks for credentials can be found in the [Azure Identity library overview](#).

For example, your app can authenticate using your Visual Studio Code sign-in credentials with when developing locally. Your app can then use a [managed identity](#) once it has been deployed to Azure. No code changes are required for this transition.

## Assign roles to your Microsoft Entra user account

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

### Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.

3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'identitymigrationstorage | Access Control (IAM)' page. The left sidebar has a red box around 'Access Control (IAM)'. The top navigation bar has a red box around '+ Add' and 'Add role assignment'. The main content area shows sections for 'My access', 'Check access', and 'Grant access to this resource'. A large red box highlights the 'Add role assignment' button under 'Grant access to this resource'.

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Sign-in and connect your app code to Azure using DefaultAzureCredential

You can authorize access to data in your storage account using the following steps:

1. Make sure you're authenticated with the same Microsoft Entra account you assigned the role to on your storage account. You can authenticate via the Azure CLI, Visual Studio Code, or Azure PowerShell.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

Azure CLI

```
az login
```

2. To use `DefaultAzureCredential`, make sure that the `azure-identity` dependency is added in `pom.xml`:

XML

```
<dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-identity</artifactId>
</dependency>
```

3. Add this code to the `Main` method. When the code runs on your local workstation, it will use the developer credentials of the prioritized tool you're logged into to authenticate to Azure, such as the Azure CLI or Visual Studio Code.

Java

```
/*
 * The default credential first checks environment variables for
 * configuration
 * If environment configuration is incomplete, it will try managed
 * identity
 */
DefaultAzureCredential defaultCredential = new
DefaultAzureCredentialBuilder().build();

// Azure SDK client builders accept the credential as a parameter
// TODO: Replace <storage-account-name> with your actual storage
account name
BlobServiceClient blobServiceClient = new
BlobServiceClientBuilder()
 .endpoint("https://<storage-account-
name>.blob.core.windows.net/")
 .credential(defaultCredential)
 .buildClient();
```

4. Make sure to update the storage account name in the URI of your `BlobServiceClient`. The storage account name can be found on the overview page of the Azure portal.

A screenshot of the Azure Storage account settings page for 'identitymigrationstorage'. The account name is at the top, followed by a search bar and navigation icons. On the left is a sidebar with links like Overview, Activity log, Tags, Diagnose and solve problems, Access Control (IAM), Data migration, Events, and Storage browser. The main area is titled 'Essentials' and contains the following information:

- Resource group ([move](#)) : alexw-identity-revamp
- Location : East US
- Primary/Secondary Location : Primary: East US, Secondary: West US
- Subscription ([move](#)) : C&L Cross Service Content Team Testing
- Subscription ID :
- Disk state : Primary: Available, Secondary: Available
- Tags ([edit](#)) : Click here to add tags

### ⓘ Note

When deployed to Azure, this same code can be used to authorize requests to Azure Storage from an application running in Azure. However, you'll need to enable managed identity on your app in Azure. Then configure your storage account to allow that managed identity to connect. For detailed instructions on configuring this connection between Azure services, see the [Auth from Azure-hosted apps](#) tutorial.

## Create a container

Create a new container in your storage account by calling the `createBlobContainer` method on the `blobServiceClient` object. In this example, the code appends a GUID value to the container name to ensure that it's unique.

Add this code to the end of the `Main` method:

Java

```
// Create a unique name for the container
String containerName = "quickstartblobs" + java.util.UUID.randomUUID();

// Create the container and return a container client object
BlobContainerClient blobContainerClient =
blobServiceClient.createBlobContainer(containerName);
```

To learn more about creating a container, and to explore more code samples, see [Create a blob container with Java](#).

## ⓘ Important

Container names must be lowercase. For more information about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

## Upload blobs to a container

Upload a blob to a container by calling the [uploadFromFile](#) method. The example code creates a text file in the local *data* directory to upload to the container.

Add this code to the end of the `Main` method:

Java

```
// Create the ./data/ directory and a file for uploading and downloading
String localPath = "./data/";
new File(localPath).mkdirs();
String fileName = "quickstart" + java.util.UUID.randomUUID() + ".txt";

// Get a reference to a blob
BlobClient blobClient = blobContainerClient.getBlobClient(fileName);

// Write text to the file
FileWriter writer = null;
try
{
 writer = new FileWriter(localPath + fileName, true);
 writer.write("Hello, World!");
 writer.close();
}
catch (IOException ex)
{
 System.out.println(ex.getMessage());
}

System.out.println("\nUploading to Blob storage as blob:\n\t" +
blobClient.getBlobUrl());

// Upload the blob
blobClient.uploadFromFile(localPath + fileName);
```

To learn more about uploading blobs, and to explore more code samples, see [Upload a blob with Java](#).

## List the blobs in a container

List the blobs in the container by calling the [listBlobs](#) method. In this case, only one blob has been added to the container, so the listing operation returns just that one blob.

Add this code to the end of the `Main` method:

Java

```
System.out.println("\nListing blobs...");

// List the blob(s) in the container.
for (BlobItem blobItem : blobContainerClient.listBlobs()) {
 System.out.println("\t" + blobItem.getName());
}
```

To learn more about listing blobs, and to explore more code samples, see [List blobs with Java](#).

## Download blobs

Download the previously created blob by calling the [downloadToFile](#) method. The example code adds a suffix of "DOWNLOAD" to the file name so that you can see both files in local file system.

Add this code to the end of the `Main` method:

Java

```
// Download the blob to a local file

// Append the string "DOWNLOAD" before the .txt extension for comparison
// purposes
String downloadFileName = fileName.replace(".txt", "DOWNLOAD.txt");

System.out.println("\nDownloading blob to\n\t" + localPath +
 downloadFileName);

blobClient.downloadToFile(localPath + downloadFileName);
```

To learn more about downloading blobs, and to explore more code samples, see [Download a blob with Java](#).

## Delete a container

The following code cleans up the resources the app created by removing the entire container using the [delete](#) method. It also deletes the local files created by the app.

The app pauses for user input by calling `System.console().readLine()` before it deletes the blob, container, and local files. This is a good chance to verify that the resources were created correctly, before they're deleted.

Add this code to the end of the `Main` method:

Java

```
File downloadedFile = new File(localPath + downloadFileName);
File localFile = new File(localPath + fileName);

// Clean up resources
System.out.println("\nPress the Enter key to begin clean up");
System.console().readLine();

System.out.println("Deleting blob container...");
blobContainerClient.delete();

System.out.println("Deleting the local source and downloaded files...");
localFile.delete();
downloadedFile.delete();

System.out.println("Done");
```

To learn more about deleting a container, and to explore more code samples, see [Delete and restore a blob container with Java](#).

## Run the code

This app creates a test file in your local folder and uploads it to Blob storage. The example then lists the blobs in the container and downloads the file with a new name so that you can compare the old and new files.

Follow steps to compile, package, and run the code

1. Navigate to the directory containing the `pom.xml` file and compile the project by using the following `mvn` command:

Console

```
mvn compile
```

2. Package the compiled code in its distributable format:

Console

```
mvn package
```

3. Run the following `mvn` command to execute the app:

Console

```
mvn exec:java -D exec.mainClass=com.blobs.quickstart.App -D
exec.cleanupDaemonThreads=false
```

To simplify the run step, you can add `exec-maven-plugin` to `pom.xml` and configure as shown below:

XML

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>exec-maven-plugin</artifactId>
 <version>1.4.0</version>
 <configuration>
 <mainClass>com.blobs.quickstart.App</mainClass>
 <cleanupDaemonThreads>false</cleanupDaemonThreads>
 </configuration>
</plugin>
```

With this configuration, you can execute the app with the following command:

Console

```
mvn exec:java
```

The output of the app is similar to the following example (UUID values omitted for readability):

Output

```
Azure Blob Storage - Java quickstart sample
```

```
Uploading to Blob storage as blob:
```

```
https://mystorageacct.blob.core.windows.net/quickstartblobsUUID/quickstartUU
ID.txt
```

```
Listing blobs...
 quickstartUUID.txt
```

```
Downloading blob to
./data/quickstartUUIDDOWNLOAD.txt
```

```
Press the Enter key to begin clean up
```

```
Deleting blob container...
Deleting the local source and downloaded files...
Done
```

Before you begin the cleanup process, check your *data* folder for the two files. You can compare them and observe that they're identical.

## Clean up resources

After you've verified the files and finished testing, press the **Enter** key to delete the test files along with the container you created in the storage account. You can also use [Azure CLI](#) to delete resources.

## Next steps

In this quickstart, you learned how to upload, download, and list blobs using Java.

To see Blob storage sample apps, continue to:

### Azure Blob Storage library for Java samples

- To learn more, see the [Azure Blob Storage client libraries for Java](#).
- For tutorials, samples, quickstarts, and other documentation, visit [Azure for Java developers](#).

# Quickstart: Azure Queue Storage client library for Java

Article • 06/29/2023

Get started with the Azure Queue Storage client library for Java. Azure Queue Storage is a service for storing large numbers of messages for later retrieval and processing. Follow these steps to install the package and try out example code for basic tasks.

[API reference documentation](#) | [Library source code](#) | [Package \(Maven\)](#) | [Samples](#)

Use the Azure Queue Storage client library for Java to:

- Create a queue
- Add messages to a queue
- Peek at messages in a queue
- Update a message in a queue
- Get the queue length
- Receive messages from a queue
- Delete messages from a queue
- Delete a queue

## Prerequisites

- [Java Development Kit \(JDK\)](#) version 8 or above
- [Apache Maven](#)
- Azure subscription - [create one for free](#)
- Azure Storage account - [create a storage account](#)

## Setting up

This section walks you through preparing a project to work with the Azure Queue Storage client library for Java.

### Create the project

Create a Java application named *queues-quickstart*.

1. In a console window (such as cmd, PowerShell, or Bash), use Maven to create a new console app with the name *queues-quickstart*. Type the following `mvn`

command to create a "Hello, world!" Java project.

PowerShell

```
mvn archetype:generate `--define interactiveMode=n`
--define groupId=com.queues.quickstart`
--define artifactId=queues-quickstart`
--define archetypeArtifactId=maven-archetype-quickstart`
--define archetypeVersion=1.4
```

2. The output from generating the project should look something like this:

Console

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.apache.maven:standalone-pom >-----

[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----[pom]-----

[INFO]
[INFO] >>> maven-archetype-plugin:3.1.2:generate (default-cli) >
generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:3.1.2:generate (default-cli) <
generate-sources @ standalone-pom <<<
[INFO]
[INFO]
[INFO] --- maven-archetype-plugin:3.1.2:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] -----

[INFO] Using following parameters for creating project from Archetype:
maven-archetype-quickstart:1.4
[INFO] -----

[INFO] Parameter: groupId, Value: com.queues.quickstart
[INFO] Parameter: artifactId, Value: queues-quickstart
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.queues.quickstart
[INFO] Parameter: packageInPathFormat, Value: com/queues/quickstart
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.queues.quickstart
[INFO] Parameter: groupId, Value: com.queues.quickstart
[INFO] Parameter: artifactId, Value: queues-quickstart
[INFO] Project created from Archetype in dir:
```

```
C:\quickstarts\queues\queues-quickstart
[INFO] -----

[INFO] BUILD SUCCESS
[INFO] -----

[INFO] Total time: 6.394 s
[INFO] Finished at: 2019-12-03T09:58:35-08:00
[INFO] -----

```

3. Switch to the newly created *queues-quickstart* directory.

```
Console
cd queues-quickstart
```

## Install the packages

Open the `pom.xml` file in your text editor.

Add **azure-sdk-bom** to take a dependency on the latest version of the library. In the following snippet, replace the `{bom_version_to_target}` placeholder with the version number. Using **azure-sdk-bom** keeps you from having to specify the version of each individual dependency. To learn more about the BOM, see the [Azure SDK BOM README](#).

```
XML
<dependencyManagement>
 <dependencies>
 <dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-sdk-bom</artifactId>
 <version>{bom_version_to_target}</version>
 <type>pom</type>
 <scope>import</scope>
 </dependency>
 </dependencies>
</dependencyManagement>
```

Then add the following dependency elements to the group of dependencies. The **azure-identity** dependency is needed for passwordless connections to Azure services.

```
XML
```

```
<dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-storage-queue</artifactId>
</dependency>
<dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-identity</artifactId>
</dependency>
```

## Set up the app framework

From the project directory:

1. Navigate to the `/src/main/java/com/queues/quickstart` directory
2. Open the `App.java` file in your editor
3. Delete the `System.out.println("Hello, world");` statement
4. Add `import` directives

Here's the code:

Java

```
package com.queues.quickstart;

/**
 * Azure Queue Storage client library quickstart
 */
import com.azure.identity.*;
import com.azure.storage.queue.*;
import com.azure.storage.queue.models.*;
import java.io.*;

public class App
{
 public static void main(String[] args) throws IOException
 {
 // Quickstart code goes here
 }
}
```

## Authenticate to Azure

Application requests to most Azure services must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the

recommended approach for implementing passwordless connections to Azure services in your code.

You can also authorize requests to Azure services using passwords, connection strings, or other credentials directly. However, this approach should be used with caution.

Developers must be diligent to never expose these secrets in an unsecure location.

Anyone who gains access to the password or secret key is able to authenticate.

`DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

#### Passwordless (Recommended)

`DefaultAzureCredential` is a class provided by the Azure Identity client library for Java. To learn more about `DefaultAzureCredential`, see the [DefaultAzureCredential overview](#). `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

For example, your app can authenticate using your Azure CLI sign-in credentials when developing locally, and then use a [managed identity](#) once it has been deployed to Azure. No code changes are required for this transition.

When developing locally, make sure that the user account that is accessing the queue data has the correct permissions. You'll need [Storage Queue Data Contributor](#) to read and write queue data. To assign yourself this role, you'll need to be assigned the [User Access Administrator](#) role, or another role that includes the [Microsoft.Authorization/roleAssignments/write](#) action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

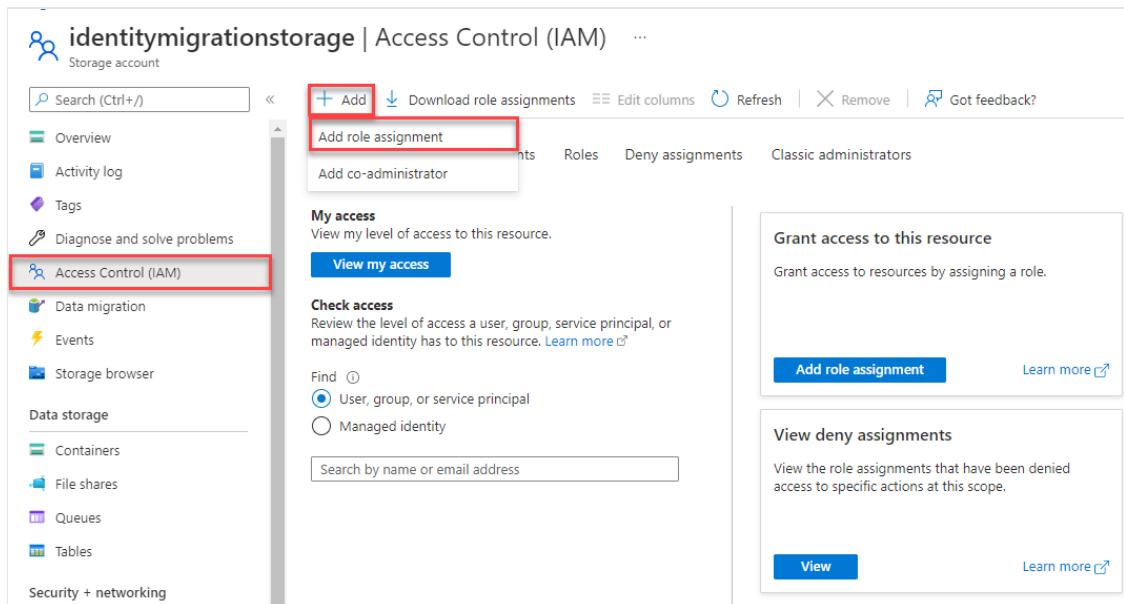
The following example will assign the [Storage Queue Data Contributor](#) role to your user account, which provides both read and write access to queue data in your storage account.

#### ⓘ Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

## Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



The screenshot shows the Azure portal interface for managing access control. The left sidebar lists various storage account components like Overview, Activity log, Tags, and Data storage. The 'Access Control (IAM)' section is highlighted with a red box. The main area shows a table with columns for Role, Principal, and Start date. At the top of the table, there's a '+ Add' button and a 'Download role assignments' link. A dropdown menu is open over the '+ Add' button, with 'Add role assignment' highlighted by a red box. To the right of the table, there are sections for 'My access', 'Check access', and 'Grant access to this resource', each with its own 'Add role assignment' button.

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Queue Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.

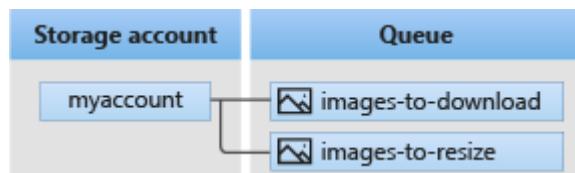
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Object model

Azure Queue Storage is a service for storing large numbers of messages. A queue message can be up to 64 KB in size. A queue may contain millions of messages, up to the total capacity limit of a storage account. Queues are commonly used to create a backlog of work to process asynchronously. Queue Storage offers three types of resources:

- **Storage account:** All access to Azure Storage is done through a storage account. For more information about storage accounts, see [Storage account overview](#)
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. A message can remain in the queue for a maximum of 7 days. For version 2017-07-29 or later, the maximum time-to-live can be any positive number, or -1 indicating that the message doesn't expire. If this parameter is omitted, the default time-to-live is seven days.

The following diagram shows the relationship between these resources.



Use the following Java classes to interact with these resources:

- **QueueClientBuilder:** The `QueueClientBuilder` class configures and instantiates a `QueueClient` object.
- **QueueServiceClient:** The `QueueServiceClient` allows you to manage the all queues in your storage account.
- **QueueClient:** The `QueueClient` class allows you to manage and manipulate an individual queue and its messages.
- **QueueMessageItem:** The `QueueMessageItem` class represents the individual objects returned when calling `ReceiveMessages` on a queue.

# Code examples

These example code snippets show you how to do the following actions with the Azure Queue Storage client library for Java:

- [Authorize access and create a client object](#)
- [Create a queue](#)
- [Add messages to a queue](#)
- [Peek at messages in a queue](#)
- [Update a message in a queue](#)
- [Get the queue length](#)
- [Receive and delete messages from a queue](#)
- [Delete a queue](#)

Passwordless (Recommended)

## Authorize access and create a client object

Make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via Azure CLI, Visual Studio Code, or Azure PowerShell.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

Azure CLI

```
az login
```

Once authenticated, you can create and authorize a `QueueClient` object using `DefaultAzureCredential` to access queue data in the storage account. `DefaultAzureCredential` automatically discovers and uses the account you signed in with in the previous step.

To authorize using `DefaultAzureCredential`, make sure you've added the `azure-identity` dependency in `pom.xml`, as described in [Install the packages](#). Also, be sure to add an import directive for `com.azure.identity` in the `App.java` file:

Java

```
import com.azure.identity.*;
```

Decide on a name for the queue and create an instance of the [QueueClient](#) class, using `DefaultAzureCredential` for authorization. We use this client object to create and interact with the queue resource in the storage account.

### ⓘ Important

Queue names may only contain lowercase letters, numbers, and hyphens, and must begin with a letter or a number. Each hyphen must be preceded and followed by a non-hyphen character. The name must also be between 3 and 63 characters long. For more information about naming queues, see [Naming queues and metadata](#).

Add this code inside the `main` method, and make sure to replace the `<storage-account-name>` placeholder value:

Java

```
System.out.println("Azure Queue Storage client library - Java quickstart sample\n");

// Create a unique name for the queue
String queueName = "quickstartqueues-" + java.util.UUID.randomUUID();

// Instantiate a QueueClient
// We'll use this client object to create and interact with the queue
// TODO: replace <storage-account-name> with the actual name
QueueClient queueClient = new QueueClientBuilder()
 .endpoint("https://<storage-account-name>.queue.core.windows.net/")
 .queueName(queueName)
 .credential(new DefaultAzureCredentialBuilder().build())
 .buildClient();
```

### ⓘ Note

Messages sent using the `QueueClient` class must be in a format that can be included in an XML request with UTF-8 encoding. You can optionally set the `QueueMessageEncoding` option to `BASE64` to handle non-compliant messages.

## Create a queue

Using the `QueueClient` object, call the `create` method to create the queue in your storage account.

Add this code to the end of the `main` method:

Java

```
System.out.println("Creating queue: " + queueName);

// Create the queue
queueClient.create();
```

## Add messages to a queue

The following code snippet adds messages to queue by calling the `sendMessage` method. It also saves a `SendMessageResult` returned from a `sendMessage` call. The result is used to update the message later in the program.

Add this code to the end of the `main` method:

Java

```
System.out.println("\nAdding messages to the queue...");

// Send several messages to the queue
queueClient.sendMessage("First message");
queueClient.sendMessage("Second message");

// Save the result so we can update this message later
SendMessageResult result = queueClient.sendMessage("Third message");
```

## Peek at messages in a queue

Peek at the messages in the queue by calling the `peekMessages` method. This method retrieves one or more messages from the front of the queue but doesn't alter the visibility of the message.

Add this code to the end of the `main` method:

Java

```
System.out.println("\nPeek at the messages in the queue...");
```

```
// Peek at messages in the queue
queueClient.peekMessages(10, null, null).forEach(
 peekedMessage -> System.out.println("Message: " +
peekedMessage.getMessageText()));
```

## Update a message in a queue

Update the contents of a message by calling the [updateMessage](#) method. This method can change a message's visibility timeout and contents. The message content must be a UTF-8 encoded string that is up to 64 KB in size. Along with new content for the message, pass in the message ID and pop receipt by using the [SendMessageResult](#) that was saved earlier in the code. The message ID and pop receipt identify which message to update.

Java

```
System.out.println("\nUpdating the third message in the queue...");

// Update a message using the result that
// was saved when sending the message
queueClient.updateMessage(result.getMessageId(),
 result.getPopReceipt(),
 "Third message has been updated",
 Duration.ofSeconds(1));
```

## Get the queue length

You can get an estimate of the number of messages in a queue.

The [getProperties](#) method returns several values including the number of messages currently in a queue. The count is only approximate because messages can be added or removed after your request. The [getApproximateMessageCount](#) method returns the last value retrieved by the call to [getProperties](#), without calling Queue Storage.

Java

```
QueueProperties properties = queueClient.getProperties();
long messageCount = properties.getApproximateMessagesCount();

System.out.println(String.format("Queue length: %d", messageCount));
```

## Receive and delete messages from a queue

Download previously added messages by calling the `receiveMessages` method. The example code also deletes messages from the queue after they're received and processed. In this case, processing is just displaying the message on the console.

The app pauses for user input by calling `System.console().readLine();` before it receives and deletes the messages. Verify in your [Azure portal](#) that the resources were created correctly, before they're deleted. Any messages not explicitly deleted eventually become visible in the queue again for another chance to process them.

Add this code to the end of the `main` method:

Java

```
System.out.println("\nPress Enter key to receive messages and delete them
from the queue...");
System.console().readLine();

// Get messages from the queue
queueClient.receiveMessages(10).forEach(
 // "Process" the message
 receivedMessage -> {
 System.out.println("Message: " + receivedMessage.getMessageText());

 // Let the service know we're finished with
 // the message and it can be safely deleted.
 queueClient.deleteMessage(receivedMessage.getMessageId(),
 receivedMessage.getPopReceipt());
 }
);
```

When calling the `receiveMessages` method, you can optionally specify a value for `maxMessages`, which is the number of messages to retrieve from the queue. The default is 1 message and the maximum is 32 messages. You can also specify a value for `visibilityTimeout`, which hides the messages from other operations for the timeout period. The default is 30 seconds.

## Delete a queue

The following code cleans up the resources the app created by deleting the queue using the [Delete](#) method.

Add this code to the end of the `main` method:

Java

```
System.out.println("\nPress Enter key to delete the queue...");
System.console().readLine();

// Clean up
System.out.println("Deleting queue: " + queueClient.getQueueName());
queueClient.delete();

System.out.println("Done");
```

## Run the code

This app creates and adds three messages to an Azure queue. The code lists the messages in the queue, then retrieves and deletes them, before finally deleting the queue.

In your console window, navigate to your application directory, then build and run the application.

```
Console
mvn compile
```

Then, build the package.

```
Console
mvn package
```

Use the following `mvn` command to run the app.

```
Console
mvn exec:java -Dexec.mainClass="com.queues.quickstart.App" -
Dexec.cleanupDaemonThreads=false
```

The output of the app is similar to the following example:

```
Output
Azure Queue Storage client library - Java quickstart sample
Adding messages to the queue...
Peek at the messages in the queue...
Message: First message
```

```
Message: Second message
Message: Third message
```

Updating the third message in the queue...

Press Enter key to receive messages and delete them from the queue...

```
Message: First message
Message: Second message
Message: Third message has been updated
```

Press Enter key to delete the queue...

```
Deleting queue: quickstartqueues-fbf58f33-4d5a-41ac-ac0e-1a05d01c7003
Done
```

When the app pauses before receiving messages, check your storage account in the [Azure portal](#). Verify the messages are in the queue.

Press the `Enter` key to receive and delete the messages. When prompted, press the `Enter` key again to delete the queue and finish the demo.

## Next steps

In this quickstart, you learned how to create a queue and add messages to it using Java code. Then you learned to peek, retrieve, and delete messages. Finally, you learned how to delete a message queue.

For tutorials, samples, quick starts, and other documentation, visit:

### Azure for Java cloud developers

- For related code samples using deprecated Java version 8 SDKs, see [Code samples using Java version 8](#).
- For more Azure Queue Storage sample apps, see [Azure Queue Storage client library for Java - samples](#).

# Tutorial: Deploy a Spring application to Azure Spring Apps with a passwordless connection to an Azure database

Article • 04/24/2023

This article shows you how to use passwordless connections to Azure databases in Spring Boot applications deployed to Azure Spring Apps.

In this tutorial, you complete the following tasks using the Azure portal or the Azure CLI. Both methods are explained in the following procedures.

- ✓ Provision an instance of Azure Spring Apps.
- ✓ Build and deploy apps to Azure Spring Apps.
- ✓ Run apps connected to Azure databases using managed identity.

## ⓘ Note

This tutorial doesn't work for R2DBC.

## Prerequisites

- An Azure subscription. If you don't already have one, create a [free account](#) before you begin.
- [Azure CLI](#) 2.45.0 or higher required.
- The Azure Spring Apps extension. You can install the extension by using the command: `az extension add --name spring`.
- [Java Development Kit \(JDK\)](#), version 8, 11, or 17.
- A [Git](#) client.
- [cURL](#) or a similar HTTP utility to test functionality.
- MySQL command line client if you choose to run Azure Database for MySQL. You can connect to your server with Azure Cloud Shell using a popular client tool, the [mysql.exe](#) command-line tool. Alternatively, you can use the `mysql` command line in your local environment.
- [ODBC Driver 18 for SQL Server](#) if you choose to run Azure SQL Database.

## Prepare the working environment

First, set up some environment variables by using the following commands:

Bash

```
export AZ_RESOURCE_GROUP=passwordless-tutorial-rg
export AZ_DATABASE_SERVER_NAME=<YOUR_DATABASE_SERVER_NAME>
export AZ_DATABASE_NAME=demodb
export AZ_LOCATION=<YOUR_AZURE_REGION>
export AZ_SPRING_APPS_SERVICE_NAME=<YOUR_AZURE_SPRING_APPS_SERVICE_NAME>
export AZ_SPRING_APPS_APP_NAME=hellospring
export AZ_DB_ADMIN_USERNAME=<YOUR_DB_ADMIN_USERNAME>
export AZ_DB_ADMIN_PASSWORD=<YOUR_DB_ADMIN_PASSWORD>
export AZ_USER_IDENTITY_NAME=<YOUR_USER_ASSIGNED_MANAGEMED_IDENTITY_NAME>
```

Replace the placeholders with the following values, which are used throughout this article:

- <YOUR\_DATABASE\_SERVER\_NAME>: The name of your Azure Database server, which should be unique across Azure.
- <YOUR\_AZURE\_REGION>: The Azure region you want to use. You can use `eastus` by default, but we recommend that you configure a region closer to where you live. You can see the full list of available regions by using `az account list-locations`.
- <YOUR\_AZURE\_SPRING\_APPS\_SERVICE\_NAME>: The name of your Azure Spring Apps instance. The name must be between 4 and 32 characters long and can contain only lowercase letters, numbers, and hyphens. The first character of the service name must be a letter and the last character must be either a letter or a number.
- <AZ\_DB\_ADMIN\_USERNAME>: The admin username of your Azure database server.
- <AZ\_DB\_ADMIN\_PASSWORD>: The admin password of your Azure database server.
- <YOUR\_USER\_ASSIGNED\_MANAGEMED\_IDENTITY\_NAME>: The name of your user assigned managed identity server, which should be unique across Azure.

## Provision an instance of Azure Spring Apps

Use the following steps to provision an instance of Azure Spring Apps.

1. Update Azure CLI with the Azure Spring Apps extension by using the following command:

Azure CLI

```
az extension update --name spring
```

2. Sign in to the Azure CLI and choose your active subscription by using the following commands:

```
Azure CLI
```

```
az login
az account list --output table
az account set --subscription <name-or-ID-of-subscription>
```

3. Use the following commands to create a resource group to contain your Azure Spring Apps service and an instance of the Azure Spring Apps service:

```
Azure CLI
```

```
az group create \
 --name $AZ_RESOURCE_GROUP \
 --location $AZ_LOCATION
az spring create \
 --resource-group $AZ_RESOURCE_GROUP \
 --name $AZ_SPRING_APPS_SERVICE_NAME
```

## Create an Azure database instance

Use the following steps to provision an Azure Database instance.

Azure Database for MySQL

1. Create an Azure Database for MySQL server by using the following command:

```
Azure CLI
```

```
az mysql flexible-server create \
 --resource-group $AZ_RESOURCE_GROUP \
 --name $AZ_DATABASE_SERVER_NAME \
 --location $AZ_LOCATION \
 --admin-user $AZ_DB_ADMIN_USERNAME \
 --admin-password $AZ_DB_ADMIN_PASSWORD \
 --yes
```

### ⓘ Note

If you don't provide `admin-user` or `admin-password` parameters, the system will generate a default admin user or a random admin password by default.

1. Create a new database by using the following command:

```
Azure CLI
```

```
az mysql flexible-server db create \
--resource-group $AZ_RESOURCE_GROUP \
--database-name $AZ_DATABASE_NAME \
--server-name $AZ_DATABASE_SERVER_NAME
```

## Create an app with a public endpoint assigned

Use the following command to create the app.

```
Azure CLI
```

```
az spring app create \
--resource-group $AZ_RESOURCE_GROUP \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--name $AZ_SPRING_APPS_APP_NAME \
--runtime-version=Java_17
--assign-endpoint true
```

## Connect Azure Spring Apps to the Azure database

First, install the [Service Connector](#) passwordless extension for the Azure CLI:

```
Azure CLI
```

```
az extension add --name serviceconnector-passwordless --upgrade
```

Azure Database for MySQL

Then, use the following command to create a user-assigned managed identity for Microsoft Entra authentication. For more information, see [Set up Microsoft Entra authentication for Azure Database for MySQL - Flexible Server](#).

```
Azure CLI
```

```
export AZ_IDENTITY_RESOURCE_ID=$(az identity create \
--name $AZ_USER_IDENTITY_NAME \
--resource-group $AZ_RESOURCE_GROUP \
```

```
--query id \
--output tsv)
```

## ⓘ Important

After creating the user-assigned identity, ask your *Global Administrator* or *Privileged Role Administrator* to grant the following permissions for this identity: `User.Read.All`, `GroupMember.Read.All`, and `Application.Read.ALL`. For more information, see the [Permissions](#) section of [Active Directory authentication](#).

Next, use the following command to create a passwordless connection to the database.

### Azure CLI

```
az spring connection create mysql-flexible \
 --resource-group $AZ_RESOURCE_GROUP \
 --service $AZ_SPRING_APPS_SERVICE_NAME \
 --app $AZ_SPRING_APPS_APP_NAME \
 --target-resource-group $AZ_RESOURCE_GROUP \
 --server $AZ_DATABASE_SERVER_NAME \
 --database $AZ_DATABASE_NAME \
 --system-identity mysql-identity-id=$AZ_IDENTITY_RESOURCE_ID
```

This Service Connector command does the following tasks in the background:

- Enable system-assigned managed identity for the app `$AZ_SPRING_APPS_APP_NAME` hosted by Azure Spring Apps.
- Set the Microsoft Entra admin to the current signed-in user.
- Add a database user named `$AZ_SPRING_APPS_SERVICE_NAME/apps/$AZ_SPRING_APPS_APP_NAME` for the managed identity created in step 1 and grant all privileges of the database `$AZ_DATABASE_NAME` to this user.
- Add two configurations to the app `$AZ_SPRING_APPS_APP_NAME`: `spring.datasource.url` and `spring.datasource.username`.

## ⓘ Note

If you see the error message `The subscription is not registered to use Microsoft.ServiceLinker`, run the command `az provider register --namespace Microsoft.ServiceLinker` to register the Service Connector resource provider, then run the connection command again.

## Build and deploy the app

The following steps describe how to download, configure, build, and deploy the sample application.

1. Use the following command to clone the sample code repository:

```
Azure Database for MySQL
Bash
git clone https://github.com/Azure-Samples/quickstart-spring-data-jdbc-mysql passwordless-sample
```

2. Add the following dependency to your `pom.xml` file:

```
Azure Database for MySQL
XML
<dependency>
 <groupId>com.azure.spring</groupId>
 <artifactId>spring-cloud-azure-starter-jdbc-mysql</artifactId>
</dependency>
```

This dependency adds support for the Spring Cloud Azure starter.

 **Note**

For more information about how to manage Spring Cloud Azure library versions by using a bill of materials (BOM), see the [Getting started](#) section of the [Spring Cloud Azure developer guide](#).

3. Use the following command to update the *application.properties* file:

Azure Database for MySQL

Bash

```
cat << EOF > passwordless-
sample/src/main/resources/application.properties

logging.level.org.springframework.jdbc.core=DEBUG
spring.datasource.azure.passwordless-enabled=true
spring.sql.init.mode=always

EOF
```

4. Use the following commands to build the project using Maven:

Bash

```
cd passwordless-sample
./mvnw clean package -DskipTests
```

5. Use the following command to deploy the *target/demo-0.0.1-SNAPSHOT.jar* file for the app:

Azure CLI

```
az spring app deploy \
--name $AZ_SPRING_APPS_APP_NAME \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--resource-group $AZ_RESOURCE_GROUP \
--artifact-path target/demo-0.0.1-SNAPSHOT.jar
```

6. Query the app status after deployment by using the following command:

Azure CLI

```
az spring app list \
--service $AZ_SPRING_APPS_SERVICE_NAME \
--resource-group $AZ_RESOURCE_GROUP \
--output table
```

You should see output similar to the following example.

Name	Location	ResourceGroup	Production	Deployment
Public Url			Provisioning	
Status	CPU	Memory	Running Instance	Registered Instance
Persistent Storage				
<app name>	eastus	<resource group>	default	
Succeeded	1	2	1/1	0/1
-				

## Test the application

To test the application, you can use cURL. First, create a new "todo" item in the database by using the following command:

Bash

```
curl --header "Content-Type: application/json" \
--request POST \
--data '{"description":"configuration","details":"congratulations, you
have set up JDBC correctly!","done": "true"}' \
https://${AZ_SPRING_APPS_SERVICE_NAME}-
hellospring.azuremicroservices.io
```

This command returns the created item, as shown in the following example:

JSON

```
{"id":1,"description":"configuration","details":"congratulations, you have
set up JDBC correctly!","done":true}
```

Next, retrieve the data by using the following cURL request:

Bash

```
curl https://${AZ_SPRING_APPS_SERVICE_NAME}-
hellospring.azuremicroservices.io
```

This command returns the list of "todo" items, including the item you've created, as shown in the following example:

JSON

```
[{"id":1,"description":"configuration","details":"congratulations, you have
set up JDBC correctly!","done":true}]
```

## Clean up resources

To clean up all resources used during this tutorial, delete the resource group by using the following command:

Azure CLI

```
az group delete \
 --name $AZ_RESOURCE_GROUP \
 --yes
```

## Next steps

- [Spring Cloud Azure documentation](#)

# Use a managed identity to connect Azure SQL Database to an app deployed to Azure Spring Apps

Article • 01/30/2024

## ⓘ Note

Azure Spring Apps is the new name for the Azure Spring Cloud service. Although the service has a new name, you'll see the old name in some places for a while as we work to update assets such as screenshots, videos, and diagrams.

This article applies to:  Java  C#

This article applies to:  Basic/Standard  Enterprise

This article shows you how to create a managed identity for an app deployed to Azure Spring Apps and use it to access Azure SQL Database.

[Azure SQL Database](#) is the intelligent, scalable, relational database service built for the cloud. It's always up to date, with AI-powered and automated features that optimize performance and durability. Serverless compute and Hyperscale storage options automatically scale resources on demand, so you can focus on building new applications without worrying about storage size or resource management.

## Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Azure CLI](#) version 2.45.0 or higher.
- Follow the [Spring Data JPA tutorial](#) to provision an Azure SQL Database and get it work with a Java app locally.
- Follow the [Azure Spring Apps system-assigned managed identity tutorial](#) to provision an app in Azure Spring Apps with managed identity enabled.

## Connect to Azure SQL Database with a managed identity

You can connect your application to an Azure SQL Database with a managed identity by following manual steps or using [Service Connector](#).

Manual configuration

## Grant permission to the managed identity

Connect to your SQL server and run the following SQL query:

SQL

```
CREATE USER [<managed-identity-name>] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [<managed-identity-name>];
ALTER ROLE db_datawriter ADD MEMBER [<managed-identity-name>];
ALTER ROLE db_ddladmin ADD MEMBER [<managed-identity-name>];
GO
```

The value of the `<managed-identity-name>` placeholder follows the rule `<service-instance-name>/apps/<app-name>`; for example: `myspringcloud/apps/sqldemo`. You can also use the following command to query the managed identity name with Azure CLI:

Azure CLI

```
az ad sp show --id <identity-object-ID> --query displayName
```

## Configure your Java app to use a managed identity

Open the `src/main/resources/application.properties` file, then add

`Authentication=ActiveDirectoryMSI;` at the end of the `spring.datasource.url` line, as shown in the following example. Be sure to use the correct value for the `$AZ_DATABASE_NAME` variable.

properties

```
spring.datasource.url=jdbc:sqlserver://$AZ_DATABASE_NAME.database.windows.net:1433;database=demo;encrypt=true;trustServerCertificate=false;hostNameInCertificate=*.database.windows.net;loginTimeout=30;Authentication=ActiveDirectoryMSI;
```

# Build and deploy the app to Azure Spring Apps

Rebuild the app and deploy it to the Azure Spring Apps provisioned in the second bullet point under Prerequisites. You now have a Spring Boot application authenticated by a managed identity that uses JPA to store and retrieve data from an Azure SQL Database in Azure Spring Apps.

## Next steps

- [How to access Storage blob with managed identity in Azure Spring Apps ↗](#)
- [How to enable system-assigned managed identity for applications in Azure Spring Apps](#)
- [What are managed identities for Azure resources?](#)
- [Authenticate Azure Spring Apps with Key Vault in GitHub Actions](#)

# Connect an Azure Database for MySQL instance to your application in Azure Spring Apps

Article • 01/30/2024

## ⓘ Note

Azure Spring Apps is the new name for the Azure Spring Cloud service. Although the service has a new name, you'll see the old name in some places for a while as we work to update assets such as screenshots, videos, and diagrams.

This article applies to: Java C#

This article applies to: Basic/Standard Enterprise

With Azure Spring Apps, you can connect selected Azure services to your applications automatically, instead of having to configure your Spring Boot application manually. This article shows you how to connect your application to your Azure Database for MySQL instance.

## Prerequisites

- An application deployed to Azure Spring Apps. For more information, see [Quickstart: Deploy your first application to Azure Spring Apps](#).
- An Azure Database for MySQL Flexible Server instance.
- [Azure CLI](#) version 2.45.0 or higher.

## Prepare your project

Java

1. In your project's *pom.xml* file, add the following dependency:

XML

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
</dependency>
<dependency>
 <groupId>com.azure.spring</groupId>
 <artifactId>spring-cloud-azure-starter-jdbc-mysql</artifactId>
</dependency>
```

2. In the `application.properties` file, remove any `spring.datasource.*` properties.
3. Update the current app by running `az spring app deploy`, or create a new deployment for this change by running `az spring app deployment create`.

## Connect your app to the Azure Database for MySQL instance

Service Connector

### ① Note

By default, Service Connectors are created at the application level. To override the connections, you can create other connections again in the deployments.

Follow these steps to configure your Spring app to connect to an Azure Database for MySQL Flexible Server with a system-assigned managed identity.

1. Use the following command to install the Service Connector passwordless extension for the Azure CLI.

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

2. Then, use the following command to create a user-assigned managed identity for Microsoft Entra authentication. Be sure to replace the variables in the example with actual values. For more information, see [Set up Microsoft Entra authentication for Azure Database for MySQL - Flexible Server](#).

Azure CLI

```
export AZ_IDENTITY_RESOURCE_ID=$(az identity create \
--name $AZURE_USER_IDENTITY_NAME \
--resource-group $AZURE_IDENTITY_RESOURCE_GROUP \
```

```
--query id \
--output tsv)
```

3. Run the `az spring connection create` command, as shown in the following example. Be sure to replace the variables in the example with actual values.

Azure CLI

```
az spring connection create mysql-flexible \
--resource-group $AZURE_SPRING_APPS_RESOURCE_GROUP \
--service $AZURE_SPRING_APPS_SERVICE_INSTANCE_NAME \
--app $APP_NAME \
--target-resource-group $MYSQL_RESOURCE_GROUP \
--server $MYSQL_SERVER_NAME \
--database $DATABASE_NAME \
--system-identity mysql-identity-id=$AZ_IDENTITY_RESOURCE_ID
```

## Next steps

In this article, you learned how to connect an application in Azure Spring Apps to an Azure Database for MySQL instance. To learn more about connecting services to an application, see [Connect an Azure Cosmos DB database to an application in Azure Spring Apps](#).

# Bind an Azure Database for PostgreSQL to your application in Azure Spring Apps

Article • 01/30/2024

## ⓘ Note

Azure Spring Apps is the new name for the Azure Spring Cloud service. Although the service has a new name, you'll see the old name in some places for a while as we work to update assets such as screenshots, videos, and diagrams.

This article applies to:  Java  C#

This article applies to:  Basic/Standard  Enterprise

With Azure Spring Apps, you can bind select Azure services to your applications automatically, instead of having to configure your Spring Boot application manually. This article shows you how to bind your application to your Azure Database for PostgreSQL instance.

In this article, we include two authentication methods: Microsoft Entra authentication and PostgreSQL authentication. The Passwordless tab shows the Microsoft Entra authentication and the Password tab shows the PostgreSQL authentication.

Microsoft Entra authentication is a mechanism for connecting to Azure Database for PostgreSQL using identities defined in Microsoft Entra ID. With Microsoft Entra authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

PostgreSQL authentication uses accounts stored in PostgreSQL. If you choose to use passwords as credentials for the accounts, these credentials are stored in the user table. Because these passwords are stored in PostgreSQL, you need to manage the rotation of the passwords by yourself.

## Prerequisites

- An application deployed to Azure Spring Apps. For more information, see [Quickstart: Deploy your first application to Azure Spring Apps](#).
- An Azure Database for PostgreSQL Single Server instance.

- Azure CLI version 2.45.0 or higher.

## Prepare your project

Java

Use the following steps to prepare your project.

1. In your project's *pom.xml* file, add the following dependency:

XML

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
 <groupId>com.azure.spring</groupId>
 <artifactId>spring-cloud-azure-starter-jdbc-
postgresql</artifactId>
</dependency>
```

2. In the *application.properties* file, remove any `spring.datasource.*` properties.
3. Update the current app by running `az spring app deploy`, or create a new deployment for this change by running `az spring app deployment create`.

## Bind your app to the Azure Database for PostgreSQL instance

### Note

Service Connectors are created at the deployment level. So if another deployment is created, you need to create the connections again.

Passwordless

1. Install the Service Connector passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

2. Configure Azure Spring Apps to connect to the PostgreSQL Database with a system-assigned managed identity using the `az spring connection create` command.

Azure CLI

```
az spring connection create postgres \
--resource-group $AZ_SPRING_APPS_RESOURCE_GROUP \
--service $AZ_SPRING_APPS_SERVICE_INSTANCE_NAME \
--app $APP_NAME \
--deployment $DEPLOYMENT_NAME \
--target-resource-group $POSTGRES_RESOURCE_GROUP \
--server $POSTGRES_SERVER_NAME \
--database $DATABASE_NAME \
--system-identity
```

## Next steps

In this article, you learned how to bind an application in Azure Spring Apps to an Azure Database for PostgreSQL instance. To learn more about binding services to an application, see [Bind an Azure Cosmos DB database to an application in Azure Spring Apps](#).

# Tutorial: Connect to a MySQL Database from Java JBoss EAP App Service with passwordless connection

Article • 10/11/2023

Azure App Service provides a highly scalable, self-patching web hosting service in Azure. It also provides a [managed identity](#) for your app, which is a turn-key solution for securing access to [Azure Database for MySQL](#) and other Azure services. Managed identities in App Service make your app more secure by eliminating secrets from your app, such as credentials in the environment variables. In this tutorial, you learn how to:

- ✓ Create a MySQL database.
- ✓ Deploy a sample JBoss EAP app to Azure App Service using a WAR package.
- ✓ Configure a Spring Boot web application to use Microsoft Entra authentication with MySQL Database.
- ✓ Connect to MySQL Database with Managed Identity using Service Connector.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

## Prerequisites

- [Git](#)
- [Java JDK](#)
- [Maven](#)
- [Azure CLI](#) version 2.46.0 or higher.
- [Azure CLI serviceconnector-passwordless extension](#) version 0.2.2 or higher.
- [jq](#)

## Clone the sample app and prepare the repo

Run the following commands in your terminal to clone the sample repo and set up the sample app environment.

Bash

```
git clone https://github.com/Azure-Samples/Passwordless-Connections-for-Java-Apps
cd Passwordless-Connections-for-Java-Apps/JakartaEE/jboss-eap/
```

# Create an Azure Database for MySQL

Follow these steps to create an Azure Database for MySQL in your subscription. The Spring Boot app connects to this database and store its data when running, persisting the application state no matter where you run the application.

1. Sign into the Azure CLI, and optionally set your subscription if you have more than one connected to your login credentials.

Azure CLI

```
az login
az account set --subscription <subscription-ID>
```

2. Create an Azure Resource Group, noting the resource group name.

Azure CLI

```
export RESOURCE_GROUP=<resource-group-name>
export LOCATION=eastus

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

3. Create an Azure Database for MySQL server. The server is created with an administrator account, but it isn't used because we're going to use the Microsoft Entra admin account to perform administrative tasks.

Azure CLI

```
export MYSQL_ADMIN_USER=azureuser
MySQL admin access rights won't be used because Azure AD
authentication is leveraged to administer the database.
export MYSQL_ADMIN_PASSWORD=<admin-password>
export MYSQL_HOST=<mysql-host-name>

Create a MySQL server.
az mysql flexible-server create \
 --name $MYSQL_HOST \
 --resource-group $RESOURCE_GROUP \
 --location $LOCATION \
 --admin-user $MYSQL_ADMIN_USER \
 --admin-password $MYSQL_ADMIN_PASSWORD \
 --public-access 0.0.0.0 \
 --tier Burstable \
 --sku-name Standard_B1ms \
 --storage-size 32
```

4. Create a database for the application.

Azure CLI

```
export DATABASE_NAME=checklist

az mysql flexible-server db create \
 --resource-group $RESOURCE_GROUP \
 --server-name $MYSQL_HOST \
 --database-name $DATABASE_NAME
```

## Create an App Service

Create an Azure App Service resource on Linux. JBoss EAP requires Premium SKU.

Azure CLI

```
export APPSERVICE_PLAN=<app-service-plan>
export APPSERVICE_NAME=<app-service-name>
Create an App Service plan
az appservice plan create \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_PLAN \
 --location $LOCATION \
 --sku P1V3 \
 --is-linux

Create an App Service resource.
az webapp create \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 --plan $APPSERVICE_PLAN \
 --runtime "JBOSSEAP:7-java8"
```

## Connect the MySQL database with identity connectivity

Next, connect the database using [Service Connector](#).

Install the Service Connector passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

Then, use the following command to create a user-assigned managed identity for Microsoft Entra authentication. For more information, see [Set up Microsoft Entra authentication for Azure Database for MySQL - Flexible Server](#).

#### Azure CLI

```
export USER_IDENTITY_NAME=<your-user-assigned-managed-identity-name>
export IDENTITY_RESOURCE_ID=$(az identity create \
 --name $USER_IDENTITY_NAME \
 --resource-group $RESOURCE_GROUP \
 --query id \
 --output tsv)
```

#### ⓘ Important

After creating the user-assigned identity, ask your *Global Administrator* or *Privileged Role Administrator* to grant the following permissions for this identity: `User.Read.All`, `GroupMember.Read.All`, and `Application.Read.ALL`. For more information, see the [Permissions](#) section of [Active Directory authentication](#).

Then, connect your app to a MySQL database with a system-assigned managed identity using Service Connector. To make this connection, run the [az webapp connection create](#) command.

#### Azure CLI

```
az webapp connection create mysql-flexible \
 --resource-group $RESOURCE_GROUP \
 --name $APPNAME \
 --target-resource-group $RESOURCE_GROUP \
 --server $MYSQL_HOST \
 --database $DATABASE_NAME \
 --system-identity mysql-identity-id=$IDENTITY_RESOURCE_ID \
 --client-type java
```

This Service Connector command does the following tasks in the background:

- Enable system-assigned managed identity for the app `$APPNAME` hosted by Azure App Service.
- Set the Microsoft Entra admin to the current signed-in user.
- Add a database user for the system-assigned managed identity in step 1 and grant all privileges of the database `$DATABASE_NAME` to this user. You can get the user name from the connection string in the output from the previous command.

- Add a connection string to App Settings in the app named

AZURE\_MYSQL\_CONNECTIONSTRING.

 Note

If you see the error message `The subscription is not registered to use Microsoft.ServiceLinker`, run the command `az provider register --namespace Microsoft.ServiceLinker` to register the Service Connector resource provider, then run the connection command again.

## Deploy the application

Follow these steps to prepare data in a database and deploy the application.

### Create Database schema

1. Open a firewall to allow connection from your current IP address.

Azure CLI

```
Create a temporary firewall rule to allow connections from your
current machine to the MySQL server
export MY_IP=$(curl http://whatismyip.akamai.com)
az mysql flexible-server firewall-rule create \
--resource-group $RESOURCE_GROUP \
--name $MYSQL_HOST \
--rule-name AllowCurrentMachineToConnect \
--start-ip-address ${MY_IP} \
--end-ip-address ${MY_IP}
```

2. Connect to the database and create tables.

Azure CLI

```
export DATABASE_FQDN=${MYSQL_HOST}.mysql.database.azure.com
export CURRENT_USER=$(az account show --query user.name --output tsv)
export RDBMS_ACCESS_TOKEN=$(az account get-access-token \
--resource-type oss-rdbms \
--output tsv \
--query accessToken)
mysql -h "${DATABASE_FQDN}" --user "${CURRENT_USER}" --enable-
cleartext-plugin --password="$RDBMS_ACCESS_TOKEN" < azure/init-db.sql
```

3. Remove the temporary firewall rule.

## Azure CLI

```
az mysql flexible-server firewall-rule delete \
--resource-group $RESOURCE_GROUP \
--name $MYSQL_HOST \
--rule-name AllowCurrentMachineToConnect
```

## Deploy the application

1. Update the connection string in App Settings.

Get the connection string generated by Service Connector and add passwordless authentication plugin. This connection string is referenced in the startup script.

## Azure CLI

```
export PASSWORDLESS_URL=$(\
 az webapp config appsettings list \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 | jq -c '.[]' \
 | select (.name == "AZURE_MYSQL_CONNECTIONSTRING") \
 | .value' \
 | sed 's//g')
Create a new environment variable with the connection string
including the passwordless authentication plugin
export
PASSWORDLESS_URL=${PASSWORDLESS_URL}'&defaultAuthenticationPlugin=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin&authenticationPlugins=com.azure.identity.extensions.jdbc.mysql.AzureMysqlAuthenticationPlugin'
az webapp config appsettings set \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 --settings
"AZURE_MYSQL_CONNECTIONSTRING_PASSWORDLESS=${PASSWORDLESS_URL}"
```

2. The sample app contains a *pom.xml* file that can generate the WAR file. Run the following command to build the app.

## Bash

```
mvn clean package -DskipTests
```

3. Deploy the WAR and the startup script to the app service.

## Azure CLI

```
az webapp deploy \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 --src-path target/ROOT.war \
 --type war
az webapp deploy \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 --src-path src/main/webapp/WEB-INF/createMySQLDataSource.sh \
 --type startup
```

## Test sample web app

Run the following command to test the application.

Bash

```
export WEBAPP_URL=$(az webapp show \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 --query defaultHostName \
 --output tsv)

Create a list
curl -X POST -H "Content-Type: application/json" -d '{"name": "list1", "date": "2022-03-21T00:00:00", "description": "Sample checklist"}' \
https://${WEBAPP_URL}/checklist

Create few items on the list 1
curl -X POST -H "Content-Type: application/json" -d '{"description": "item 1"}' https://${WEBAPP_URL}/checklist/1/item
curl -X POST -H "Content-Type: application/json" -d '{"description": "item 2"}' https://${WEBAPP_URL}/checklist/1/item
curl -X POST -H "Content-Type: application/json" -d '{"description": "item 3"}' https://${WEBAPP_URL}/checklist/1/item

Get all lists
curl https://${WEBAPP_URL}/checklist

Get list 1
curl https://${WEBAPP_URL}/checklist/1
```

## 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 App Service on Linux in the developer guide.

[Java in App Service Linux dev guide](#)

# Tutorial: Connect to a PostgreSQL Database from Java Tomcat App Service without secrets using a managed identity

Article • 10/11/2023

Azure App Service provides a highly scalable, self-patching web hosting service in Azure. It also 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 App Service make your app more secure by eliminating secrets from your app, such as credentials in the environment variables. In this tutorial, you learn how to:

- ✓ Create a PostgreSQL database.
- ✓ Deploy the sample app to Azure App Service on Tomcat using WAR packaging.
- ✓ Configure a Tomcat web application to use Microsoft Entra authentication with PostgreSQL Database.
- ✓ Connect to PostgreSQL Database with Managed Identity using Service Connector.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

## Prerequisites

- [Git](#)
- Java JDK
- Maven
- [Azure CLI](#) version 2.45.0 or higher.

## Clone the sample app and prepare the repo

Run the following commands in your terminal to clone the sample repo and set up the sample app environment.

Bash

```
git clone https://github.com/Azure-Samples/Passwordless-Connections-for-Java-Apps
cd Passwordless-Connections-for-Java-Apps/Tomcat/
```

# Create an Azure Database for PostgreSQL

Follow these steps to create an Azure Database for Postgres in your subscription. The Tomcat app connects to this database and store its data when running, persisting the application state no matter where you run the application.

1. Sign into the Azure CLI, and optionally set your subscription if you have more than one connected to your login credentials.

Azure CLI

```
az login
az account set --subscription <subscription-ID>
```

2. Create an Azure Resource Group, noting the resource group name.

Azure CLI

```
export RESOURCE_GROUP=<resource-group-name>
export LOCATION=eastus

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

3. Create an Azure Database for PostgreSQL server. The server is created with an administrator account, but it isn't used because we're going to use the Microsoft Entra admin account to perform administrative tasks.

Flexible Server

Azure CLI

```
export POSTGRESQL_ADMIN_USER=azureuser
PostgreSQL admin access rights won't be used because Azure AD
authentication is leveraged to administer the database.
export POSTGRESQL_ADMIN_PASSWORD=<admin-password>
export POSTGRESQL_HOST=<postgresql-host-name>

Create a PostgreSQL server.
az postgres flexible-server create \
 --resource-group $RESOURCE_GROUP \
 --name $POSTGRESQL_HOST \
 --location $LOCATION \
 --admin-user $POSTGRESQL_ADMIN_USER \
 --admin-password $POSTGRESQL_ADMIN_PASSWORD \
 --ssl-encryption-enabled true
```

```
--public-access 0.0.0.0 \
--sku-name Standard_D2s_v3
```

4. Create a database for the application.

Flexible Server

Azure CLI

```
export DATABASE_NAME=checklist

az postgres flexible-server db create \
--resource-group $RESOURCE_GROUP \
--server-name $POSTGRESQL_HOST \
--database-name $DATABASE_NAME
```

## Deploy the application to App Service

Follow these steps to build a WAR file and deploy to Azure App Service on Tomcat using a WAR packaging.

1. The sample app contains a *pom.xml* file that can generate the WAR file. Run the following command to build the app.

Bash

```
mvn clean package -f pom.xml
```

2. Create an Azure App Service resource on Linux using Tomcat 9.0.

Azure CLI

```
export APPSERVICE_PLAN=<app-service-plan>
export APPSERVICE_NAME=<app-service-name>
Create an App Service plan
az appservice plan create \
--resource-group $RESOURCE_GROUP \
--name $APPSERVICE_PLAN \
--location $LOCATION \
--sku B1 \
--is-linux

Create an App Service resource.
az webapp create \
--resource-group $RESOURCE_GROUP \
```

```
--name $APPSERVICE_NAME \
--plan $APPSERVICE_PLAN \
--runtime "TOMCAT:10.0-javal1"
```

### 3. Deploy the WAR package to App Service.

Azure CLI

```
az webapp deploy \
--resource-group $RESOURCE_GROUP \
--name $APPSERVICE_NAME \
--src-path target/app.war \
--type war
```

## Connect the Postgres database with identity connectivity

Next, connect the database using [Service Connector](#).

Install the Service Connector passwordless extension for the Azure CLI:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
```

Then, connect your app to a Postgres database with a system-assigned managed identity using Service Connector.

Flexible Server

To make this connection, run the `az webapp connection create` command.

Azure CLI

```
az webapp connection create postgres-flexible \
--resource-group $RESOURCE_GROUP \
--name $APPSERVICE_NAME \
--target-resource-group $RESOURCE_GROUP \
--server $POSTGRESQL_HOST \
--database $DATABASE_NAME \
--system-identity \
--client-type java
```

This command creates a connection between your web app and your PostgreSQL server, and manages authentication through a system-assigned managed identity.

Next, update App Settings and add plugin in connection string

Azure CLI

```
export AZURE_POSTGRESQL_CONNECTIONSTRING=$(\
 az webapp config appsettings list \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 | jq -c -r '.[]' \
 | select (.name == "AZURE_POSTGRESQL_CONNECTIONSTRING") \
 | .value')

az webapp config appsettings set \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 --settings 'CATALINA_OPTS=-
DdbUrl=""${AZURE_POSTGRESQL_CONNECTIONSTRING}"&authenticationPluginClassName=com.azure.identity.extensions.jdbc.postgresql.AzurePostgresqlAuthenticationPlugin'"
```

## Test the sample web app

Run the following command to test the application.

Bash

```
export WEBAPP_URL=$(az webapp show \
 --resource-group $RESOURCE_GROUP \
 --name $APPSERVICE_NAME \
 --query defaultHostName \
 --output tsv)

Create a list
curl -X POST -H "Content-Type: application/json" -d '{"name": "list1", "date": "2022-03-21T00:00:00", "description": "Sample checklist"}' \
https://$WEBAPP_URL/checklist

Create few items on the list 1
curl -X POST -H "Content-Type: application/json" -d '{"description": "item 1"}' https://$WEBAPP_URL/checklist/1/item
curl -X POST -H "Content-Type: application/json" -d '{"description": "item 2"}' https://$WEBAPP_URL/checklist/1/item
curl -X POST -H "Content-Type: application/json" -d '{"description": "item 3"}' https://$WEBAPP_URL/checklist/1/item

Get all lists
curl https://$WEBAPP_URL/checklist
```

```
Get list 1
curl https://${WEBAPP_URL}/checklist/1
```

## 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 App Service on Linux in the developer guide.

[Java in App Service Linux dev guide](#)

Learn how to secure your app with a custom domain and certificate.

[Secure with custom domain and certificate](#)

# Tutorial: Connect to PostgreSQL Database from a Java Quarkus Container App without secrets using a managed identity

Article • 10/12/2023

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 Microsoft Entra ID 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}.postgres.database.azure.com:5432/${DBNAME}?\
authenticationPluginClassName=com.azure.identity.providers.postgresql.AzureIdentityPostgresqlAuthenticationPlugin\
&sslmode=require\
&azure.clientId=${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}.postgres.database.azure.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](#)

# Use Azure OpenAI without keys

Article • 05/20/2024

Application requests to most Azure services must be authenticated with keys or [passwordless connections](#). Developers must be diligent to never expose the keys in an unsecure location. Anyone who gains access to the key is able to authenticate to the service. Passwordless authentication offers improved management and security benefits over the account key because there's no key (or connection string) to store.

Passwordless connections are enabled with the following steps:

- Configure your authentication.
- Set environment variables, as needed.
- Use an Azure Identity library credential type to create an Azure OpenAI client object.

## Authentication

Authentication to Microsoft Entra ID is required to use the Azure client libraries.

Authentication differs based on the environment in which the app is running:

- [Local development](#)
- [Azure](#)

## Authenticate for local development

Java

Select a tool for [authentication during local development](#).

## Authenticate for Azure-hosted environments

Java

Learn about how to manage the [DefaultAzureCredential](#) for applications deployed to Azure.

# Configure roles for authorization

1. Find the [role](#) for your usage of Azure OpenAI. Depending on how you intend to set that role, you'll need either the name or ID.

[+] [Expand table](#)

Role name	Role ID
For Azure CLI or Azure PowerShell, you can use role name. For Bicep, you need the role ID.	

2. For many Azure OpenAI chat completion use cases, the following role and ID are suggested.

[+] [Expand table](#)

Role name	Role ID
Cognitive Services OpenAI User	5e0bd9bd-7b93-4f28-af87-19fc36ad61bd

3. Select an identity type to use.

- **Personal identity:** This is your personal identity tied to your sign in to Azure.
- **Managed identity:** This is an identity managed by and created for use on Azure. For [managed identity](#), create a [user-assigned managed identity](#). When you create the managed identity, you need the `Client ID`, also known as the `app ID`.

4. To find your personal identity, use one of the following commands. Use the ID as the `<identity-id>` in the next step.

Azure CLI

For local development, to get your own identity ID, use the following command. You need to sign in with `az login` before using this command.

Azure CLI

```
az ad signed-in-user show \
--query id -o tsv
```

5. Assign the role-based access control (RBAC) role to the identity for the resource group.

## Azure CLI

To grant your identity permissions to your resource through RBAC, assign a role using the Azure CLI command [az role assignment create](#).

### Azure CLI

```
az role assignment create \
 --role "Cognitive Services OpenAI User" \
 --assignee "<identity-id>" \
 --scope "/subscriptions/<subscription-
 id>/resourceGroups/<resource-group-name>"
```

Where applicable, replace `<identity-id>`, `<subscription-id>`, and `<resource-group-name>` with your actual values.

## Configure environment variables

To connect to Azure OpenAI, your code needs to know your resource endpoint, and *may* need additional environment variables.

1. Create an environment variable for your Azure OpenAI endpoint.

- `AZURE_OPENAI_ENDPOINT`: This URL is the access point for your Azure OpenAI resource.

2. Create environment variables based on the location in which your app runs:

[+] [Expand table](#)

Location	Identity	Description
Local	Personal	For local runtimes with your <a href="#">personal identity</a> , <a href="#">sign in</a> to create your credential with a tool.
Azure cloud	User-assigned managed identity	Create an <code>AZURE_CLIENT_ID</code> environment variable containing the client ID of the user-assigned managed identity to authenticate as.

## Install Azure Identity client library

### Java

Install the Java [Azure Identity client library](#) with the following POM file:

XML

```
<dependencyManagement>
 <dependencies>
 <dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-identity</artifactId>
 <version>1.10.0</version>
 <type>pom</type>
 <scope>import</scope>
 </dependency>
 </dependencies>
</dependencyManagement>
```

## Use DefaultAzureCredential

The Azure Identity library's `DefaultAzureCredential` allows the customer to run the same code in the local development environment and in the Azure Cloud.

Java

For more information on `DefaultAzureCredential` for Java, see [Azure Identity client library for Java](#).

Java

```
import com.azure.identity.DefaultAzureCredentialBuilder;
import com.azure.ai.openai.OpenAIclient;
import com.azure.ai.openai.OpenAIclientBuilder;

String endpoint = System.getenv("AZURE_OPENAI_ENDPOINT");

DefaultAzureCredential credential = new
DefaultAzureCredentialBuilder().build();
OpenAIclient client = new OpenAIclientBuilder()
 .credential(credential)
 .endpoint(endpoint)
 .buildClient();
```

## Additional resources

- Passwordless connections developer guide
- 

## Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

# Connect to and query Azure SQL Database using Node.js and mssql npm package

Article • 09/29/2023

Applies to:  Azure SQL Database

This quickstart describes how to connect an application to a database in Azure SQL Database and perform queries using Node.js and mssql. This quickstart follows the recommended passwordless approach to connect to the database.

## Passwordless connections for developers

Passwordless connections offer a more secure mechanism for accessing Azure resources. The following high-level steps are used to connect to Azure SQL Database using passwordless connections in this article:

- Prepare your environment for password-free authentication.
  - For a local environment: Your personal identity is used. This identity can be pulled from an IDE, CLI, or other local development tools.
  - For a cloud environment: A [managed identity](#) is used.
- Authenticate in the environment using the `DefaultAzureCredential` from the Azure Identity library to obtain a verified credential.
- Use the verified credential to create Azure SDK client objects for resource access.

You can learn more about passwordless connections on the [passwordless hub](#).

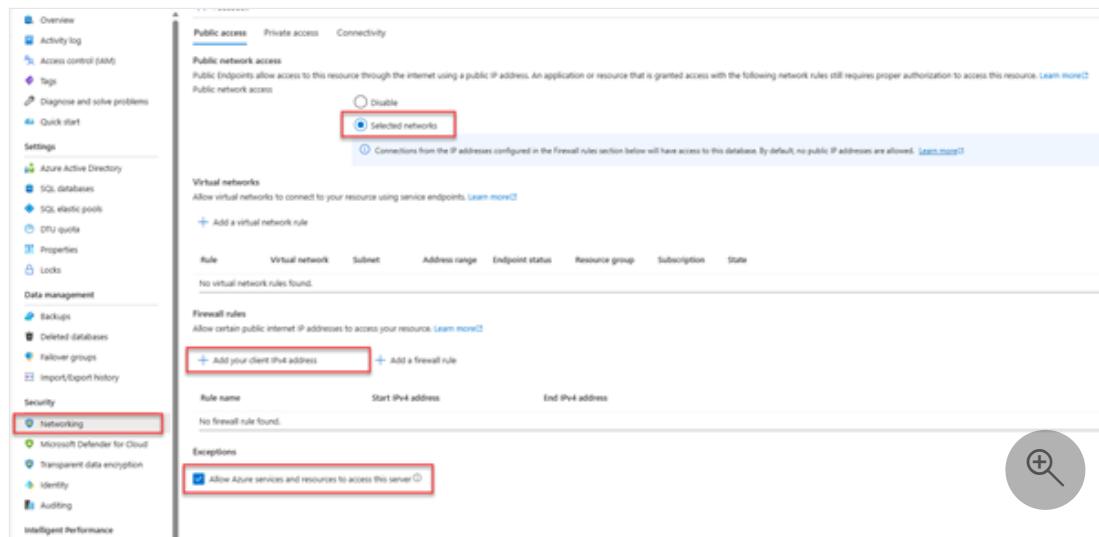
## Prerequisites

- An [Azure subscription](#) ↗
- A database in Azure SQL Database configured for authentication with Microsoft Entra ID ([formerly Azure Active Directory](#)). You can create one using the [Create database quickstart](#).
- Bash-enabled shell
- [Node.js LTS](#) ↗
- [Visual Studio Code](#) ↗
- [Visual Studio Code App Service extension](#) ↗
- The latest version of the [Azure CLI](#)

# Configure the database server

Secure, passwordless connections to Azure SQL Database require certain database configurations. Verify the following settings on your [logical server in Azure](#) to properly connect to Azure SQL Database in both local and hosted environments:

1. For local development connections, make sure your logical server is configured to allow your local machine IP address and other Azure services to connect:
  - Navigate to the **Networking** page of your server.
  - Toggle the **Selected networks** radio button to show additional configuration options.
  - Select **Add your client IPv4 address(xx.xx.xx.xx)** to add a firewall rule that will enable connections from your local machine IPv4 address. Alternatively, you can also select **+ Add a firewall rule** to enter a specific IP address of your choice.
  - Make sure the **Allow Azure services and resources to access this server** checkbox is selected.



## ⚠️ Warning

Enabling the **Allow Azure services and resources to access this server** setting is not a recommended security practice for production scenarios. Real applications should implement more secure approaches, such as stronger firewall restrictions or virtual network configurations.

You can read more about database security configurations on the following resources:

- [Configure Azure SQL Database firewall rules.](#)
- [Configure a virtual network with private endpoints.](#)

2. The server must also have Microsoft Entra authentication enabled and have a Microsoft Entra admin account assigned. For local development connections, the Microsoft Entra admin account should be an account you can also log into Visual Studio or the Azure CLI with locally. You can verify whether your server has Microsoft Entra authentication enabled on the **Microsoft Entra ID** page of your logical server.

The screenshot shows the Microsoft Entra ID settings for a SQL server. The 'Microsoft Entra ID' section is highlighted with a red box. It displays the admin name 'testing123@microsoft.com' and a checked checkbox for 'Support only Microsoft Entra authentication for this server server'.

3. If you're using a personal Azure account, make sure you have [Microsoft Entra setup and configured for Azure SQL Database](#) in order to assign your account as a server admin. If you're using a corporate account, Microsoft Entra ID will most likely already be configured for you.

## Create the project

The steps in this section create a Node.js REST API.

1. Create a new directory for the project and navigate into it.
2. Initialize the project by running the following command in the terminal:

```
Bash
```

```
npm init -y
```

3. Install the required packages used in the sample code in this article:

```
Bash
```

```
npm install mssql swagger-ui-express yamljs
```

4. Install the development package used in the sample code in this article:

```
Bash
```

```
npm install --save-dev dotenv
```

5. Open the project in Visual Studio Code.

```
Bash
```

```
code .
```

6. Open the **package.json** file and add the following property and value after the *name* property to configure the project for ESM modules.

```
JSON
```

```
"type": "module",
```

## Create Express.js application code

To create the Express.js OpenAPI application, you'll create several files:

[+] Expand table

File	Description
.env.development	Local development-only environment file.
index.js	Main application file, which starts the Express.js app on port 3000.
person.js	Express.js <b>/person</b> route API file to handle CRUD operations.
openapi.js	Express.js <b>/api-docs</b> route for OpenAPI explorer UI. Root redirects to this route.
openApiSchema.yml	OpenAPI 3.0 schema file defining Person API.
config.js	Configuration file to read environment variables and construct appropriate mssql connection object.
database.js	Database class to handle Azure SQL CRUD operations using the <b>mssql</b> npm package.
./vscode/settings.json	Ignore files by glob pattern during deployment.

1. Create an **index.js** file and add the following code:

```
JavaScript

import express from 'express';
import { config } from './config.js';
import Database from './database.js';

// Import App routes
import person from './person.js';
import openapi from './openapi.js';

const port = process.env.PORT || 3000;

const app = express();

// Development only - don't do in production
// Run this to create the table in the database
if (process.env.NODE_ENV === 'development') {
 const database = new Database(config);
 database
 .executeQuery(
 `CREATE TABLE Person (id int NOT NULL IDENTITY, firstName
varchar(255), lastName varchar(255));`
)
 .then(() => {
 console.log('Table created');
 })
 .catch((err) => {
 // Table may already exist
 console.error(`Error creating table: ${err}`);
 });
}

// Connect App routes
app.use('/api-docs', openapi);
app.use('/persons', person);
app.use('*', (_, res) => {
 res.redirect('/api-docs');
});

// Start the server
app.listen(port, () => {
 console.log(`Server started on port ${port}`);
});
```

2. Create a **person.js** route file and add the following code:

```
JavaScript

import express from 'express';
import { config } from './config.js';
```

```
import Database from './database.js';

const router = express.Router();
router.use(express.json());

// Development only - don't do in production
console.log(config);

// Create database object
const database = new Database(config);

router.get('/', async (_, res) => {
 try {
 // Return a list of persons
 const persons = await database.readAll();
 console.log(`persons: ${JSON.stringify(persons)}`);
 res.status(200).json(persons);
 } catch (err) {
 res.status(500).json({ error: err?.message });
 }
});

router.post('/', async (req, res) => {
 try {
 // Create a person
 const person = req.body;
 console.log(`person: ${JSON.stringify(person)}`);
 const rowsAffected = await database.create(person);
 res.status(201).json({ rowsAffected });
 } catch (err) {
 res.status(500).json({ error: err?.message });
 }
});

router.get('/:id', async (req, res) => {
 try {
 // Get the person with the specified ID
 const personId = req.params.id;
 console.log(`personId: ${personId}`);
 if (personId) {
 const result = await database.read(personId);
 console.log(`persons: ${JSON.stringify(result)}`);
 res.status(200).json(result);
 } else {
 res.status(404);
 }
 } catch (err) {
 res.status(500).json({ error: err?.message });
 }
});

router.put('/:id', async (req, res) => {
 try {
 // Update the person with the specified ID
 const personId = req.params.id;
 const updatedPerson = req.body;
 const result = await database.update(personId, updatedPerson);
 console.log(`updatedPerson: ${JSON.stringify(updatedPerson)}`);
 res.status(200).json(result);
 } catch (err) {
 res.status(500).json({ error: err?.message });
 }
});
```

```

 console.log(`personId: ${personId}`);
 const person = req.body;

 if (personId && person) {
 delete person.id;
 console.log(`person: ${JSON.stringify(person)}`);
 const rowsAffected = await database.update(personId, person);
 res.status(200).json({ rowsAffected });
 } else {
 res.status(404);
 }
} catch (err) {
 res.status(500).json({ error: err?.message });
}
});

router.delete('/:id', async (req, res) => {
 try {
 // Delete the person with the specified ID
 const personId = req.params.id;
 console.log(`personId: ${personId}`);

 if (!personId) {
 res.status(404);
 } else {
 const rowsAffected = await database.delete(personId);
 res.status(204).json({ rowsAffected });
 }
 } catch (err) {
 res.status(500).json({ error: err?.message });
 }
});

export default router;

```

3. Create an `openapi.js` route file and add the following code for the OpenAPI UI explorer:

JavaScript

```

import express from 'express';
import { join, dirname } from 'path';
import swaggerUi from 'swagger-ui-express';
import yaml from 'yamljs';
import { fileURLToPath } from 'url';

const __dirname = dirname(fileURLToPath(import.meta.url));

const router = express.Router();
router.use(express.json());

const pathToSpec = join(__dirname, './openApiSchema.yml');

```

```

const openApiSpec = yaml.load(pathToSpec);

router.use('/', swaggerUi.serve, swaggerUi.setup(openApiSpec));

export default router;

```

4. Create a `openApiSchema.yml` schema file and add the following YAML:

```

yml

openapi: 3.0.0
info:
 version: 1.0.0
 title: Persons API
paths:
 /persons:
 get:
 summary: Get all persons
 responses:
 '200':
 description: OK
 content:
 application/json:
 schema:
 type: array
 items:
 $ref: '#/components/schemas/Person'
 post:
 summary: Create a new person
 requestBody:
 required: true
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/Person'
 responses:
 '201':
 description: Created
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/Person'
 /persons/{id}:
 parameters:
 - name: id
 in: path
 required: true
 schema:
 type: integer
 get:
 summary: Get a person by ID
 responses:
 '200':

```

```
 description: OK
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/Person'
 '404':
 description: Person not found
 put:
 summary: Update a person by ID
 requestBody:
 required: true
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/Person'
 responses:
 '200':
 description: OK
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/Person'
 '404':
 description: Person not found
 delete:
 summary: Delete a person by ID
 responses:
 '204':
 description: No Content
 '404':
 description: Person not found
 components:
 schemas:
 Person:
 type: object
 properties:
 id:
 type: integer
 readOnly: true
 firstName:
 type: string
 lastName:
 type: string
```

## Configure the `mssql` connection object

The `mssql` package implements the connection to Azure SQL Database by providing a configuration setting for an authentication type.

Passwordless (recommended)

1. In Visual Studio Code, create a **config.js** file and add the following mssql configuration code to authenticate to Azure SQL Database.

```
JavaScript

import * as dotenv from 'dotenv';
dotenv.config({ path: `./env.${process.env.NODE_ENV}`, debug: true });

const server = process.env.AZURE_SQL_SERVER;
const database = process.env.AZURE_SQL_DATABASE;
const port = parseInt(process.env.AZURE_SQL_PORT);
const type = process.env.AZURE_SQL_AUTHENTICATIONTYPE;

export const config = {
 server,
 port,
 database,
 authentication: {
 type
 },
 options: {
 encrypt: true
 }
};
```

2. Create a **.env.development** file for your local environment variables and add the following text and update with your values for <YOURSERVERNAME> and <YOURDATABASENAME>.

```
text

AZURE_SQL_SERVER=<YOURSERVERNAME>.database.windows.net
AZURE_SQL_DATABASE=<YOURDATABASENAME>
AZURE_SQL_PORT=1433
AZURE_SQL_AUTHENTICATIONTYPE=azure-active-directory-default
```

### ⓘ Note

Passwordless configuration objects are safe to commit to source control, since they do not contain any secrets such as usernames, passwords, or access keys.

3. Create a **.vscode** folder and create a **settings.json** file in the folder.

4. Add the following to ignore environment variables and dependencies during the zip deployment.

JSON

```
{
 "appService.zipIgnorePattern": ["./.env*", "node_modules{,/**}"]
}
```

## Add the code to connect to Azure SQL Database

1. Create a `database.js` file and add the following code:

JavaScript

```
import sql from 'mssql';

export default class Database {
 config = {};
 poolconnection = null;
 connected = false;

 constructor(config) {
 this.config = config;
 console.log(`Database: config: ${JSON.stringify(config)})`);
 }

 async connect() {
 try {
 console.log(`Database connecting...${this.connected}`);
 if (this.connected === false) {
 this.poolconnection = await sql.connect(this.config);
 this.connected = true;
 console.log('Database connection successful');
 } else {
 console.log('Database already connected');
 }
 } catch (error) {
 console.error(`Error connecting to database:
${JSON.stringify(error)})`);
 }
 }

 async disconnect() {
 try {
 this.poolconnection.close();
 console.log('Database connection closed');
 } catch (error) {
```

```
 console.error(`Error closing database connection: ${error}`);
 }
}

async executeQuery(query) {
 await this.connect();
 const request = this.poolconnection.request();
 const result = await request.query(query);

 return result.rowsAffected[0];
}

async create(data) {
 await this.connect();
 const request = this.poolconnection.request();

 request.input('firstName', sql.NVarChar(255), data.firstName);
 request.input('lastName', sql.NVarChar(255), data.lastName);

 const result = await request.query(
 `INSERT INTO Person (firstName, lastName) VALUES (@firstName,
@lastName)`
);

 return result.rowsAffected[0];
}

async readAll() {
 await this.connect();
 const request = this.poolconnection.request();
 const result = await request.query(`SELECT * FROM Person`);

 return result.recordsets[0];
}

async read(id) {
 await this.connect();

 const request = this.poolconnection.request();
 const result = await request
 .input('id', sql.Int, +id)
 .query(`SELECT * FROM Person WHERE id = @id`);

 return result.recordset[0];
}

async update(id, data) {
 await this.connect();

 const request = this.poolconnection.request();

 request.input('id', sql.Int, +id);
 request.input('firstName', sql.NVarChar(255), data.firstName);
 request.input('lastName', sql.NVarChar(255), data.lastName);
```

```

 const result = await request.query(
 `UPDATE Person SET firstName=@firstName, lastName=@lastName WHERE
id = @id`;
);

 return result.rowsAffected[0];
}

async delete(id) {
 await this.connect();

 const idAsNumber = Number(id);

 const request = this.poolconnection.request();
 const result = await request
 .input('id', sql.Int, idAsNumber)
 .query(`DELETE FROM Person WHERE id = @id`);

 return result.rowsAffected[0];
}
}

```

## Test the app locally

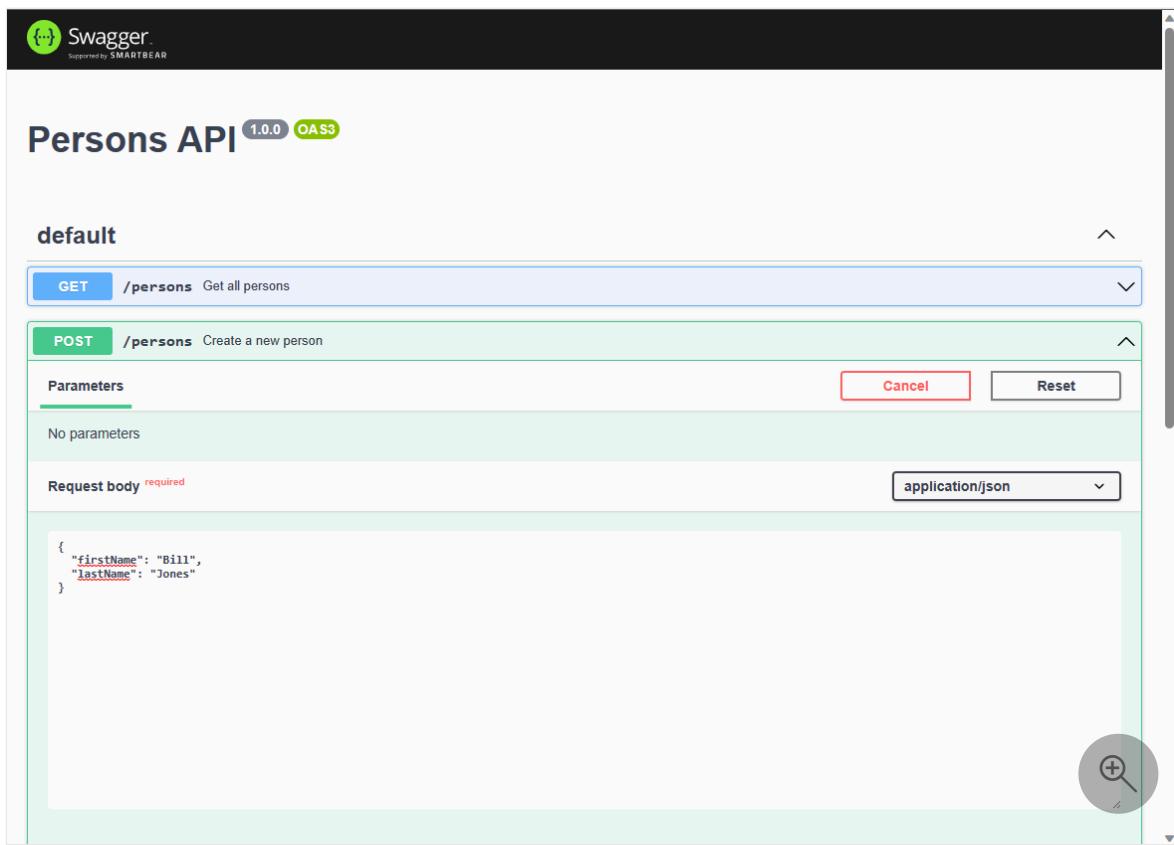
The app is ready to be tested locally. Make sure you're signed in to the Azure Cloud in Visual Studio Code with the same account you set as the admin for your database.

1. Run the application with the following command. The app starts on port 3000.

```
Bash
NODE_ENV=development node index.js
```

The **Person** table is created in the database when you run this application.

2. In a browser, navigate to the OpenAPI explorer at <http://localhost:3000>.
3. On the Swagger UI page, expand the POST method and select **Try it**.
4. Modify the sample JSON to include values for the properties. The ID property is ignored.

The screenshot shows the Swagger UI interface for a Persons API. At the top, it says "Persons API 1.0.0 OAS3". Below that, there's a section titled "default". Under "default", there are two methods: "GET /persons" and "POST /persons". The "GET /persons" method is described as "Get all persons". The "POST /persons" method is described as "Create a new person". Under the "POST /persons" method, there is a "Parameters" section which says "No parameters". There are also "Cancel" and "Reset" buttons. Below the parameters, there is a "Request body" section with the note "required". A dropdown menu shows "application/json". The request body example is: { "firstName": "Bill", "lastName": "Jones" }.

The screenshot shows the Swagger UI interface for a Persons API. At the top, it says "Persons API 1.0.0 OAS3". Below that, there's a section titled "default". Under "default", there are two methods: "GET /persons" and "POST /persons". The "GET /persons" method is described as "Get all persons". The "POST /persons" method is described as "Create a new person". Under the "POST /persons" method, there is a "Parameters" section which says "No parameters". There are also "Cancel" and "Reset" buttons. Below the parameters, there is a "Request body" section with the note "required". A dropdown menu shows "application/json". The request body example is: { "firstName": "Bill", "lastName": "Jones" }.

5. Select **Execute** to add a new record to the database. The API returns a successful response.
6. Expand the **GET** method on the Swagger UI page and select **Try it**. Select **Execute**, and the person you just created is returned.

## Deploy to Azure App Service

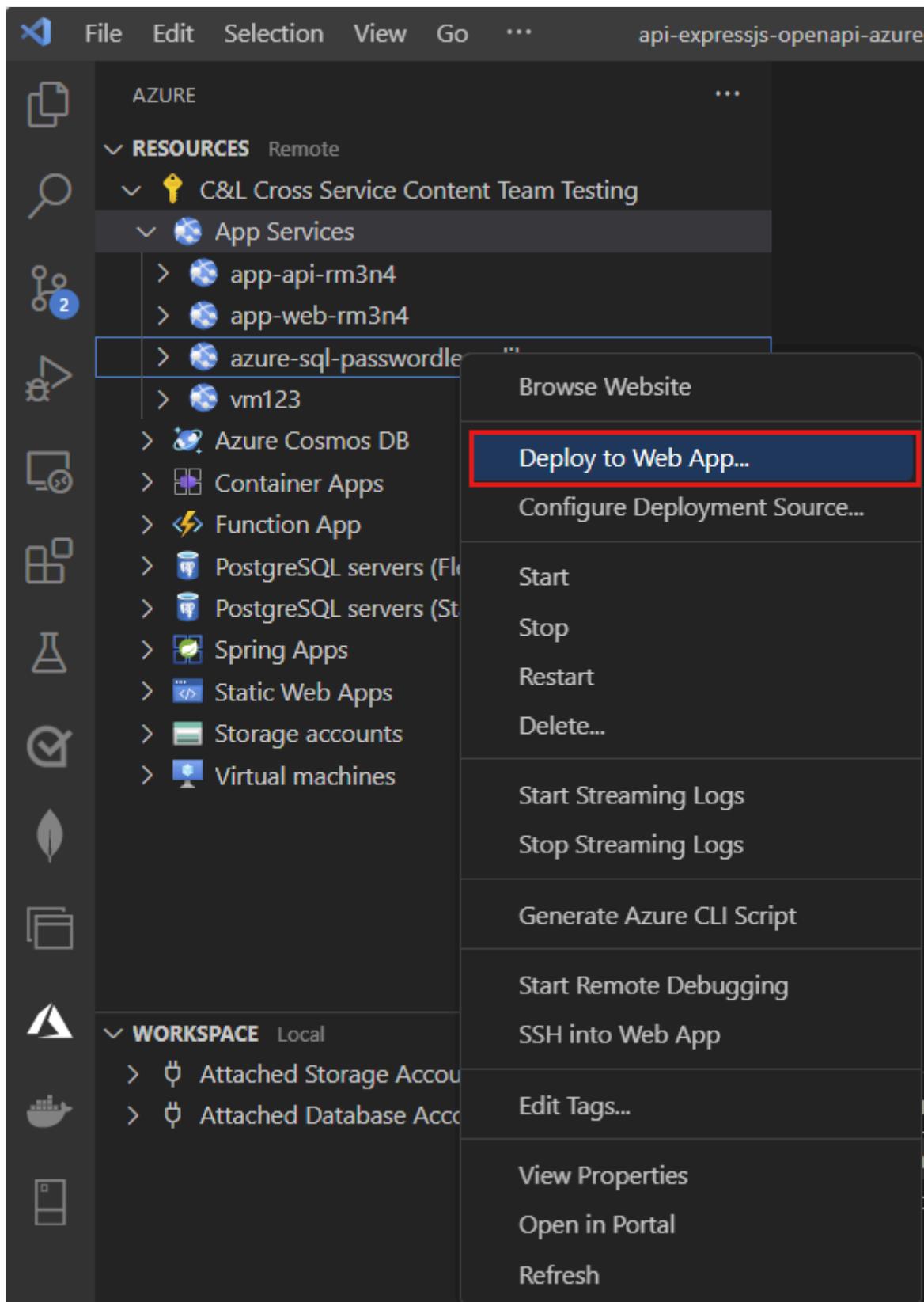
The app is ready to be deployed to Azure. Visual Studio Code can create an Azure App Service and deploy your application in a single workflow.

1. Make sure the app is stopped.
2. Sign in to Azure, if you haven't already, by selecting the **Azure: Sign In to Azure Cloud** command in the Command Palette (`Ctrl + Shift + P`)
3. In Visual Studio Code's **Azure Explorer** window, right-click on the **App Services** node and select **Create New Web App (Advanced)**.
4. Use the following table to create the App Service:

[] [Expand table](#)

Prompt	Value
Enter a globally unique name for the new web app.	Enter a prompt such as <code>azure-sql-passwordless</code> . Postpend a unique string such as <code>123</code> .
Select a resource group for new resources.	Select <b>+Create a new resource group</b> then select the default name.
Select a runtime stack.	Select an LTS version of the Node.js stack.
Select an OS.	Select <b>Linux</b> .
Select a location for new resources.	Select a location close to you.
Select a Linux App Service plan.	Select <b>Create new App Service plan</b> , then select the default name.
Select a pricing tier.	Select <b>Free (F1)</b> .
Select an Application Insights resource for your app.	Select <b>Skip for now</b> .

5. Wait until the notification that your app was created before continuing.
6. In the **Azure Explorer**, expand the **App Services** node and right-click your new app.
7. Select **Deploy to Web App**.



8. Select the root folder of the JavaScript project.

9. When the Visual Studio Code pop-up appears, select Deploy.

When the deployment finishes, the app doesn't work correctly on Azure. You still need to configure the secure connection between the App Service and the SQL database to retrieve your data.

# Connect the App Service to Azure SQL Database

Passwordless (recommended)

The following steps are required to connect the App Service instance to Azure SQL Database:

1. Create a managed identity for the App Service.
2. Create an SQL database user and associate it with the App Service managed identity.
3. Assign SQL roles to the database user that allow for read, write, and potentially other permissions.

There are multiple tools available to implement these steps:

Service Connector (Recommended)

Service Connector is a tool that streamlines authenticated connections between different services in Azure. Service Connector currently supports connecting an App Service to an Azure SQL database via the Azure CLI using the `az webapp connection create sql` command. This single command completes the three steps mentioned above for you.

## Create the managed identity with Service Connector

Run the following command in the Azure portal's Cloud Shell. The Cloud Shell has the latest version of the Azure CLI. Replace the variables in `<>` with your own values.

Azure CLI

```
az webapp connection create sql \
-g <app-service-resource-group> \
-n <app-service-name> \
--tg <database-server-resource-group> \
--server <database-server-name> \
--database <database-name> \
--system-identity
```

## Verify the App Service app settings

You can verify the changes made by Service Connector on the App Service settings.

1. In Visual Studio Code, in the Azure explorer, right-click your App Service and select **Open in portal**.
2. Navigate to the **Identity** page for your App Service. Under the **System assigned** tab, the **Status** should be set to **On**. This value means that a system-assigned managed identity was enabled for your app.
3. Navigate to the **Configuration** page for your App Service. Under the **Application Settings** tab, you should see several environment variables, which were already in the **mssql** configuration object.
  - AZURE\_SQL\_SERVER
  - AZURE\_SQL\_DATABASE
  - AZURE\_SQL\_PORT
  - AZURE\_SQL\_AUTHENTICATIONTYPE

Don't delete or change the property names or values.

## Test the deployed application

Browse to the URL of the app to test that the connection to Azure SQL Database is working. You can locate the URL of your app on the App Service overview page.

The person you created locally should display in the browser. Congratulations! Your application is now connected to Azure SQL Database in both local and hosted environments.

### 💡 Tip

If you receive a 500 Internal Server error while testing, it may be due to your database networking configurations. Verify that your logical server is configured with the settings outlined in the [Configure the database](#) section.

## Clean up the resources

When you are finished working with the Azure SQL Database, delete the resource to avoid unintended costs.

Azure portal

1. In the Azure portal search bar, search for *Azure SQL* and select the matching result.
2. Locate and select your database in the list of databases.
3. On the **Overview** page of your Azure SQL Database, select **Delete**.
4. On the **Azure you sure you want to delete...** page that opens, type the name of your database to confirm, and then select **Delete**.

## Sample code

The sample code for this application is available on [GitHub](#).

## Next steps

- [Tutorial: Secure a database in Azure SQL Database](#)
- [Authorize database access to SQL Database](#)
- [An overview of Azure SQL Database security capabilities](#)
- [Azure SQL Database security best practices](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

# Quickstart: Azure Cosmos DB for NoSQL library for Node.js

Article • 01/08/2024

APPLIES TO:  NoSQL

Get started with the Azure Cosmos DB for NoSQL client library for Node.js to query data in your containers and perform common operations on individual items. Follow these steps to deploy a minimal solution to your environment using the Azure Developer CLI.

[API reference documentation](#) | [Library source code](#) | [Package \(npm\)](#) | [Azure Developer CLI](#)

## Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [GitHub account](#)

## Setting up

Deploy this project's development container to your environment. Then, use the Azure Developer CLI (`azd`) to create an Azure Cosmos DB for NoSQL account and deploy a containerized sample application. The sample application uses the client library to manage, create, read, and query sample data.

 GITHUB CODESPACES 

### Important

GitHub accounts include an entitlement of storage and core hours at no cost. For more information, see [included storage and core hours for GitHub accounts](#).

1. Open a terminal in the root directory of the project.
2. Authenticate to the Azure Developer CLI using `azd auth login`. Follow the steps specified by the tool to authenticate to the CLI using your preferred Azure credentials.

Azure CLI

```
azd auth login
```

3. Use `azd init` to initialize the project.

```
Azure CLI
```

```
azd init
```

4. During initialization, configure a unique environment name.

 **Tip**

The environment name will also be used as the target resource group name. For this quickstart, consider using `msdocs-cosmos-db-nosql`.

5. Deploy the Azure Cosmos DB for NoSQL account using `azd up`. The Bicep templates also deploy a sample web application.

```
Azure CLI
```

```
azd up
```

6. During the provisioning process, select your subscription and desired location. Wait for the provisioning process to complete. The process can take **approximately five minutes**.
7. Once the provisioning of your Azure resources is done, a URL to the running web application is included in the output.

```
Output
```

```
Deploying services (azd deploy)
```

```
(✓) Done: Deploying service web
- Endpoint: <https://[container-app-sub-domain].azurecontainerapps.io>
```

```
SUCCESS: Your application was provisioned and deployed to Azure in 5
minutes 0 seconds.
```

8. Use the URL in the console to navigate to your web application in the browser. Observe the output of the running app.

## Azure Cosmos DB for NoSQL | Go Quickstart

```
Current Status: Starting...
Get database: cosmicworks
Get container: products
Upserter item: {70b63682-b93a-4c77-aad2-65501347265f gear-surf-surfboards Yamba Surfboard 12 850 false}
Status code: 201
Request charge: 7.24
Upserter item: {25a68543-b90c-439d-8332-7ef41e06a0e0 gear-surf-surfboards Kiama Classic Surfboard 25 790 true}
Status code: 201
Request charge: 7.24
Read item id: 70b63682-b93a-4c77-aad2-65501347265f
Read item: {70b63682-b93a-4c77-aad2-65501347265f gear-surf-surfboards Yamba Surfboard 12 850 false}
Status code: 200
Request charge: 1.00
```

## Install the client library

The client library is available through the Node Package Manager, as the `@azure/cosmos` package.

1. Open a terminal and navigate to the `/src` folder.

```
Bash
```

```
cd ./src
```

2. If not already installed, install the `@azure/cosmos` package using `npm install`.

```
Bash
```

```
npm install --save @azure/cosmos
```

3. Also, install the `@azure/identity` package if not already installed.

```
Bash
```

```
npm install --save @azure/identity
```

4. Open and review the `src/package.json` file to validate that the `azure-cosmos` and `azure-identity` entries both exist.

## Object model

[] Expand table

Name	Description
<a href="#">CosmosClient</a>	This class is the primary client class and is used to manage account-wide

Name	Description
	metadata or databases.
Database	This class represents a database within the account.
Container	This class is primarily used to perform read, update, and delete operations on either the container or the items stored within the container.
PartitionKey	This class represents a logical partition key. This class is required for many common operations and queries.
SqlQuerySpec	This interface represents a SQL query and any query parameters.

## Code examples

- [Authenticate the client](#)
- [Get a database](#)
- [Get a container](#)
- [Create an item](#)
- [Get an item](#)
- [Query items](#)

The sample code in the template uses a database named `cosmicworks` and container named `products`. The `products` container contains details such as name, category, quantity, a unique identifier, and a sale flag for each product. The container uses the `/category` property as a logical partition key.

## Authenticate the client

Application requests to most Azure services must be authorized. Use the `DefaultAzureCredential` type as the preferred way to implement a passwordless connection between your applications and Azure Cosmos DB for NoSQL. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime.

### Important

You can also authorize requests to Azure services using passwords, connection strings, or other credentials directly. However, this approach should be used with caution. Developers must be diligent to never expose these secrets in an unsecure location. Anyone who gains access to the password or secret key is able to authenticate to the database service. `DefaultAzureCredential` offers improved

management and security benefits over the account key to allow passwordless authentication without the risk of storing keys.

This sample creates a new instance of the `CosmosClient` type and authenticates using a `DefaultAzureCredential` instance.

JavaScript

```
const credential = new DefaultAzureCredential();

const client = new CosmosClient({
 endpoint,
 aadCredentials: credential
});
```

## Get a database

Use `client.database` to retrieve the existing database named `cosmicworks`.

JavaScript

```
const database = client.database('cosmicworks');
```

## Get a container

Retrieve the existing `products` container using `database.container`.

JavaScript

```
const container = database.container('products');
```

## Create an item

Build a new object with all of the members you want to serialize into JSON. In this example, the type has a unique identifier, and fields for category, name, quantity, price, and sale. Create an item in the container using `container.items.upsert`. This method "upserts" the item effectively replacing the item if it already exists.

JavaScript

```
var item = {
 'id': '70b63682-b93a-4c77-aad2-65501347265f',
```

```
'category': 'gear-surf-surfboards',
'name': 'Yamba Surfboard',
'quantity': 12,
'price': 850.00,
'clearance': false
};

var response = await container.items.upsert(item);
```

## Read an item

Perform a point read operation by using both the unique identifier (`id`) and partition key fields. Use `container.item` to get a pointer to an item and `item.read` to efficiently retrieve the specific item.

JavaScript

```
var id = '70b63682-b93a-4c77-aad2-65501347265f';
var partitionKey = 'gear-surf-surfboards';

var response = await container.item(id, partitionKey).read();
var read_item = response.resource;
```

## Query items

Perform a query over multiple items in a container using `container.items.query`. Find all items within a specified category using this parameterized query:

nosql

```
SELECT * FROM products p WHERE p.category = @category
```

Fetch all of the results of the query using `query.fetchAll`. Loop through the results of the query.

JavaScript

```
const querySpec = {
 query: 'SELECT * FROM products p WHERE p.category = @category',
 parameters: [
 {
 name: '@category',
 value: 'gear-surf-surfboards'
 }
]
};
```

```
var response = await container.items.query(querySpec).fetchAll();
for (var item of response.resources) {
}
```

## Related content

- [.NET Quickstart](#)
- [Java Quickstart](#)
- [Python Quickstart](#)
- [Go Quickstart](#)

## Next step

[Tutorial: Build a Node.js web app](#)

# Quickstart: Send events to or receive events from event hubs by using JavaScript

Article • 04/05/2024

In this Quickstart, you learn how to send events to and receive events from an event hub using the [@azure/event-hubs](#) npm package.

## Prerequisites

If you're new to Azure Event Hubs, see [Event Hubs overview](#) before you do this quickstart.

To complete this quickstart, you need the following prerequisites:

- **Microsoft Azure subscription.** To use Azure services, including Azure Event Hubs, you need a subscription. If you don't have an existing Azure account, you can sign up for a [free trial](#).
- Node.js LTS. Download the latest [long-term support \(LTS\) version](#).
- Visual Studio Code (recommended) or any other integrated development environment (IDE).
- **Create an Event Hubs namespace and an event hub.** The first step is to use the [Azure portal](#) to create a namespace of type Event Hubs, and obtain the management credentials your application needs to communicate with the event hub. To create a namespace and an event hub, follow the procedure in [this article](#).

## Install npm packages to send events

To install the [Node Package Manager \(npm\)](#) package for Event Hubs, open a command prompt that has *npm* in its path, change the directory to the folder where you want to keep your samples.

Passwordless (Recommended)

Run these commands:

shell

```
npm install @azure/event-hubs
npm install @azure/identity
```

## Authenticate the app to Azure

This quickstart shows you two ways of connecting to Azure Event Hubs: passwordless and connection string. The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to an Event Hubs namespace. You don't need to worry about having hard-coded connection strings in your code or in a configuration file or in a secure storage like Azure Key Vault. The second option shows you how to use a connection string to connect to an Event Hubs namespace. If you're new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

## Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Event Hubs has the correct permissions. You'll need the [Azure Event Hubs Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Event Hubs Data Owner` role to your user account, which provides full access to Azure Event Hubs resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

## Azure built-in roles for Azure Event Hubs

For Azure Event Hubs, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already

protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to an Event Hubs namespace:

- [Azure Event Hubs Data Owner](#): Enables data access to Event Hubs namespace and its entities (queues, topics, subscriptions, and filters)
- [Azure Event Hubs Data Sender](#): Use this role to give the sender access to Event Hubs namespace and its entities.
- [Azure Event Hubs Data Receiver](#): Use this role to give the receiver access to Event Hubs namespace and its entities.

If you want to create a custom role, see [Rights required for Event Hubs operations](#).

 **Important**

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your Event Hubs namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'Access control (IAM)' blade in the Azure portal. The left sidebar lists various service bus entities: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Geo-Recovery, Migrate to premium, Encryption, Configuration, Properties, Locks), Entities (Queues, Topics). The 'Access control (IAM)' item is selected and highlighted with a red box. The main area has a tooltip: 'You do not have access to this customer directory, so some options are disabled. If you require access to these options, contact your administrator.' Below this, there are two main sections: 'Grant access to this resource' (with 'Add role assignment' and 'Learn more' buttons) and 'View access to this resource' (with 'View' and 'Learn more' buttons).

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Event Hubs Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Send events

In this section, you create a JavaScript application that sends events to an event hub.

1. Open your favorite editor, such as [Visual Studio Code](#).
2. Create a file called *send.js*, and paste the following code into it:

Passwordless (Recommended)

In the code, use real values to replace the following placeholders:

- **EVENT HUBS NAMESPACE NAME**

- EVENT HUB NAME

JavaScript

```
const { EventHubProducerClient } = require("@azure/event-hubs");
const { DefaultAzureCredential } = require("@azure/identity");

// Event hubs
const eventHubsNamespaceName = "EVENT HUBS NAMESPACE NAME";
const fullyQualifiedNamespace =
`${eventHubsNamespaceName}.servicebus.windows.net`;
const eventHubName = "EVENT HUB NAME";

// Azure Identity - passwordless authentication
const credential = new DefaultAzureCredential();

async function main() {

 // Create a producer client to send messages to the event hub.
 const producer = new
EventHubProducerClient(fullyQualifiedNamespace, eventHubName,
credential);

 // Prepare a batch of three events.
 const batch = await producer.createBatch();
 batch.tryAdd({ body: "passwordless First event" });
 batch.tryAdd({ body: "passwordless Second event" });
 batch.tryAdd({ body: "passwordless Third event" });

 // Send the batch to the event hub.
 await producer.sendBatch(batch);

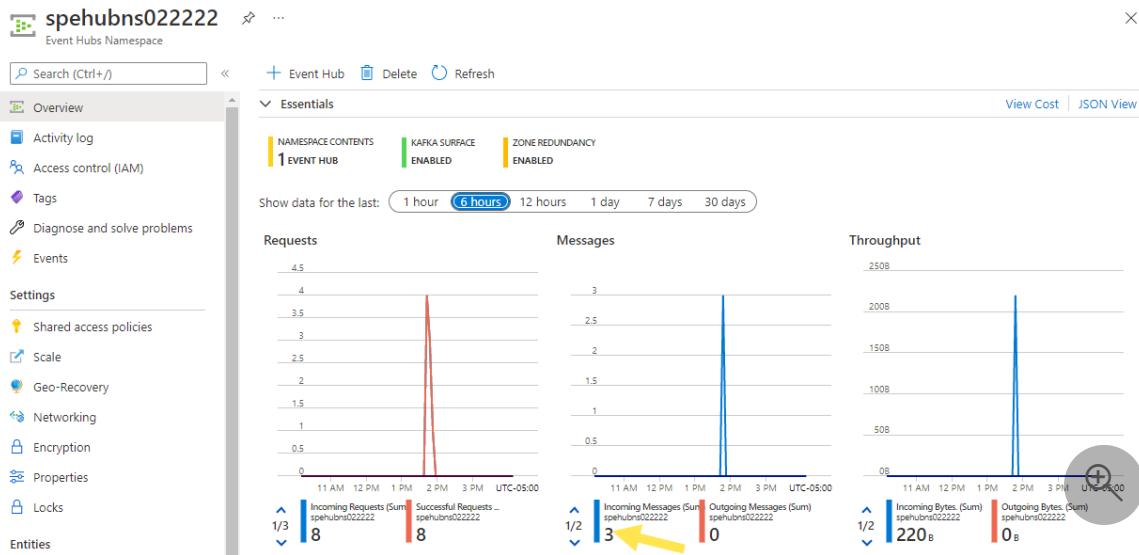
 // Close the producer client.
 await producer.close();

 console.log("A batch of three events have been sent to the event
hub");
}

main().catch((err) => {
 console.log("Error occurred: ", err);
});
```

3. Run `node send.js` to execute this file. This command sends a batch of three events to your event hub. If you're using the Passwordless (Azure Active Directory's Role-based Access Control) authentication, you might want to run `az login` and sign into Azure using the account that was added to the Azure Event Hubs Data Owner role.

4. In the Azure portal, verify that the event hub received the messages. Refresh the page to update the chart. It might take a few seconds for it to show that the messages are received.



### ⓘ Note

For the complete source code, including additional informational comments, go to the [GitHub sendEvents.js page](#).

## Receive events

In this section, you receive events from an event hub by using an Azure Blob storage checkpoint store in a JavaScript application. It performs metadata checkpoints on received messages at regular intervals in an Azure Storage blob. This approach makes it easy to continue receiving messages later from where you left off.

Follow these recommendations when using Azure Blob Storage as a checkpoint store:

- Use a separate container for each consumer group. You can use the same storage account, but use one container per each group.
- Don't use the container for anything else, and don't use the storage account for anything else.
- Storage account should be in the same region as the deployed application is located in. If the application is on-premises, try to choose the closest region possible.

On the **Storage account** page in the Azure portal, in the **Blob service** section, ensure that the following settings are disabled.

- Hierarchical namespace
- Blob soft delete
- Versioning

## Create an Azure storage account and a blob container

To create an Azure storage account and a blob container in it, do the following actions:

1. [Create an Azure storage account](#)
2. [Create a blob container in the storage account](#)
3. Authenticate to the blob container

Passwordless (Recommended)

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

### Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'Access Control (IAM)' page for a storage account named 'identitymigrationstorage'. The left sidebar has a red box around the 'Access Control (IAM)' item. The top navigation bar has a red box around the '+ Add' button. A dropdown menu is open, with 'Add role assignment' highlighted and also has a red box around it. Other options in the dropdown include 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. The main content area has sections for 'My access', 'Check access', and 'Grant access to this resource'. There is a search bar at the bottom of the main content area.

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Install the npm packages to receive events

For the receiving side, you need to install two more packages. In this quickstart, you use Azure Blob storage to persist checkpoints so that the program doesn't read the events that it already read. It performs metadata checkpoints on received messages at regular intervals in a blob. This approach makes it easy to continue receiving messages later from where you left off.

Passwordless (Recommended)

Run these commands:

shell

```
npm install @azure/storage-blob
npm install @azure/eventhubs-checkpointstore-blob
npm install @azure/identity
```

## Write code to receive events

1. Open your favorite editor, such as [Visual Studio Code](#).
2. Create a file called *receive.js*, and paste the following code into it:

Passwordless (Recommended)

In the code, use real values to replace the following placeholders:

- EVENT HUBS NAMESPACE NAME
- EVENT HUB NAME
- STORAGE ACCOUNT NAME
- STORAGE CONTAINER NAME

JavaScript

```
const { DefaultAzureCredential } = require("@azure/identity");
const { EventHubConsumerClient, earliestEventPosition } =
require("@azure/event-hubs");
const { ContainerClient } = require("@azure/storage-blob");
const { BlobCheckpointStore } = require("@azure/eventhubs-
checkpointstore-blob");

// Event hubs
const eventHubsResourceName = "EVENT HUBS NAMESPACE NAME";
const fullyQualifiedNamespace =
`${eventHubsResourceName}.servicebus.windows.net`;
const eventHubName = "EVENT HUB NAME";
```

```
const consumerGroup = "$Default"; // name of the default consumer group

// Azure Storage
const storageAccountName = "STORAGE ACCOUNT NAME";
const storageContainerName = "STORAGE CONTAINER NAME";
const baseUrl =
`https://${storageAccountName}.blob.core.windows.net`;

// Azure Identity - passwordless authentication
const credential = new DefaultAzureCredential();

async function main() {

 // Create a blob container client and a blob checkpoint store using the client.
 const containerClient = new ContainerClient(
 `${baseUrl}/${storageContainerName}`,
 credential
);
 const checkpointStore = new BlobCheckpointStore(containerClient);

 // Create a consumer client for the event hub by specifying the checkpoint store.
 const consumerClient = new EventHubConsumerClient(consumerGroup,
 fullyQualifiedNamespace, eventHubName, credential,
 checkpointStore);

 // Subscribe to the events, and specify handlers for processing the events and errors.
 const subscription = consumerClient.subscribe({
 processEvents: async (events, context) => {
 if (events.length === 0) {
 console.log(`No events received within wait time. Waiting for next interval`);
 return;
 }

 for (const event of events) {
 console.log(`Received event: '${event.body}' from partition: '${context.partitionId}' and consumer group: '${context.consumerGroup}'`);
 }
 // Update the checkpoint.
 await context.updateCheckpoint(events[events.length - 1]);
 },
 processError: async (err, context) => {
 console.log(`Error : ${err}`);
 }
 },
 { startPosition: earliestEventPosition
 });

 // After 30 seconds, stop processing.
}
```

```
 await new Promise((resolve) => {
 setTimeout(async () => {
 await subscription.close();
 await consumerClient.close();
 resolve();
 }, 3000);
 });
}

main().catch((err) => {
 console.log("Error occurred: ", err);
});
```

3. Run `node receive.js` in a command prompt to execute this file. The window should display messages about received events.

Bash

```
C:\Self Study\Event Hubs\JavaScript>node receive.js
Received event: 'First event' from partition: '0' and consumer group:
'$Default'
Received event: 'Second event' from partition: '0' and consumer group:
'$Default'
Received event: 'Third event' from partition: '0' and consumer group:
'$Default'
```

#### ⚠ Note

For the complete source code, including additional informational comments, go to the [GitHub receiveEventsUsingCheckpointStore.js page](#).

The receiver program receives events from all the partitions of the default consumer group in the event hub.

## Clean up resources

Delete the resource group that has the Event Hubs namespace or delete only the namespace if you want to keep the resource group.

## Related content

Check out these samples on GitHub:

- [JavaScript samples](#)

- [TypeScript samples ↗](#)

# Quickstart: Azure Key Vault certificate client library for JavaScript

Article • 04/07/2024

Get started with the Azure Key Vault certificate client library for JavaScript. [Azure Key Vault](#) is a cloud service that provides a secure store for certificates. You can securely store keys, passwords, certificates, and other secrets. Azure key vaults may be created and managed through the Azure portal. In this quickstart, you learn how to create, retrieve, and delete certificates from an Azure key vault using the JavaScript client library

Key Vault client library resources:

[API reference documentation](#) | [Library source code](#) | [Package \(npm\)](#)

For more information about Key Vault and certificates, see:

- [Key Vault Overview](#)
- [Certificates Overview](#)

## Prerequisites

- An Azure subscription - [create one for free](#).
- Current [Node.js LTS](#).
- [Azure CLI](#)
- An existing Key Vault - you can create one using:
  - [Azure CLI](#)
  - [Azure portal](#)
  - [Azure PowerShell](#)

This quickstart assumes you're running [Azure CLI](#).

## Sign in to Azure

1. Run the `login` command.

```
Azure CLI
```

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

## Create new Node.js application

Create a Node.js application that uses your key vault.

1. In a terminal, create a folder named `key-vault-node-app` and change into that folder:

```
terminal

mkdir key-vault-node-app && cd key-vault-node-app
```

2. Initialize the Node.js project:

```
terminal

npm init -y
```

## Install Key Vault packages

1. Using the terminal, install the Azure Key Vault secrets library, [@azure/keyvault-certificates](#) for Node.js.

```
terminal

npm install @azure/keyvault-certificates
```

2. Install the Azure Identity client library, [@azure/identity](#), to authenticate to a Key Vault.

```
terminal

npm install @azure/identity
```

# Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

## Set environment variables

This application is using key vault name as an environment variable called

`KEY_VAULT_NAME`.

Windows

Windows Command Prompt

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

## Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) method provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this code, the name of your key vault is used to create the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

## Code example

This code uses the following [Key Vault Certificate classes and methods](#):

- `DefaultAzureCredential` class
- `CertificateClient` class
  - `beginCreateCertificate`
  - `getCertificate`
  - `getCertificateVersion`
  - `updateCertificateProperties`
  - `updateCertificatePolicy`
  - `beginDeleteCertificate`
- `PollerLike` interface
  - `getResult`

## Set up the app framework

1. Create new text file and paste the following code into the `index.js` file.

JavaScript

```
const { CertificateClient, DefaultCertificatePolicy } =
require("@azure/keyvault-certificates");
const { DefaultAzureCredential } = require("@azure/identity");

async function main() {
 // If you're using MSI, DefaultAzureCredential should "just work".
 // Otherwise, DefaultAzureCredential expects the following three
 environment variables:
 // - AZURE_TENANT_ID: The tenant ID in Azure Active Directory
 // - AZURE_CLIENT_ID: The application (client) ID registered in the
 AAD tenant
 // - AZURE_CLIENT_SECRET: The client secret for the registered
```

```
application
 const credential = new DefaultAzureCredential();

 const keyVaultName = process.env["KEY_VAULT_NAME"];
 if(!keyVaultName) throw new Error("KEY_VAULT_NAME is empty");
 const url = "https://" + keyVaultName + ".vault.azure.net";

 const client = new CertificateClient(url, credential);

 const uniqueString = new Date().getTime();
 const certificateName = `cert${uniqueString}`;

 // Creating a self-signed certificate
 const createPoller = await client.beginCreateCertificate(
 certificateName,
 DefaultCertificatePolicy
);

 const pendingCertificate = createPoller.getResult();
 console.log("Certificate: ", pendingCertificate);

 // To read a certificate with their policy:
 let certificateWithPolicy = await
client.getCertificate(certificateName);
 // Note: It will always read the latest version of the certificate.

 console.log("Certificate with policy:", certificateWithPolicy);

 // To read a certificate from a specific version:
 const certificateFromVersion = await client.getCertificateVersion(
 certificateName,
 certificateWithPolicy.properties.version
);
 // Note: It will not retrieve the certificate's policy.
 console.log("Certificate from a specific version:",
certificateFromVersion);

 const updatedCertificate = await
client.updateCertificateProperties(certificateName, "", {
 tags: {
 customTag: "value"
 }
});
 console.log("Updated certificate:", updatedCertificate);

 // Updating the certificate's policy:
 await client.updateCertificatePolicy(certificateName, {
 issuerName: "Self",
 subject: "cn=MyOtherCert"
 });
 certificateWithPolicy = await client.getCertificate(certificateName);
 console.log("updatedCertificate certificate's policy:",
certificateWithPolicy.policy);

 // delete certificate
```

```

 const deletePoller = await
client.beginDeleteCertificate(certificateName);
 const deletedCertificate = await deletePoller.pollUntilDone();
 console.log("Recovery Id: ", deletedCertificate.recoveryId);
 console.log("Deleted Date: ", deletedCertificate.deletedOn);
 console.log("Scheduled Purge Date: ",
deletedCertificate.scheduledPurgeDate);
}

main().catch((error) => {
 console.error("An error occurred:", error);
 process.exit(1);
});

```

## Run the sample application

1. Run the app:

terminal

```
node index.js
```

2. The create and get methods return a full JSON object for the certificate:

JSON

```
{
 "keyId": undefined,
 "secretId": undefined,
 "name": "YOUR-CERTIFICATE-NAME",
 "reuseKey": false,
 "keyCurveName": undefined,
 "exportable": true,
 "issuerName": 'Self',
 "certificateType": undefined,
 "certificateTransparency": undefined
},
"properties": {
 "createdOn": 2021-11-29T20:17:45.000Z,
 "updatedOn": 2021-11-29T20:17:45.000Z,
 "expiresOn": 2022-11-29T20:17:45.000Z,
 "id": "https://YOUR-KEY-VAULT-
NAME.vault.azure.net/certificates/YOUR-CERTIFICATE-NAME/YOUR-
CERTIFICATE-VERSION",
 "enabled": false,
 "notBefore": 2021-11-29T20:07:45.000Z,
 "recoveryLevel": "Recoverable+Purgeable",
 "name": "YOUR-CERTIFICATE-NAME",
 "vaultUrl": "https://YOUR-KEY-VAULT-NAME.vault.azure.net",
 "version": "YOUR-CERTIFICATE-VERSION",
}
```

```
 "tags": undefined,
 "x509Thumbprint": undefined,
 "recoverableDays": 90
 }
}
```

## Integrating with App Configuration

The Azure SDK provides a helper method, [parseKeyVaultCertificateIdentifier](#), to parse the given Key Vault certificate ID, which is necessary if you use [App Configuration](#) references to Key Vault. App Config stores the Key Vault certificate ID. You need the *parseKeyVaultCertificateIdentifier* method to parse that ID to get the certificate name. Once you have the certificate name, you can get the current certificate using code from this quickstart.

## Next steps

In this quickstart, you created a key vault, stored a certificate, and retrieved that certificate. To learn more about Key Vault and how to integrate it with your applications, continue on to these articles.

- Read an [Overview of Azure Key Vault](#)
- Read an [Overview of certificates](#)
- See an [Access Key Vault from App Service Application Tutorial](#)
- See an [Access Key Vault from Virtual Machine Tutorial](#)
- See the [Azure Key Vault developer's guide](#)
- Review the [Key Vault security overview](#)

# Quickstart: Azure Key Vault key client library for JavaScript

Article • 04/07/2024

Get started with the Azure Key Vault key client library for JavaScript. [Azure Key Vault](#) is a cloud service that provides a secure store for cryptographic keys. You can securely store keys, passwords, certificates, and other secrets. Azure key vaults may be created and managed through the Azure portal. In this quickstart, you learn how to create, retrieve, and delete keys from an Azure key vault using the JavaScript key client library.

Key Vault client library resources:

[API reference documentation](#) | [Library source code](#) | [Package \(npm\)](#)

For more information about Key Vault and keys, see:

- [Key Vault Overview](#)
- [Keys Overview](#).

## Prerequisites

- An Azure subscription - [create one for free](#).
- Current [Node.js LTS](#).
- [Azure CLI](#)
- An existing Key Vault - you can create one using:
  - [Azure CLI](#)
  - [Azure portal](#)
  - [Azure PowerShell](#)

This quickstart assumes you're running [Azure CLI](#).

## Sign in to Azure

1. Run the `login` command.

Azure CLI

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

## Create new Node.js application

Create a Node.js application that uses your key vault.

1. In a terminal, create a folder named `key-vault-node-app` and change into that folder:

```
terminal
mkdir key-vault-node-app && cd key-vault-node-app
```

2. Initialize the Node.js project:

```
terminal
npm init -y
```

## Install Key Vault packages

1. Using the terminal, install the Azure Key Vault secrets client library, [@azure/keyvault-keys](#) for Node.js.

```
terminal
npm install @azure/keyvault-keys
```

2. Install the Azure Identity client library, [@azure/identity](#) package to authenticate to a Key Vault.

```
terminal
npm install @azure/identity
```

# Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

## Set environment variables

This application is using key vault name as an environment variable called

`KEY_VAULT_NAME`.

Windows

Windows Command Prompt

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

## Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) method provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this code, the name of your key vault is used to create the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

## Code example

The code samples below will show you how to create a client, set a secret, retrieve a secret, and delete a secret.

This code uses the following [Key Vault Secret classes and methods](#):

- [DefaultAzureCredential class](#)
- [KeyClient class](#)
  - [createKey](#)
  - [createEcKey](#)
  - [createRsaKey](#)
  - [getKey](#)
  - [listPropertiesOfKeys](#)
  - [updateKeyProperties](#)
  - [beginDeleteKey](#)
  - [getDeletedKey](#)
  - [purgeDeletedKey](#)

## Set up the app framework

1. Create new text file and paste the following code into the `index.js` file.

JavaScript

```
const { KeyClient } = require("@azure/keyvault-keys");
const { DefaultAzureCredential } = require("@azure/identity");

async function main() {

 // DefaultAzureCredential expects the following three environment
 // variables:
 // - AZURE_TENANT_ID: The tenant ID in Azure Active Directory
```

```
// - AZURE_CLIENT_ID: The application (client) ID registered in the
AAD tenant
// - AZURE_CLIENT_SECRET: The client secret for the registered
application
const credential = new DefaultAzureCredential();

const keyVaultName = process.env["KEY_VAULT_NAME"];
if(!keyVaultName) throw new Error("KEY_VAULT_NAME is empty");
const url = "https://" + keyVaultName + ".vault.azure.net";

const client = new KeyClient(url, credential);

const uniqueString = Date.now();
const keyName = `sample-key-${uniqueString}`;
const ecKeyName = `sample-ec-key-${uniqueString}`;
const rsaKeyName = `sample-rsa-key-${uniqueString}`;

// Create key using the general method
const result = await client.createKey(keyName, "EC");
console.log("key: ", result);

// Create key using specialized key creation methods
const ecResult = await client.createEcKey(ecKeyName, { curve: "P-
256" });
const rsaResult = await client.createRsaKey(rsaKeyName, { keySize:
2048 });
console.log("Elliptic curve key: ", ecResult);
console.log("RSA Key: ", rsaResult);

// Get a specific key
const key = await client.getKey(keyName);
console.log("key: ", key);

// Or list the keys we have
for await (const keyProperties of client.listPropertiesOfKeys()) {
const key = await client.getKey(keyProperties.name);
console.log("key: ", key);
}

// Update the key
const updatedKey = await client.updateKeyProperties(keyName,
result.properties.version, {
enabled: false
});
console.log("updated key: ", updatedKey);

// Delete the key - the key is soft-deleted but not yet purged
const deletePoller = await client.beginDeleteKey(keyName);
await deletePoller.pollUntilDone();

const deletedKey = await client.getDeletedKey(keyName);
console.log("deleted key: ", deletedKey);

// Purge the key - the key is permanently deleted
// This operation could take some time to complete
```

```

 console.time("purge a single key");
 await client.purgeDeletedKey(keyName);
 console.timeEnd("purge a single key");
 }

main().catch((error) => {
 console.error("An error occurred:", error);
 process.exit(1);
});

```

## Run the sample application

1. Run the app:

terminal

```
node index.js
```

2. The create and get methods return a full JSON object for the key:

JSON

```

"key": {
 "key": {
 "kid": "https://YOUR-KEY-VAULT-NAME.vault.azure.net/keys/YOUR-KEY-
NAME/YOUR-KEY-VERSION",
 "kty": "YOUR-KEY-TYPE",
 "keyOps": [ARRAY-OF-VALID-OPERATIONS],
 ... other properties based on key type
 },
 "id": "https://YOUR-KEY-VAULT-NAME.vault.azure.net/keys/YOUR-KEY-
NAME/YOUR-KEY-VERSION",
 "name": "YOUR-KEY-NAME",
 "keyOperations": [ARRAY-OF-VALID-OPERATIONS],
 "keyType": "YOUR-KEY-TYPE",
 "properties": {
 "tags": undefined,
 "enabled": true,
 "notBefore": undefined,
 "expiresOn": undefined,
 "createdOn": 2021-11-29T18:29:11.000Z,
 "updatedOn": 2021-11-29T18:29:11.000Z,
 "recoverableDays": 90,
 "recoveryLevel": "Recoverable+Purgeable",
 "exportable": undefined,
 "releasePolicy": undefined,
 "vaultUrl": "https://YOUR-KEY-VAULT-NAME.vault.azure.net",
 "version": "YOUR-KEY-VERSION",
 "name": "YOUR-KEY-VAULT-NAME",
 "managed": undefined,
 }
}

```

```
 "id": "https://YOUR-KEY-VAULT-NAME.vault.azure.net/keys/YOUR-KEY-
 NAME/YOUR-KEY-VERSION"
 }
}
```

## Integrating with App Configuration

The Azure SDK provides a helper method, [parseKeyVaultKeyIdentifier](#), to parse the given Key Vault Key ID. This is necessary if you use [App Configuration](#) references to Key Vault. App Config stores the Key Vault Key ID. You need the *parseKeyVaultKeyIdentifier* method to parse that ID to get the key name. Once you have the key name, you can get the current key value using code from this quickstart.

## Next steps

In this quickstart, you created a key vault, stored a key, and retrieved that key. To learn more about Key Vault and how to integrate it with your applications, continue on to these articles.

- Read an [Overview of Azure Key Vault](#)
- Read an [Overview of Azure Key Vault Keys](#)
- How to [Secure access to a key vault](#)
- See the [Azure Key Vault developer's guide](#)
- Review the [Key Vault security overview](#)

# Quickstart: Azure Key Vault secret client library for JavaScript

Article • 04/07/2024

Get started with the Azure Key Vault secret client library for JavaScript. [Azure Key Vault](#) is a cloud service that provides a secure store for secrets. You can securely store keys, passwords, certificates, and other secrets. Azure key vaults may be created and managed through the Azure portal. In this quickstart, you learn how to create, retrieve, and delete secrets from an Azure key vault using the JavaScript client library

Key Vault client library resources:

[API reference documentation](#) | [Library source code](#) | [Package \(npm\)](#)

For more information about Key Vault and secrets, see:

- [Key Vault Overview](#)
- [Secrets Overview](#)

## Prerequisites

- An Azure subscription - [create one for free](#).
- Current [Node.js LTS](#).
- [Azure CLI](#)
- An existing Key Vault - you can create one using:
  - [Azure CLI](#)
  - [Azure portal](#)
  - [Azure PowerShell](#)

This quickstart assumes you are running [Azure CLI](#).

## Sign in to Azure

1. Run the `login` command.

```
Azure CLI
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

## Create new Node.js application

Create a Node.js application that uses your key vault.

1. In a terminal, create a folder named `key-vault-node-app` and change into that folder:

```
terminal
mkdir key-vault-node-app && cd key-vault-node-app
```

2. Initialize the Node.js project:

```
terminal
npm init -y
```

## Install Key Vault packages

1. Using the terminal, install the Azure Key Vault secrets client library, [@azure/keyvault-secrets](#) for Node.js.

```
terminal
npm install @azure/keyvault-secrets
```

2. Install the Azure Identity client library, [@azure/identity](#) package to authenticate to a Key Vault.

```
terminal
npm install @azure/identity
```

# Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

## Set environment variables

This application is using key vault name as an environment variable called

`KEY_VAULT_NAME`.

Windows

Windows Command Prompt

```
set KEY_VAULT_NAME=<your-key-vault-name>
```

## Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) method provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In this code, the name of your key vault is used to create the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`. For more information about authenticating to key vault, see [Developer's Guide](#).

## Code example

The code samples below will show you how to create a client, set a secret, retrieve a secret, and delete a secret.

This code uses the following [Key Vault Secret classes and methods](#):

- `DefaultAzureCredential`
- `SecretClient` class
  - `setSecret`
  - `getSecret`
  - `updateSecretProperties`
  - `beginDeleteSecret`

## Set up the app framework

1. Create new text file and paste the following code into the `index.js` file.

JavaScript

```
const { SecretClient } = require("@azure/keyvault-secrets");
const { DefaultAzureCredential } = require("@azure/identity");

async function main() {
 // If you're using MSI, DefaultAzureCredential should "just work".
 // Otherwise, DefaultAzureCredential expects the following three
 environment variables:
 // - AZURE_TENANT_ID: The tenant ID in Azure Active Directory
 // - AZURE_CLIENT_ID: The application (client) ID registered in the
 AAD tenant
 // - AZURE_CLIENT_SECRET: The client secret for the registered
 application
 const credential = new DefaultAzureCredential();
```

```

const keyVaultName = process.env["KEY_VAULT_NAME"];
if(!keyVaultName) throw new Error("KEY_VAULT_NAME is empty");
const url = "https://" + keyVaultName + ".vault.azure.net";

const client = new SecretClient(url, credential);

// Create a secret
// The secret can be a string of any kind. For example,
// a multiline text block such as an RSA private key with newline
// characters,
// or a stringified JSON object, like `JSON.stringify({ mySecret:
'MySecretValue'})`.
const uniqueString = new Date().getTime();
const secretName = `secret${uniqueString}`;
const result = await client.setSecret(secretName, "MySecretValue");
console.log("result: ", result);

// Read the secret we created
const secret = await client.getSecret(secretName);
console.log("secret: ", secret);

// Update the secret with different attributes
const updatedSecret = await client.updateSecretProperties(secretName,
result.properties.version, {
 enabled: false
});
console.log("updated secret: ", updatedSecret);

// Delete the secret immediately without ability to restore or purge.
await client.beginDeleteSecret(secretName);
}

main().catch((error) => {
 console.error("An error occurred:", error);
 process.exit(1);
});

```

## Run the sample application

1. Run the app:

terminal

node index.js

2. The create and get methods return a full JSON object for the secret:

JSON

```
{
 "value": "MySecretValue",
 "name": "secret1637692472606",
 "properties": {
 "createdOn": "2021-11-23T18:34:33.000Z",
 "updatedOn": "2021-11-23T18:34:33.000Z",
 "enabled": true,
 "recoverableDays": 90,
 "recoveryLevel": "Recoverable+Purgeable",
 "id": "https://YOUR-KEYVAULT-
NAME.vault.azure.net/secrets/secret1637692472606/YOUR-VERSION",
 "vaultUrl": "https://YOUR-KEYVAULT-NAME.vault.azure.net",
 "version": "YOUR-VERSION",
 "name": "secret1637692472606"
 }
}
```

The update method returns the **properties** name/values pairs:

JSON

```
"createdOn": "2021-11-23T18:34:33.000Z",
"updatedOn": "2021-11-23T18:34:33.000Z",
"enabled": true,
"recoverableDays": 90,
"recoveryLevel": "Recoverable+Purgeable",
"id": "https://YOUR-KEYVAULT-
NAME.vault.azure.net/secrets/secret1637692472606/YOUR-VERSION",
"vaultUrl": "https://YOUR-KEYVAULT-NAME.vault.azure.net",
"version": "YOUR-VERSION",
"name": "secret1637692472606"
```

## Integrating with App Configuration

The Azure SDK provides a helper method, `parseKeyVaultSecretIdentifier`, to parse the given Key Vault Secret ID. This is necessary if you use [App Configuration](#) references to Key Vault. App Config stores the Key Vault Secret ID. You need the `parseKeyVaultSecretIdentifier` method to parse that ID to get the secret name. Once you have the secret name, you can get the current secret value using code from this quickstart.

## Next steps

In this quickstart, you created a key vault, stored a secret, and retrieved that secret. To learn more about Key Vault and how to integrate it with your applications, continue on

to the articles below.

- Read an [Overview of Azure Key Vault](#)
- Read an [Overview of Azure Key Vault Secrets](#)
- How to [Secure access to a key vault](#)
- See the [Azure Key Vault developer's guide](#)
- Review the [Key Vault security overview](#)

# Send messages to and receive messages from Azure Service Bus queues (JavaScript)

Article • 12/08/2023

In this tutorial, you complete the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus queue, using the Azure portal.
3. Write a JavaScript application to use the [@azure/service-bus](#) package to:
  - a. Send a set of messages to the queue.
  - b. Receive those messages from the queue.

## ⚠ Note

This quick start provides step-by-step instructions for a simple scenario of sending messages to a Service Bus queue and receiving them. You can find pre-built JavaScript and TypeScript samples for Azure Service Bus in the [Azure SDK for JavaScript repository on GitHub](#).

## Prerequisites

If you're new to the service, see [Service Bus overview](#) before you do this quickstart.

- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign-up for a [free account](#).
- [Node.js LTS](#)

### Passwordless

To use this quickstart with your own Azure account, you need:

- Install [Azure CLI](#), which provides the passwordless authentication to your developer machine.
- Sign in with your Azure account at the terminal or command prompt with `az login`.
- Use the same account when you add the appropriate data role to your resource.

- Run the code in the same terminal or command prompt.
- Note down your **queue** name for your Service Bus namespace. You'll need that in the code.

### ⓘ Note

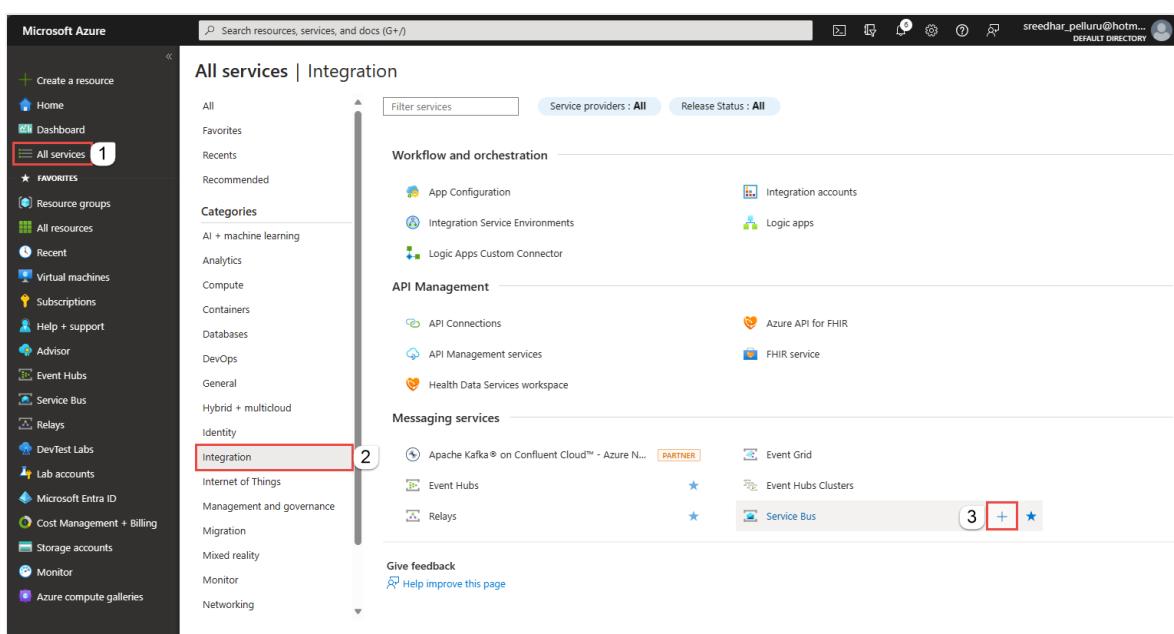
This tutorial works with samples that you can copy and run using Nodejs  [↗](#). For instructions on how to create a Node.js application, see [Create and deploy a Node.js application to an Azure Website](#), or [Node.js cloud service using Windows PowerShell](#).

## Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.

To create a namespace:

1. Sign in to the [Azure portal](#)  [↗](#).
2. In the left navigation pane of the portal, select **All services**, select **Integration** from the list of categories, hover the mouse over **Service Bus**, and then select + button on the Service Bus tile.



3. In the **Basics** tag of the [Create namespace](#) page, follow these steps:

- a. For **Subscription**, choose an Azure subscription in which to create the namespace.
- b. For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
- c. Enter a **name for the namespace**. The namespace name should adhere to the following naming conventions:
  - The name must be unique across Azure. The system immediately checks to see if the name is available.
  - The name length is at least 6 and at most 50 characters.
  - The name can contain only letters, numbers, hyphens “-”.
  - The name must start with a letter and end with a letter or number.
  - The name doesn't end with “-sb” or “-mgmt”.
- d. For **Location**, choose the region in which your namespace should be hosted.
- e. For **Pricing tier**, select the pricing tier (Basic, Standard, or Premium) for the namespace. For this quickstart, select **Standard**.

 **Important**

If you want to use **topics and subscriptions**, choose either Standard or Premium. Topics/subscriptions aren't supported in the Basic pricing tier.

If you selected the **Premium** pricing tier, specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, 4, 8 or 16 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

- f. Select **Review + create** at the bottom of the page.

 **Create namespace** ...

Service Bus

Basics Advanced Networking Tags Review + create

**Project Details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* Visual Studio Enterprise Subscription

Resource group \* (New) spsbusrg [Create new](#)

**Instance Details**

Enter required settings for this namespace.

Namespace name \* contosoordersns .servicebus.windows.net

Location \* East US

Pricing tier \* Standard [Browse the available plans and their features](#)

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

g. On the **Review + create** page, review settings, and select **Create**.

4. Once the deployment of the resource is successful, select **Go to resource** on the deployment page.

 **contosoordersns | Overview** ⚡ ...

Deployment

Search [Delete](#) [Cancel](#) [Redeploy](#) [Download](#) [Refresh](#)

**Overview**  Your deployment is complete

Deployment name: contosoordersns  
Subscription: Visual Studio Enterprise Subscription  
Resource group: spsbusrg

Start time: 10/20/2022, 4:45:03 PM  
Correlation ID: a453ace1-bab9-4c4a-81ad-a1c5366460ea [Copy](#)

[Deployment details](#) [Next steps](#)

[Go to resource](#)

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

5. You see the home page for your service bus namespace.

The screenshot shows the Azure Service Bus Namespace overview page for the resource group 'spsbusrg'. It displays various metrics and configurations. Key details include:

- Resource group (move):** spsbusrg
- Status:** Active
- Location:** East US
- Subscription (move):** Visual Studio Enterprise Subscription
- Subscription ID:** 0000000000-0000-0000-0000-000000000000
- Tags:** Add tags

Metrics shown in the Requests and Messages sections indicate zero activity over the last hour. A timeline at the bottom shows data from 5 PM UTC-05:00 to 5:15 PM UTC-05:00.

## Create a queue in the Azure portal

1. On the Service Bus Namespace page, select **Queues** in the left navigational menu.
2. On the Queues page, select **+ Queue** on the toolbar.
3. Enter a **name** for the queue, and leave the other values with their defaults.
4. Now, select **Create**.

The screenshot shows the Azure Service Bus Namespace Queues page for the 'contososbusns' namespace. A red box labeled '1' highlights the 'Queues' link in the left sidebar. The 'Create queue' dialog is open on the right, showing the following configuration:

- Name:** myqueue (highlighted with a red box labeled '3')
- Max queue size:** 1 GB
- Max delivery count:** 10
- Message time to live:** 14 Days, 0 Hours, 0 Minutes, 0 Seconds
- Lock duration:** 0 Days, 0 Hours, 0 Minutes, 30 Seconds
- Checkboxes (unchecked):** Enable auto-delete on idle queue, Enable duplicate detection, Enable dead lettering on message expiration, Enable partitioning, Enable sessions, Forward messages to queue/topic

A red box labeled '4' highlights the 'Create' button at the bottom of the dialog.

# Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Service Bus: **passwordless** and **connection string**.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace. You don't need to worry about having hard-coded connection string in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a connection string to connect to a Service Bus namespace. If you are new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

## Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Service Bus has the correct permissions. You'll need the [Azure Service Bus Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Service Bus Data Owner` role to your user account, which provides full access to Azure Service Bus resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

## Azure built-in roles for Azure Service Bus

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters). A member of this role can send and receive messages from queues or topics/subscriptions.
- [Azure Service Bus Data Sender](#): Use this role to give the send access to Service Bus namespace and its entities.
- [Azure Service Bus Data Receiver](#): Use this role to give the receive access to Service Bus namespace and its entities.

If you want to create a custom role, see [Rights required for Service Bus operations](#).

## Add Microsoft Entra user to Azure Service Bus Owner role

Add your Microsoft Entra user name to the **Azure Service Bus Data Owner** role at the Service Bus namespace level. It will allow an app running in the context of your user account to send messages to a queue or a topic, and receive messages from a queue or a topic's subscription.

### Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. If you don't have the Service Bus Namespace page open in the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Service Bus Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Use Node Package Manager (NPM) to install the package

Passwordless

1. To install the required npm package(s) for Service Bus, open a command prompt that has **npm** in its path, change the directory to the folder where you want to have your samples and then run this command.
2. Install the following packages:

Bash

```
npm install @azure/service-bus @azure/identity
```

## Send messages to a queue

The following sample code shows you how to send a message to a queue.

Passwordless

You must have signed in with the Azure CLI's `az login` in order for your local machine to provide the passwordless authentication required in this code.

1. Open your favorite editor, such as [Visual Studio Code](#).
2. Create a file called `send.js` and paste the below code into it. This code sends the names of scientists as messages to your queue.

### Important

The passwordless credential is provided with the [DefaultAzureCredential](#).

JavaScript

```
const { ServiceBusClient } = require("@azure/service-bus");
const { DefaultAzureCredential } = require("@azure/identity");

// Replace `<SERVICE-BUS-NAMESPACE>` with your namespace
const fullyQualifiedNamespace = "<SERVICE-BUS-
NAMESPACE>.servicebus.windows.net";

// Passwordless credential
const credential = new DefaultAzureCredential();

// name of the queue
const queueName = "<QUEUE NAME>

const messages = [
 { body: "Albert Einstein" },
 { body: "Werner Heisenberg" },
 { body: "Marie Curie" },
 { body: "Steven Hawking" },
 { body: "Isaac Newton" },
```

```
{ body: "Niels Bohr" },
{ body: "Michael Faraday" },
{ body: "Galileo Galilei" },
{ body: "Johannes Kepler" },
{ body: "Nikolaus Kopernikus" }
];

async function main() {
 // create a Service Bus client using the passwordless
 authentication to the Service Bus namespace
 const sbClient = new ServiceBusClient(fullyQualifiedNamespace,
 credential);

 // createSender() can also be used to create a sender for a
 topic.
 const sender = sbClient.createSender(queueName);

 try {
 // Tries to send all messages in a single batch.
 // Will fail if the messages cannot fit in a batch.
 // await sender.sendMessages(messages);

 // create a batch object
 let batch = await sender.createMessageBatch();
 for (let i = 0; i < messages.length; i++) {
 // for each message in the array

 // try to add the message to the batch
 if (!batch.tryAddMessage(messages[i])) {
 // if it fails to add the message to the current
batch
 // send the current batch as it is full
 await sender.sendMessages(batch);

 // then, create a new batch
 batch = await sender.createMessageBatch();

 // now, add the message failed to be added to the
previous batch to this batch
 if (!batch.tryAddMessage(messages[i])) {
 // if it still can't be added to the batch, the
message is probably too big to fit in a batch
 throw new Error("Message too big to fit in a
batch");
 }
 }
 }

 // Send the last created batch of messages to the queue
 await sender.sendMessages(batch);

 console.log(`Sent a batch of messages to the queue:
${queueName}`);
 }

 // Close the sender
}
```

```
 await sender.close();
 } finally {
 await sbClient.close();
 }
}

// call the main function
main().catch((err) => {
 console.log("Error occurred: ", err);
 process.exit(1);
});
```

3. Replace <SERVICE-BUS-NAMESPACE> with your Service Bus namespace.

4. Replace <QUEUE NAME> with the name of the queue.

5. Then run the command in a command prompt to execute this file.

Console

```
node send.js
```

6. You should see the following output.

Console

```
Sent a batch of messages to the queue: myqueue
```

## Receive messages from a queue

Passwordless

You must have signed in with the Azure CLI's `az login` in order for your local machine to provide the passwordless authentication required in this code.

1. Open your favorite editor, such as [Visual Studio Code](#)

2. Create a file called `receive.js` and paste the following code into it.

JavaScript

```
const { delay, ServiceBusClient, ServiceBusMessage } =
require("@azure/service-bus");
const { DefaultAzureCredential } = require("@azure/identity");
```

```

// Replace `<SERVICE-BUS-NAMESPACE>` with your namespace
const fullyQualifiedNamespace = "<SERVICE-BUS-

NAMESPACE>.servicebus.windows.net";

// Passwordless credential

const credential = new DefaultAzureCredential();

// name of the queue

const queueName = "<QUEUE NAME>"

async function main() {

 // create a Service Bus client using the passwordless

 authentication to the Service Bus namespace

 const sbClient = new ServiceBusClient(fullyQualifiedNamespace,

 credential);

 // createReceiver() can also be used to create a receiver for a

 subscription.

 const receiver = sbClient.createReceiver(queueName);

 // function to handle messages

 const myMessageHandler = async (messageReceived) => {

 console.log(`Received message: ${messageReceived.body}`);

 };

 // function to handle any errors

 const myErrorHandler = async (error) => {

 console.log(error);

 };

 // subscribe and specify the message and error handlers

 receiver.subscribe({

 processMessage: myMessageHandler,

 processError: myErrorHandler

 });

 // Waiting long enough before closing the sender to send

 messages

 await delay(20000);

 await receiver.close();

 await sbClient.close();

}

// call the main function

main().catch((err) => {

 console.log("Error occurred: ", err);

 process.exit(1);

});
```

3. Replace <SERVICE-BUS-NAMESPACE> with your Service Bus namespace.

4. Replace <QUEUE NAME> with the name of the queue.

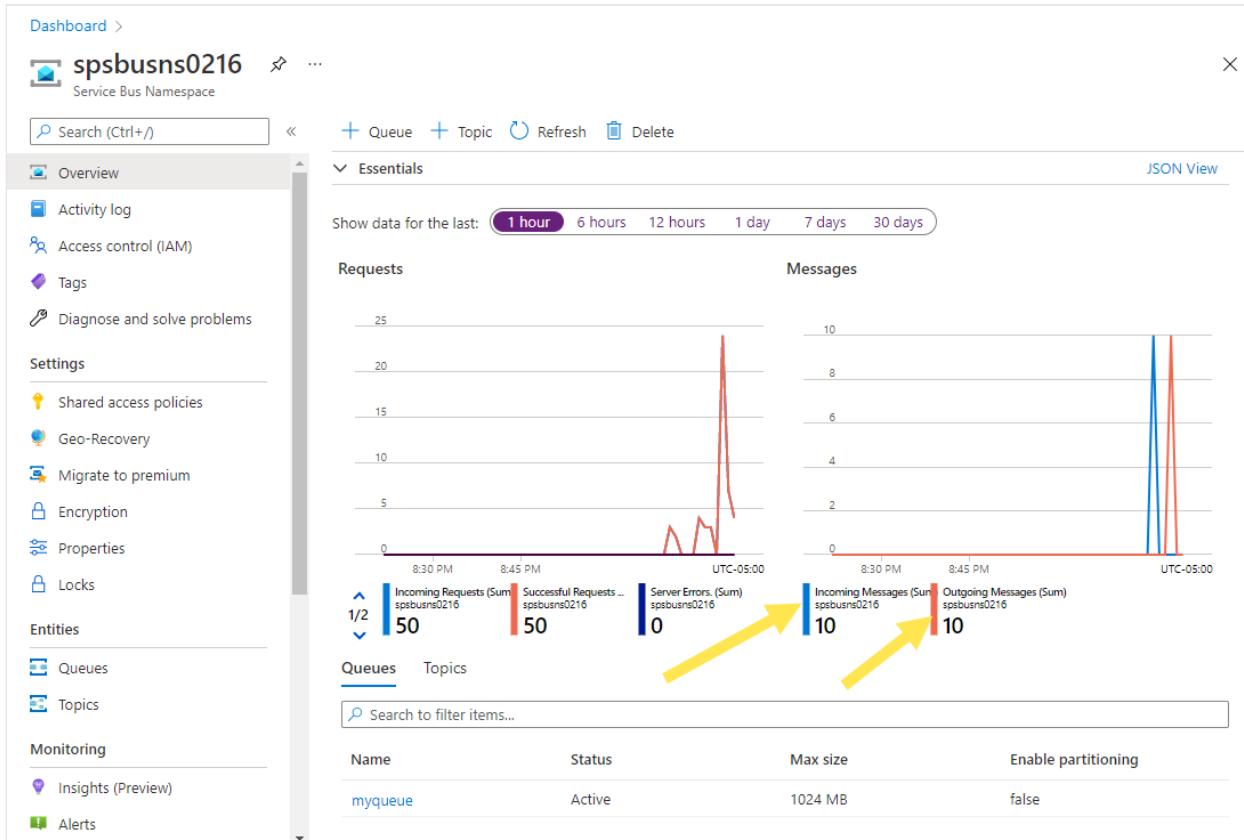
5. Then run the command in a command prompt to execute this file.

```
Console
node receive.js
```

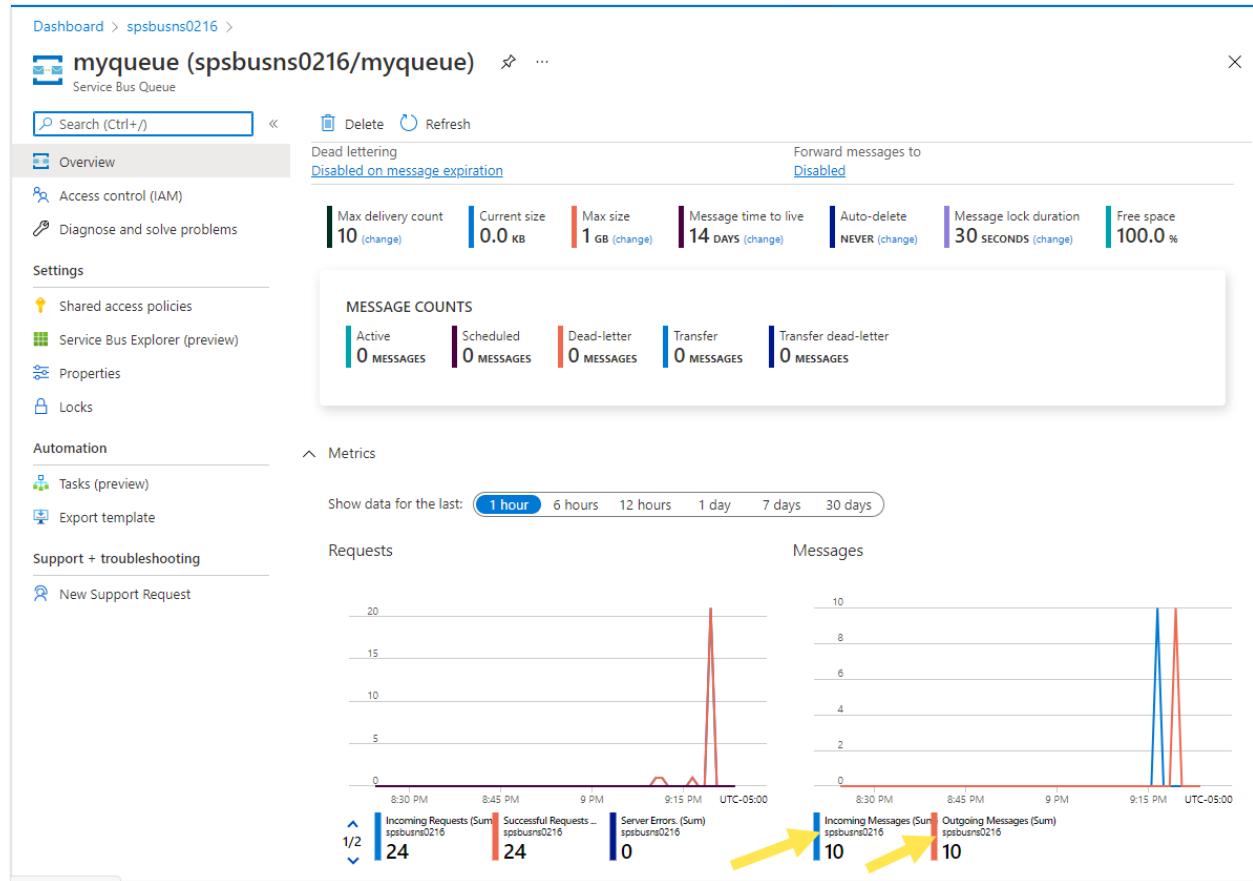
You should see the following output.

```
Console
Received message: Albert Einstein
Received message: Werner Heisenberg
Received message: Marie Curie
Received message: Steven Hawking
Received message: Isaac Newton
Received message: Niels Bohr
Received message: Michael Faraday
Received message: Galileo Galilei
Received message: Johannes Kepler
Received message: Nikolaus Kopernikus
```

On the **Overview** page for the Service Bus namespace in the Azure portal, you can see **incoming** and **outgoing** message count. You may need to wait for a minute or so and then refresh the page to see the latest values.



Select the queue on this **Overview** page to navigate to the **Service Bus Queue** page. You see the **incoming** and **outgoing** message count on this page too. You also see other information such as the **current size** of the queue, **maximum size**, **active message count**, and so on.



## Troubleshooting

If you receive one of the following errors when running the **passwordless** version of the JavaScript code, make sure you are signed in via the Azure CLI command, `az login` and the **appropriate role** is applied to your Azure user account:

- 'Send' claim(s) are required to perform this operation
- 'Receive' claim(s) are required to perform this operation

## Clean up resources

Navigate to your Service Bus namespace in the Azure portal, and select **Delete** on the Azure portal to delete the namespace and the queue in it.

## Next steps

See the following documentation and samples:

- [Azure Service Bus client library for JavaScript](#) ↗
- [JavaScript samples](#) ↗
- [TypeScript samples](#) ↗
- [API reference documentation](#)

# Send messages to an Azure Service Bus topic and receive messages from subscriptions to the topic (JavaScript)

Article • 12/08/2023

In this tutorial, you complete the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus topic, using the Azure portal.
3. Create a Service Bus subscription to that topic, using the Azure portal.
4. Write a JavaScript application to use the [@azure/service-bus](#) package to:
  - Send a set of messages to the topic.
  - Receive those messages from the subscription.

## ⓘ Note

This quick start provides step-by-step instructions for a simple scenario of sending a batch of messages to a Service Bus topic and receiving those messages from a subscription of the topic. You can find pre-built JavaScript and TypeScript samples for Azure Service Bus in the [Azure SDK for JavaScript repository on GitHub](#).

## Prerequisites

- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
- [Node.js LTS](#)
- Follow steps in the [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions to the topic](#). You will use only one subscription for this quickstart.

Passwordless

To use this quickstart with your own Azure account, you need:

- Install [Azure CLI](#), which provides the passwordless authentication to your developer machine.

- Sign in with your Azure account at the terminal or command prompt with `az login`.
- Use the same account when you add the appropriate role to your resource.
- Run the code in the same terminal or command prompt.
- Note down your **topic** name and **subscription** for your Service Bus namespace. You'll need that in the code.

### ⓘ Note

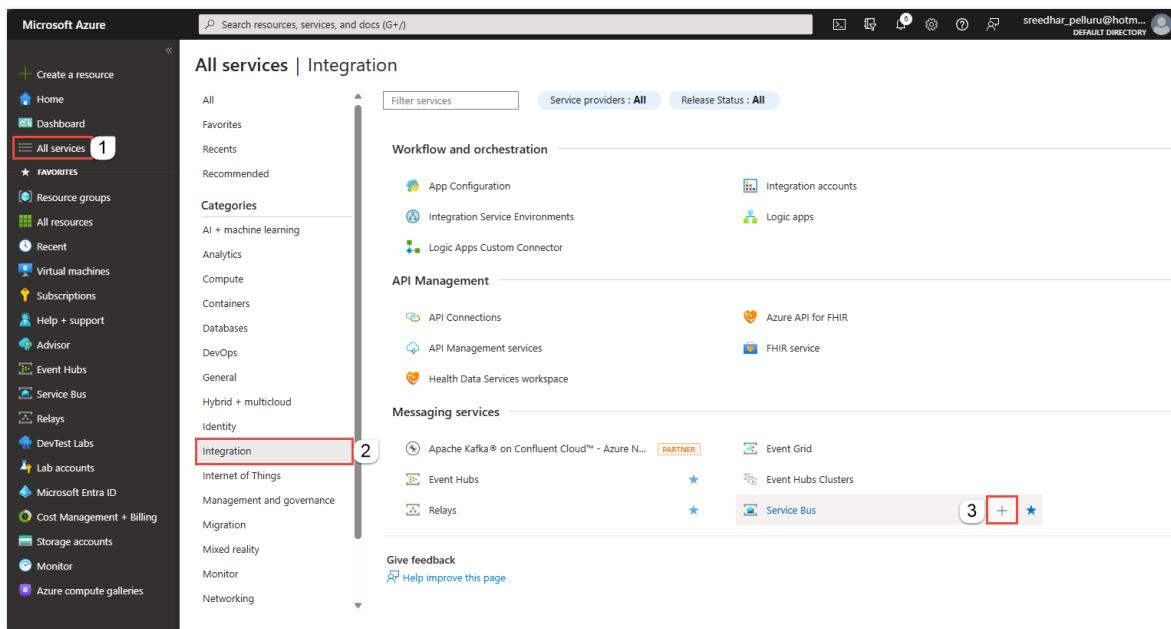
- This tutorial works with samples that you can copy and run using [Nodejs](#). For instructions on how to create a Node.js application, see [Create and deploy a Node.js application to an Azure Website](#), or [Node.js Cloud Service using Windows PowerShell](#).

## Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.

To create a namespace:

1. Sign in to the [Azure portal](#).
2. In the left navigation pane of the portal, select **All services**, select **Integration** from the list of categories, hover the mouse over **Service Bus**, and then select + button on the Service Bus tile.



3. In the **Basics** tag of the **Create namespace** page, follow these steps:

- For **Subscription**, choose an Azure subscription in which to create the namespace.
- For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
- Enter a **name for the namespace**. The namespace name should adhere to the following naming conventions:
  - The name must be unique across Azure. The system immediately checks to see if the name is available.
  - The name length is at least 6 and at most 50 characters.
  - The name can contain only letters, numbers, hyphens “-”.
  - The name must start with a letter and end with a letter or number.
  - The name doesn't end with “-sb” or “-mgmt”.
- For **Location**, choose the region in which your namespace should be hosted.
- For **Pricing tier**, select the pricing tier (Basic, Standard, or Premium) for the namespace. For this quickstart, select **Standard**.

### Important

If you want to use **topics and subscriptions**, choose either Standard or Premium. Topics/subscriptions aren't supported in the Basic pricing tier.

If you selected the **Premium** pricing tier, specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, 4, 8 or 16 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

- f. Select **Review + create** at the bottom of the page.

The screenshot shows the 'Create namespace' wizard in the Azure portal. The title bar says 'Create namespace' with a 'Service Bus' icon. Below it, tabs are labeled 'Basics', 'Advanced', 'Networking', 'Tags', and 'Review + create'. The 'Basics' tab is selected. The 'Project Details' section asks to select a subscription and resource group. The 'Subscription' dropdown shows 'Visual Studio Enterprise Subscription'. The 'Resource group' dropdown shows '(New) spsbusrg' with a 'Create new' link. The 'Instance Details' section asks for a namespace name, location, and pricing tier. The 'Namespace name' field contains 'contosoorderrsns' with '.servicebus.windows.net' suffix. The 'Location' dropdown shows 'East US'. The 'Pricing tier' dropdown shows 'Standard' with a 'Browse the available plans and their features' link. At the bottom, buttons include 'Review + create' (highlighted in blue), '< Previous', and 'Next: Advanced >'.

- g. On the **Review + create** page, review settings, and select **Create**.
4. Once the deployment of the resource is successful, select **Go to resource** on the deployment page.

The screenshot shows the Azure Deployment Overview page for a deployment named 'contosoordersns'. The status is 'Your deployment is complete'. Deployment details include a name 'contosoordersns', a subscription 'Visual Studio Enterprise Subscription', and a resource group 'spsbusrg'. The start time was 10/20/2022, 4:45:03 PM, and the correlation ID is a453ace1-bab9-4c4a-81ad-a1c5366460ea. A 'Go to resource' button is highlighted in red.

5. You see the home page for your service bus namespace.

The screenshot shows the Azure Service Bus Namespace overview page for 'spsbusns1128'. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Geo-Recovery, Migrate to premium, Encryption, Configuration, Properties, Locks), Entities (Queues, Topics), Monitoring (Insights (Preview), Alerts), and Queues (0) / Topics (0). The main area displays 'Essentials' information such as Resource group (spsbusrg), Status (Active), Location (East US), Subscription (Visual Studio Enterprise Subscription), Subscription ID, Tags, and various metrics for Requests and Messages over the last hour.

## Create a topic using the Azure portal

1. On the Service Bus Namespace page, select **Topics** on the left menu.
2. Select **+ Topic** on the toolbar.
3. Enter a name for the topic. Leave the other options with their default values.
4. Select **Create**.

All services > Resource groups > spsbusrg > spsbusns1128

spsbusns1128 | Topics

Service Bus Namespace

Search  2 + Topic Refresh Give feedback

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings Shared access policies Geo-Recovery Migrate to premium Encryption Configuration Properties Locks

Entities Queues **Topics** 1 Topics

Monitoring Insights (Preview) Alerts

Create 4 Give feedback

## Create a subscription to the topic

1. Select the **topic** that you created in the previous section.

spsbusns1128 | Topics

Service Bus Namespace

Search + Topic Refresh Give feedback

Properties Locks

Entities Queues Topics

Monitoring Insights (Preview) Alerts Metrics Diagnostic settings Logs Workbooks

Name	Status	Scheduled messages	Max size	Subscription count	Enable partitioning
mytopic	Active	0	1024 MB	0	false

2. On the **Service Bus Topic** page, select **+ Subscription** on the toolbar.

3. On the **Create subscription** page, follow these steps:

- a. Enter **S1** for **name** of the subscription.
- b. Enter **3** for **Max delivery count**.
- c. Then, select **Create** to create the subscription.

## Create subscription

Service Bus

Name \* ⓘ

S1



Max delivery count \* ⓘ

10

Auto-delete after idle for ⓘ

Days

Hours

Minutes

Seconds

14

0

0

0

Never auto-delete

Forward messages to queue/topic ⓘ

### MESSAGE SESSIONS

Service bus sessions allow ordered handling of unbounded sequences of related messages. With sessions enabled a subscription can guarantee first-in-first-out delivery of messages. [Learn more.](#)

Enable sessions

### MESSAGE TIME TO LIVE AND DEAD-LETTERING

Message time to live (default) ⓘ

Days

Hours

Minutes

Seconds

14

0

0

0

Enable dead lettering on message expiration

Move messages that cause filter evaluation exceptions to the dead-letter subqueue

### MESSAGE LOCK DURATION

Lock duration ⓘ

Days

Hours

Minutes

Seconds

0

0

1

0

**Create**

## Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Service Bus: **passwordless** and **connection string**.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace. You don't

need to worry about having hard-coded connection string in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a connection string to connect to a Service Bus namespace. If you are new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

## Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Service Bus has the correct permissions. You'll need the [Azure Service Bus Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Service Bus Data Owner` role to your user account, which provides full access to Azure Service Bus resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

## Azure built-in roles for Azure Service Bus

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters). A member of this role can send and receive messages from queues or topics/subscriptions.
- [Azure Service Bus Data Sender](#): Use this role to give the send access to Service Bus namespace and its entities.

- **Azure Service Bus Data Receiver:** Use this role to give the receive access to Service Bus namespace and its entities.

If you want to create a custom role, see [Rights required for Service Bus operations](#).

## Add Microsoft Entra user to Azure Service Bus Owner role

Add your Microsoft Entra user name to the **Azure Service Bus Data Owner** role at the Service Bus namespace level. It will allow an app running in the context of your user account to send messages to a queue or a topic, and receive messages from a queue or a topic's subscription.

### **Important**

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. If you don't have the Service Bus Namespace page open in the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Service Bus Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Use Node Package Manager (NPM) to install the package

Passwordless

1. To install the required npm package(s) for Service Bus, open a command prompt that has **npm** in its path, change the directory to the folder where you want to have your samples and then run this command.
2. Install the following packages:

Bash

```
npm install @azure/service-bus @azure/identity
```

## Send messages to a topic

The following sample code shows you how to send a batch of messages to a Service Bus topic. See code comments for details.

Passwordless

You must have signed in with the Azure CLI's `az login` in order for your local machine to provide the passwordless authentication required in this code.

1. Open your favorite editor, such as [Visual Studio Code](#)
2. Create a file called `sendtotopic.js` and paste the below code into it. This code will send a message to your topic.

### Important

The passwordless credential is provided with the [DefaultAzureCredential](#).

JavaScript

```
const { ServiceBusClient } = require("@azure/service-bus");
const { DefaultAzureCredential } = require("@azure/identity");

// Replace `<SERVICE-BUS-NAMESPACE>` with your namespace
const fullyQualifiedNamespace = "<SERVICE-BUS-
NAMESPACE>.servicebus.windows.net";

// Passwordless credential
const credential = new DefaultAzureCredential();

const topicName = "<TOPIC NAME>";

const messages = [
 { body: "Albert Einstein" },
 { body: "Werner Heisenberg" },
 { body: "Marie Curie" },
 { body: "Steven Hawking" },
 { body: "Isaac Newton" },
```

```
 { body: "Niels Bohr" },
 { body: "Michael Faraday" },
 { body: "Galileo Galilei" },
 { body: "Johannes Kepler" },
 { body: "Nikolaus Kopernikus" }
];

 async function main() {
 // create a Service Bus client using the passwordless
 authentication to the Service Bus namespace
 const sbClient = new ServiceBusClient(fullyQualifiedNamespace,
 credential);

 // createSender() can also be used to create a sender for a
 queue.
 const sender = sbClient.createSender(topicName);

 try {
 // Tries to send all messages in a single batch.
 // Will fail if the messages cannot fit in a batch.
 // await sender.sendMessages(messages);

 // create a batch object
 let batch = await sender.createMessageBatch();
 for (let i = 0; i < messages.length; i++) {
 // for each message in the array

 // try to add the message to the batch
 if (!batch.tryAddMessage(messages[i])) {
 // if it fails to add the message to the current
batch
 // send the current batch as it is full
 await sender.sendMessages(batch);

 // then, create a new batch
 batch = await sender.createMessageBatch();

 // now, add the message failed to be added to the
previous batch to this batch
 if (!batch.tryAddMessage(messages[i])) {
 // if it still can't be added to the batch, the
message is probably too big to fit in a batch
 throw new Error("Message too big to fit in a
batch");
 }
 }
 }

 // Send the last created batch of messages to the topic
 await sender.sendMessages(batch);

 console.log(`Sent a batch of messages to the topic:
${topicName}`);
 }

 // Close the sender
 }
}
```

```
 await sender.close();
 } finally {
 await sbClient.close();
 }
}

// call the main function
main().catch((err) => {
 console.log("Error occurred: ", err);
 process.exit(1);
});
```

3. Replace <SERVICE BUS NAMESPACE CONNECTION STRING> with the connection string to your Service Bus namespace.
4. Replace <TOPIC NAME> with the name of the topic.
5. Then run the command in a command prompt to execute this file.

Console

```
node sendtotoTopic.js
```

6. You should see the following output.

Console

```
Sent a batch of messages to the topic: mytopic
```

## Receive messages from a subscription

Passwordless

You must have signed in with the Azure CLI's `az login` in order for your local machine to provide the passwordless authentication required in this code.

1. Open your favorite editor, such as [Visual Studio Code](#) ↗
2. Create a file called `receivefromsubscription.js` and paste the following code into it. See code comments for details.

JavaScript

```
const { delay, ServiceBusClient, ServiceBusMessage } =
require("@azure/service-bus");
const { DefaultAzureCredential } = require("@azure/identity");

// Replace `<SERVICE-BUS-NAMESPACE>` with your namespace
const fullyQualifiedNamespace = "<SERVICE-BUS-
NAMESPACE>.servicebus.windows.net";

// Passwordless credential
const credential = new DefaultAzureCredential();

const topicName = "<TOPIC NAME>";
const subscriptionName = "<SUBSCRIPTION NAME>";

async function main() {
 // create a Service Bus client using the passwordless
 authentication to the Service Bus namespace
 const sbClient = new ServiceBusClient(fullyQualifiedNamespace,
 credential);

 // createReceiver() can also be used to create a receiver for a
 queue.
 const receiver = sbClient.createReceiver(topicName,
 subscriptionName);

 // function to handle messages
 const myMessageHandler = async (messageReceived) => {
 console.log(`Received message: ${messageReceived.body}`);
 };

 // function to handle any errors
 const myErrorHandler = async (error) => {
 console.log(error);
 };

 // subscribe and specify the message and error handlers
 receiver.subscribe({
 processMessage: myMessageHandler,
 processError: myErrorHandler
 });

 // Waiting long enough before closing the sender to send
 messages
 await delay(5000);

 await receiver.close();
 await sbClient.close();
}

// call the main function
main().catch((err) => {
 console.log("Error occurred: ", err);
```

```
 process.exit(1);
});
```

3. Replace <SERVICE BUS NAMESPACE CONNECTION STRING> with the connection string to the namespace.
4. Replace <TOPIC NAME> with the name of the topic.
5. Replace <SUBSCRIPTION NAME> with the name of the subscription to the topic.
6. Then run the command in a command prompt to execute this file.

Console

```
node receivefromsubscription.js
```

You should see the following output.

Console

```
Received message: Albert Einstein
Received message: Werner Heisenberg
Received message: Marie Curie
Received message: Steven Hawking
Received message: Isaac Newton
Received message: Niels Bohr
Received message: Michael Faraday
Received message: Galileo Galilei
Received message: Johannes Kepler
Received message: Nikolaus Kopernikus
```

In the Azure portal, navigate to your Service Bus namespace, switch to **Topics** in the bottom pane, and select your topic to see the **Service Bus Topic** page for your topic. On this page, you should see 10 incoming and 10 outgoing messages in the **Messages** chart.

**mytopic (spsbusns0216/mytopic)** Service Bus Topic

Search (Ctrl+ /) Subscription Delete Refresh

**Overview**

**Access control (IAM)** **Diagnose and solve problems**

**Settings**

- Shared access policies
- Service Bus Explorer (preview)
- Properties
- Locks

**Entities**

- Subscriptions

**Automation**

- Tasks (preview)
- Export template

**Support + troubleshooting**

- New Support Request

**Essentials**

Current size 0.0 kB	Max size 1 GB (change)	Message time to live 14 DAYS (change)	Auto-delete NEVER (change)	Free space 100.0 %
------------------------	---------------------------	------------------------------------------	-------------------------------	-----------------------

**Metrics**

Show data for the last: 1 hour 6 hours 12 hours 1 day 7 days 30 days

**Requests**

**Messages**

**Subscriptions**

Name	Status	Message count	Max delivery count	Client Scoped	Shared
S1	Active	0	3	No	N/A

If you run only the send app next time, on the **Service Bus Topic** page, you see 20 incoming messages (10 new) but 10 outgoing messages.

**Dashboard >** **mytopic (spsbusns0216/mytopic)** Service Bus Topic

Search (Ctrl+ /) Subscription Delete Refresh

**Overview**

**Access control (IAM)** **Diagnose and solve problems**

**Settings**

- Shared access policies
- Service Bus Explorer (preview)
- Properties
- Locks

**Entities**

- Subscriptions

**Automation**

- Tasks (preview)
- Export template

**Support + troubleshooting**

- New Support Request

**Essentials**

Current size 2.1 kB	Max size 1 GB (change)	Message time to live 14 DAYS (change)	Auto-delete NEVER (change)	Free space 100.0 %
------------------------	---------------------------	------------------------------------------	-------------------------------	-----------------------

**Metrics**

Show data for the last: 1 hour 6 hours 12 hours 1 day 7 days 30 days

**Requests**

**Messages**

**Subscriptions**

Name	Status	Message count	Max delivery count	Client Scoped	Shared
S1	Active	10	3	No	N/A

On this page, if you select a subscription in the bottom pane, you get to the **Service Bus Subscription** page. You can see the active message count, dead-letter message count,

and more on this page. In this example, there are 10 active messages that haven't been received by a receiver yet.

The screenshot shows the Azure Service Bus Subscription settings for a subscription named 'S1'. A yellow arrow points to the 'Active message count' which is displayed as '10 MESSAGES'. Other visible settings include 'Max delivery count' (3), 'Message time to live' (14 days), 'Message lock duration' (30 seconds), and 'Auto-delete after idle for' (14 days). The status bar at the bottom indicates '\$Default' and 'SqlFilter'.

## Troubleshooting

If you receive an error when running the **passwordless** version of the JavaScript code about required claims, make sure you are signed in via the Azure CLI command, `az login` and the [appropriate role](#) is applied to your Azure user account.

## Clean up resources

Navigate to your Service Bus namespace in the Azure portal, and select **Delete** on the Azure portal to delete the namespace and the queue in it.

## Next steps

See the following documentation and samples:

- [Azure Service Bus client library for JavaScript ↗](#)
- [JavaScript samples](#)
- [TypeScript samples](#)
- [API reference documentation](#)

# Quickstart: Azure Blob Storage client library for Node.js

Article • 03/06/2024

## ⓘ Note

The **Build from scratch** option walks you step by step through the process of creating a new project, installing packages, writing the code, and running a basic console app. This approach is recommended if you want to understand all the details involved in creating an app that connects to Azure Blob Storage. If you prefer to automate deployment tasks and start with a completed project, choose [Start with a template](#).

Get started with the Azure Blob Storage client library for Node.js to manage blobs and containers.

In this article, you follow steps to install the package and try out example code for basic tasks.

[API reference](#) | [Library source code](#) | [Package \(npm\)](#) | [Samples](#)

## Prerequisites

- Azure account with an active subscription - [create an account for free](#)
- Azure Storage account - [Create a storage account](#)
- [Node.js LTS](#)

## Setting up

This section walks you through preparing a project to work with the Azure Blob Storage client library for Node.js.

## Create the Node.js project

Create a JavaScript application named *blob-quickstart*.

1. In a console window (such as cmd, PowerShell, or Bash), create a new directory for the project:

Console

```
mkdir blob-quickstart
```

2. Switch to the newly created *blob-quickstart* directory:

Console

```
cd blob-quickstart
```

3. Create a *package.json* file:

Console

```
npm init -y
```

4. Open the project in Visual Studio Code:

Console

```
code .
```

## Install the packages

From the project directory, install the following packages using the `npm install` command.

1. Install the Azure Storage npm package:

Console

```
npm install @azure/storage-blob
```

2. Install the Azure Identity npm package for a passwordless connection:

Console

```
npm install @azure/identity
```

3. Install other dependencies used in this quickstart:

Console

```
npm install uuid dotenv
```

## Set up the app framework

From the project directory:

1. Create a new file named `index.js`
2. Copy the following code into the file:

JavaScript

```
const { BlobServiceClient } = require("@azure/storage-blob");
const { v1: uuidv1 } = require("uuid");
require("dotenv").config();

async function main() {
 try {
 console.log("Azure Blob storage v12 - JavaScript quickstart
sample");

 // Quick start code goes here

 } catch (err) {
 console.error(`Error: ${err.message}`);
 }
}

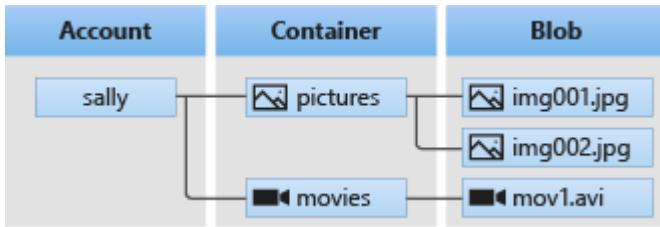
main()
 .then(() => console.log("Done"))
 .catch((ex) => console.log(ex.message));
```

## Object model

Azure Blob storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that doesn't adhere to a particular data model or definition, such as text or binary data. Blob storage offers three types of resources:

- The storage account
- A container in the storage account
- A blob in the container

The following diagram shows the relationship between these resources.



Use the following JavaScript classes to interact with these resources:

- **[BlobServiceClient](#)**: The `BlobServiceClient` class allows you to manipulate Azure Storage resources and blob containers.
- **[ContainerClient](#)**: The `ContainerClient` class allows you to manipulate Azure Storage containers and their blobs.
- **[BlobClient](#)**: The `BlobClient` class allows you to manipulate Azure Storage blobs.

## Code examples

These example code snippets show you how to do the following tasks with the Azure Blob Storage client library for JavaScript:

- [Authenticate to Azure and authorize access to blob data](#)
- [Create a container](#)
- [Upload blobs to a container](#)
- [List the blobs in a container](#)
- [Download blobs](#)
- [Delete a container](#)

Sample code is also available on [GitHub](#).

## Authenticate to Azure and authorize access to blob data

Application requests to Azure Blob Storage must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code, including Blob Storage.

You can also authorize requests to Azure Blob Storage by using the account access key. However, this approach should be used with caution. Developers must be diligent to never expose the access key in an unsecure location. Anyone who has the access key is able to authorize requests against the storage account, and effectively has access to all the data. `DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

## Passwordless (Recommended)

`DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` looks for credentials can be found in the [Azure Identity library overview](#).

For example, your app can authenticate using your Azure CLI sign-in credentials with when developing locally. Your app can then use a [managed identity](#) once it's deployed to Azure. No code changes are required for this transition.

## Assign roles to your Microsoft Entra user account

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

### Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'identitymigrationstorage | Access Control (IAM)' page. The left sidebar has a red box around the 'Access Control (IAM)' item. The top navigation bar has a red box around the '+ Add' button, which is highlighted in a dropdown menu. The dropdown menu also includes 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?' options. The main content area shows tabs for 'My access', 'Check access', and 'Add role assignment'. The 'Add role assignment' tab is selected, showing a 'Grant access to this resource' section with a 'Add role assignment' button and a 'Learn more' link. Below it is a 'View deny assignments' section with a 'View' button and a 'Learn more' link.

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Sign in and connect your app code to Azure using DefaultAzureCredential

You can authorize access to data in your storage account using the following steps:

1. Make sure you're authenticated with the same Microsoft Entra account you assigned the role to on your storage account. You can authenticate via the Azure CLI, Visual Studio Code, or Azure PowerShell.

```
Azure CLI
```

Sign-in to Azure through the Azure CLI using the following command:

```
Azure CLI
```

```
az login
```

2. To use `DefaultAzureCredential`, make sure that the `@azure\identity` package is [installed](#), and the class is imported:

```
JavaScript
```

```
const { DefaultAzureCredential } = require('@azure/identity');
```

3. Add this code inside the `try` block. When the code runs on your local workstation, `DefaultAzureCredential` uses the developer credentials of the prioritized tool you're logged into to authenticate to Azure. Examples of these tools include Azure CLI or Visual Studio Code.

```
JavaScript
```

```
const accountName = process.env.AZURE_STORAGE_ACCOUNT_NAME;
if (!accountName) throw Error('Azure Storage accountName not
found');

const blobServiceClient = new BlobServiceClient(
`https://${accountName}.blob.core.windows.net`,
new DefaultAzureCredential()
);
```

4. Make sure to update the storage account name, `AZURE_STORAGE_ACCOUNT_NAME`, in the `.env` file or your environment's variables. The storage account name can be found on the overview page of the Azure portal.

A screenshot of the Azure Storage account settings page for 'identitymigrationstorage'. The account name is at the top, followed by a storage icon. Below is a search bar and a navigation bar with options like Upload, Open in Explorer, Delete, Move, and Refresh. A sidebar on the left lists Overview, Activity log, Tags, Diagnose and solve problems, Access Control (IAM), Data migration, Events, and Storage browser. The main content area is titled 'Essentials' and contains the following information:

- Resource group ([move](#)) : alexw-identity-revamp
- Location : East US
- Primary/Secondary Location : Primary: East US, Secondary: West US
- Subscription ([move](#)) : C&L Cross Service Content Team Testing
- Subscription ID :
- Disk state : Primary: Available, Secondary: Available
- Tags ([edit](#)) : Click here to add tags

### ⓘ Note

When deployed to Azure, this same code can be used to authorize requests to Azure Storage from an application running in Azure. However, you'll need to enable managed identity on your app in Azure. Then configure your storage account to allow that managed identity to connect. For detailed instructions on configuring this connection between Azure services, see the [Auth from Azure-hosted apps](#) tutorial.

## Create a container

Create a new container in the storage account. The following code example takes a `BlobServiceClient` object and calls the `getContainerClient` method to get a reference to a container. Then, the code calls the `create` method to actually create the container in your storage account.

Add this code to the end of the `try` block:

JavaScript

```
// Create a unique name for the container
const containerName = 'quickstart' + uuidv1();

console.log('\nCreating container...');

// Get a reference to a container
const containerClient = blobServiceClient.getContainerClient(containerName);
// Create the container
```

```
const createContainerResponse = await containerClient.create();
console.log(
 `Container was created
successfully.\n\trequestId:${createContainerResponse.requestId}\n\tURL:
${containerClient.url}`
);
```

To learn more about creating a container, and to explore more code samples, see [Create a blob container with JavaScript](#).

 **Important**

Container names must be lowercase. For more information about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

## Upload blobs to a container

Upload a blob to the container. The following code gets a reference to a [BlockBlobClient](#) object by calling the [getBlockBlobClient](#) method on the [ContainerClient](#) from the [Create a container](#) section.

The code uploads the text string data to the blob by calling the [upload](#) method.

Add this code to the end of the `try` block:

JavaScript

```
// Create a unique name for the blob
const blobName = 'quickstart' + uuidv1() + '.txt';

// Get a block blob client
const blockBlobClient = containerClient.getBlockBlobClient(blobName);

// Display blob name and url
console.log(
 `\nUploading to Azure storage as blob\n\tname: ${blobName}:\n\tURL:
${blockBlobClient.url}`
);

// Upload data to the blob
const data = 'Hello, World!';
const uploadBlobResponse = await blockBlobClient.upload(data, data.length);
console.log(
 `Blob was uploaded successfully. requestId:
```

```
`${uploadBlobResponse.requestId}`
);
```

To learn more about uploading blobs, and to explore more code samples, see [Upload a blob with JavaScript](#).

## List the blobs in a container

List the blobs in the container. The following code calls the [listBlobsFlat](#) method. In this case, only one blob is in the container, so the listing operation returns just that one blob.

Add this code to the end of the `try` block:

```
JavaScript

console.log('\nListing blobs...');

// List the blob(s) in the container.
for await (const blob of containerClient.listBlobsFlat()) {
 // Get Blob Client from name, to get the URL
 const tempBlockBlobClient = containerClient.getBlockBlobClient(blob.name);

 // Display blob name and URL
 console.log(`\n\tname: ${blob.name}\n\tURL: ${tempBlockBlobClient.url}\n`
);
}
```

To learn more about listing blobs, and to explore more code samples, see [List blobs with JavaScript](#).

## Download blobs

Download the blob and display the contents. The following code calls the [download](#) method to download the blob.

Add this code to the end of the `try` block:

```
JavaScript

// Get blob content from position 0 to the end
// In Node.js, get downloaded data by accessing
downloadBlockBlobResponse.readableStreamBody
// In browsers, get downloaded data by accessing
downloadBlockBlobResponse.blobBody
```

```
const downloadBlockBlobResponse = await blockBlobClient.download(0);
console.log('\nDownloaded blob content...');
console.log(
 '\t',
 await streamToText(downloadBlockBlobResponse.readableStreamBody)
);
```

The following code converts a stream back into a string to display the contents.

Add this code *after* the `main` function:

JavaScript

```
// Convert stream to text
async function streamToText(readable) {
 readable.setEncoding('utf8');
 let data = '';
 for await (const chunk of readable) {
 data += chunk;
 }
 return data;
}
```

To learn more about downloading blobs, and to explore more code samples, see [Download a blob with JavaScript](#).

## Delete a container

Delete the container and all blobs within the container. The following code cleans up the resources created by the app by removing the entire container using the `delete` method.

Add this code to the end of the `try` block:

JavaScript

```
// Delete container
console.log('\nDeleting container...');

const deleteContainerResponse = await containerClient.delete();
console.log(
 'Container was deleted successfully. requestId: ',
 deleteContainerResponse.requestId
);
```

To learn more about deleting a container, and to explore more code samples, see [Delete and restore a blob container with JavaScript](#).

# Run the code

From a Visual Studio Code terminal, run the app.

```
Console

node index.js
```

The output of the app is similar to the following example:

```
Output

Azure Blob storage - JavaScript quickstart sample

Creating container...
 quickstart4a0780c0-fb72-11e9-b7b9-b387d3c488da

Uploading to Azure Storage as blob:
 quickstart4a3128d0-fb72-11e9-b7b9-b387d3c488da.txt

Listing blobs...
 quickstart4a3128d0-fb72-11e9-b7b9-b387d3c488da.txt

Downloaded blob content...
 Hello, World!

Deleting container...
Done
```

Step through the code in your debugger and check your [Azure portal](#) throughout the process. Check to see that the container is being created. You can open the blob inside the container and view the contents.

## Clean up resources

1. When you're done with this quickstart, delete the `blob-quickstart` directory.
2. If you're done using your Azure Storage resource, use the [Azure CLI to remove the Storage resource](#).

## Next steps

In this quickstart, you learned how to upload, download, and list blobs using JavaScript.

To see Blob storage sample apps, continue to:

## Azure Blob Storage library for JavaScript samples

- To learn more, see the [Azure Blob Storage client libraries for JavaScript](#).
- For tutorials, samples, quickstarts, and other documentation, visit [Azure for JavaScript and Node.js developers](#).

# Quickstart: Azure Queue Storage client library for JavaScript

Article • 06/29/2023

Get started with the Azure Queue Storage client library for JavaScript. Azure Queue Storage is a service for storing large numbers of messages for later retrieval and processing. Follow these steps to install the package and try out example code for basic tasks.

[API reference documentation](#) | [Library source code](#) | [Package \(npm\)](#) | [Samples](#)

Use the Azure Queue Storage client library for JavaScript to:

- Create a queue
- Add messages to a queue
- Peek at messages in a queue
- Update a message in a queue
- Get the queue length
- Receive messages from a queue
- Delete messages from a queue
- Delete a queue

## Prerequisites

- Azure subscription - [create one for free](#)
- Azure Storage account - [create a storage account](#)
- Current [Node.js](#) for your operating system.

## Setting up

This section walks you through preparing a project to work with the Azure Queue Storage client library for JavaScript.

## Create the project

Create a Node.js application named `queues-quickstart`.

1. In a console window (such as cmd, PowerShell, or Bash), create a new directory for the project:

Console

```
mkdir queues-quickstart
```

2. Switch to the newly created `queues-quickstart` directory:

Console

```
cd queues-quickstart
```

3. Create a `package.json` file:

Console

```
npm init -y
```

4. Open the project in Visual Studio Code:

Console

```
code .
```

## Install the packages

From the project directory, install the following packages using the `npm install` command.

1. Install the Azure Queue Storage npm package:

Console

```
npm install @azure/storage-queue
```

2. Install the Azure Identity npm package to support passwordless connections:

Console

```
npm install @azure/identity
```

3. Install other dependencies used in this quickstart:

Console

```
npm install uuid dotenv
```

## Set up the app framework

From the project directory:

1. Open a new text file in your code editor
2. Add `require` calls to load Azure and Node.js modules
3. Create the structure for the program, including basic exception handling

Here's the code:

JavaScript

```
const { QueueClient } = require("@azure/storage-queue");
const { DefaultAzureCredential } = require('@azure/identity');
const { v1: uuidv1 } = require("uuid");

async function main() {
 console.log("Azure Queue Storage client library - JavaScript
quickstart sample");

 // Quickstart code goes here
}

main().then(() => console.log("\nDone")).catch((ex) =>
 console.log(ex.message));
```

4. Save the new file as `index.js` in the `queues-quickstart` directory.

## Authenticate to Azure

Application requests to most Azure services must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code.

You can also authorize requests to Azure services using passwords, connection strings, or other credentials directly. However, this approach should be used with caution. Developers must be diligent to never expose these secrets in an unsecure location. Anyone who gains access to the password or secret key is able to authenticate.

`DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

#### Passwordless (Recommended)

`DefaultAzureCredential` is a class provided by the Azure Identity client library for JavaScript. To learn more about `DefaultAzureCredential`, see the [DefaultAzureCredential overview](#). `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

For example, your app can authenticate using your Azure CLI sign-in credentials when developing locally, and then use a [managed identity](#) once it has been deployed to Azure. No code changes are required for this transition.

When developing locally, make sure that the user account that is accessing the queue data has the correct permissions. You'll need **Storage Queue Data Contributor** to read and write queue data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Queue Data Contributor** role to your user account, which provides both read and write access to queue data in your storage account.

#### ⓘ Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the Azure portal interface for managing access to a storage account. The left sidebar has a red box around the 'Access Control (IAM)' item. The top navigation bar has a red box around the '+ Add' button, which is part of a dropdown menu that also includes 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. The 'Add role assignment' option in the dropdown is also highlighted with a red box. The main content area displays 'My access' (viewing level of access), 'Check access' (reviewing access for users, groups, service principals, or managed identities), and 'Grant access to this resource' (a section for adding role assignments). The 'Add role assignment' button is visible in this section.

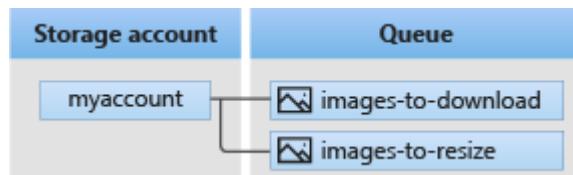
5. Use the search box to filter the results to the desired role. For this example, search for *Storage Queue Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

# Object model

Azure Queue Storage is a service for storing large numbers of messages. A queue message can be up to 64 KB in size. A queue may contain millions of messages, up to the total capacity limit of a storage account. Queues are commonly used to create a backlog of work to process asynchronously. Queue Storage offers three types of resources:

- **Storage account:** All access to Azure Storage is done through a storage account. For more information about storage accounts, see [Storage account overview](#)
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. A message can remain in the queue for a maximum of 7 days. For version 2017-07-29 or later, the maximum time-to-live can be any positive number, or -1 indicating that the message doesn't expire. If this parameter is omitted, the default time-to-live is seven days.

The following diagram shows the relationship between these resources.



Use the following JavaScript classes to interact with these resources:

- [QueueServiceClient](#): A `QueueServiceClient` instance represents a connection to a given storage account in the Azure Storage Queue service. This client allows you to manage the all queues in your storage account.
- [QueueClient](#): A `QueueClient` instance represents a single queue in a storage account. This client allows you to manage and manipulate an individual queue and its messages.

## Code examples

These example code snippets show you how to do the following actions with the Azure Queue Storage client library for JavaScript:

- [Authorize access and create a client object](#)
- [Create a queue](#)
- [Add messages to a queue](#)
- [Peek at messages in a queue](#)

- Update a message in a queue
- Get the queue length
- Receive messages from a queue
- Delete messages from a queue
- Delete a queue

Passwordless (Recommended)

## Authorize access and create a client object

Make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via Azure CLI, Visual Studio Code, or Azure PowerShell.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

Azure CLI

```
az login
```

Once authenticated, you can create and authorize a `QueueClient` object using

`DefaultAzureCredential` to access queue data in the storage account.

`DefaultAzureCredential` automatically discovers and uses the account you signed in with in the previous step.

To authorize using `DefaultAzureCredential`, make sure you've added the `@azure/identity` package, as described in [Install the packages](#). Also, be sure to load the `@azure/identity` module in the `index.js` file:

JavaScript

```
const { DefaultAzureCredential } = require('@azure/identity');
```

Decide on a name for the queue and create an instance of the `QueueClient` class, using `DefaultAzureCredential` for authorization. We use this client object to create and interact with the queue resource in the storage account.

 **Important**

Queue names may only contain lowercase letters, numbers, and hyphens, and must begin with a letter or a number. Each hyphen must be preceded and followed by a non-hyphen character. The name must also be between 3 and 63 characters long. For more information about naming queues, see [Naming queues and metadata](#).

Add the following code inside the `main` method, and make sure to replace the `<storage-account-name>` placeholder value:

JavaScript

```
// Create a unique name for the queue
const queueName = "quickstart" + uuidv1();

// Instantiate a QueueClient which will be used to create and interact
// with a queue
// TODO: replace <storage-account-name> with the actual name
const queueClient = new QueueClient(`https://<storage-account-
name>.queue.core.windows.net/${queueName}`, new
DefaultAzureCredential());
```

## ⓘ Note

Messages sent using the `QueueClient` class must be in a format that can be included in an XML request with UTF-8 encoding. To include markup in the message, the contents of the message must either be XML-escaped or Base64-encoded.

Queues messages are stored as strings. If you need to send a different data type, you must serialize that data type into a string when sending the message and deserialize the string format when reading the message.

To convert JSON to a string format and back again in Node.js, use the following helper functions:

JavaScript

```
function jsonToBase64(jsonObj) {
 const jsonString = JSON.stringify(jsonObj)
 return Buffer.from(jsonString).toString('base64')
}
function encodeBase64ToJson(base64String) {
 const jsonString = Buffer.from(base64String, 'base64').toString()
```

```
 return JSON.parse(jsonString)
}
```

## Create a queue

Using the `queueClient` object, call the `create` method to create the queue in your storage account.

Add this code to the end of the `main` method:

JavaScript

```
console.log("\nCreating queue...");
console.log("\t", queueName);

// Create the queue
const createQueueResponse = await queueClient.create();
console.log("Queue created, requestId:", createQueueResponse.requestId);
```

## Add messages to a queue

The following code snippet adds messages to queue by calling the `sendMessage` method. It also saves the `QueueSendMessageResponse` returned from the third `sendMessage` call. The returned `sendMessageResponse` is used to update the message content later in the program.

Add this code to the end of the `main` function:

JavaScript

```
console.log("\nAdding messages to the queue...");

// Send several messages to the queue
await queueClient.sendMessage("First message");
await queueClient.sendMessage("Second message");
const sendMessageResponse = await queueClient.sendMessage("Third message");

console.log("Messages added, requestId:", sendMessageResponse.requestId);
```

## Peek at messages in a queue

Peek at the messages in the queue by calling the `peekMessages` method. This method retrieves one or more messages from the front of the queue but doesn't alter the visibility of the message. By default, `peekMessages` peeks at a single message.

Add this code to the end of the `main` function:

JavaScript

```
console.log("\nPeek at the messages in the queue...");

// Peek at messages in the queue
const peekedMessages = await queueClient.peekMessages({ numberOfMessages : 5 });

for (i = 0; i < peekedMessages.peekedMessageItems.length; i++) {
 // Display the peeked message
 console.log("\t", peekedMessages.peekedMessageItems[i].messageText);
}
```

## Update a message in a queue

Update the contents of a message by calling the `updateMessage` method. This method can change a message's visibility timeout and contents. The message content must be a UTF-8 encoded string that is up to 64 KB in size. Along with the new content, pass in `messageId` and `popReceipt` from the response that was saved earlier in the code. The `sendMessageResponse` properties identify which message to update.

JavaScript

```
console.log("\nUpdating the third message in the queue...");

// Update a message using the response saved when calling sendMessage
earlier
updateMessageResponse = await queueClient.updateMessage(
 sendMessageResponse.messageId,
 sendMessageResponse.popReceipt,
 "Third message has been updated"
);

console.log("Message updated, requestId:", updateMessageResponse.requestId);
```

## Get the queue length

The `getProperties` method returns metadata about the queue, including the approximate number of messages waiting in the queue.

JavaScript

```
const properties = await queueClient.getProperties();
console.log("Approximate queue length: ",
```

```
properties.approximateMessagesCount);
```

## Receive messages from a queue

Download previously added messages by calling the [receiveMessages](#) method. In the `numberOfMessages` field, pass in the maximum number of messages to receive for this call.

Add this code to the end of the `main` function:

JavaScript

```
console.log("\nReceiving messages from the queue...");

// Get messages from the queue
const receivedMessagesResponse = await queueClient.receiveMessages({
 numberOfMessages : 5 });

console.log("Messages received, requestId:",
receivedMessagesResponse.requestId);
```

When calling the `receiveMessages` method, you can optionally specify values in [QueueReceiveMessageOptions](#) to customize message retrieval. You can specify a value for `numberOfMessages`, which is the number of messages to retrieve from the queue. The default is 1 message and the maximum is 32 messages. You can also specify a value for `visibilityTimeout`, which hides the messages from other operations for the timeout period. The default is 30 seconds.

## Delete messages from a queue

You can delete messages from the queue after they're received and processed. In this case, processing is just displaying the message on the console.

Delete messages by calling the [deleteMessage](#) method. Any messages not explicitly deleted eventually become visible in the queue again for another chance to process them.

Add this code to the end of the `main` function:

JavaScript

```
// 'Process' and delete messages from the queue
for (i = 0; i < receivedMessagesResponse.receivedMessageItems.length; i++) {
 receivedMessage = receivedMessagesResponse.receivedMessageItems[i];
```

```
// 'Process' the message
console.log("\tProcessing:", receivedMessage.messageText);

// Delete the message
const deleteMessageResponse = await queueClient.deleteMessage(
 receivedMessage.messageId,
 receivedMessage.popReceipt
);
console.log("\tMessage deleted, requestId:",
deleteMessageResponse.requestId);
}
```

## Delete a queue

The following code cleans up the resources the app created by deleting the queue using the [delete](#) method.

Add this code to the end of the `main` function and save the file:

JavaScript

```
// Delete the queue
console.log("\nDeleting queue...");
const deleteQueueResponse = await queueClient.delete();
console.log("Queue deleted, requestId:", deleteQueueResponse.requestId);
```

## Run the code

This app creates and adds three messages to an Azure queue. The code lists the messages in the queue, then retrieves and deletes them, before finally deleting the queue.

In your console window, navigate to the directory containing the `index.js` file, then use the following `node` command to run the app.

Console

```
node index.js
```

The output of the app is similar to the following example:

Output

## Azure Queue Storage client library - JavaScript quickstart sample

```
Creating queue...
```

```
 quickstart<UUID>
```

```
Queue created, requestId: 5c0bc94c-6003-011b-7c11-b13d06000000
```

```
Adding messages to the queue...
```

```
Messages added, requestId: a0390321-8003-001e-0311-b18f2c000000
```

```
Peek at the messages in the queue...
```

```
 First message
```

```
 Second message
```

```
 Third message
```

```
Updating the third message in the queue...
```

```
Message updated, requestId: cb172c9a-5003-001c-2911-b18dd6000000
```

```
Receiving messages from the queue...
```

```
Messages received, requestId: a039036f-8003-001e-4811-b18f2c000000
```

```
 Processing: First message
```

```
 Message deleted, requestId: 4a65b82b-d003-00a7-5411-b16c22000000
```

```
 Processing: Second message
```

```
 Message deleted, requestId: 4f0b2958-c003-0030-2a11-b10feb000000
```

```
 Processing: Third message has been updated
```

```
 Message deleted, requestId: 6c978fcb-5003-00b6-2711-b15b39000000
```

```
Deleting queue...
```

```
Queue deleted, requestId: 5c0bca05-6003-011b-1e11-b13d06000000
```

```
Done
```

Step through the code in your debugger and check your [Azure portal](#) throughout the process. Check your storage account to verify messages in the queue are created and deleted.

## Next steps

In this quickstart, you learned how to create a queue and add messages to it using JavaScript code. Then you learned to peek, retrieve, and delete messages. Finally, you learned how to delete a message queue.

For tutorials, samples, quick starts and other documentation, visit:

[Azure for JavaScript documentation](#)

- To learn more, see the [Azure Queue Storage client library for JavaScript](#).
- For more Azure Queue Storage sample apps, see [Azure Queue Storage client library for JavaScript - samples](#).

# Use Azure OpenAI without keys

Article • 05/20/2024

Application requests to most Azure services must be authenticated with keys or [passwordless connections](#). Developers must be diligent to never expose the keys in an unsecure location. Anyone who gains access to the key is able to authenticate to the service. Passwordless authentication offers improved management and security benefits over the account key because there's no key (or connection string) to store.

Passwordless connections are enabled with the following steps:

- Configure your authentication.
- Set environment variables, as needed.
- Use an Azure Identity library credential type to create an Azure OpenAI client object.

## Authentication

Authentication to Microsoft Entra ID is required to use the Azure client libraries.

Authentication differs based on the environment in which the app is running:

- [Local development](#)
- [Azure](#)

## Authenticate for local development

JavaScript

Select a tool for [authentication during local development](#).

## Authenticate for Azure-hosted environments

JavaScript

Learn about how to manage the [DefaultAzureCredential](#) for applications deployed to Azure.

# Configure roles for authorization

1. Find the [role](#) for your usage of Azure OpenAI. Depending on how you intend to set that role, you'll need either the name or ID.

[+] [Expand table](#)

Role name	Role ID
For Azure CLI or Azure PowerShell, you can use role name. For Bicep, you need the role ID.	

2. For many Azure OpenAI chat completion use cases, the following role and ID are suggested.

[+] [Expand table](#)

Role name	Role ID
Cognitive Services OpenAI User	5e0bd9bd-7b93-4f28-af87-19fc36ad61bd

3. Select an identity type to use.

- **Personal identity:** This is your personal identity tied to your sign in to Azure.
- **Managed identity:** This is an identity managed by and created for use on Azure. For [managed identity](#), create a [user-assigned managed identity](#). When you create the managed identity, you need the `Client ID`, also known as the `app ID`.

4. To find your personal identity, use one of the following commands. Use the ID as the `<identity-id>` in the next step.

Azure CLI

For local development, to get your own identity ID, use the following command. You need to sign in with `az login` before using this command.

Azure CLI

```
az ad signed-in-user show \
--query id -o tsv
```

5. Assign the role-based access control (RBAC) role to the identity for the resource group.

## Azure CLI

To grant your identity permissions to your resource through RBAC, assign a role using the Azure CLI command [az role assignment create](#).

### Azure CLI

```
az role assignment create \
 --role "Cognitive Services OpenAI User" \
 --assignee "<identity-id>" \
 --scope "/subscriptions/<subscription-
 id>/resourceGroups/<resource-group-name>"
```

Where applicable, replace `<identity-id>`, `<subscription-id>`, and `<resource-group-name>` with your actual values.

## Configure environment variables

To connect to Azure OpenAI, your code needs to know your resource endpoint, and *may* need additional environment variables.

1. Create an environment variable for your Azure OpenAI endpoint.

- `AZURE_OPENAI_ENDPOINT`: This URL is the access point for your Azure OpenAI resource.

2. Create environment variables based on the location in which your app runs:

[+] [Expand table](#)

Location	Identity	Description
Local	Personal	For local runtimes with your <a href="#">personal identity</a> , <a href="#">sign in</a> to create your credential with a tool.
Azure cloud	User-assigned managed identity	Create an <code>AZURE_CLIENT_ID</code> environment variable containing the client ID of the user-assigned managed identity to authenticate as.

## Install Azure Identity client library

### JavaScript

Install the JavaScript [Azure Identity client library](#):

Console

```
npm install --save @azure/identity
```

## Use DefaultAzureCredential

The Azure Identity library's `DefaultAzureCredential` allows the customer to run the same code in the local development environment and in the Azure Cloud.

JavaScript

For more information on `DefaultAzureCredential` for JavaScript, see [Azure Identity client library for JavaScript](#).

JavaScript

```
import { DefaultAzureCredential } from "@azure/identity";
import { OpenAIClient } from "@azure/openai";

const AZURE_OPENAI_ENDPOINT = process.env.AZURE_OPENAI_ENDPOINT;

const credential = new OpenAIClient(AZURE_OPENAI_ENDPOINT, new
DefaultAzureCredential());
```

## Additional resources

- [Passwordless connections developer guide](#)

## Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

# Connect to and query Azure SQL Database using Python and the pyodbc driver

Article • 03/21/2024

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

Applies to:  [Azure SQL Database](#)

This quickstart describes how to connect an application to a database in Azure SQL Database and perform queries using Python and the [Python SQL Driver - pyodbc](#). This quickstart follows the recommended passwordless approach to connect to the database. You can learn more about passwordless connections on the [passwordless hub](#).

## Prerequisites

- An [Azure subscription](#).
- An Azure SQL database configured with Microsoft Entra authentication. You can create one using the [Create database quickstart](#).
- The latest version of the [Azure CLI](#).
- Visual Studio Code with the [Python extension](#).
- Python 3.8 or later. If you're using a Linux client machine, see [Install the ODBC driver](#).

## Configure the database

Secure, passwordless connections to Azure SQL Database require certain database configurations. Verify the following settings on your [logical server in Azure](#) to properly connect to Azure SQL Database in both local and hosted environments:

1. For local development connections, make sure your logical server is configured to allow your local machine IP address and other Azure services to connect:
  - Navigate to the **Networking** page of your server.
  - Toggle the **Selected networks** radio button to show additional configuration options.

- Select **Add your client IPv4 address(xx.xx.xx.xx)** to add a firewall rule that will enable connections from your local machine IPv4 address. Alternatively, you can also select **+ Add a firewall rule** to enter a specific IP address of your choice.
- Make sure the **Allow Azure services and resources to access this server** checkbox is selected.

The screenshot shows the Azure portal's 'Networking' settings for a database. On the left, the 'Networking' option is highlighted. In the main pane, under 'Public access', the 'Selected networks' radio button is selected. In the 'Firewall rules' section, the 'Add your client IPv4 address' button is highlighted with a red box. The 'Allow Azure services and resources to access this server' checkbox is also highlighted with a red box.

### ⚠ Warning

Enabling the **Allow Azure services and resources to access this server** setting is not a recommended security practice for production scenarios. Real applications should implement more secure approaches, such as stronger firewall restrictions or virtual network configurations.

You can read more about database security configurations on the following resources:

- [Configure Azure SQL Database firewall rules.](#)
- [Configure a virtual network with private endpoints.](#)

2. The server must also have Microsoft Entra authentication enabled and have a Microsoft Entra admin account assigned. For local development connections, the Microsoft Entra admin account should be an account you can also log into Visual Studio or the Azure CLI with locally. You can verify whether your server has Microsoft Entra authentication enabled on the **Microsoft Entra ID** page of your logical server.

The screenshot shows the Microsoft Entra admin center interface. At the top, there's a search bar and navigation links for 'Set admin', 'Remove admin', and 'Save'. Below this, the 'Microsoft Entra admin' section is displayed, with a note about centrally managing identity and access to Azure SQL Database. It shows an 'Admin name' as 'testing123@microsoft.com'. Under 'Microsoft Entra authentication only', it states that only Microsoft Entra ID will be used for authentication, and SQL authentication will be disabled. A checkbox for 'Support only Microsoft Entra authentication for this server' is checked. On the left, a sidebar lists 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', and 'Settings'. The 'Microsoft Entra ID' option under 'Settings' is highlighted with a red box. On the right, there's a circular button with a magnifying glass icon.

3. If you're using a personal Azure account, make sure you have [Microsoft Entra setup and configured for Azure SQL Database](#) in order to assign your account as a server admin. If you're using a corporate account, Microsoft Entra ID will most likely already be configured for you.

## Create the project

Create a new Python project using Visual Studio Code.

1. Open Visual Studio Code and create a new folder for your project and change directory into it.

```
Cmd
mkdir python-sql-azure
cd python-sql-azure
```

2. Create a virtual environment for the app.

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

3. Create a new Python file called `app.py`.

## Install the pyodbc driver

To connect to Azure SQL Database using Python, install the `pyodbc` driver. This package acts as a data provider for connecting to databases, executing commands, and

retrieving results. In this quickstart, you also install `flask`, `uvicorn`, and `pydantic` packages to create and run an API.

For details and specific instructions for installing the `pyodbc` driver on all operating systems, see [Configure development environment for pyodbc Python development](#).

1. Create a `requirements.txt` file with the following lines:

```
pyodbc
fastapi
uvicorn[standard]
pydantic
azure-identity
```

2. Install the requirements.

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

## Configure the local connection string

For local development and connecting to Azure SQL Database, add the following `AZURE_SQL_CONNECTIONSTRING` environment variable. Replace the `<database-server-name>` and `<database-name>` placeholders with your own values. Example environment variables are shown for the Bash shell.

Interactive authentication provides a passwordless option when you're running locally.

### Interactive Authentication

In Windows, Microsoft Entra Interactive Authentication can use Microsoft Entra multifactor authentication technology to set up connection. In this mode, by providing the sign in ID, an Azure Authentication dialog is triggered and allows the user to input the password to complete the connection.

### Bash

```
export AZURE_SQL_CONNECTIONSTRING='Driver={ODBC Driver 18 for SQL
Server};Server=tcp:<database-server-
```

```
name>.database.windows.net,1433;Database=<database-name>;Encrypt=yes;TrustServerCertificate=no;Connection Timeout=30'
```

For more information, see [Using Microsoft Entra ID with the ODBC Driver](#). If you use this option, look for the window that prompts you for credentials.

You can get the details to create your connection string from the Azure portal:

1. Go to the Azure SQL Server, select the **SQL databases** page to find your database name, and select the database.
2. On the database, go to the **Connection strings** page to get connection string information. Look under the **ODBC** tab.

 **Note**

If you've installed [Azure Arc](#) and associated it with your Azure subscription, you can also use the managed identity approach shown for the app deployed to App Service.

## Add code to connect to Azure SQL Database

In the project folder, create an *app.py* file and add the sample code. This code creates an API that:

- Retrieves an Azure SQL Database connection string from an environment variable.
- Creates a `Persons` table in the database during startup (for testing scenarios only).
- Defines a function to retrieve all `Person` records from the database.
- Defines a function to retrieve one `Person` record from the database.
- Defines a function to add new `Person` records to the database.

Python

```
import os
import pyodbc, struct
from azure import identity

from typing import Union
from fastapi import FastAPI
from pydantic import BaseModel

class Person(BaseModel):
 first_name: str
 last_name: Union[str, None] = None
```

```
connection_string = os.environ["AZURE_SQL_CONNECTIONSTRING"]

app = FastAPI()

@app.get("/")
def root():
 print("Root of Person API")
 try:
 conn = get_conn()
 cursor = conn.cursor()

 # Table should be created ahead of time in production app.
 cursor.execute("""
 CREATE TABLE Persons (
 ID int NOT NULL PRIMARY KEY IDENTITY,
 FirstName varchar(255),
 LastName varchar(255)
);
 """)

 conn.commit()
 except Exception as e:
 # Table may already exist
 print(e)
 return "Person API"

@app.get("/all")
def get_persons():
 rows = []
 with get_conn() as conn:
 cursor = conn.cursor()
 cursor.execute("SELECT * FROM Persons")

 for row in cursor.fetchall():
 print(row.FirstName, row.LastName)
 rows.append(f"{row.ID}, {row.FirstName}, {row.LastName}")
 return rows

@app.get("/person/{person_id}")
def get_person(person_id: int):
 with get_conn() as conn:
 cursor = conn.cursor()
 cursor.execute("SELECT * FROM Persons WHERE ID = ?", person_id)

 row = cursor.fetchone()
 return f"{row.ID}, {row.FirstName}, {row.LastName}"

@app.post("/person")
def create_person(item: Person):
 with get_conn() as conn:
 cursor = conn.cursor()
 cursor.execute(f"INSERT INTO Persons (FirstName, LastName) VALUES (?, ?)", item.first_name, item.last_name)
 conn.commit()
```

```
 return item

def get_conn():
 credential =
 identity.DefaultAzureCredential(exclude_interactive_browser_credential=False)
 token_bytes =
 credential.get_token("https://database.windows.net/.default").token.encode("UTF-16-LE")
 token_struct = struct.pack(f'<I{len(token_bytes)}s', len(token_bytes),
 token_bytes)
 SQL_COPT_SS_ACCESS_TOKEN = 1256 # This connection option is defined by
 microsoft in msodbcsql.h
 conn = pyodbc.connect(connection_string, attrs_before=
 {SQL_COPT_SS_ACCESS_TOKEN: token_struct})
 return conn
```

### ⚠️ Warning

The sample code shows raw SQL statements, which shouldn't be used in production code. Instead, use an Object Relational Mapper (ORM) package like [SqlAlchemy](#) that generates a more secure object layer to access your database.

## Run and test the app locally

The app is ready to be tested locally.

1. Run the `app.py` file in Visual Studio Code.

```
Console
```

```
uvicorn app:app --reload
```

2. On the Swagger UI page for the app <http://127.0.0.1:8000/docs>, expand the POST method and select Try it out.

You can also use try /redoc to see another form of generated documentation for the API.

3. Modify the sample JSON to include values for the first and last name. Select Execute to add a new record to the database. The API returns a successful response.

4. Expand the **GET** method on the Swagger UI page and select **Try it**. Choose **Execute**, and the person you just created is returned.

## Deploy to Azure App Service

The app is ready to be deployed to Azure.

1. Create a *start.sh* file so that gunicorn in Azure App Service can run unicorn. The *start.sh* has one line:

Console

```
gunicorn -w 4 -k unicorn.workers.UvicornWorker app:app
```

2. Use the [az webapp up](#) to deploy the code to App Service. (You can use the option `-dryrun` to see what the command does without creating the resource.)

Azure CLI

```
az webapp up \
--resource-group <resource-group-name> \
--name <web-app-name>
```

3. Use the [az webapp config set](#) command to configure App Service to use the *start.sh* file.

Azure CLI

```
az webapp config set \
--resource-group <resource-group-name> \
--name <web-app-name> \
--startup-file start.sh
```

4. Use the [az webapp identity assign](#) command to enable a system-assigned managed identity for the App Service.

Azure CLI

```
az webapp identity assign \
--resource-group <resource-group-name> \
--name <web-app-name>
```

In this quickstart, a system-assigned managed identity is used for demonstration. A user-assigned managed identity is more efficient in a broader range of scenarios.

For more information, see [Managed identity best practice recommendations](#). For an example of using a user-assigned managed identity with pyodbc, see [Migrate a Python application to use passwordless connections with Azure SQL Database](#).

# Connect the App Service to Azure SQL Database

In the [Configure the database](#) section, you configured networking and Microsoft Entra authentication for the Azure SQL database server. In this section, you complete the database configuration and configure the App Service with a connection string to access the database server.

To run these commands you can use any tool or IDE that can connect to Azure SQL Database, including [SQL Server Management Studio \(SSMS\)](#), [Azure Data Studio](#), and Visual Studio Code with the [SQL server mssql](#) extension. As well, you can use the Azure portal as described in [Quickstart: Use the Azure portal query editor to query Azure SQL Database](#).

1. Add a user to the Azure SQL Database with SQL commands to create a user and role for passwordless access.

SQL

```
CREATE USER [<web-app-name>] FROM EXTERNAL PROVIDER
ALTER ROLE db_datareader ADD MEMBER [<web-app-name>]
ALTER ROLE db_datawriter ADD MEMBER [<web-app-name>]
```

For more information, see [Contained Database Users - Making Your Database Portable](#). For an example that shows the same principle but applied to Azure VM, see [Tutorial: Use a Windows VM system-assigned managed identity to access Azure SQL](#). For more information about the roles assigned, see [Fixed-database Roles](#).

If you disable and then enable the App Service system-assigned managed identity, then drop the user and recreate it. Run `DROP USER [<web-app-name>]` and rerun the `CREATE` and `ALTER` commands. To see users, use `SELECT * FROM sys.database_principals`.

2. Use the `az webapp config appsettings set` command to add an app setting for the connection string.

Azure CLI

```
az webapp config appsettings set \
 --resource-group <resource-group-name> \
 --name <web-app-name> \
 --settings AZURE_SQL_CONNECTIONSTRING="<connection-string>"
```

For the deployed app, the connection string should resemble:

```
Driver={ODBC Driver 18 for SQL Server};Server=tcp:<database-server-
name>.database.windows.net,1433;Database=<database-
name>;Encrypt=yes;TrustServerCertificate=no;Connection Timeout=30
```

Fill in the `<database-server-name>` and `<database-name>` with your values.

The passwordless connection string doesn't contain a user name or password.

Instead, when the app runs in Azure, the code uses `DefaultAzureCredential` from the [Azure Identity library](#) to get a token to use with `pyodbc`.

## Test the deployed application

Browse to the URL of the app to test that the connection to Azure SQL Database is working. You can locate the URL of your app on the App Service overview page.

HTTP

```
https://<web-app-name>.azurewebsites.net
```

Append `/docs` to the URL to see the Swagger UI and test the API methods.

Congratulations! Your application is now connected to Azure SQL Database in both local and hosted environments.

## Related content

- [Migrate a Python application to use passwordless connections with Azure SQL Database](#) - Shows user-assigned managed identity.
- [Passwordless connections for Azure services](#)
- [Managed identity best practice recommendations](#)

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

# Quickstart: Azure Cosmos DB for NoSQL library for Python

Article • 01/08/2024

APPLIES TO: NoSQL

Get started with the Azure Cosmos DB for NoSQL client library for Python to query data in your containers and perform common operations on individual items. Follow these steps to deploy a minimal solution to your environment using the Azure Developer CLI.

[API reference documentation](#) | [Library source code](#) | [Package \(PyPI\)](#) | [Azure Developer CLI](#)

## Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [GitHub account](#)

## Setting up

Deploy this project's development container to your environment. Then, use the Azure Developer CLI (`azd`) to create an Azure Cosmos DB for NoSQL account and deploy a containerized sample application. The sample application uses the client library to manage, create, read, and query sample data.

GITHUB CODESPACES OPEN

### Important

GitHub accounts include an entitlement of storage and core hours at no cost. For more information, see [included storage and core hours for GitHub accounts](#).

1. Open a terminal in the root directory of the project.
2. Authenticate to the Azure Developer CLI using `azd auth login`. Follow the steps specified by the tool to authenticate to the CLI using your preferred Azure credentials.

Azure CLI

```
azd auth login
```

3. Use `azd init` to initialize the project.

```
Azure CLI
```

```
azd init
```

4. During initialization, configure a unique environment name.

 **Tip**

The environment name will also be used as the target resource group name. For this quickstart, consider using `msdocs-cosmos-db-nosql`.

5. Deploy the Azure Cosmos DB for NoSQL account using `azd up`. The Bicep templates also deploy a sample web application.

```
Azure CLI
```

```
azd up
```

6. During the provisioning process, select your subscription and desired location. Wait for the provisioning process to complete. The process can take **approximately five minutes**.

7. Once the provisioning of your Azure resources is done, a URL to the running web application is included in the output.

```
Output
```

```
Deploying services (azd deploy)
```

```
(✓) Done: Deploying service web
- Endpoint: <https://[container-app-sub-domain].azurecontainerapps.io>
```

```
SUCCESS: Your application was provisioned and deployed to Azure in 5
minutes 0 seconds.
```

8. Use the URL in the console to navigate to your web application in the browser. Observe the output of the running app.

## Azure Cosmos DB for NoSQL | Go Quickstart

```
Current Status: Starting...
Get database: cosmicworks
Get container: products
Upserter item: {70b63682-b93a-4c77-aad2-65501347265f gear-surf-surfboards Yamba Surfboard 12 850 false}
Status code: 201
Request charge: 7.24
Upserter item: {25a68543-b90c-439d-8332-7ef41e06a0e0 gear-surf-surfboards Kiama Classic Surfboard 25 790 true}
Status code: 201
Request charge: 7.24
Read item id: 70b63682-b93a-4c77-aad2-65501347265f
Read item: {70b63682-b93a-4c77-aad2-65501347265f gear-surf-surfboards Yamba Surfboard 12 850 false}
Status code: 200
Request charge: 1.00
```

## Install the client library

The client library is available through the Python Package Index, as the `azure-cosmos` library.

1. Open a terminal and navigate to the `/src` folder.

```
Bash
```

```
cd ./src
```

2. If not already installed, install the `azure-cosmos` package using `pip install`.

```
Bash
```

```
pip install azure-cosmos
```

3. Also, install the `azure-identity` package if not already installed.

```
Bash
```

```
pip install azure-identity
```

4. Open and review the `src/requirements.txt` file to validate that the `azure-cosmos` and `azure-identity` entries both exist.

## Object model

[] Expand table

Name	Description
<a href="#">CosmosClient</a>	This class is the primary client class and is used to manage account-wide

Name	Description
	metadata or databases.
DatabaseProxy	This class represents a database within the account.
CotnainerProxy	This class is primarily used to perform read, update, and delete operations on either the container or the items stored within the container.
PartitionKey	This class represents a logical partition key. This class is required for many common operations and queries.

## Code examples

- [Authenticate the client](#)
- [Get a database](#)
- [Get a container](#)
- [Create an item](#)
- [Get an item](#)
- [Query items](#)

The sample code in the template uses a database named `cosmicworks` and container named `products`. The `products` container contains details such as name, category, quantity, a unique identifier, and a sale flag for each product. The container uses the `/category` property as a logical partition key.

## Authenticate the client

Application requests to most Azure services must be authorized. Use the `DefaultAzureCredential` type as the preferred way to implement a passwordless connection between your applications and Azure Cosmos DB for NoSQL. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime.

### ⓘ Important

You can also authorize requests to Azure services using passwords, connection strings, or other credentials directly. However, this approach should be used with caution. Developers must be diligent to never expose these secrets in an unsecure location. Anyone who gains access to the password or secret key is able to authenticate to the database service. `DefaultAzureCredential` offers improved

management and security benefits over the account key to allow passwordless authentication without the risk of storing keys.

This sample creates a new instance of the `CosmosClient` type and authenticates using a `DefaultAzureCredential` instance.

Python

```
credential = DefaultAzureCredential()
client = CosmosClient(url=endpoint, credential=credential)
```

## Get a database

Use `client.get_database_client` to retrieve the existing database named `cosmicworks`.

Python

```
database = client.get_database_client("cosmicworks")
```

## Get a container

Retrieve the existing `products` container using `database.get_container_client`.

Python

```
container = database.get_container_client("products")
```

## Create an item

Build a new object with all of the members you want to serialize into JSON. In this example, the type has a unique identifier, and fields for category, name, quantity, price, and sale. Create an item in the container using `container.upsert_item`. This method "upserts" the item effectively replacing the item if it already exists.

Python

```
new_item = {
 "id": "70b63682-b93a-4c77-aad2-65501347265f",
 "category": "gear-surf-surfboards",
 "name": "Yamba Surfboard",
 "quantity": 12,
 "sale": False,
```

```
}
```

```
 created_item = container.upsert_item(new_item)
```

## Read an item

Perform a point read operation by using both the unique identifier (`id`) and partition key fields. Use `container.read_item` to efficiently retrieve the specific item.

Python

```
existing_item = container.read_item(
 item="70b63682-b93a-4c77-aad2-65501347265f",
 partition_key="gear-surf-surfboards",
)
```

## Query items

Perform a query over multiple items in a container using `container.GetItemQueryIterator`. Find all items within a specified category using this parameterized query:

nosql

```
SELECT * FROM products p WHERE p.category = @category
```

Python

```
queryText = "SELECT * FROM products p WHERE p.category = @category"
results = container.query_items(
 query=queryText,
 parameters=[
 dict(
 name="@category",
 value="gear-surf-surfboards",
)
],
 enable_cross_partition_query=False,
)
```

Loop through the results of the query.

Python

```
items = [item for item in results]
```

```
output = json.dumps(items, indent=True)
```

## Related content

- [.NET Quickstart](#)
- [JavaScript/Node.js Quickstart](#)
- [Java Quickstart](#)
- [Go Quickstart](#)

## Next step

[PyPI package](#)

# Send events to or receive events from event hubs by using Python

Article • 02/07/2024

This quickstart shows how to send events to and receive events from an event hub using the `azure-eventhub` Python package.

## Prerequisites

If you're new to Azure Event Hubs, see [Event Hubs overview](#) before you do this quickstart.

To complete this quickstart, you need the following prerequisites:

- **Microsoft Azure subscription.** To use Azure services, including Azure Event Hubs, you need a subscription. If you don't have an existing Azure account, sign up for a [free trial ↗](#).
- Python 3.8 or later, with pip installed and updated.
- Visual Studio Code (recommended) or any other integrated development environment (IDE).
- **Create an Event Hubs namespace and an event hub.** The first step is to use the [Azure portal ↗](#) to create an Event Hubs namespace, and obtain the management credentials that your application needs to communicate with the event hub. To create a namespace and an event hub, follow the procedure in [this article](#).

## Install the packages to send events

To install the Python packages for Event Hubs, open a command prompt that has Python in its path. Change the directory to the folder where you want to keep your samples.

Passwordless (Recommended)

shell

```
pip install azure-eventhub
pip install azure-identity
pip install aiohttp
```

# Authenticate the app to Azure

This quickstart shows you two ways of connecting to Azure Event Hubs: passwordless and connection string. The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to an Event Hubs namespace. You don't need to worry about having hard-coded connection strings in your code or in a configuration file or in a secure storage like Azure Key Vault. The second option shows you how to use a connection string to connect to an Event Hubs namespace. If you're new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

## Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Event Hubs has the correct permissions. You'll need the [Azure Event Hubs Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Event Hubs Data Owner` role to your user account, which provides full access to Azure Event Hubs resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

## Azure built-in roles for Azure Event Hubs

For Azure Event Hubs, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to an Event Hubs namespace:

- [Azure Event Hubs Data Owner](#): Enables data access to Event Hubs namespace and its entities (queues, topics, subscriptions, and filters)

- **Azure Event Hubs Data Sender**: Use this role to give the sender access to Event Hubs namespace and its entities.
- **Azure Event Hubs Data Receiver**: Use this role to give the receiver access to Event Hubs namespace and its entities.

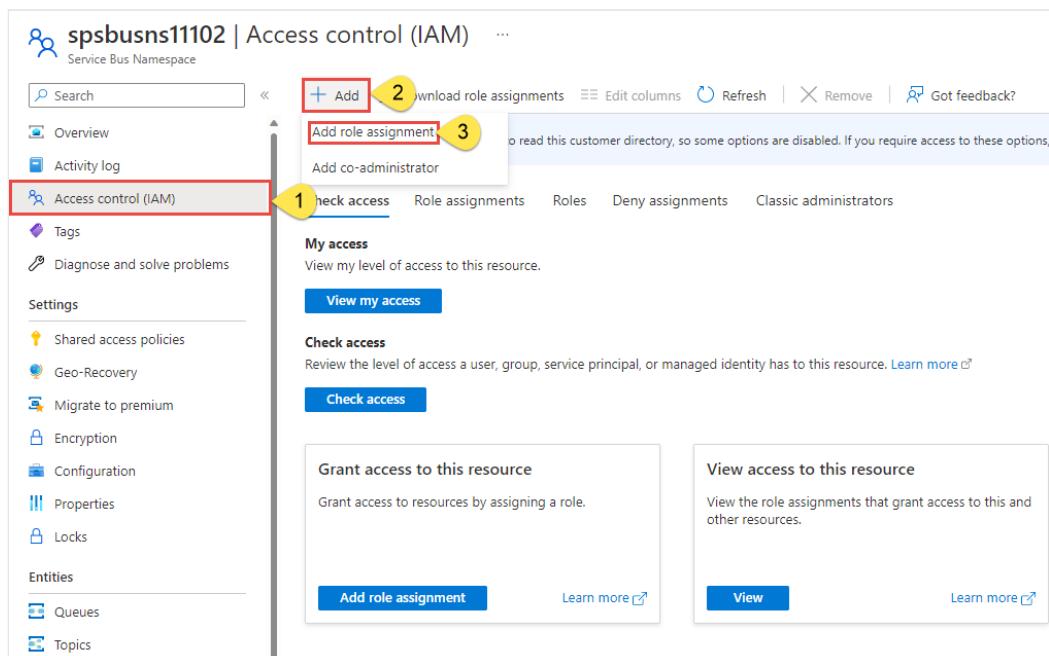
If you want to create a custom role, see [Rights required for Event Hubs operations](#).

## ⓘ Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your Event Hubs namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



5. Use the search box to filter the results to the desired role. For this example, search for `Azure Event Hubs Data Owner` and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your `user@domain` email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Send events

In this section, create a Python script to send events to the event hub that you created earlier.

1. Open your favorite Python editor, such as [Visual Studio Code](#).
2. Create a script called `send.py`. This script sends a batch of events to the event hub that you created earlier.
3. Paste the following code into `send.py`:

Passwordless (Recommended)

In the code, use real values to replace the following placeholders:

- `EVENT_HUB_FULLY_QUALIFIED_NAMESPACE`
- `EVENT_HUB_NAME`

Python

```
import asyncio

from azure.eventhub import EventData
from azure.eventhub.aio import EventHubProducerClient
from azure.identity.aio import DefaultAzureCredential

EVENT_HUB_FULLY_QUALIFIED_NAMESPACE =
"EVENT_HUB_FULLY_QUALIFIED_NAMESPACE"
EVENT_HUB_NAME = "EVENT_HUB_NAME"
```

```
credential = DefaultAzureCredential()

async def run():
 # Create a producer client to send messages to the event hub.
 # Specify a credential that has correct role assigned to access
 # event hubs namespace and the event hub name.
 producer = EventHubProducerClient(
 fully_qualified_namespace=EVENT_HUB_FULLY_QUALIFIED_NAMESPACE,
 eventhub_name=EVENT_HUB_NAME,
 credential=credential,
)
 async with producer:
 # Create a batch.
 event_data_batch = await producer.create_batch()

 # Add events to the batch.
 event_data_batch.add(EventData("First event "))
 event_data_batch.add(EventData("Second event"))
 event_data_batch.add(EventData("Third event"))

 # Send the batch of events to the event hub.
 await producer.send_batch(event_data_batch)

 # Close credential when no longer needed.
 await credential.close()

asyncio.run(run())
```

### ⓘ Note

For examples of other options for sending events to Event Hub asynchronously using a connection string, see the [GitHub send\\_async.py page](#). The patterns shown there are also applicable to sending events passwordless.

## Receive events

This quickstart uses Azure Blob storage as a checkpoint store. The checkpoint store is used to persist checkpoints (that is, the last read positions).

Follow these recommendations when using Azure Blob Storage as a checkpoint store:

- Use a separate container for each processor group. You can use the same storage account, but use one container per each group.

- Don't use the container for anything else, and don't use the storage account for anything else.
- Storage account should be in the same region as the deployed application is located in. If the application is on-premises, try to choose the closest region possible.

On the **Storage account** page in the Azure portal, in the **Blob service** section, ensure that the following settings are disabled.

- Hierarchical namespace
- Blob soft delete
- Versioning

## Create an Azure storage account and a blob container

Create an Azure storage account and a blob container in it by doing the following steps:

1. [Create an Azure Storage account](#)
2. [Create a blob container](#).
3. Authenticate to the blob container.

Be sure to record the connection string and container name for later use in the receive code.

Passwordless (Recommended)

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

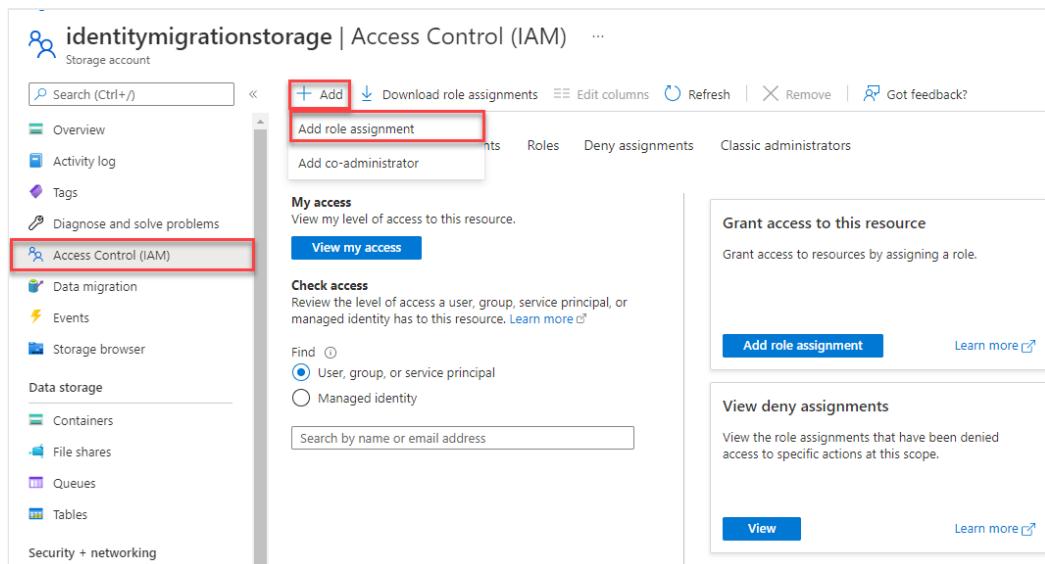
The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

## ⓘ Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the

dialog.

8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Install the packages to receive events

For the receiving side, you need to install one or more packages. In this quickstart, you use Azure Blob storage to persist checkpoints so that the program doesn't read the events that it has already read. It performs metadata checkpoints on received messages at regular intervals in a blob. This approach makes it easy to continue receiving messages later from where you left off.

Passwordless (Recommended)

shell

```
pip install azure-eventhub-checkpointstoreblob-aio
pip install azure-identity
```

## Create a Python script to receive events

In this section, you create a Python script to receive events from your event hub:

1. Open your favorite Python editor, such as [Visual Studio Code](#).
2. Create a script called *recv.py*.
3. Paste the following code into *recv.py*:

Passwordless (Recommended)

In the code, use real values to replace the following placeholders:

- `BLOB_STORAGE_ACCOUNT_URL`
- `BLOB_CONTAINER_NAME`
- `EVENT_HUB_FULLY_QUALIFIED_NAMESPACE`
- `EVENT_HUB_NAME`

Python

```
import asyncio

from azure.eventhub.aio import EventHubConsumerClient
from azure.eventhub.extensions.checkpointstoreblobai import (
 BlobCheckpointStore,
)
from azure.identity.aio import DefaultAzureCredential

BLOB_STORAGE_ACCOUNT_URL = "BLOB_STORAGE_ACCOUNT_URL"
BLOB_CONTAINER_NAME = "BLOB_CONTAINER_NAME"
EVENT_HUB_FULLY_QUALIFIED_NAMESPACE =
"EVENT_HUB_FULLY_QUALIFIED_NAMESPACE"
EVENT_HUB_NAME = "EVENT_HUB_NAME"

credential = DefaultAzureCredential()

async def on_event(partition_context, event):
 # Print the event data.
 print(
 'Received the event: "{}" from the partition with ID: "{}"'.format(
 event.body_as_str(encoding="UTF-8"),
 partition_context.partition_id
)
)

 # Update the checkpoint so that the program doesn't read the
 events
 # that it has already read when you run it next time.
 await partition_context.update_checkpoint(event)

async def main():
 # Create an Azure blob checkpoint store to store the
 checkpoints.
 checkpoint_store = BlobCheckpointStore(
 blob_account_url=BLOB_STORAGE_ACCOUNT_URL,
 container_name=BLOB_CONTAINER_NAME,
 credential=credential,
)

 # Create a consumer client for the event hub.
 client = EventHubConsumerClient(
 fully_qualified_namespace=EVENT_HUB_FULLY_QUALIFIED_NAMESPACE,
 eventhub_name=EVENT_HUB_NAME,
 consumer_group="$Default",
 checkpoint_store=checkpoint_store,
 credential=credential,
)
 async with client:
 # Call the receive method. Read from the beginning of the
 partition
 # (starting_position: "-1")
```

```
 await client.receive(on_event=on_event,
starting_position="-1")

 # Close credential when no longer needed.
 await credential.close()

if __name__ == "__main__":
 # Run the main method.
 asyncio.run(main())
```

### ⓘ Note

For examples of other options for receiving events from Event Hub asynchronously using a connection string, see the [GitHub recv\\_with\\_checkpoint\\_store\\_async.py page](#). The patterns shown there are also applicable to receiving events passwordless.

## Run the receiver app

To run the script, open a command prompt that has Python in its path, and then run this command:

Bash

```
python recv.py
```

## Run the sender app

To run the script, open a command prompt that has Python in its path, and then run this command:

Bash

```
python send.py
```

The receiver window should display the messages that were sent to the event hub.

## Troubleshooting

If you don't see events in the receiver window or the code reports an error, try the following troubleshooting tips:

- If you don't see results from `recy.py`, run `send.py` several times.
- If you see errors about "coroutine" when using the passwordless code (with credentials), make sure you're using importing from `azure.identity.aio`.
- If you see "Unclosed client session" with passwordless code (with credentials), make sure you close the credential when finished. For more information, see [Async credentials](#).
- If you see authorization errors with `recv.py` when accessing storage, make sure you followed the steps in [Create an Azure storage account and a blob container](#) and assigned the **Storage Blob Data Contributor** role to the service principal.
- If you receive events with different partition IDs, this result is expected. Partitions are a data organization mechanism that relates to the downstream parallelism required in consuming applications. The number of partitions in an event hub directly relates to the number of concurrent readers you expect to have. For more information, see [Learn more about partitions](#).

## Next steps

In this quickstart, you've sent and received events asynchronously. To learn how to send and receive events synchronously, go to the [GitHub sync\\_samples page](#).

For all the samples (both synchronous and asynchronous) on GitHub, go to [Azure Event Hubs client library for Python samples](#).

# Quickstart: Azure Key Vault certificate client library for Python

Article • 04/07/2024

Get started with the Azure Key Vault certificate client library for Python. Follow these steps to install the package and try out example code for basic tasks. By using Key Vault to store certificates, you avoid storing certificates in your code, which increases the security of your app.

[API reference documentation](#) | [Library source code](#) | [Package \(Python Package Index\)](#)

## Prerequisites

- An Azure subscription - [create one for free](#).
- [Python 3.7+](#)
- [Azure CLI](#)

This quickstart assumes you're running [Azure CLI](#) or [Azure PowerShell](#) in a Linux terminal window.

## Set up your local environment

This quickstart uses the Azure Identity library with Azure CLI or Azure PowerShell to authenticate the user to Azure services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls. For more information, see [Authenticate the client with Azure Identity client library](#).

### Sign in to Azure

Azure CLI

1. Run the `login` command.

Azure CLI

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

## Install the packages

1. In a terminal or command prompt, create a suitable project folder, and then create and activate a Python virtual environment as described on [Use Python virtual environments](#)
2. Install the Microsoft Entra identity library:

```
terminal
pip install azure.identity
```

3. Install the Key Vault certificate client library:

```
terminal
pip install azure-keyvault-certificates
```

## Create a resource group and key vault

Azure CLI

1. Use the `az group create` command to create a resource group:

```
Azure CLI
az group create --name myResourceGroup --location eastus
```

You can change "eastus" to a location nearer to you, if you prefer.

2. Use `az keyvault create` to create the key vault:

```
Azure CLI
```

```
az keyvault create --name <your-unique-keyvault-name> --resource-group myResourceGroup
```

Replace `<your-unique-keyvault-name>` with a name that's unique across all of Azure. You typically use your personal or company name along with other numbers and identifiers.

## Set the KEY\_VAULT\_NAME environmental variable

Our script will use the value assigned to the `KEY_VAULT_NAME` environment variable as the name of the key vault. You must therefore set this value using the following command:

Console

```
export KEY_VAULT_NAME=<your-unique-keyvault-name>
```

## Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

## Create the sample code

The Azure Key Vault certificate client library for Python allows you to manage certificates. The following code sample demonstrates how to create a client, set a certificate, retrieve a certificate, and delete a certificate.

Create a file named *kv\_certificates.py* that contains this code.

Python

```
import os
from azure.keyvault.certificates import CertificateClient, CertificatePolicy
from azure.identity import DefaultAzureCredential

keyVaultName = os.environ["KEY_VAULT_NAME"]
KVUri = "https://" + keyVaultName + ".vault.azure.net"

credential = DefaultAzureCredential()
client = CertificateClient(vault_url=KVUri, credential=credential)

certificateName = input("Input a name for your certificate > ")

print(f"Creating a certificate in {keyVaultName} called '{certificateName}'\n...")

policy = CertificatePolicy.get_default()
poller = client.begin_create_certificate(certificate_name=certificateName,
 policy=policy)
certificate = poller.result()

print(" done.")

print(f"Retrieving your certificate from {keyVaultName}.")

retrieved_certificate = client.get_certificate(certificateName)

print(f"Certificate with name '{retrieved_certificate.name}' was found'.")
print(f"Deleting your certificate from {keyVaultName} ...")

poller = client.begin_delete_certificate(certificateName)
deleted_certificate = poller.result()

print(" done.")
```

## Run the code

Make sure the code in the previous section is in a file named *kv\_certificates.py*. Then run the code with the following command:

terminal

```
python kv_certificates.py
```

- If you encounter permissions errors, make sure you ran the [az keyvault set-policy](#) or [Set-AzKeyVaultAccessPolicy command](#).
- Rerunning the code with the same key name may produce the error, "(Conflict) Certificate <name> is currently in a deleted but recoverable state." Use a different key name.

## Code details

### Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) class provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In the example code, the name of your key vault is expanded to the key vault URI, in the format `https://<your-key-vault-name>.vault.azure.net`.

Python

```
credential = DefaultAzureCredential()
client = CertificateClient(vault_url=KVUri, credential=credential)
```

## Save a certificate

Once you've obtained the client object for the key vault, you can create a certificate using the [begin\\_create\\_certificate](#) method:

Python

```
policy = CertificatePolicy.get_default()
poller = client.begin_create_certificate(certificate_name=certificateName,
policy=policy)
certificate = poller.result()
```

Here, the certificate requires a policy obtained with the [CertificatePolicy.get\\_default](#) method.

Calling a `begin_create_certificate` method generates an asynchronous call to the Azure REST API for the key vault. The asynchronous call returns a poller object. To wait for the result of the operation, call the poller's `result` method.

When Azure handles the request, it authenticates the caller's identity (the service principal) using the credential object you provided to the client.

## Retrieve a certificate

To read a certificate from Key Vault, use the [get\\_certificate](#) method:

Python

```
retrieved_certificate = client.get_certificate(certificateName)
```

You can also verify that the certificate has been set with the Azure CLI command [az keyvault certificate show](#) or the Azure PowerShell cmdlet [Get-AzKeyVaultCertificate](#)

## Delete a certificate

To delete a certificate, use the [begin\\_delete\\_certificate](#) method:

Python

```
poller = client.begin_delete_certificate(certificateName)
deleted_certificate = poller.result()
```

The `begin_delete_certificate` method is asynchronous and returns a poller object. Calling the poller's `result` method waits for its completion.

You can verify that the certificate is deleted with the Azure CLI command [az keyvault certificate show](#) or the Azure PowerShell cmdlet [Get-AzKeyVaultCertificate](#).

Once deleted, a certificate remains in a deleted but recoverable state for a time. If you run the code again, use a different certificate name.

## Clean up resources

If you want to also experiment with [secrets](#) and [keys](#), you can reuse the Key Vault created in this article.

Otherwise, when you're finished with the resources created in this article, use the following command to delete the resource group and all its contained resources:



Azure CLI

```
az group delete --resource-group myResourceGroup
```

## Next steps

- [Overview of Azure Key Vault](#)
- [Secure access to a key vault](#)
- [Azure Key Vault developer's guide](#)
- [Key Vault security overview](#)
- [Authenticate with Key Vault](#)

# Quickstart: Azure Key Vault keys client library for Python

Article • 04/07/2024

Get started with the Azure Key Vault client library for Python. Follow these steps to install the package and try out example code for basic tasks. By using Key Vault to store cryptographic keys, you avoid storing such keys in your code, which increases the security of your app.

[API reference documentation](#) | [Library source code](#) | [Package \(Python Package Index\)](#)

## Prerequisites

- An Azure subscription - [create one for free](#).
- [Python 3.7+](#)
- [Azure CLI](#)

This quickstart assumes you're running [Azure CLI](#) or [Azure PowerShell](#) in a Linux terminal window.

## Set up your local environment

This quickstart is using the Azure Identity library with Azure CLI or Azure PowerShell to authenticate the user to Azure services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls. For more information, see [Authenticate the client with Azure Identity client library](#).

### Sign in to Azure

Azure CLI

1. Run the `login` command.

Azure CLI

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

## Install the packages

1. In a terminal or command prompt, create a suitable project folder, and then create and activate a Python virtual environment as described on [Use Python virtual environments](#).
2. Install the Microsoft Entra identity library:

```
terminal
pip install azure-identity
```

3. Install the Key Vault key client library:

```
terminal
pip install azure-keyvault-keys
```

## Create a resource group and key vault

Azure CLI

1. Use the `az group create` command to create a resource group:

```
Azure CLI
az group create --name myResourceGroup --location eastus
```

You can change "eastus" to a location nearer to you, if you prefer.

2. Use `az keyvault create` to create the key vault:

```
Azure CLI
```

```
az keyvault create --name <your-unique-keyvault-name> --resource-group myResourceGroup
```

Replace `<your-unique-keyvault-name>` with a name that's unique across all of Azure. You typically use your personal or company name along with other numbers and identifiers.

## Set the KEY\_VAULT\_NAME environmental variable

Our script will use the value assigned to the `KEY_VAULT_NAME` environment variable as the name of the key vault. You must therefore set this value using the following command:

Console

```
export KEY_VAULT_NAME=<your-unique-keyvault-name>
```

## Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

## Create the sample code

The Azure Key Vault key client library for Python allows you to manage cryptographic keys. The following code sample demonstrates how to create a client, set a key, retrieve a key, and delete a key.

Create a file named *kv\_keys.py* that contains this code.

Python

```
import os
from azure.keyvault.keys import KeyClient
from azure.identity import DefaultAzureCredential

keyVaultName = os.environ["KEY_VAULT_NAME"]
KVUri = "https://" + keyVaultName + ".vault.azure.net"

credential = DefaultAzureCredential()
client = KeyClient(vault_url=KVUri, credential=credential)

keyName = input("Input a name for your key > ")

print(f"Creating a key in {keyVaultName} called '{keyName}' ...")

rsa_key = client.create_rsa_key(keyName, size=2048)

print(" done.")

print(f"Retrieving your key from {keyVaultName}.")

retrieved_key = client.get_key(keyName)

print(f"Key with name '{retrieved_key.name}' was found.")
print(f"Deleting your key from {keyVaultName} ...")

poller = client.begin_delete_key(keyName)
deleted_key = poller.result()

print(" done.")
```

## Run the code

Make sure the code in the previous section is in a file named *kv\_keys.py*. Then run the code with the following command:

terminal

```
python kv_keys.py
```

Rerunning the code with the same key name may produce the error, "(Conflict) Key <name> is currently in a deleted but recoverable state." Use a different key name.

## Code details

### Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) class provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In the example code, the name of your key vault is expanded using the value of the `KVUri` variable, in the format: "https://<your-key-vault-name>.vault.azure.net".

Python

```
credential = DefaultAzureCredential()
client = KeyClient(vault_url=KVUri, credential=credential)
```

## Save a key

Once you've obtained the client object for the key vault, you can store a key using the [create\\_rsa\\_key](#) method:

Python

```
rsa_key = client.create_rsa_key(keyName, size=2048)
```

You can also use [create\\_key](#) or [create\\_ec\\_key](#).

Calling a `create` method generates a call to the Azure REST API for the key vault.

When Azure handles the request, it authenticates the caller's identity (the service principal) using the credential object you provided to the client.

## Retrieve a key

To read a key from Key Vault, use the `get_key` method:

Python

```
retrieved_key = client.get_key(keyName)
```

You can also verify that the key has been set with the Azure CLI command `az keyvault key show` or the Azure PowerShell cmdlet `Get-AzKeyVaultKey`.

## Delete a key

To delete a key, use the `begin_delete_key` method:

Python

```
poller = client.begin_delete_key(keyName)
deleted_key = poller.result()
```

The `begin_delete_key` method is asynchronous and returns a poller object. Calling the poller's `result` method waits for its completion.

You can verify that the key is deleted with the Azure CLI command `az keyvault key show` or the Azure PowerShell cmdlet `Get-AzKeyVaultKey`.

Once deleted, a key remains in a deleted but recoverable state for a time. If you run the code again, use a different key name.

## Clean up resources

If you want to also experiment with `certificates` and `secrets`, you can reuse the Key Vault created in this article.

Otherwise, when you're finished with the resources created in this article, use the following command to delete the resource group and all its contained resources:

Azure CLI

Azure CLI

```
az group delete --resource-group myResourceGroup
```

## Next steps

- [Overview of Azure Key Vault](#)
- [Secure access to a key vault](#)
- [RBAC Guide](#)
- [Azure Key Vault developer's guide](#)
- [Authenticate with Key Vault](#)

# Quickstart: Azure Key Vault secret client library for Python

Article • 04/07/2024

Get started with the Azure Key Vault secret client library for Python. Follow these steps to install the package and try out example code for basic tasks. By using Key Vault to store secrets, you avoid storing secrets in your code, which increases the security of your app.

[API reference documentation](#) | [Library source code](#) | [Package \(Python Package Index\)](#)

## Prerequisites

- An Azure subscription - [create one for free](#).
- [Python 3.7+](#).
- [Azure CLI](#) or [Azure PowerShell](#).

This quickstart assumes you're running [Azure CLI](#) or [Azure PowerShell](#) in a Linux terminal window.

## Set up your local environment

This quickstart is using Azure Identity library with Azure CLI or Azure PowerShell to authenticate user to Azure Services. Developers can also use Visual Studio or Visual Studio Code to authenticate their calls, for more information, see [Authenticate the client with Azure Identity client library](#).

### Sign in to Azure

Azure CLI

1. Run the `az login` command.

Azure CLI

```
az login
```

If the CLI can open your default browser, it will do so and load an Azure sign-in page.

Otherwise, open a browser page at <https://aka.ms/devicelogin> and enter the authorization code displayed in your terminal.

2. Sign in with your account credentials in the browser.

## Install the packages

1. In a terminal or command prompt, create a suitable project folder, and then create and activate a Python virtual environment as described on [Use Python virtual environments](#).
2. Install the Microsoft Entra identity library:

```
terminal
pip install azure-identity
```

3. Install the Key Vault secrets library:

```
terminal
pip install azure-keyvault-secrets
```

## Create a resource group and key vault

Azure CLI

1. Use the `az group create` command to create a resource group:

```
Azure CLI
az group create --name myResourceGroup --location eastus
```

You can change "eastus" to a location nearer to you, if you prefer.

2. Use `az keyvault create` to create the key vault:

```
Azure CLI
```

```
az keyvault create --name <your-unique-keyvault-name> --resource-group myResourceGroup
```

Replace `<your-unique-keyvault-name>` with a name that's unique across all of Azure. You typically use your personal or company name along with other numbers and identifiers.

## Set the KEY\_VAULT\_NAME environmental variable

Our script will use the value assigned to the `KEY_VAULT_NAME` environment variable as the name of the key vault. You must therefore set this value using the following command:

Console

```
export KEY_VAULT_NAME=<your-unique-keyvault-name>
```

## Grant access to your key vault

Azure CLI

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

Azure CLI

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

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

## Create the sample code

The Azure Key Vault secret client library for Python allows you to manage secrets. The following code sample demonstrates how to create a client, set a secret, retrieve a secret, and delete a secret.

Create a file named *kv\_secrets.py* that contains this code.

Python

```
import os
from azure.keyvault.secrets import SecretClient
from azure.identity import DefaultAzureCredential

keyVaultName = os.environ["KEY_VAULT_NAME"]
KVUri = f"https://{keyVaultName}.vault.azure.net"

credential = DefaultAzureCredential()
client = SecretClient(vault_url=KVUri, credential=credential)

secretName = input("Input a name for your secret > ")
secretValue = input("Input a value for your secret > ")

print(f"Creating a secret in {keyVaultName} called '{secretName}' with the
 value '{secretValue}' ...")

client.set_secret(secretName, secretValue)

print(" done.")

print(f"Retrieving your secret from {keyVaultName}.")
retrieved_secret = client.get_secret(secretName)

print(f"Your secret is '{retrieved_secret.value}'.")
print(f"Deleting your secret from {keyVaultName} ...")

poller = client.begin_delete_secret(secretName)
deleted_secret = poller.result()

print(" done.")
```

## Run the code

Make sure the code in the previous section is in a file named *kv\_secrets.py*. Then run the code with the following command:

terminal

```
python kv_secrets.py
```

- If you encounter permissions errors, make sure you ran the [az keyvault set-policy](#) or [Set-AzKeyVaultAccessPolicy command](#).
- Rerunning the code with the same secret name may produce the error, "(Conflict) Secret <name> is currently in a deleted but recoverable state." Use a different secret name.

## Code details

### Authenticate and create a client

Application requests to most Azure services must be authorized. Using the [DefaultAzureCredential](#) class provided by the [Azure Identity client library](#) is the recommended approach for implementing passwordless connections to Azure services in your code. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

In this quickstart, `DefaultAzureCredential` authenticates to key vault using the credentials of the local development user logged into the Azure CLI. When the application is deployed to Azure, the same `DefaultAzureCredential` code can automatically discover and use a managed identity that is assigned to an App Service, Virtual Machine, or other services. For more information, see [Managed Identity Overview](#).

In the example code, the name of your key vault is expanded using the value of the `KVUri` variable, in the format: "https://<your-key-vault-name>.vault.azure.net".

Python

```
credential = DefaultAzureCredential()
client = SecretClient(vault_url=KVUri, credential=credential)
```

### Save a secret

Once you've obtained the client object for the key vault, you can store a secret using the [set\\_secret](#) method:

Python

```
client.set_secret(secretName, secretValue)
```

Calling `set_secret` generates a call to the Azure REST API for the key vault.

When Azure handles the request, it authenticates the caller's identity (the service principal) using the credential object you provided to the client.

## Retrieve a secret

To read a secret from Key Vault, use the [get\\_secret](#) method:

Python

```
retrieved_secret = client.get_secret(secretName)
```

The secret value is contained in `retrieved_secret.value`.

You can also retrieve a secret with the Azure CLI command [az keyvault secret show](#) or the Azure PowerShell cmdlet [Get-AzKeyVaultSecret](#).

## Delete a secret

To delete a secret, use the [begin\\_delete\\_secret](#) method:

Python

```
poller = client.begin_delete_secret(secretName)
deleted_secret = poller.result()
```

The `begin_delete_secret` method is asynchronous and returns a poller object. Calling the poller's `result` method waits for its completion.

You can verify that the secret had been removed with the Azure CLI command [az keyvault secret show](#) or the Azure PowerShell cmdlet [Get-AzKeyVaultSecret](#).

Once deleted, a secret remains in a deleted but recoverable state for a time. If you run the code again, use a different secret name.

## Clean up resources

If you want to also experiment with [certificates](#) and [keys](#), you can reuse the Key Vault created in this article.

Otherwise, when you're finished with the resources created in this article, use the following command to delete the resource group and all its contained resources:

Azure CLI

Azure CLI

```
az group delete --resource-group myResourceGroup
```

## Next steps

- [Overview of Azure Key Vault](#)
- [Azure Key Vault developer's guide](#)
- [Key Vault security overview](#)
- [Authenticate with Key Vault](#)

# Send messages to and receive messages from Azure Service Bus queues (Python)

Article • 02/06/2024

In this tutorial, you complete the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus queue, using the Azure portal.
3. Write Python code to use the [azure-servicebus](#) package to:
  - a. Send a set of messages to the queue.
  - b. Receive those messages from the queue.

## Note

This quick start provides step-by-step instructions for a simple scenario of sending messages to a Service Bus queue and receiving them. You can find pre-built JavaScript and TypeScript samples for Azure Service Bus in the [Azure SDK for Python repository on GitHub](#).

## Prerequisites

If you're new to the service, see [Service Bus overview](#) before you do this quickstart.

- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign-up for a [free account](#).
- [Python 3.8](#) or higher.

Passwordless (Recommended)

To use this quickstart with your own Azure account:

- Install [Azure CLI](#), which provides the passwordless authentication to your developer machine.
- Sign in with your Azure account at the terminal or command prompt with `az login`.
- Use the same account when you add the appropriate data role to your resource.
- Run the code in the same terminal or command prompt.

- Note the **queue** name for your Service Bus namespace. You'll need that in the code.

### Note

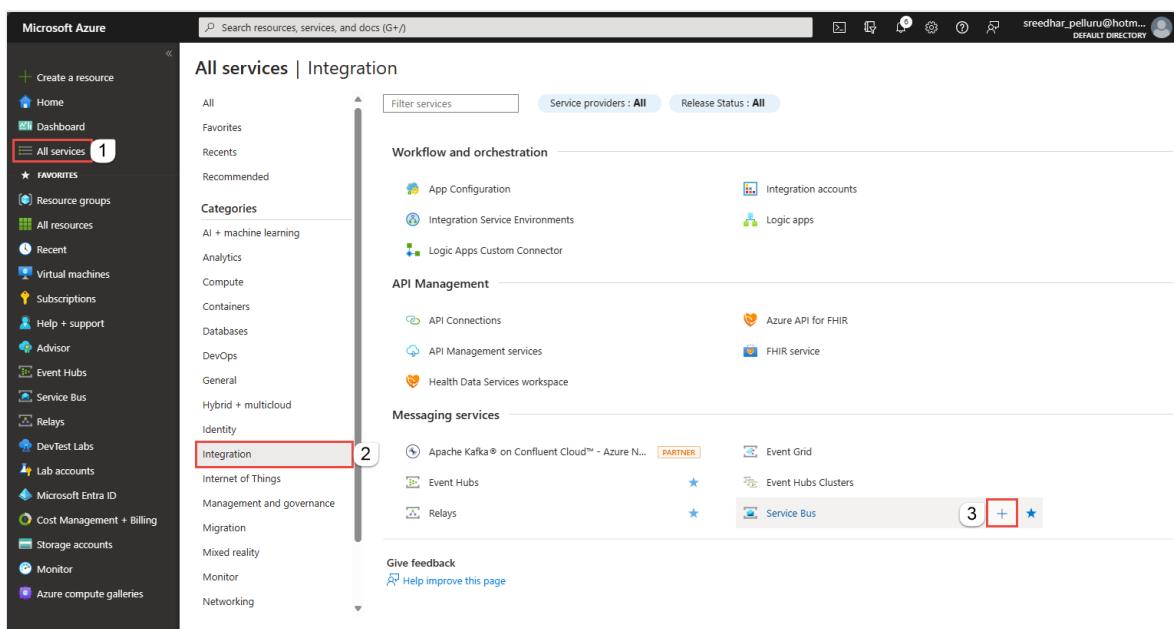
This tutorial works with samples that you can copy and run using Python. For instructions on how to create a Python application, see [Create and deploy a Python application to an Azure Website](#). For more information about installing packages used in this tutorial, see the [Python Installation Guide](#).

## Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.

To create a namespace:

1. Sign in to the [Azure portal](#).
2. In the left navigation pane of the portal, select **All services**, select **Integration** from the list of categories, hover the mouse over **Service Bus**, and then select + button on the Service Bus tile.



3. In the **Basics** tag of the [Create namespace](#) page, follow these steps:

- a. For **Subscription**, choose an Azure subscription in which to create the namespace.
- b. For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
- c. Enter a **name for the namespace**. The namespace name should adhere to the following naming conventions:
  - The name must be unique across Azure. The system immediately checks to see if the name is available.
  - The name length is at least 6 and at most 50 characters.
  - The name can contain only letters, numbers, hyphens “-”.
  - The name must start with a letter and end with a letter or number.
  - The name doesn't end with “-sb” or “-mgmt”.
- d. For **Location**, choose the region in which your namespace should be hosted.
- e. For **Pricing tier**, select the pricing tier (Basic, Standard, or Premium) for the namespace. For this quickstart, select **Standard**.

 **Important**

If you want to use **topics and subscriptions**, choose either Standard or Premium. Topics/subscriptions aren't supported in the Basic pricing tier.

If you selected the **Premium** pricing tier, specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, 4, 8 or 16 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

- f. Select **Review + create** at the bottom of the page.

 **Create namespace** ...

Service Bus

Basics Advanced Networking Tags Review + create

**Project Details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* Visual Studio Enterprise Subscription

Resource group \* (New) spsbusrg [Create new](#)

**Instance Details**

Enter required settings for this namespace.

Namespace name \* contosoordersns .servicebus.windows.net

Location \* East US

Pricing tier \* Standard [Browse the available plans and their features](#)

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

g. On the **Review + create** page, review settings, and select **Create**.

4. Once the deployment of the resource is successful, select **Go to resource** on the deployment page.

 **contosoordersns | Overview** ⚡ ...

Deployment

Search [Delete](#) [Cancel](#) [Redeploy](#) [Download](#) [Refresh](#)

**Overview**  Your deployment is complete

Deployment name: contosoordersns  
Subscription: Visual Studio Enterprise Subscription  
Resource group: spsbusrg

Start time: 10/20/2022, 4:45:03 PM  
Correlation ID: a453ace1-bab9-4c4a-81ad-a1c5366460ea [Copy](#)

[Deployment details](#) [Next steps](#)

[Go to resource](#)

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

5. You see the home page for your service bus namespace.

The screenshot shows the Azure Service Bus Namespace overview page for the resource group 'spsbusrg'. It displays various metrics and configurations. Key details include:

- Resource group (move):** spsbusrg
- Status:** Active
- Location:** East US
- Subscription (move):** Visual Studio Enterprise Subscription
- Subscription ID:** 0000000000-0000-0000-0000-000000000000
- Tags (edit):** Add tags

Metrics shown in the Requests and Messages sections indicate zero activity over the last hour. A timeline at the bottom shows data from 5 PM UTC-05:00 to 5:15 PM UTC-05:00.

## Create a queue in the Azure portal

1. On the Service Bus Namespace page, select **Queues** in the left navigational menu.
2. On the Queues page, select **+ Queue** on the toolbar.
3. Enter a **name** for the queue, and leave the other values with their defaults.
4. Now, select **Create**.

The screenshot shows the 'Create queue' dialog for the 'contososbusns' namespace. The configuration fields are as follows:

- Name:** myqueue (highlighted with a red box and circled with a red marker)
- Max queue size:** 1 GB
- Max delivery count:** 10
- Message time to live:** 14 Days, 0 Hours, 0 Minutes, 0 Seconds
- Lock duration:** 0 Days, 0 Hours, 0 Minutes, 30 Seconds
- Checkboxes (unchecked):** Enable auto-delete on idle queue, Enable duplicate detection, Enable dead lettering on message expiration, Enable partitioning, Enable sessions, Forward messages to queue/topic

At the bottom right, the **Create** button is highlighted with a red box and circled with a red marker.

# Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Service Bus: **passwordless** and **connection string**.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace. You don't need to worry about having hard-coded connection string in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a connection string to connect to a Service Bus namespace. If you are new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

## Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Service Bus has the correct permissions. You'll need the [Azure Service Bus Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Service Bus Data Owner` role to your user account, which provides full access to Azure Service Bus resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

## Azure built-in roles for Azure Service Bus

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters). A member of this role can send and receive messages from queues or topics/subscriptions.
- [Azure Service Bus Data Sender](#): Use this role to give the send access to Service Bus namespace and its entities.
- [Azure Service Bus Data Receiver](#): Use this role to give the receive access to Service Bus namespace and its entities.

If you want to create a custom role, see [Rights required for Service Bus operations](#).

## Add Microsoft Entra user to Azure Service Bus Owner role

Add your Microsoft Entra user name to the **Azure Service Bus Data Owner** role at the Service Bus namespace level. It will allow an app running in the context of your user account to send messages to a queue or a topic, and receive messages from a queue or a topic's subscription.

### Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. If you don't have the Service Bus Namespace page open in the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'Access control (IAM)' section of the Azure Service Bus Namespace. The left sidebar has a red box around the 'Access control (IAM)' item. The top navigation bar has a red box around the 'Add role assignment' button, with a yellow arrow labeled '2' pointing to it. A tooltip '3' with a yellow arrow points to the same button.

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Service Bus Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Use pip to install packages

Passwordless (Recommended)

1. To install the required Python packages for this Service Bus tutorial, open a command prompt that has Python in its path, change the directory to the folder where you want to have your samples.
2. Install the following packages:

shell

```
pip install azure-servicebus
pip install azure-identity
pip install aiohttp
```

## Send messages to a queue

The following sample code shows you how to send a message to a queue. Open your favorite editor, such as [Visual Studio Code](#), create a file `send.py`, and add the following code into it.

Passwordless (Recommended)

1. Add import statements.

Python

```
import asyncio
from azure.servicebus.aio import ServiceBusClient
from azure.servicebus import ServiceBusMessage
from azure.identity.aio import DefaultAzureCredential
```

2. Add constants and define a credential.

Python

```
FULLY_QUALIFIED_NAMESPACE = "FULLY_QUALIFIED_NAMESPACE"
QUEUE_NAME = "QUEUE_NAME"

credential = DefaultAzureCredential()
```

### ⓘ Important

- Replace `FULLY_QUALIFIED_NAMESPACE` with the fully qualified namespace for your Service Bus namespace.
- Replace `QUEUE_NAME` with the name of the queue.

3. Add a method to send a single message.

Python

```
async def send_single_message(sender):
 # Create a Service Bus message and send it to the queue
 message = ServiceBusMessage("Single Message")
 await sender.send_messages(message)
 print("Sent a single message")
```

The sender is an object that acts as a client for the queue you created. You'll create it later and send as an argument to this function.

#### 4. Add a method to send a list of messages.

Python

```
async def send_a_list_of_messages(sender):
 # Create a list of messages and send it to the queue
 messages = [ServiceBusMessage("Message in list") for _ in
range(5)]
 await sender.send_messages(messages)
 print("Sent a list of 5 messages")
```

#### 5. Add a method to send a batch of messages.

Python

```
async def send_batch_message(sender):
 # Create a batch of messages
 async with sender:
 batch_message = await sender.create_message_batch()
 for _ in range(10):
 try:
 # Add a message to the batch

 batch_message.add_message(ServiceBusMessage("Message inside a
ServiceBusMessageBatch"))
 except ValueError:
 # ServiceBusMessageBatch object reaches max_size.
 # New ServiceBusMessageBatch object can be created
here to send more data.
 break
 # Send the batch of messages to the queue
 await sender.send_messages(batch_message)
 print("Sent a batch of 10 messages")
```

#### 6. Create a Service Bus client and then a queue sender object to send messages.

Python

```
async def run():
 # create a Service Bus client using the credential
```

```
async with ServiceBusClient(
 fully_qualified_namespace=FULLY_QUALIFIED_NAMESPACE,
 credential=credential,
 logging_enable=True) as servicebus_client:
 # get a Queue Sender object to send messages to the queue
 sender =
 servicebus_client.get_queue_sender(queue_name=QUEUE_NAME)
 async with sender:
 # send one message
 await send_single_message(sender)
 # send a list of messages
 await send_a_list_of_messages(sender)
 # send a batch of messages
 await send_batch_message(sender)

 # Close credential when no longer needed.
 await credential.close()
```

7. Call the `run` method and print a message.

Python

```
asyncio.run(run())
print("Done sending messages")
print("-----")
```

## Receive messages from a queue

The following sample code shows you how to receive messages from a queue. The code shown receives new messages until it doesn't receive any new messages for 5 (`max_wait_time`) seconds.

Open your favorite editor, such as [Visual Studio Code](#), create a file `recv.py`, and add the following code into it.

Passwordless (Recommended)

1. Similar to the send sample, add `import` statements, define constants that you should replace with your own values, and define a credential.

Python

```
import asyncio
```

```
from azure.servicebus.aio import ServiceBusClient
from azure.identity.aio import DefaultAzureCredential

FULLY_QUALIFIED_NAMESPACE = "FULLY_QUALIFIED_NAMESPACE"
QUEUE_NAME = "QUEUE_NAME"

credential = DefaultAzureCredential()
```

2. Create a Service Bus client and then a queue receiver object to receive messages.

Python

```
async def run():
 # create a Service Bus client using the connection string
 async with ServiceBusClient(
 fully_qualified_namespace=FULLY_QUALIFIED_NAMESPACE,
 credential=credential,
 logging_enable=True) as servicebus_client:

 async with servicebus_client:
 # get the Queue Receiver object for the queue
 receiver =
 servicebus_client.get_queue_receiver(queue_name=QUEUE_NAME)
 async with receiver:
 received_msgs = await
 receiver.receive_messages(max_wait_time=5, max_message_count=20)
 for msg in received_msgs:
 print("Received: " + str(msg))
 # complete the message so that the message is
 removed from the queue
 await receiver.complete_message(msg)

 # Close credential when no longer needed.
 await credential.close()
```

3. Call the `run` method.

Python

```
asyncio.run(run())
```

## Run the app

Open a command prompt that has Python in its path, and then run the code to send and receive messages from the queue.

```
shell
```

```
python send.py; python recv.py
```

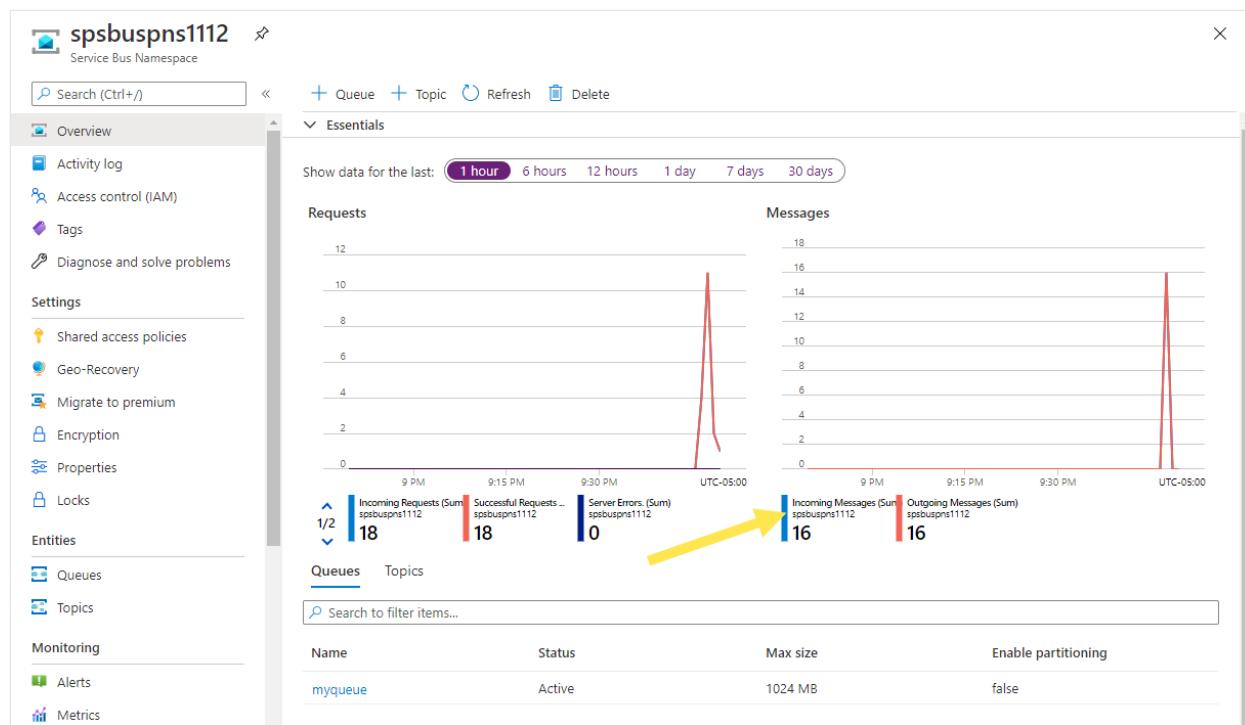
You should see the following output:

Console

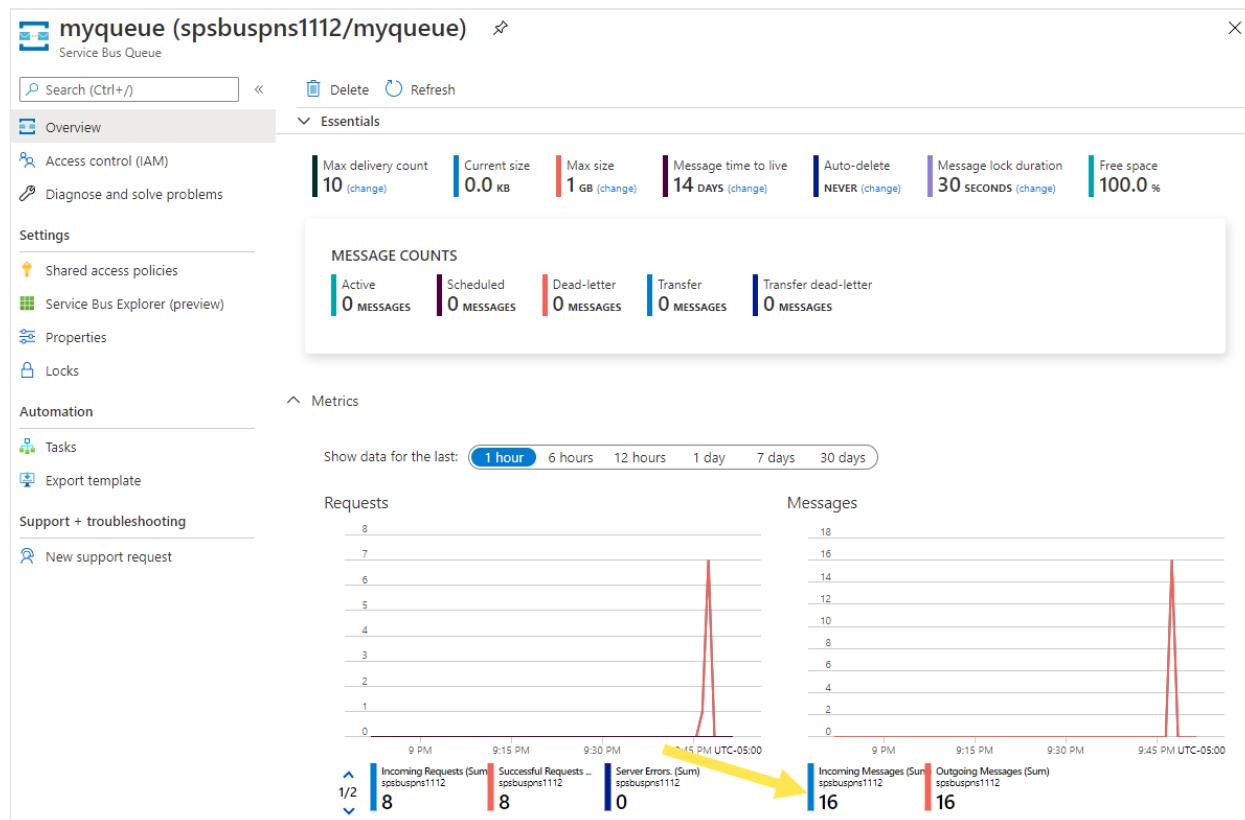
```
Sent a single message
Sent a list of 5 messages
Sent a batch of 10 messages
Done sending messages

Received: Single Message
Received: Message in list
Received: Message inside a ServiceBusMessageBatch
```

In the Azure portal, navigate to your Service Bus namespace. On the [Overview](#) page, verify that the **incoming** and **outgoing** message counts are 16. If you don't see the counts, refresh the page after waiting for a few minutes.



Select the queue on this **Overview** page to navigate to the **Service Bus Queue** page. You can also see the **incoming** and **outgoing** message count on this page. You also see other information such as the **current size** of the queue and **active message count**.



## Next steps

See the following documentation and samples:

- Azure Service Bus client library for Python ↗

- [Samples ↗](#).
  - The **sync\_samples** folder has samples that show you how to interact with Service Bus in a synchronous manner.
  - The **async\_samples** folder has samples that show you how to interact with Service Bus in an asynchronous manner. In this quick start, you used this method.
- [azure-servicebus reference documentation](#)

# Send messages to an Azure Service Bus topic and receive messages from subscriptions to the topic (Python)

Article • 02/06/2024

In this tutorial, you complete the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus topic, using the Azure portal.
3. Create a Service Bus subscription to that topic, using the Azure portal.
4. Write a Python application to use the [azure-servicebus](#) package to:
  - Send a set of messages to the topic.
  - Receive those messages from the subscription.

## ⓘ Note

This quickstart provides step-by-step instructions for a simple scenario of sending a batch of messages to a Service Bus topic and receiving those messages from a subscription of the topic. You can find pre-built Python samples for Azure Service Bus in the [Azure SDK for Python repository on GitHub](#).

## Prerequisites

- An [Azure subscription](#).
- [Python 3.8](#) or higher

## ⓘ Note

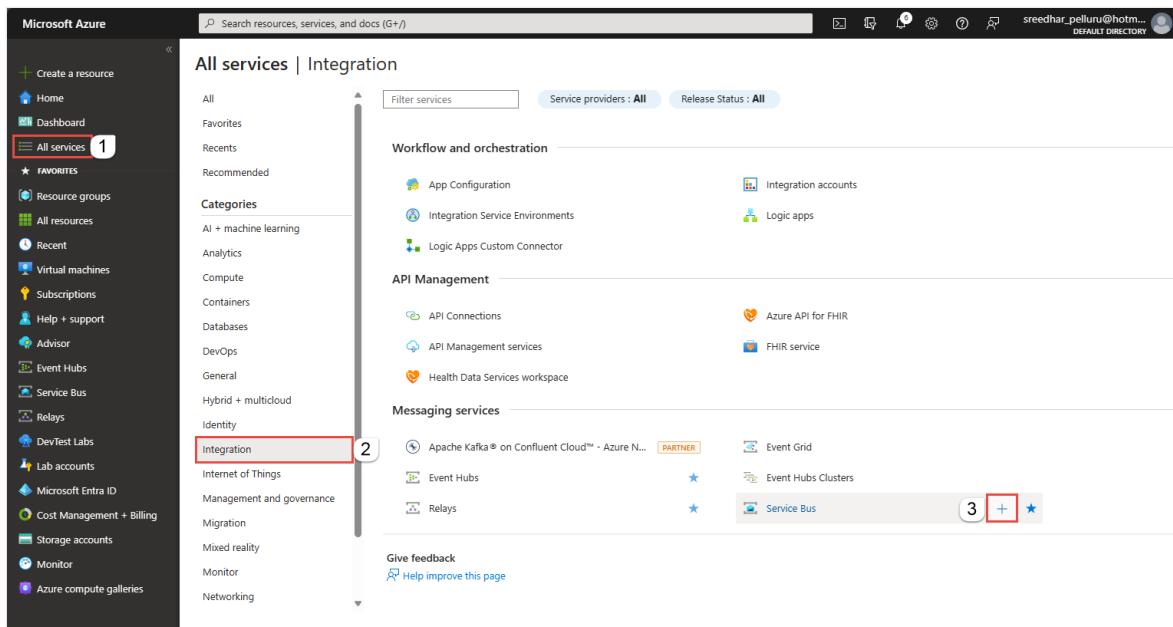
This tutorial works with samples that you can copy and run using Python. For instructions on how to create a Python application, see [Create and deploy a Python application to an Azure Website](#). For more information about installing packages used in this tutorial, see the [Python Installation Guide](#).

## Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.

To create a namespace:

1. Sign in to the [Azure portal](#).
2. In the left navigation pane of the portal, select **All services**, select **Integration** from the list of categories, hover the mouse over **Service Bus**, and then select + button on the Service Bus tile.



3. In the **Basics** tag of the [Create namespace](#) page, follow these steps:
  - a. For **Subscription**, choose an Azure subscription in which to create the namespace.
  - b. For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
  - c. Enter a **name for the namespace**. The namespace name should adhere to the following naming conventions:
    - The name must be unique across Azure. The system immediately checks to see if the name is available.
    - The name length is at least 6 and at most 50 characters.
    - The name can contain only letters, numbers, hyphens “-”.
    - The name must start with a letter and end with a letter or number.
    - The name doesn't end with “-sb” or “-mgmt”.
  - d. For **Location**, choose the region in which your namespace should be hosted.

- e. For **Pricing tier**, select the pricing tier (Basic, Standard, or Premium) for the namespace. For this quickstart, select **Standard**.

**ⓘ Important**

If you want to use **topics and subscriptions**, choose either Standard or Premium. Topics/subscriptions aren't supported in the Basic pricing tier.

If you selected the **Premium** pricing tier, specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, 4, 8 or 16 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

- f. Select **Review + create** at the bottom of the page.

The screenshot shows the 'Create namespace' dialog box. The 'Basics' tab is selected. In the 'Project Details' section, the 'Subscription' dropdown is set to 'Visual Studio Enterprise Subscription'. Below it, the 'Resource group' dropdown shows '(New) spsbusrg' with a 'Create new' link. In the 'Instance Details' section, the 'Namespace name' input field contains 'contosoordersns' with a green checkmark and '.servicebus.windows.net' suffix. The 'Location' dropdown is set to 'East US'. The 'Pricing tier' dropdown is set to 'Standard' with a link to 'Browse the available plans and their features'. At the bottom, there are three buttons: 'Review + create' (highlighted in blue), '< Previous', and 'Next: Advanced >'.

- g. On the **Review + create** page, review settings, and select **Create**.
4. Once the deployment of the resource is successful, select **Go to resource** on the deployment page.

The screenshot shows the Azure Deployment Overview page for a deployment named 'contosoordersns'. The status is 'Your deployment is complete'. Deployment details include a name 'contosoordersns', a subscription 'Visual Studio Enterprise Subscription', and a resource group 'spsbusrg'. The start time was 10/20/2022, 4:45:03 PM, and the correlation ID is a453ace1-bab9-4c4a-81ad-a1c5366460ea. A 'Go to resource' button is highlighted in red.

5. You see the home page for your service bus namespace.

The screenshot shows the Azure Service Bus Namespace overview page for 'spsbusns1128'. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Geo-Recovery, Migrate to premium, Encryption, Configuration, Properties, Locks), Entities (Queues, Topics), Monitoring (Insights (Preview), Alerts), and Queues (0) / Topics (0). The main area displays 'Essentials' information such as Resource group (spsbusrg), Status (Active), Location (East US), Subscription (Visual Studio Enterprise Subscription), Subscription ID, Tags, and various metrics for Requests and Messages over the last hour.

## Create a topic using the Azure portal

1. On the Service Bus Namespace page, select **Topics** on the left menu.
2. Select **+ Topic** on the toolbar.
3. Enter a name for the topic. Leave the other options with their default values.
4. Select **Create**.

All services > Resource groups > spsbusrg > spsbusns1128

spsbusns1128 | Topics

Service Bus Namespace

Search  2 + Topic Refresh Give feedback

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings Shared access policies Geo-Recovery Migrate to premium Encryption Configuration Properties Locks

Entities Queues **Topics** 1 Topics

Monitoring Insights (Preview) Alerts

Create 4 Give feedback

## Create a subscription to the topic

1. Select the **topic** that you created in the previous section.

spsbusns1128 | Topics

Service Bus Namespace

Search + Topic Refresh Give feedback

Properties Locks

Entities Queues Topics

Monitoring Insights (Preview) Alerts Metrics Diagnostic settings Logs Workbooks

Name	Status	Scheduled messages	Max size	Subscription count	Enable partitioning
mytopic	Active	0	1024 MB	0	false

2. On the **Service Bus Topic** page, select **+ Subscription** on the toolbar.

3. On the **Create subscription** page, follow these steps:

- a. Enter **S1** for **name** of the subscription.
- b. Enter **3** for **Max delivery count**.
- c. Then, select **Create** to create the subscription.

## Create subscription

Service Bus

Name \* ⓘ

S1



Max delivery count \* ⓘ

10

Auto-delete after idle for ⓘ

Days

Hours

Minutes

Seconds

14

0

0

0

Never auto-delete

Forward messages to queue/topic ⓘ

### MESSAGE SESSIONS

Service bus sessions allow ordered handling of unbounded sequences of related messages. With sessions enabled a subscription can guarantee first-in-first-out delivery of messages. [Learn more.](#)

Enable sessions

### MESSAGE TIME TO LIVE AND DEAD-LETTERING

Message time to live (default) ⓘ

Days

Hours

Minutes

Seconds

14

0

0

0

Enable dead lettering on message expiration

Move messages that cause filter evaluation exceptions to the dead-letter subqueue

### MESSAGE LOCK DURATION

Lock duration ⓘ

Days

Hours

Minutes

Seconds

0

0

1

0

**Create**

## Authenticate the app to Azure

This quick start shows you two ways of connecting to Azure Service Bus: **passwordless** and **connection string**.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace. You don't

need to worry about having hard-coded connection string in your code or in a configuration file or in a secure storage like Azure Key Vault.

The second option shows you how to use a connection string to connect to a Service Bus namespace. If you are new to Azure, you may find the connection string option easier to follow. We recommend using the passwordless option in real-world applications and production environments. For more information, see [Authentication and authorization](#). You can also read more about passwordless authentication on the [overview page](#).

Passwordless (Recommended)

## Assign roles to your Microsoft Entra user

When developing locally, make sure that the user account that connects to Azure Service Bus has the correct permissions. You'll need the [Azure Service Bus Data Owner](#) role in order to send and receive messages. To assign yourself this role, you'll need the User Access Administrator role, or another role that includes the `Microsoft.Authorization/roleAssignments/write` action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. Learn more about the available scopes for role assignments on the [scope overview](#) page.

The following example assigns the `Azure Service Bus Data Owner` role to your user account, which provides full access to Azure Service Bus resources. In a real scenario, follow the [Principle of Least Privilege](#) to give users only the minimum permissions needed for a more secure production environment.

## Azure built-in roles for Azure Service Bus

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the Azure RBAC model. Azure provides the below Azure built-in roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters). A member of this role can send and receive messages from queues or topics/subscriptions.
- [Azure Service Bus Data Sender](#): Use this role to give the send access to Service Bus namespace and its entities.

- **Azure Service Bus Data Receiver:** Use this role to give the receive access to Service Bus namespace and its entities.

If you want to create a custom role, see [Rights required for Service Bus operations](#).

## Add Microsoft Entra user to Azure Service Bus Owner role

Add your Microsoft Entra user name to the **Azure Service Bus Data Owner** role at the Service Bus namespace level. It will allow an app running in the context of your user account to send messages to a queue or a topic, and receive messages from a queue or a topic's subscription.

### Important

In most cases, it will take a minute or two for the role assignment to propagate in Azure. In rare cases, it may take up to **eight minutes**. If you receive authentication errors when you first run your code, wait a few moments and try again.

1. If you don't have the Service Bus Namespace page open in the Azure portal, locate your Service Bus namespace using the main search bar or left navigation.
2. On the overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

5. Use the search box to filter the results to the desired role. For this example, search for **Azure Service Bus Data Owner** and select the matching result. Then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Code setup

Passwordless (Recommended)

To follow this quickstart using passwordless authentication and your own Azure account:

- Install the [Azure CLI](#).
- Sign in with your Azure account at the terminal or command prompt with `az login`.

- Use the same account when you add the appropriate role to your resource later in the tutorial.
- Run the tutorial code in the same terminal or command prompt.

### ⓘ Important

Make sure you sign in with `az login`. The `DefaultAzureCredential` class in the passwordless code uses the Azure CLI credentials to authenticate with Microsoft Entra ID.

To use the passwordless code, you'll need to specify a:

- fully qualified service bus namespace, for example: `<service-bus-namespace>.servicebus.windows.net`
- topic name
- subscription name

## Use pip to install packages

Passwordless (Recommended)

1. To install the required Python packages for this Service Bus tutorial, open a command prompt that has Python in its path. Change the directory to the folder where you want to have your samples.
2. Install packages:

shell

```
pip install azure-servicebus
pip install azure-identity
pip install aiohttp
```

## Send messages to a topic

The following sample code shows you how to send a batch of messages to a Service Bus topic. See code comments for details.

Open your favorite editor, such as [Visual Studio Code](#), create a file `send.py`, and add the following code into it.

#### Passwordless (Recommended)

1. Add the following `import` statements.

Python

```
import asyncio
from azure.servicebus.aio import ServiceBusClient
from azure.servicebus import ServiceBusMessage
from azure.identity.aio import DefaultAzureCredential
```

2. Add the constants and define a credential.

Python

```
FULLY_QUALIFIED_NAMESPACE = "FULLY_QUALIFIED_NAMESPACE"
TOPIC_NAME = "TOPIC_NAME"

credential = DefaultAzureCredential()
```

#### ⓘ Important

- Replace `FULLY_QUALIFIED_NAMESPACE` with the fully qualified namespace for your Service Bus namespace.
- Replace `TOPIC_NAME` with the name of the topic.

In the preceding code, you used the Azure Identity client library's `DefaultAzureCredential` class. When the app runs locally during development, `DefaultAzureCredential` will automatically discover and authenticate to Azure using the account you logged into the Azure CLI with. When the app is deployed to Azure, `DefaultAzureCredential` can authenticate your app to Microsoft Entra ID via a managed identity without any code changes.

3. Add a method to send a single message.

Python

```
async def send_single_message(sender):
 # Create a Service Bus message
 message = ServiceBusMessage("Single Message")
```

```
send the message to the topic
await sender.send_messages(message)
print("Sent a single message")
```

The sender is an object that acts as a client for the topic you created. You'll create it later and send as an argument to this function.

#### 4. Add a method to send a list of messages.

Python

```
async def send_a_list_of_messages(sender):
 # Create a list of messages
 messages = [ServiceBusMessage("Message in list") for _ in
range(5)]
 # send the list of messages to the topic
 await sender.send_messages(messages)
 print("Sent a list of 5 messages")
```

#### 5. Add a method to send a batch of messages.

Python

```
async def send_batch_message(sender):
 # Create a batch of messages
 async with sender:
 batch_message = await sender.create_message_batch()
 for _ in range(10):
 try:
 # Add a message to the batch

 batch_message.add_message(ServiceBusMessage("Message inside a
ServiceBusMessageBatch"))
 except ValueError:
 # ServiceBusMessageBatch object reaches max_size.
 # New ServiceBusMessageBatch object can be created
here to send more data.
 break
 # Send the batch of messages to the topic
 await sender.send_messages(batch_message)
 print("Sent a batch of 10 messages")
```

#### 6. Create a Service Bus client and then a topic sender object to send messages.

Python

```
async def run():
 # create a Service Bus client using the credential.
 async with ServiceBusClient(
 fully_qualified_namespace=FULLY_QUALIFIED_NAMESPACE,
```

```
 credential=credential,
 logging_enable=True) as servicebus_client:
 # Get a Topic Sender object to send messages to the topic
 sender =
 servicebus_client.get_topic_sender(topic_name=TOPIC_NAME)
 async with sender:
 # Send one message
 await send_single_message(sender)
 # Send a list of messages
 await send_a_list_of_messages(sender)
 # Send a batch of messages
 await send_batch_message(sender)
 # Close credential when no longer needed.
 await credential.close()

asyncio.run(run())
print("Done sending messages")
print("-----")
```

## Receive messages from a subscription

The following sample code shows you how to receive messages from a subscription. This code continually receives new messages until it doesn't receive any new messages for 5 (`max_wait_time`) seconds.

Open your favorite editor, such as [Visual Studio Code](#), create a file `recv.py`, and add the following code into it.

Passwordless (Recommended)

1. Similar to the send sample, add `import` statements, define constants that you should replace with your own values, and define a credential.

Python

```
import asyncio
from azure.servicebus.aio import ServiceBusClient
from azure.identity.aio import DefaultAzureCredential

FULLY_QUALIFIED_NAMESPACE = "FULLY_QUALIFIED_NAMESPACE"
SUBSCRIPTION_NAME = "SUBSCRIPTION_NAME"
TOPIC_NAME = "TOPIC_NAME"

credential = DefaultAzureCredential()
```

2. Create a Service Bus client and then a subscription receiver object to receive messages.

Python

```
async def run():
 # create a Service Bus client using the credential
 async with ServiceBusClient(
 fully_qualified_namespace=FULLY_QUALIFIED_NAMESPACE,
 credential=credential,
 logging_enable=True) as servicebus_client:

 async with servicebus_client:
 # get the Subscription Receiver object for the
 subscription
 receiver =
servicebus_client.get_subscription_receiver(topic_name=TOPIC_NAME,
 subscription_name=SUBSCRIPTION_NAME, max_wait_time=5)
 async with receiver:
 received_msgs = await
receiver.receive_messages(max_wait_time=5, max_message_count=20)
 for msg in received_msgs:
 print("Received: " + str(msg))
 # complete the message so that the message is
removed from the subscription
 await receiver.complete_message(msg)
 # Close credential when no longer needed.
 await credential.close()
```

3. Call the `run` method.

Python

```
asyncio.run(run())
```

## Run the app

Open a command prompt that has Python in its path, and then run the code to send and receive messages for a subscription under a topic.

shell

```
python send.py; python recv.py
```

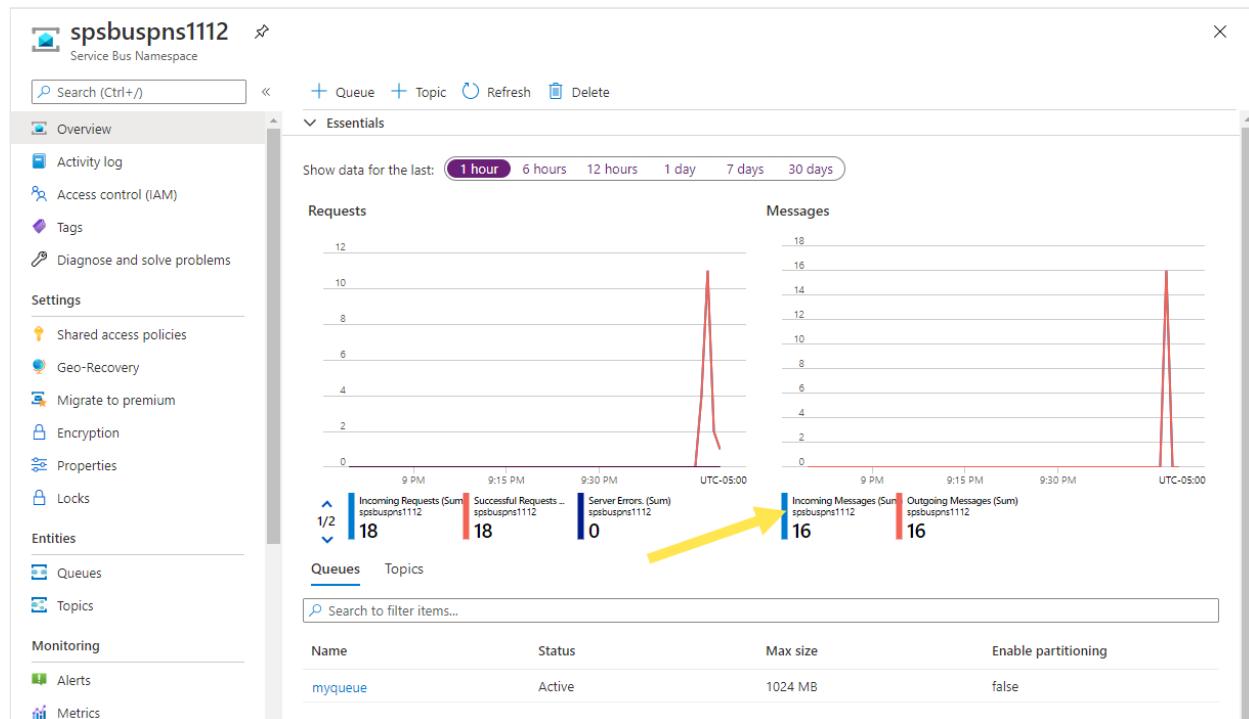
You should see the following output:

## Console

```
Sent a single message
Sent a list of 5 messages
Sent a batch of 10 messages
Done sending messages

Received: Single Message
Received: Message in list
Received: Message inside a ServiceBusMessageBatch
```

In the Azure portal, navigate to your Service Bus namespace. On the **Overview** page, verify that the **incoming** and **outgoing** message counts are 16. If you don't see the counts, refresh the page after waiting for a few minutes.



Select the topic in the bottom pane to see the **Service Bus Topic** page for your topic. On this page, you should see three incoming and three outgoing messages in the **Messages** chart.

**mytopic (spsbuspns1112/mytopic)**

Service Bus Topic

Search (Ctrl+ /) <> + Subscription Delete Refresh

Overview

Access control (IAM)  
Diagnose and solve problems  
Settings Shared access policies Service Bus Explorer (preview)  
Properties Locks  
Entities Subscriptions Automation Tasks Export template Support + troubleshooting New support request

Essentials

Current size 0.0 kB Max size 1 GB (change) Message time to live 14 DAYS (change) Auto-delete NEVER (change) Free space 100.0 %

Metrics

Show data for the last: 1 hour 6 hours 12 hours 1 day 7 days 30 days

Requests

17 Incoming Requests (Sum) spibuspns1112 14 Successful Requests ... spibuspns1112 0 Server Errors, (Sum) spibuspns1112

Messages

16 Incoming Messages (Sum) spibuspns1112 16 Outgoing Messages (Sum) spibuspns1112

Subscriptions

Search to filter items...

Name	Status	Message count	Max delivery count	Sessions Enabled
S1	Active	0	3	false

On this page, if you select a subscription, you get to the **Service Bus Subscription** page. You can see the active message count, dead-letter message count, and more on this page. In this example, all the messages have been received, so the active message count is zero.

**S1 (spsbuspns1112/mytopic/S1)**

Service Bus Subscription

Search (Ctrl+ /) <> Delete Refresh

Overview

Diagnose and solve problems  
Automation Tasks Export template Support + troubleshooting New support request

Essentials

Max delivery count 3 (change) Message time to live 14 DAYS (change) Message lock duration 30 SECONDS (change) Auto-delete after idle for 14 DAYS (change)

Active message count 0 MESSAGES Dead-letter message count 0 MESSAGES Transfer message count 0 MESSAGES Scheduled message count 0 MESSAGES Transfer message count 0 MESSAGES

FILTERS

+ Add filter

Name	Filter Type
\$Default	SqlFilter

If you comment out the receive code, you'll see the active message count as 16.

S1 (spsbuspns1112/mytopic/S1)  
Service Bus Subscription

Search (Ctrl+ /) < Delete Refresh

Overview Essentials

Diagnose and solve problems

Automation

Tasks → Active message count 16 MESSAGES

Export template

Support + troubleshooting

New support request

FILTERS

+ Add filter

Name	Filter Type
\$Default	SqlFilter

## Next steps

See the following documentation and samples:

- [Azure Service Bus client library for Python ↗](#)
- [Samples ↗](#).
  - The **sync\_samples** folder has samples that show you how to interact with Service Bus in a synchronous manner. In this quick start, you used this method.
  - The **async\_samples** folder has samples that show you how to interact with Service Bus in an asynchronous manner.
- [azure-servicebus reference documentation](#)

# Quickstart: Azure Blob Storage client library for Python

Article • 02/23/2024

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

## ⓘ Note

The **Build from scratch** option walks you step by step through the process of creating a new project, installing packages, writing the code, and running a basic console app. This approach is recommended if you want to understand all the details involved in creating an app that connects to Azure Blob Storage. If you prefer to automate deployment tasks and start with a completed project, choose [Start with a template](#).

Get started with the Azure Blob Storage client library for Python to manage blobs and containers.

In this article, you follow steps to install the package and try out example code for basic tasks.

[API reference documentation](#) | [Library source code](#) ↗ | [Package \(PyPi\)](#) ↗ | [Samples](#)

This video shows you how to start using the Azure Blob Storage client library for Python.  
<https://learn-video.azurefd.net/vod/player?id=f663a554-96ca-4bc3-b3b1-48376a7efbd&locale=en-us&embedUrl=%2Fazure%2Fstorage%2Fblobs%2Fstorage-quickstart-blobs-python> ↗

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

## Prerequisites

- Azure account with an active subscription - [create an account for free](#) ↗
- Azure Storage account - [create a storage account](#)
- [Python](#) ↗ 3.8+

## Setting up

This section walks you through preparing a project to work with the Azure Blob Storage client library for Python.

## Create the project

Create a Python application named *blob-quickstart*.

1. In a console window (such as PowerShell or Bash), create a new directory for the project:

```
Console
mkdir blob-quickstart
```

2. Switch to the newly created *blob-quickstart* directory:

```
Console
cd blob-quickstart
```

## Install the packages

From the project directory, install packages for the Azure Blob Storage and Azure Identity client libraries using the `pip install` command. The `azure-identity` package is needed for passwordless connections to Azure services.

```
Console
pip install azure-storage-blob azure-identity
```

## Set up the app framework

From the project directory, follow steps to create the basic structure of the app:

1. Open a new text file in your code editor.
2. Add `import` statements, create the structure for the program, and include basic exception handling, as shown below.
3. Save the new file as `blob_quickstart.py` in the `blob-quickstart` directory.

```
Python
```

```

import os, uuid
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient, BlobClient,
ContainerClient

try:
 print("Azure Blob Storage Python quickstart sample")

 # Quickstart code goes here

except Exception as ex:
 print('Exception:')
 print(ex)

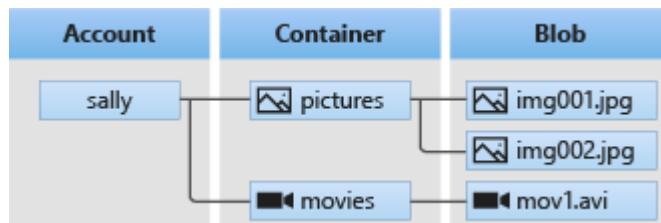
```

## Object model

Azure Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that doesn't adhere to a particular data model or definition, such as text or binary data. Blob storage offers three types of resources:

- The storage account
- A container in the storage account
- A blob in the container

The following diagram shows the relationship between these resources:



Use the following Python classes to interact with these resources:

- [BlobServiceClient](#): The `BlobServiceClient` class allows you to manipulate Azure Storage resources and blob containers.
- [ContainerClient](#): The `ContainerClient` class allows you to manipulate Azure Storage containers and their blobs.
- [BlobClient](#): The `BlobClient` class allows you to manipulate Azure Storage blobs.

## Code examples

These example code snippets show you how to do the following tasks with the Azure Blob Storage client library for Python:

- Authenticate to Azure and authorize access to blob data
- Create a container
- Upload blobs to a container
- List the blobs in a container
- Download blobs
- Delete a container

## Authenticate to Azure and authorize access to blob data

Application requests to Azure Blob Storage must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code, including Blob Storage.

You can also authorize requests to Azure Blob Storage by using the account access key. However, this approach should be used with caution. Developers must be diligent to never expose the access key in an unsecure location. Anyone who has the access key is able to authorize requests against the storage account, and effectively has access to all the data. `DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

### Passwordless (Recommended)

`DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` looks for credentials can be found in the [Azure Identity library overview](#).

For example, your app can authenticate using your Azure CLI sign-in credentials with when developing locally. Your app can then use a [managed identity](#) once it has been deployed to Azure. No code changes are required for this transition.

## Assign roles to your Microsoft Entra user account

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the

User Access Administrator role, or another role that includes the [Microsoft.Authorization/roleAssignments/write](#) action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

 **Important**

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the 'identitymigrationstorage | Access Control (IAM)' blade in the Azure portal. The left sidebar has 'Access Control (IAM)' selected. The main area shows a table with columns: 'Add', 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. A red box highlights the 'Add' button. Below the table, there's a section titled 'My access' with a 'View my access' button. Another section titled 'Check access' allows selecting between 'User, group, or service principal' (which is selected) and 'Managed identity'. A search bar is also present. To the right, there are two sections: 'Grant access to this resource' (with a 'Add role assignment' button) and 'View deny assignments' (with a 'View' button and a magnifying glass icon).

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Sign in and connect your app code to Azure using DefaultAzureCredential

You can authorize access to data in your storage account using the following steps:

1. Make sure you're authenticated with the same Microsoft Entra account you assigned the role to on your storage account. You can authenticate via the Azure CLI, Visual Studio Code, or Azure PowerShell.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

Azure CLI

```
az login
```

2. To use `DefaultAzureCredential`, make sure that the `azure-identity` package is [installed](#), and the class is imported:

Python

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

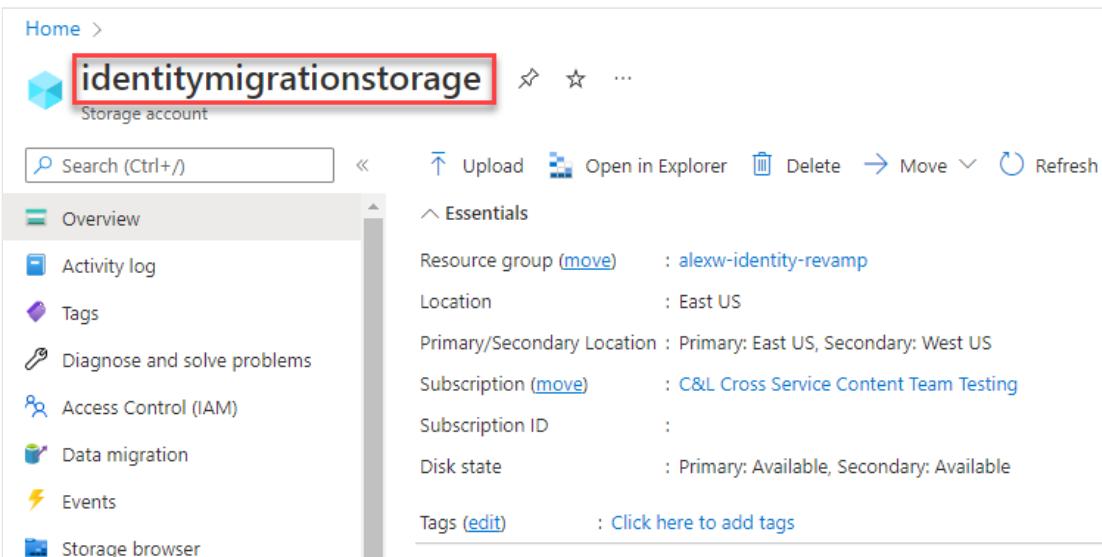
3. Add this code inside the `try` block. When the code runs on your local workstation, `DefaultAzureCredential` uses the developer credentials of the prioritized tool you're logged into to authenticate to Azure. Examples of these tools include Azure CLI or Visual Studio Code.

Python

```
account_url = "https://<storageaccountname>.blob.core.windows.net"
default_credential = DefaultAzureCredential()

Create the BlobServiceClient object
blob_service_client = BlobServiceClient(account_url,
 credential=default_credential)
```

4. Make sure to update the storage account name in the URI of your `BlobServiceClient` object. The storage account name can be found on the overview page of the Azure portal.



The screenshot shows the Azure Storage Account overview page for a storage account named "identitymigrationstorage". The account name is highlighted with a red box. The page displays various details about the storage account, including its resource group, location, primary/secondary location, subscription, disk state, and tags. A sidebar on the left provides links to other account management features like Activity log, Tags, and Data migration.

Resource group	: alexw-identity-revamp
Location	: East US
Primary/Secondary Location	: Primary: East US, Secondary: West US
Subscription	: C&L Cross Service Content Team Testing
Subscription ID	:
Disk state	: Primary: Available, Secondary: Available
Tags	: Click here to add tags

 Note

When deployed to Azure, this same code can be used to authorize requests to Azure Storage from an application running in Azure. However, you'll need to enable managed identity on your app in Azure. Then configure your storage account to allow that managed identity to connect. For detailed instructions on configuring this connection between Azure services, see the [Auth from Azure-hosted apps](#) tutorial.

## Create a container

Create a new container in your storage account by calling the `create_container` method on the `blob_service_client` object. In this example, the code appends a GUID value to the container name to ensure that it's unique.

Add this code to the end of the `try` block:

Python

```
Create a unique name for the container
container_name = str(uuid.uuid4())

Create the container
container_client = blob_service_client.create_container(container_name)
```

To learn more about creating a container, and to explore more code samples, see [Create a blob container with Python](#).

### ⓘ Important

Container names must be lowercase. For more information about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

## Upload blobs to a container

Upload a blob to a container using `upload_blob`. The example code creates a text file in the local `data` directory to upload to the container.

Add this code to the end of the `try` block:

Python

```

Create a local directory to hold blob data
local_path = "./data"
os.mkdir(local_path)

Create a file in the local data directory to upload and download
local_file_name = str(uuid.uuid4()) + ".txt"
upload_file_path = os.path.join(local_path, local_file_name)

Write text to the file
file = open(file=upload_file_path, mode='w')
file.write("Hello, World!")
file.close()

Create a blob client using the local file name as the name for the blob
blob_client = blob_service_client.get_blob_client(container=container_name,
blob=local_file_name)

print("\nUploading to Azure Storage as blob:\n\t" + local_file_name)

Upload the created file
with open(file=upload_file_path, mode="rb") as data:
 blob_client.upload_blob(data)

```

To learn more about uploading blobs, and to explore more code samples, see [Upload a blob with Python](#).

## List the blobs in a container

List the blobs in the container by calling the `list_blobs` method. In this case, only one blob has been added to the container, so the listing operation returns just that one blob.

Add this code to the end of the `try` block:

Python

```

print("\nListing blobs...")

List the blobs in the container
blob_list = container_client.list_blobs()
for blob in blob_list:
 print("\t" + blob.name)

```

To learn more about listing blobs, and to explore more code samples, see [List blobs with Python](#).

## Download blobs

Download the previously created blob by calling the `download_blob` method. The example code adds a suffix of "DOWNLOAD" to the file name so that you can see both files in local file system.

Add this code to the end of the `try` block:

Python

```
Download the blob to a local file
Add 'DOWNLOAD' before the .txt extension so you can see both files in the
data directory
download_file_path = os.path.join(local_path, str.replace(local_file_name
 , '.txt', 'DOWNLOAD.txt'))
container_client = blob_service_client.get_container_client(container=
 container_name)
print("\nDownloading blob to \n\t" + download_file_path)

with open(file=download_file_path, mode="wb") as download_file:
 download_file.write(container_client.download_blob(blob.name).readall())
```

To learn more about downloading blobs, and to explore more code samples, see [Download a blob with Python](#).

## Delete a container

The following code cleans up the resources the app created by removing the entire container using the `delete_container` method. You can also delete the local files, if you like.

The app pauses for user input by calling `input()` before it deletes the blob, container, and local files. Verify that the resources were created correctly before they're deleted.

Add this code to the end of the `try` block:

Python

```
Clean up
print("\nPress the Enter key to begin clean up")
input()

print("Deleting blob container...")
container_client.delete_container()

print("Deleting the local source and downloaded files...")
os.remove(upload_file_path)
os.remove(download_file_path)
os.rmdir(local_path)
```

```
print("Done")
```

To learn more about deleting a container, and to explore more code samples, see [Delete and restore a blob container with Python](#).

## Run the code

This app creates a test file in your local folder and uploads it to Azure Blob Storage. The example then lists the blobs in the container, and downloads the file with a new name. You can compare the old and new files.

Navigate to the directory containing the *blob\_quickstart.py* file, then execute the following `python` command to run the app:

```
Console
python blob_quickstart.py
```

The output of the app is similar to the following example (UUID values omitted for readability):

```
Output

Azure Blob Storage Python quickstart sample

Uploading to Azure Storage as blob:
 quickstartUUID.txt

Listing blobs...
 quickstartUUID.txt

Downloading blob to
 ./data/quickstartUUIDDOWNLOAD.txt

Press the Enter key to begin clean up

Deleting blob container...
Deleting the local source and downloaded files...
Done
```

Before you begin the cleanup process, check your *data* folder for the two files. You can compare them and observe that they're identical.

## Clean up resources

After you've verified the files and finished testing, press the **Enter** key to delete the test files along with the container you created in the storage account. You can also use [Azure CLI](#) to delete resources.

## Next steps

In this quickstart, you learned how to upload, download, and list blobs using Python.

To see Blob storage sample apps, continue to:

[Azure Blob Storage library for Python samples](#)

- To learn more, see the [Azure Blob Storage client libraries for Python](#).
- For tutorials, samples, quickstarts, and other documentation, visit [Azure for Python Developers](#).

# Quickstart: Azure Queue Storage client library for Python

Article • 06/29/2023

Get started with the Azure Queue Storage client library for Python. Azure Queue Storage is a service for storing large numbers of messages for later retrieval and processing. Follow these steps to install the package and try out example code for basic tasks.

[API reference documentation](#) | [Library source code](#) | [Package \(Python Package Index\)](#) | [Samples](#)

Use the Azure Queue Storage client library for Python to:

- Create a queue
- Add messages to a queue
- Peek at messages in a queue
- Update a message in a queue
- Get the queue length
- Receive messages from a queue
- Delete messages from a queue
- Delete a queue

## Prerequisites

- Azure subscription - [create one for free](#)
- Azure Storage account - [create a storage account](#)
- [Python](#) 3.7+

## Setting up

This section walks you through preparing a project to work with the Azure Queue Storage client library for Python.

### Create the project

Create a Python application named *queues-quickstart*.

1. In a console window (such as cmd, PowerShell, or Bash), create a new directory for the project.

Console

```
mkdir queues-quickstart
```

2. Switch to the newly created *queues-quickstart* directory.

Console

```
cd queues-quickstart
```

## Install the packages

From the project directory, install the Azure Queue Storage client library for Python package by using the `pip install` command. The `azure-identity` package is needed for passwordless connections to Azure services.

Console

```
pip install azure-storage-queue azure-identity
```

## Set up the app framework

1. Open a new text file in your code editor
2. Add `import` statements
3. Create the structure for the program, including basic exception handling

Here's the code:

Python

```
import os, uuid
from azure.identity import DefaultAzureCredential
from azure.storage.queue import QueueServiceClient, QueueClient,
QueueMessage, BinaryBase64DecodePolicy, BinaryBase64EncodePolicy

try:
 print("Azure Queue storage - Python quickstart sample")
 # Quickstart code goes here
except Exception as ex:
 print('Exception:')
 print(ex)
```

4. Save the new file as `queues-quickstart.py` in the `queues-quickstart` directory.

## Authenticate to Azure

Application requests to most Azure services must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code.

You can also authorize requests to Azure services using passwords, connection strings, or other credentials directly. However, this approach should be used with caution. Developers must be diligent to never expose these secrets in an unsecure location. Anyone who gains access to the password or secret key is able to authenticate.

`DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

### Passwordless (Recommended)

`DefaultAzureCredential` is a class provided by the Azure Identity client library for Python. To learn more about `DefaultAzureCredential`, see the [DefaultAzureCredential overview](#). `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

For example, your app can authenticate using your Visual Studio Code sign-in credentials when developing locally, and then use a [managed identity](#) once it has been deployed to Azure. No code changes are required for this transition.

When developing locally, make sure that the user account that is accessing the queue data has the correct permissions. You'll need [Storage Queue Data Contributor](#) to read and write queue data. To assign yourself this role, you'll need to be assigned the [User Access Administrator](#) role, or another role that includes the [Microsoft.Authorization/roleAssignments/write](#) action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the [scope overview](#) page.

In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the [Principle of Least Privilege](#). This practice gives users

only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Queue Data Contributor** role to your user account, which provides both read and write access to queue data in your storage account.

### ⓘ Important

In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.

The screenshot shows the Azure portal interface for managing access control. The left sidebar lists various storage account components like Overview, Activity log, Tags, and Queues. The 'Access Control (IAM)' section is highlighted with a red box. The main content area shows the 'Role assignments' tab selected. A red box highlights the '+ Add' button in the top navigation bar. A dropdown menu is open, with 'Add role assignment' highlighted by a red box. To the right, there are sections for 'Grant access to this resource' and 'View deny assignments', each with a 'Learn more' link.

5. Use the search box to filter the results to the desired role. For this example, search for *Storage Queue Data Contributor* and select the

matching result and then choose **Next**.

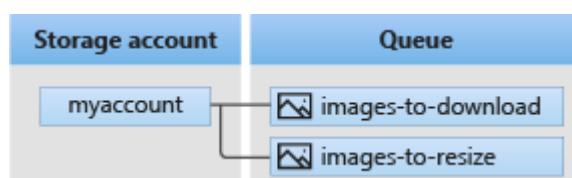
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the dialog.
8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

## Object model

Azure Queue Storage is a service for storing large numbers of messages. A queue message can be up to 64 KB in size. A queue may contain millions of messages, up to the total capacity limit of a storage account. Queues are commonly used to create a backlog of work to process asynchronously. Queue Storage offers three types of resources:

- **Storage account:** All access to Azure Storage is done through a storage account. For more information about storage accounts, see [Storage account overview](#)
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. A message can remain in the queue for a maximum of 7 days. For version 2017-07-29 or later, the maximum time-to-live can be any positive number, or -1 indicating that the message doesn't expire. If this parameter is omitted, the default time-to-live is seven days.

The following diagram shows the relationship between these resources.



Use the following Python classes to interact with these resources:

- **QueueServiceClient:** The `QueueServiceClient` allows you to manage the all queues in your storage account.

- **QueueClient**: The `QueueClient` class allows you to manage and manipulate an individual queue and its messages.
- **QueueMessage**: The `QueueMessage` class represents the individual objects returned when calling `receive_messages` on a queue.

## Code examples

These example code snippets show you how to do the following actions with the Azure Queue Storage client library for Python:

- Authorize access and create a client object
- Create a queue
- Add messages to a queue
- Peek at messages in a queue
- Update a message in a queue
- Get the queue length
- Receive messages from a queue
- Delete messages from a queue
- Delete a queue

Passwordless (Recommended)

### Authorize access and create a client object

Make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via Azure CLI, Visual Studio Code, or Azure PowerShell.

Azure CLI

Sign-in to Azure through the Azure CLI using the following command:

Azure CLI

```
az login
```

Once authenticated, you can create and authorize a `QueueClient` object using `DefaultAzureCredential` to access queue data in the storage account.

`DefaultAzureCredential` automatically discovers and uses the account you signed in with in the previous step.

To authorize using `DefaultAzureCredential`, make sure you've added the `azure-identity` package, as described in [Install the packages](#). Also, be sure to add the following import statement in the `queues-quickstart.py` file:

Python

```
from azure.identity import DefaultAzureCredential
```

Decide on a name for the queue and create an instance of the `QueueClient` class, using `DefaultAzureCredential` for authorization. We use this client object to create and interact with the queue resource in the storage account.

### ⓘ Important

Queue names may only contain lowercase letters, numbers, and hyphens, and must begin with a letter or a number. Each hyphen must be preceded and followed by a non-hyphen character. The name must also be between 3 and 63 characters long. For more information about naming queues, see [Naming queues and metadata](#).

Add the following code inside the `try` block, and make sure to replace the `<storage-account-name>` placeholder value:

Python

```
print("Azure Queue storage - Python quickstart sample")

Create a unique name for the queue
queue_name = "quickstartqueues-" + str(uuid.uuid4())

account_url = "https://<storageaccountname>.queue.core.windows.net"
default_credential = DefaultAzureCredential()

Create the QueueClient object
We'll use this object to create and interact with the queue
queue_client = QueueClient(account_url, queue_name=queue_name
, credential=default_credential)
```

Queue messages are stored as text. If you want to store binary data, set up Base64 encoding and decoding functions before putting a message in the queue.

You can configure Base64 encoding and decoding functions when creating the client object:

Python

```
Setup Base64 encoding and decoding functions
base64_queue_client = QueueClient.from_connection_string(
 conn_str=connect_str, queue_name=q_name,
 message_encode_policy =
BinaryBase64EncodePolicy(),
 message_decode_policy =
BinaryBase64DecodePolicy()
)
```

## Create a queue

Using the `QueueClient` object, call the `create_queue` method to create the queue in your storage account.

Add this code to the end of the `try` block:

Python

```
print("Creating queue: " + queue_name)

Create the queue
queue_client.create_queue()
```

## Add messages to a queue

The following code snippet adds messages to queue by calling the `send_message` method. It also saves the `QueueMessage` returned from the third `send_message` call. The `saved_message` is used to update the message content later in the program.

Add this code to the end of the `try` block:

Python

```
print("\nAdding messages to the queue...")

Send several messages to the queue
queue_client.send_message(u"First message")
queue_client.send_message(u"Second message")
saved_message = queue_client.send_message(u"Third message")
```

## Peek at messages in a queue

Peek at the messages in the queue by calling the `peek_messages` method. This method retrieves one or more messages from the front of the queue but doesn't alter the visibility of the message.

Add this code to the end of the `try` block:

Python

```
print("\nPeek at the messages in the queue...")

Peek at messages in the queue
peeked_messages = queue_client.peek_messages(max_messages=5)

for peeked_message in peeked_messages:
 # Display the message
 print("Message: " + peeked_message.content)
```

## Update a message in a queue

Update the contents of a message by calling the `update_message` method. This method can change a message's visibility timeout and contents. The message content must be a UTF-8 encoded string that is up to 64 KB in size. Along with the new content, pass in values from the message that was saved earlier in the code. The `saved_message` values identify which message to update.

Python

```
print("\nUpdating the third message in the queue...")

Update a message using the message saved when calling send_message
earlier
queue_client.update_message(saved_message,
pop_receipt=saved_message.pop_receipt, \
content="Third message has been updated")
```

## Get the queue length

You can get an estimate of the number of messages in a queue.

The `get_queue_properties` method returns queue properties including the `approximate_message_count`.

Python

```
properties = queue_client.get_queue_properties()
count = properties.approximate_message_count
print("Message count: " + str(count))
```

The result is approximate since messages can be added or removed after the service responds to your request.

## Receive messages from a queue

You can download previously added messages by calling the `receive_messages` method.

Add this code to the end of the `try` block:

Python

```
print("\nReceiving messages from the queue...")

Get messages from the queue
messages = queue_client.receive_messages(max_messages=5)
```

When calling the `receive_messages` method, you can optionally specify a value for `max_messages`, which is the number of messages to retrieve from the queue. The default is 1 message and the maximum is 32 messages. You can also specify a value for `visibility_timeout`, which hides the messages from other operations for the timeout period. The default is 30 seconds.

## Delete messages from a queue

Delete messages from the queue after they're received and processed. In this case, processing is just displaying the message on the console.

The app pauses for user input by calling `input` before it processes and deletes the messages. Verify in your [Azure portal](#) that the resources were created correctly, before they're deleted. Any messages not explicitly deleted eventually become visible in the queue again for another chance to process them.

Add this code to the end of the `try` block:

Python

```
print("\nPress Enter key to 'process' messages and delete them from the
queue...")
input()
```

```
for msg_batch in messages.by_page():
 for msg in msg_batch:
 # "Process" the message
 print(msg.content)
 # Let the service know we're finished with
 # the message and it can be safely deleted.
 queue_client.delete_message(msg)
```

## Delete a queue

The following code cleans up the resources the app created by deleting the queue using the [delete\\_queue](#) method.

Add this code to the end of the `try` block and save the file:

Python

```
print("\nPress Enter key to delete the queue...")
input()

Clean up
print("Deleting queue...")
queue_client.delete_queue()

print("Done")
```

## Run the code

This app creates and adds three messages to an Azure queue. The code lists the messages in the queue, then retrieves and deletes them, before finally deleting the queue.

In your console window, navigate to the directory containing the `queues-quickstart.py` file, then use the following `python` command to run the app.

Console

```
python queues-quickstart.py
```

The output of the app is similar to the following example:

Output

```
Azure Queue Storage client library - Python quickstart sample
Creating queue: quickstartqueues-<UUID>
```

```
Adding messages to the queue...
```

```
Peek at the messages in the queue...
```

```
Message: First message
```

```
Message: Second message
```

```
Message: Third message
```

```
Updating the third message in the queue...
```

```
Receiving messages from the queue...
```

```
Press Enter key to 'process' messages and delete them from the queue...
```

```
First message
```

```
Second message
```

```
Third message has been updated
```

```
Press Enter key to delete the queue...
```

```
Deleting queue...
```

```
Done
```

When the app pauses before receiving messages, check your storage account in the [Azure portal](#). Verify the messages are in the queue.

Press the `Enter` key to receive and delete the messages. When prompted, press the `Enter` key again to delete the queue and finish the demo.

## Next steps

In this quickstart, you learned how to create a queue and add messages to it using Python code. Then you learned to peek, retrieve, and delete messages. Finally, you learned how to delete a message queue.

For tutorials, samples, quick starts and other documentation, visit:

### Azure for Python developers

- For related code samples using deprecated Python version 2 SDKs, see [Code samples using Python version 2](#).
- To learn more, see the [Azure Storage libraries for Python](#).
- For more Azure Queue Storage sample apps, see [Azure Queue Storage client library for Python - samples](#).

# Use Azure OpenAI without keys

Article • 05/20/2024

Application requests to most Azure services must be authenticated with keys or [passwordless connections](#). Developers must be diligent to never expose the keys in an unsecure location. Anyone who gains access to the key is able to authenticate to the service. Passwordless authentication offers improved management and security benefits over the account key because there's no key (or connection string) to store.

Passwordless connections are enabled with the following steps:

- Configure your authentication.
- Set environment variables, as needed.
- Use an Azure Identity library credential type to create an Azure OpenAI client object.

## Authentication

Authentication to Microsoft Entra ID is required to use the Azure client libraries.

Authentication differs based on the environment in which the app is running:

- [Local development](#)
- [Azure](#)

## Authenticate for local development

Python

Select a tool for [authentication during local development](#).

## Authenticate for Azure-hosted environments

Python

Learn about how to manage the [DefaultAzureCredential](#) for applications deployed to Azure.

# Configure roles for authorization

1. Find the [role](#) for your usage of Azure OpenAI. Depending on how you intend to set that role, you'll need either the name or ID.

[+] [Expand table](#)

Role name	Role ID
For Azure CLI or Azure PowerShell, you can use role name. For Bicep, you need the role ID.	

2. For many Azure OpenAI chat completion use cases, the following role and ID are suggested.

[+] [Expand table](#)

Role name	Role ID
Cognitive Services OpenAI User	5e0bd9bd-7b93-4f28-af87-19fc36ad61bd

3. Select an identity type to use.

- **Personal identity:** This is your personal identity tied to your sign in to Azure.
- **Managed identity:** This is an identity managed by and created for use on Azure. For [managed identity](#), create a [user-assigned managed identity](#). When you create the managed identity, you need the `Client ID`, also known as the `app ID`.

4. To find your personal identity, use one of the following commands. Use the ID as the `<identity-id>` in the next step.

Azure CLI

For local development, to get your own identity ID, use the following command. You need to sign in with `az login` before using this command.

Azure CLI

```
az ad signed-in-user show \
--query id -o tsv
```

5. Assign the role-based access control (RBAC) role to the identity for the resource group.

## Azure CLI

To grant your identity permissions to your resource through RBAC, assign a role using the Azure CLI command [az role assignment create](#).

### Azure CLI

```
az role assignment create \
 --role "Cognitive Services OpenAI User" \
 --assignee "<identity-id>" \
 --scope "/subscriptions/<subscription-
 id>/resourceGroups/<resource-group-name>"
```

Where applicable, replace `<identity-id>`, `<subscription-id>`, and `<resource-group-name>` with your actual values.

## Configure environment variables

To connect to Azure OpenAI, your code needs to know your resource endpoint, and *may* need additional environment variables.

1. Create an environment variable for your Azure OpenAI endpoint.

- `AZURE_OPENAI_ENDPOINT`: This URL is the access point for your Azure OpenAI resource.

2. Create environment variables based on the location in which your app runs:

[+] [Expand table](#)

Location	Identity	Description
Local	Personal	For local runtimes with your <a href="#">personal identity</a> , <a href="#">sign in</a> to create your credential with a tool.
Azure cloud	User-assigned managed identity	Create an <code>AZURE_CLIENT_ID</code> environment variable containing the client ID of the user-assigned managed identity to authenticate as.

## Install Azure Identity client library

### Python

Install the Python Azure Identity client library [↗](#):

Console

```
pip install azure-identity
```

## Use DefaultAzureCredential

The Azure Identity library's `DefaultAzureCredential` allows the customer to run the same code in the local development environment and in the Azure Cloud.

Python

For more information on `DefaultAzureCredential` for Python, see [Azure Identity client library for Python](#).

Python

```
import openai
from azure.identity import DefaultAzureCredential,
get_bearer_token_provider

token_provider = get_bearer_token_provider(DefaultAzureCredential(),
"https://cognitiveservices.azure.com/.default")

openai_client = openai.AzureOpenAI(
 api_version=os.getenv("AZURE_OPENAI_VERSION"),
 azure_endpoint=os.getenv("AZURE_OPENAI_ENDPOINT"),
 azure_ad_token_provider=token_provider
)
```

## Additional resources

- [Passwordless connections developer guide](#)

## Feedback

Was this page helpful?

Yes

No

Get help at [Microsoft Q&A](#)

# Versioning policy for Azure services, SDKs, and CLI tools

Article • 12/12/2023

Most Azure services let you programmatically control and manage their resources with REST APIs. Services evolve through new published versions of their APIs with different contracts that add new features and/or modify their behaviors.

This article outlines the policy that the Azure service, SDK, and CLI teams use for versioning the Azure REST APIs. While Azure teams make every effort to adhere to this policy, deviations may occasionally occur.

## Service versioning

Each published version of an API is identified by a date value in `YYYY-MM-DD` format, called the `api-version`. Newer versions have later dates.

All API operations require clients to specify a valid API version for the service via the `api-version` query string parameter in the URL. For example:

`https://management.azure.com/subscriptions?api-version=2020-01-01`. Client SDKs and tools include the `api-version` value automatically. For more considerations, see the [Client SDKs and service versions](#) section later in this article.

Usually, published service versions remain available and supported for many years, even as newer versions become available. In most cases, the only time you should adopt a new service version within existing code is to take advantage of new features.

## Stable versions

Most service versions published are *stable versions*. Stable versions are backwards compatible, meaning that any code you write that relies on one version of a service can adopt a newer stable version without requiring any code changes to maintain correctness or existing functionality.

## Breaking change versions

A *breaking change version* of a service isn't backwards compatible. Adopting a breaking change version in existing client code may require code changes to ensure the client behaves exactly as it did when targeting the previous version.

Breaking change versions are rare, announced via documentation, and are typically preceded by publication of a preview version. Publication of a breaking change version may prompt the eventual retirement of existing stable versions, which will remain available for at least three years after the breaking change version releases. For breaking changes published due to security or compliance issues, existing stable service versions may remain available for one year or less depending on the severity of the issue.

Due to the rapid innovation and developments in AI, AI-driven services may have a reduced minimum availability of one year. Each service will publish its breaking change policy.

Any Azure service dependent on a non-Microsoft component can shrink its support policy to match that of the component's policy. Any breaking change due to this will link to the component vendor's policy showing the date when the component is no longer supported.

## Preview versions

Occasionally, Microsoft publishes a *preview version* of a service to gather feedback about proposed changes and new features. Preview service versions are identified with the suffix `-preview` in their `api-version` - for example, `2022-07-07-preview`.

Unless explicitly intended to introduce a breaking change from the previous stable version, new preview versions include all the features of the most recent stable version and add new preview features. However, between preview versions, a service may break any of the newly added preview features.

Previews aren't intended for long-term use. Anytime a new stable or preview version of a service becomes available, existing preview versions may become unavailable as early as 90 days from the availability of the new version. Use preview versions only in situations where you're actively developing against new service features and you're prepared to adopt a new, non-preview version soon after it's released. If some features from a preview version are released in a new stable version, remaining features still in preview will typically be published in a new preview version.

## Client SDKs and service versions

The [Azure SDKs](#) aim to eliminate service versioning as a concern when writing code. Each SDK is composed of client libraries, one for each service, and each client library version targets a single version of the service it relies on.

When you use an SDK to access an Azure service, taking advantage of new versions and features typically requires upgrading the client library version used by the application. New stable versions of services are accompanied by new point releases of client libraries. For new breaking change versions, new client libraries are published as either point release versions or major release versions. The type of release depends on the nature of the service's change and the way the library is able to accommodate it. Only beta-version client libraries use preview service versions.

SDK client libraries support manual overriding of the service version. Overriding a client library's default service version is an advanced scenario and may lead to unexpected behavior. If you make use of this feature, test your application thoroughly to ensure it works as desired.

## Azure command line tools

As with the SDKs, the Azure command line tools (including the [Azure CLI](#) and [Azure PowerShell](#)) are designed to allow usage of Azure management services without regard for versions. Accessing new service features often requires a new version of a tool. New backward-compatible tool versions are released monthly. Versions with breaking changes are released approximately twice a year, or as required to fix critical security issues.

The Azure command line tools may occasionally expose preview features. These commands are marked with a `Preview` label and will output a warning indicating limited support and potential changes in future tool versions.

## Next steps

- [Azure REST API specifications ↗](#)
- [Microsoft REST API guidelines ↗](#)
- [Azure SDK general guidelines ↗](#)