The Microsoft identity platform for developers is a set of tools that includes authentication service, open-source libraries, and application management tools.

After completing this module, you'll be able to:

- Identify the components of the Microsoft identity platform
- Describe the three types of service principals and how they relate to application objects
- Explain how permissions and user consent operate, and how conditional access impacts your application

Explore the Microsoft identity platform

Completed100 XP

- 3 minutes

The Microsoft identity platform helps you build applications your users and customers can sign in to using their Microsoft identities or social accounts, and provide authorized access to your own APIs or Microsoft APIs like Microsoft Graph.

There are several components that make up the Microsoft identity platform:

- **OAuth 2.0 and OpenID Connect standard-compliant authentication service** enabling developers to authenticate several identity types, including:
  - Work or school accounts, provisioned through Azure Active Directory
  - Personal Microsoft account, like Skype, Xbox, and Outlook.com
  - Social or local accounts, by using Azure Active Directory B2C
- **Open-source libraries**: Microsoft Authentication Libraries (MSAL) and support for other standards-compliant libraries
- **Application management portal**: A registration and configuration experience in the Azure portal, along with the other Azure management capabilities.
- **Application configuration API and PowerShell**: Programmatic configuration of your applications through the Microsoft Graph API and PowerShell so you can automate your DevOps tasks.

For developers, the Microsoft identity platform offers integration of modern innovations in the identity and security space like passwordless authentication, step-up authentication, and Conditional Access. You don’t need to implement such functionality yourself: applications integrated with the Microsoft identity platform natively take advantage of such innovations.

Explore service principals

Completed100 XP

- 3 minutes

To delegate Identity and Access Management functions to Azure Active Directory, an application must be registered with an Azure Active Directory tenant. When you register your application with Azure Active Directory, you're creating an identity configuration for your application that allows it to integrate with Azure Active Directory. When you register an app in the Azure portal, you choose whether it is:

- **Single tenant**: only accessible in your tenant
- **Multi-tenant**: accessible in other tenants

If you register an application in the portal, an application object (the globally unique instance of the app) and a service principal object are automatically created in your home tenant. You also have a globally unique ID for your app (the app or client ID). In the portal, you can then add secrets or certificates and scopes to make your app work, customize the branding of your app in the sign-in dialog, and more.

Note

You can also create service principal objects in a tenant using Azure PowerShell, Azure CLI, Microsoft Graph, and other tools.

Application object

An Azure Active Directory application is defined by its one and only application object. The application object resides in the Azure Active Directory tenant where the application was registered (known as the application's "home" tenant). An application object is used as a template or blueprint to create one or more service principal objects. A service principal is created in every tenant where the application is used. Similar to a class in object-oriented programming, the application object has some static properties that are applied to all the created service principals (or application instances).

The application object describes three aspects of an application: how the service can issue tokens in order to access the application, resources that the application might need to access, and the actions that the application can take.

The Microsoft Graph Application entity defines the schema for an application object's properties.

Service principal object

To access resources secured by an Azure Active Directory tenant, the entity that requires access must be represented by a security principal. This is true for both users (user principal) and applications (service principal).

The security principal defines the access policy and permissions for the user/application in the Azure Active Directory tenant. This enables core features such as authentication of the user/application during sign-in, and authorization during resource access.

There are three types of service principal:

- **Application** - This type of service principal is the local representation, or application instance, of a global application object in a single tenant or directory. A service principal is created in each tenant where the application is used and references the globally unique app object. The service principal object defines what the app can actually do in the specific tenant, who can access the app, and what resources the app can access.
- **Managed identity** - This type of service principal is used to represent a managed identity. Managed identities provide an identity for applications to use when connecting to resources that support Azure Active Directory authentication. When a managed identity is enabled, a service principal representing that managed identity is created in your tenant. Service principals representing managed identities can be granted access and permissions, but can't be updated or modified directly.
- **Legacy** - This type of service principal represents a legacy app, which is an app created before app registrations were introduced or an app created through legacy experiences. A legacy service principal can have credentials, service principal names, reply URLs, and other properties that an authorized user can edit, but doesn't have an associated app registration. The service principal can only be used in the tenant where it was created.

Relationship between application objects and service principals

The application object is the *global* representation of your application for use across all tenants, and the service principal is the *local* representation for use in a specific tenant. The application object serves as the template from which common and default properties are *derived* for use in creating corresponding service principal objects.

An application object has:

- A one to one relationship with the software application, and
- A one to many relationship with its corresponding service principal object(s).

A service principal must be created in each tenant where the application is used to enable it to establish an identity for sign-in and/or access to resources being secured by the tenant. A single-tenant application has only one service principal (in its home tenant), created and consented for use during application registration. A multi-tenant application also has a service principal created in each tenant where a user from that tenant has consented to its use.

Discover permissions and consent

Completed100 XP

- 3 minutes

Applications that integrate with the Microsoft identity platform follow an authorization model that gives users and administrators control over how data can be accessed.

The Microsoft identity platform implements the OAuth 2.0 authorization protocol. OAuth 2.0 is a method through which a third-party app can access web-hosted resources on behalf of a user. Any web-hosted resource that integrates with the Microsoft identity platform has a resource identifier, or *application ID URI*.

Here are some examples of Microsoft web-hosted resources:

- Microsoft Graph: `https://graph.microsoft.com`
- Microsoft 365 Mail API: `https://outlook.office.com`
- Azure Key Vault: `https://vault.azure.net`

The same is true for any third-party resources that have integrated with the Microsoft identity platform. Any of these resources also can define a set of permissions that can be used to divide the functionality of that resource into smaller chunks. When a resource's functionality is chunked into small permission sets, third-party apps can be built to request only the permissions that they need to perform their function. Users and administrators can know what data the app can access.

In OAuth 2.0, these types of permission sets are called *scopes*. They're also often referred to as *permissions*. In the Microsoft identity platform, a permission is represented as a string value. An app requests the permissions it needs by specifying the permission in the `scope` query parameter. Identity platform supports several well-defined OpenID Connect scopes and resource-based permissions (each permission is indicated by appending the permission value to the resource's identifier or application ID URI). For example, the permission string `https://graph.microsoft.com/Calendars.Read` is used to request permission to read users calendars in Microsoft Graph.

An app most commonly requests these permissions by specifying the scopes in requests to the Microsoft identity platform authorize endpoint. However, some high-privilege permissions can be granted only through administrator consent. They can be requested or granted by using the administrator consent endpoint.

Note

resource's identifier == the application ID URI

In requests to the authorization, token or consent endpoints for the Microsoft Identity platform, if the resource identifier is omitted in the scope parameter, the resource is assumed to be Microsoft Graph. For example, `scope=User.Read` is equivalent to `https://graph.microsoft.com/User.Read`.

Permission types

The Microsoft identity platform supports two types of permissions: *delegated permissions* and *app-only access*.

- **Delegated permissions** are used by apps that have a signed-in user present. For these apps, either the user or an administrator consents to the permissions that the app requests. The app is delegated with the permission to act as a signed-in user when it makes calls to the target resource.

  ** App acts on behalf of user

- **App-only access permissions** are used by apps that run without a signed-in user present, for example, apps that run as background services or daemons. Only an administrator can consent to app-only access permissions.

Consent types

Applications in Microsoft identity platform rely on consent in order to gain access to necessary resources or APIs. There are many kinds of consent that your app may need to know about in order to be successful. If you're defining permissions, you'll also need to understand how your users gain access to your app or API.

There are three consent types: *static user consent*, *incremental and dynamic user consent*, and *admin consent*.

Static user consent

In the static user consent scenario, you must specify all the permissions it needs in the app's configuration in the Azure portal. If the user (or administrator, as appropriate) hasn't granted consent for this app, then Microsoft identity platform prompts the user to provide consent at this time. Static permissions also enable administrators to consent on behalf of all users in the organization.

While static permissions of the app defined in the Azure portal keep the code nice and simple, it presents some possible issues for developers:

- The app needs to request all the permissions it would ever need upon the user's first sign-in. This can lead to a long list of permissions that discourages end users from approving the app's access on initial sign-in.
- The app needs to know all of the resources it would ever access ahead of time. It's difficult to create apps that could access an arbitrary number of resources.

Incremental and dynamic user consent

With the Microsoft identity platform endpoint, you can ignore the static permissions defined in the app registration information in the Azure portal and request permissions incrementally instead. You can ask for a minimum set of permissions upfront and request more over time as the customer uses more app features.

To do so, you can specify the scopes your app needs at any time by including the new scopes in the `scope` parameter when requesting an access token - without the need to predefine them in the application registration information. If the user hasn't yet consented to new scopes added to the request, they're prompted to consent only to the new permissions. Incremental, or dynamic consent, only applies to delegated permissions and not to app-only access permissions.

Important

Dynamic consent can be convenient, but presents a big challenge for permissions that require admin consent, since the admin consent experience doesn't know about those permissions at consent time. If you require admin privileged permissions or if your app uses dynamic consent, you must register all of the permissions in the Azure portal (not just the subset of permissions that require admin consent). This enables tenant admins to consent on behalf of all their users.

Admin consent

Admin consent is required when your app needs access to certain high-privilege permissions. Admin consent ensures that administrators have some other controls before authorizing apps or users to access highly privileged data from the organization.

Admin consent done on behalf of an organization still requires the static permissions registered for the app. Set those permissions for apps in the app registration portal if you need an admin to give consent on behalf of the entire organization. This reduces the cycles required by the organization admin to set up the application.

Requesting individual user consent

In an OpenID Connect or OAuth 2.0 authorization request, an app can request the permissions it needs by using the scope query parameter. For example, when a user signs in to an app, the app sends a request like the following example. Line breaks are added for legibility.

HTTPCopy

```
GET https://login.microsoftonline.com/common/oauth2/v2.0/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&response_type=code
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&response_mode=query
&scope=
https%3A%2F%2Fgraph.microsoft.com%2Fcalendars.read%20
https%3A%2F%2Fgraph.microsoft.com%2Fmail.send
&state=12345
```

The `scope` parameter is a space-separated list of delegated permissions that the app is requesting. Each permission is indicated by appending the permission value to the resource's identifier (the application ID URI). In the request example, the app needs permission to read the user's calendar and send mail as the user.

After the user enters their credentials, the Microsoft identity platform checks for a matching record of *user consent*. If the user hasn't consented to any of the requested permissions in the past, and if the administrator hasn't consented to these permissions on behalf of the entire organization, the Microsoft identity platform asks the user to grant the requested permissions.

Discover conditional access

Completed100 XP

- 3 minutes

The Conditional Access feature in Azure Active Directory offers one of several ways that you can use to secure your app and protect a service. Conditional Access enables developers and enterprise customers to protect services in a multitude of ways including:

- Multifactor authentication
- Allowing only Intune enrolled devices to access specific services
- Restricting user locations and IP ranges

How does Conditional Access impact an app?

In most common cases, Conditional Access doesn't change an app's behavior or require any changes from the developer. Only in certain cases when an app indirectly or silently requests a token for a service does an app require code changes to handle Conditional Access challenges. It may be as simple as performing an interactive sign-in request.

Specifically, the following scenarios require code to handle Conditional Access challenges:

- Apps performing the on-behalf-of flow
- Apps accessing multiple services/resources

- Single-page apps using MSAL.js
- Web apps calling a resource

Conditional Access policies can be applied to the app and also a web API your app accesses. Depending on the scenario, an enterprise customer can apply and remove Conditional Access policies at any time. For your app to continue functioning when a new policy is applied, implement challenge handling.

Conditional Access examples

Some scenarios require code changes to handle Conditional Access whereas others work as is. Here are a few scenarios using Conditional Access to do multifactor authentication that gives some insight into the difference.

- You're building a single-tenant iOS app and apply a Conditional Access policy. The app signs in a user and doesn't request access to an API. When the user signs in, the policy is automatically invoked and the user needs to perform multifactor authentication.
- You're building an app that uses a middle tier service to access a downstream API. An enterprise customer at the company using this app applies a policy to the downstream API. When an end user signs in, the app requests access to the middle tier and sends the token. The middle tier performs on-behalf-of flow to request access to the downstream API. At this point, a claims "challenge" is presented to the middle tier. The middle tier sends the challenge back to the app, which needs to comply with the Conditional Access policy.

On-Behalf-Of flows require conditional Access Policy

Check your knowledge

1. Which of the types of permissions supported by the Microsoft identity platform is used by apps that have a signed-in user present?

- ● Delegated permissions

  ✓ **Correct. Delegated permissions are used by apps that have a signed-in user present. The app is delegated with the permission to act as a signed-in user when it makes calls to the target resource.**

- ○ App-only access permissions
- ○ Both delegated and app-only access permissions

2. Which of the following app scenarios require code to handle Conditional Access challenges?

- ○ Apps performing the device-code flow
- ● Apps performing the on-behalf-of flow

  ✓ **Correct. Apps performing the on-behalf-of flow require code to handle Conditional Access challenges.**

- ○ Apps performing the Integrated Windows authentication flow

Implement authentication by using the Microsoft Authentication Library

Introduction

Completed100 XP

- 3 minutes

The Microsoft Authentication Library (MSAL) enables developers to acquire tokens from the Microsoft identity platform in order to authenticate users and access secured web APIs.

After completing this module, you'll be able to:

- Explain the benefits of using MSAL and the application types and scenarios it supports
- Instantiate both public and confidential client apps from code
- Register an app with the Microsoft identity platform
- Create an app that retrieves a token by using the MSAL.NET library

Explore the Microsoft Authentication Library

Completed100 XP

- 3 minutes

The Microsoft Authentication Library (MSAL) can be used to provide secure access to Microsoft Graph, other Microsoft APIs, third-party web APIs, or your own web API. MSAL supports many different application architectures and platforms including .NET, JavaScript, Java, Python, Android, and iOS.

MSAL gives you many ways to get tokens, with a consistent API for many platforms. Using MSAL provides the following benefits:

- No need to directly use the OAuth libraries or code against the protocol in your application.
- Acquires tokens on behalf of a user or on behalf of an application (when applicable to the platform).

- Maintains a token cache and refreshes tokens for you when they're close to expire. You don't need to handle token expiration on your own.
- Helps you specify which audience you want your application to sign in.
- Helps you set up your application from configuration files.
- Helps you troubleshoot your app by exposing actionable exceptions, logging, and telemetry.

Application types and scenarios

Using MSAL, a token can be acquired from many application types: web applications, web APIs, single-page apps (JavaScript), mobile and native applications, and daemons and server-side applications. MSAL currently supports the platforms and frameworks listed in the following table.

Library	**Supported platforms and frameworks**
MSAL for Android	Android
MSAL Angular	Single-page apps with Angular and Angular.js frameworks
MSAL for iOS and macOS	iOS and macOS
MSAL Go (Preview)	Windows, macOS, Linux
MSAL Java	Windows, macOS, Linux
MSAL.js	JavaScript/TypeScript frameworks such as Vue.js, Ember.js, or Durandal.js
MSAL.NET	.NET Framework, .NET Core, Xamarin Android, Xamarin iOS, Universal Windows Platform
MSAL Node	Web apps with Express, desktop apps with Electron, Cross-platform console apps
MSAL Python	Windows, macOS, Linux
MSAL React	Single-page apps with React and React-based libraries (Next.js, Gatsby.js)

Authentication flows

The following table shows some of the different authentication flows provided by Microsoft Authentication Library (MSAL). These flows can be used in various application scenarios.

Flow	**Description**
Authorization code	Native and web apps securely obtain tokens in the name of the user
Client credentials	Service applications run without user interaction
On-behalf-of	The application calls a service/web API, which in turns calls Microsoft Graph
Implicit	Used in browser-based applications
Device code	Enables sign-in to a device by using another device that has a browser
Integrated Windows	Windows computers silently acquire an access token when they're domain joined
Interactive	Mobile and desktops applications call Microsoft Graph in the name of a user
Username/password	The application signs in a user by using their username and password

Public client, and confidential client applications

Security tokens can be acquired by multiple types of applications. These applications tend to be separated into the following two categories. Each is used with different libraries and objects.

- **Public client applications**: Are apps that run on devices or desktop computers or in a web browser. They're not trusted to safely keep application secrets, so they only access web APIs on behalf of the user. (They support only public client flows.) Public clients can't hold configuration-time secrets, so they don't have client secrets.

- **Confidential client applications**: Are apps that run on servers (web apps, web API apps, or even service/daemon apps). They're considered difficult to access, and for that reason capable of keeping an application secret. Confidential clients can hold configuration-time secrets. Each instance of the client has a distinct configuration (including client ID and client secret).

Initialize client applications

Completed100 XP

- 3 minutes

With MSAL.NET 3.x, the recommended way to instantiate an application is by using the application builders: `PublicClientApplicationBuilder` and `ConfidentialClientApplicationBuilder`. They offer a powerful mechanism to configure the application either from the code, or from a configuration file, or even by mixing both approaches.

Before initializing an application, you first need to register it so that your app can be integrated with the Microsoft identity platform. After registration, you may need the following information (which can be found in the Azure portal):

- The client ID (a string representing a GUID)
- The identity provider URL (named the instance) and the sign-in audience for your application. These two parameters are collectively known as the authority.
- The tenant ID if you're writing a line of business application solely for your organization (also named single-tenant application).
- The application secret (client secret string) or certificate (of type X509Certificate2) if it's a confidential client app.
- For web apps, and sometimes for public client apps (in particular when your app needs to use a broker), you have to also set the `redirectUri` where the identity provider connects back to your application with the security tokens.

Initializing public and confidential client applications from code

The following code instantiates a public client application, signing-in users in the Microsoft Azure public cloud, with their work and school accounts, or their personal Microsoft accounts.

C#Copy

```
IPublicClientApplication app =
PublicClientApplicationBuilder.Create(clientId).Build();
```

In the same way, the following code instantiates a confidential application (a Web app located at `https://myapp.azurewebsites.net`) handling tokens from users in the Microsoft Azure public cloud, with their work and school accounts, or their personal Microsoft accounts. The application is identified with the identity provider by sharing a client secret:

C#Copy

```
string redirectUri = "https://myapp.azurewebsites.net";
IConfidentialClientApplication app =
ConfidentialClientApplicationBuilder.Create(clientId)
    .WithClientSecret(clientSecret)
    .WithRedirectUri(redirectUri )
    .Build();
```

Builder modifiers

In the code snippets using application builders, `.With` methods can be applied as modifiers (for example, `.WithAuthority` and `.WithRedirectUri`).

- `.WithAuthority` modifier: The `.WithAuthority` modifier sets the application default authority to an Azure Active Directory authority, with the possibility of choosing the Azure Cloud, the audience, the tenant (tenant ID or domain name), or providing directly the authority URI.

  C#Copy

  ```
  var clientApp = PublicClientApplicationBuilder.Create(client_id)
      .WithAuthority(AzureCloudInstance.AzurePublic, tenant_id)
      .Build();
  ```

- `.WithRedirectUri` modifier: The `.WithRedirectUri` modifier overrides the default redirect URI.

  C#Copy

  ```
  var clientApp = PublicClientApplicationBuilder.Create(client_id)
      .WithAuthority(AzureCloudInstance.AzurePublic, tenant_id)
      .WithRedirectUri("http://localhost")
      .Build();
  ```

Modifiers common to public and confidential client applications

The table below lists some of the modifiers you can set on a public, or confidential client.

Modifier	**Description**
`.WithAuthority()`	Sets the application default authority to an Azure Active Directory authority, v choosing the Azure Cloud, the audience, the tenant (tenant ID or domain nar directly the authority URI.

Modifier	Description
`.WithTenantId(string tenantId)`	Overrides the tenant ID, or the tenant description.
`.WithClientId(string)`	Overrides the client ID.
`.WithRedirectUri(string redirectUri)`	Overrides the default redirect URI. This is useful for scenarios requiring a brol
`.WithComponent(string)`	Sets the name of the library using MSAL.NET (for telemetry reasons).
`.WithDebugLoggingCallback()`	If called, the application calls `Debug.Write` simply enabling debugging traces.
`.WithLogging()`	If called, the application calls a callback with debugging traces.
`.WithTelemetry(TelemetryCallback telemetryCallback)`	Sets the delegate used to send telemetry.

Modifiers specific to confidential client applications

The modifiers you can set on a confidential client application builder are:

Modifier	Description
`.WithCertificate(X509Certificate2 certificate)`	Sets the certificate identifying the application with Azure Active Direc
`.WithClientSecret(string clientSecret)`	Sets the client secret (app password) identifying the application with

Exercise - Implement interactive authentication by using MSAL.NET

Completed 100 XP

- 10 minutes

In this exercise you learn how to perform the following actions:

- Register an application with the Microsoft identity platform
- Use the `PublicClientApplicationBuilder` class in MSAL.NET
- Acquire a token interactively in a console application

Prerequisites

- An **Azure account** with an active subscription. If you don't already have one, you can sign up for a free trial at https://azure.com/free

- **Visual Studio Code**: You can install Visual Studio Code from https://code.visualstudio.com.

Register a new application

1. Sign in to the portal: https://portal.azure.com
2. Search for and select **Azure Active Directory**.
3. Under **Manage**, select **App registrations** > **New registration**.
4. When the **Register an application** page appears, enter your application's registration information:

Field	Value
Name	`az204appreg`
Supported account types	Select **Accounts in this organizational directory only**
Redirect URI (optional)	Select **Public client/native (mobile & desktop)** and enter `http://localhost` in

5. Select **Register**.

Azure Active Directory assigns a unique application (client) ID to your app, and you're taken to your application's **Overview** page.

Set up the console application

1. Launch Visual Studio Code and open a terminal by selecting **Terminal** and then **New Terminal**.
2. Create a folder for the project and change in to the folder.

psCopy

```
md az204-auth
cd az204-auth
```

3. Create the .NET console app.

psCopy

```
dotnet new console
```

4. Open the *az204-auth* folder in Visual Studio Code.

psCopy

```
code . -r
```

Build the console app

In this section, you add the necessary packages and code to the project.

Add packages and using statements

1. Add the `Microsoft.Identity.Client` package to the project in a terminal in Visual Studio Code.

   psCopy

   ```
   dotnet add package Microsoft.Identity.Client
   ```

2. Open the *Program.cs* file and add `using` statements to include `Microsoft.Identity.Client` and to enable async operations.

   C#Copy

   ```
   using System.Threading.Tasks;
   using Microsoft.Identity.Client;
   ```

3. Change the Main method to enable async.

   C#Copy

   ```
   public static async Task Main(string[] args)
   ```

Add code for the interactive authentication

1. You need two variables to hold the Application (client) and Directory (tenant) IDs. You can copy those values from the portal. Add the following code and replace the string values with the appropriate values from the portal.

   C#Copy

   ```
   private const string _clientId = "APPLICATION_CLIENT_ID";
   private const string _tenantId = "DIRECTORY_TENANT_ID";
   ```

2. Use the `PublicClientApplicationBuilder` class to build out the authorization context.

   C#Copy

   ```
   var app = PublicClientApplicationBuilder
       .Create(_clientId)
       .WithAuthority(AzureCloudInstance.AzurePublic, _tenantId)
       .WithRedirectUri("http://localhost")
       .Build();
   ```

Code	Description
`.Create`	Creates a `PublicClientApplicationBuilder` from a clientID.
`.WithAuthority`	Adds a known Authority corresponding to an ADFS server. In the code we're specifying the tenant for the app we registered.

Acquire a token

When you registered the *az204appreg* app, it automatically generated an API permission `user.read` for Microsoft Graph. You use that permission to acquire a token.

1. Set the permission scope for the token request. Add the following code below the `PublicClientApplicationBuilder`.

   C#Copy

   ```
   string[] scopes = { "user.read" };
   ```

2. Add code to request the token and write the result out to the console.

   C#Copy

   ```
   AuthenticationResult result = await
   app.AcquireTokenInteractive(scopes).ExecuteAsync();

   Console.WriteLine($"Token:\t{result.AccessToken}");
   ```

Review completed application

The contents of the *Program.cs* file should resemble the following example:

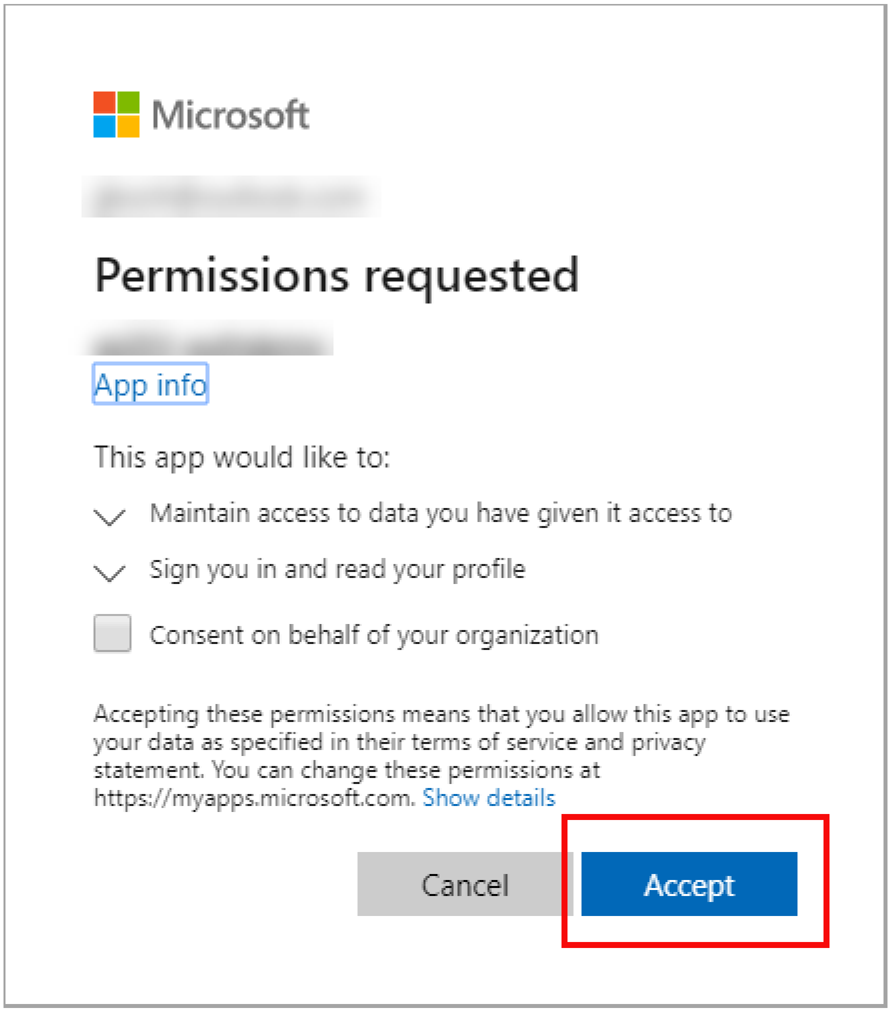
C#Copy

```
using System;
using System.Threading.Tasks;
```

```
using Microsoft.Identity.Client;

namespace az204_auth
{
    class Program
    {
        private const string _clientId = "APPLICATION_CLIENT_ID";
        private const string _tenantId = "DIRECTORY_TENANT_ID";

        public static async Task Main(string[] args)
        {
            var app = PublicClientApplicationBuilder
                .Create(_clientId)
                .WithAuthority(AzureCloudInstance.AzurePublic, _tenantId)
                .WithRedirectUri("http://localhost")
                .Build();
            string[] scopes = { "user.read" };
            AuthenticationResult result = await
app.AcquireTokenInteractive(scopes).ExecuteAsync();

            Console.WriteLine($"Token:\t{result.AccessToken}");
        }
    }
}
```

Run the application

1. In the Visual Studio Code terminal run `dotnet build` to check for errors, then `dotnet run` to run the app.
2. The app opens the default browser prompting you to select the account you want to authenticate with. If there are multiple accounts listed select the one associated with the tenant used in the app.
3. If this is the first time you've authenticated to the registered app you receive a **Permissions requested** notification asking you to approve the app to read data associated with your account. Select **Accept**.



[redacted]

Permissions requested

[redacted]

This app would like to:

Maintain access to data you have given it access to

Sign you in and read your profile

□ Consent on behalf of your organization

Accepting these permissions means that you allow this app to use your data as specified in their terms of service and privacy statement. You can change these permissions at https://myapps.microsoft.com. Show details

4. You should see the results similar to the example below in the console.

Copy

```
Token:  eyJ0eXAiOiJKV1QiLCJub25jZSI6IlVhU.....
```

Explore Microsoft Graph

Introduction

Completed 100 XP

- 3 minutes

Use the wealth of data in Microsoft Graph to build apps for organizations and consumers that interact with millions of users.

After completing this module, you'll be able to:

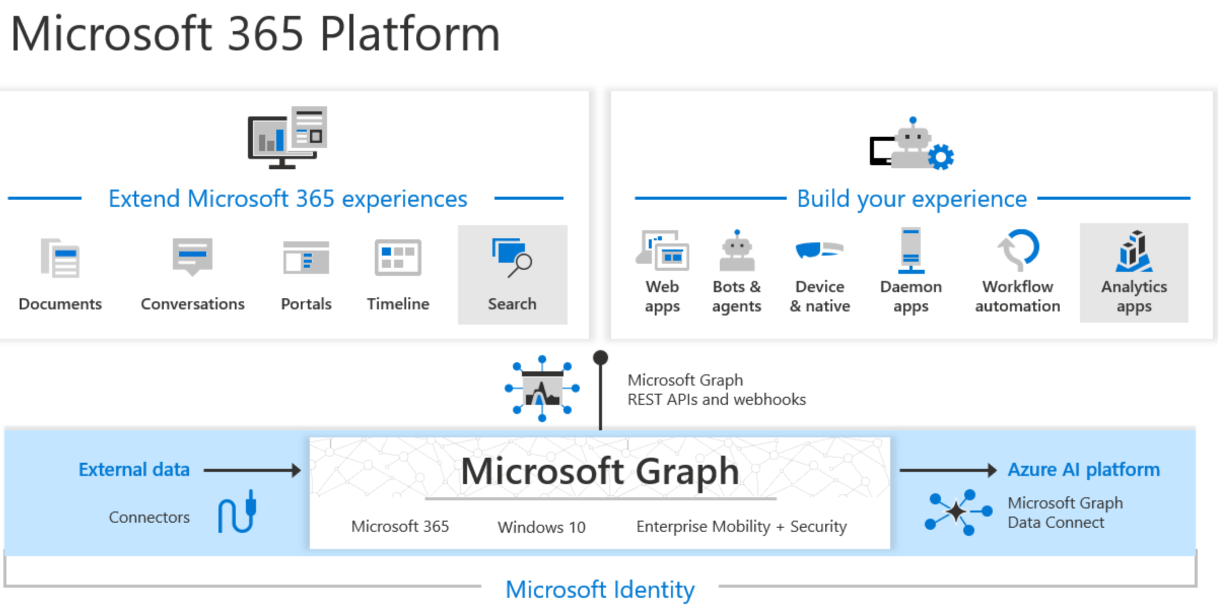
- Explain the benefits of using Microsoft Graph
- Perform operations on Microsoft Graph by using REST and SDKs
- Apply best practices to help your applications get the most out of Microsoft Graph

Discover Microsoft Graph

Completed 100 XP

- 3 minutes

Microsoft Graph is the gateway to data and intelligence in Microsoft 365. It provides a unified programmability model that you can use to access the tremendous amount of data in Microsoft 365, Windows 10, and Enterprise Mobility + Security.



In the Microsoft 365 platform, three main components facilitate the access and flow of data:

- The Microsoft Graph API offers a single endpoint, `https://graph.microsoft.com`. You can use REST APIs or SDKs to access the endpoint. Microsoft Graph also includes a powerful set of services that manage user and device identity, access, compliance, security, and help protect organizations from data leakage or loss.
- Microsoft Graph connectors work in the incoming direction, **delivering data external to the Microsoft cloud into Microsoft Graph services and applications**, to enhance Microsoft 365 experiences such as Microsoft Search. Connectors exist for many commonly used data sources such as Box, Google Drive, Jira, and Salesforce.
- Microsoft Graph Data Connect provides a set of tools to streamline secure and scalable **delivery of Microsoft Graph data to popular Azure data stores**. The cached data serves as data sources for Azure development tools that you can use to build intelligent applications.

Query Microsoft Graph by using REST

Completed100 XP

- 3 minutes

Microsoft Graph is a RESTful web API that enables you to access Microsoft Cloud service resources. After you register your app and get authentication tokens for a user or service, you can make requests to the Microsoft Graph API.

The Microsoft Graph API defines most of its resources, methods, and enumerations in the OData namespace, `microsoft.graph`, in the Microsoft Graph metadata. A few API sets are defined in their sub-namespaces, such as the call records API which defines resources like callRecord in `microsoft.graph.callRecords`.

Unless explicitly specified in the corresponding topic, assume types, methods, and enumerations are part of the `microsoft.graph` namespace.

Call a REST API method

To read from or write to a resource such as a user or an email message, construct a request that looks like the following:

HTTPCopy

```
{HTTP method} https://graph.microsoft.com/{version}/{resource}?{query-parameters}
```

The components of a request include:

- `{HTTP method}` - The HTTP method used on the request to Microsoft Graph.
- `{version}` - The version of the Microsoft Graph API your application is using.
- `{resource}` - The resource in Microsoft Graph that you're referencing.
- `{query-parameters}` - Optional OData query options or REST method parameters that customize the response.

After you make a request, a response is returned that includes:

- Status code - An HTTP status code that indicates success or failure.
- Response message - The data that you requested or the result of the operation. The response message can be empty for some operations.
- `nextLink` - If your request returns a lot of data, you need to page through it by using the URL returned in `@odata.nextLink`.

HTTP methods

Microsoft Graph uses the HTTP method on your request to determine what your request is doing. The API supports the following methods.

Method	**Description**
GET	Read data from a resource.
POST	Create a new resource, or perform an action.
PATCH	Update a resource with new values.
PUT	Replace a resource with a new one.
DELETE	Remove a resource.

- For the CRUD methods `GET` and `DELETE`, no request body is required.
- The `POST`, `PATCH`, and `PUT` methods require a request body specified in JSON format that contains additional information. Such as the values for properties of the resource.

Version

Microsoft Graph currently supports two versions: `v1.0` and `beta`.

- `v1.0` includes generally available APIs. Use the v1.0 version for all production apps.
- `beta` includes APIs that are currently in preview. Because we might introduce breaking changes to our beta APIs, we recommend that you use the beta version only to test apps that are in development; don't use beta APIs in your production apps.

Resource

A resource can be an entity or complex type, commonly defined with properties. Entities differ from complex types by always including an **id** property.

Your URL includes the resource you're interacting with in the request, such as `me`, **user**, **group**, **drive**, and **site**. Often, top-level resources also include *relationships*, which you can use to access other resources, like `me/messages` or `me/drive`. You can also interact with resources using *methods*; for example, to send an email, use `me/sendMail`.

Each resource might require different permissions to access it. You often need a higher level of permissions to create or update a resource than to read it. For details about required permissions, see the method reference topic.

Query parameters

Query parameters can be OData system query options, or other strings that a method accepts to customize its response.

You can use optional OData system query options to include more or fewer properties than the default response, filter the response for items that match a custom query, or provide another parameters for a method.

For example, adding the following `filter` parameter restricts the messages returned to only those with the `emailAddress` property of `jon@contoso.com`.

HTTPCopy

```
GET https://graph.microsoft.com/v1.0/me/messages?filter=emailAddress eq
'jon@contoso.com'
```

Other resources

Following are links to some tools you can use to build and test requests using Microsoft Graph APIs.

- Graph Explorer
- Postman

Query Microsoft Graph by using SDKs

Completed100 XP

- 3 minutes

The Microsoft Graph SDKs are designed to simplify building high-quality, efficient, and resilient applications that access Microsoft Graph. The SDKs include two components: a service library and a core library.

The service library contains models and request builders that are generated from Microsoft Graph metadata to provide a rich, strongly typed, and discoverable experience when working with the many datasets available in Microsoft Graph.

The core library provides a set of features that enhance working with all the Microsoft Graph services. Embedded support for retry handling, secure redirects, transparent authentication, and payload compression, improve the quality of your application's interactions with Microsoft Graph, with no added complexity, while leaving you completely in control. The core library also provides support for common tasks such as paging through collections and creating batch requests.

In this unit, you learn about the available SDKs and see some code examples of some of the most common operations.

Install the Microsoft Graph .NET SDK

The Microsoft Graph .NET SDK is included in the following NuGet packages:

- Microsoft.Graph - Contains the models and request builders for accessing the `v1.0` endpoint with the fluent API. Microsoft.Graph has a dependency on Microsoft.Graph.Core.
- Microsoft.Graph.Beta - Contains the models and request builders for accessing the `beta` endpoint with the fluent API. Microsoft.Graph.Beta has a dependency on Microsoft.Graph.Core.
- Microsoft.Graph.Core - The core library for making calls to Microsoft Graph.

Create a Microsoft Graph client

The Microsoft Graph client is designed to make it simple to make calls to Microsoft Graph. You can use a single client instance for the lifetime of the application. The following code examples show how to create an instance of a Microsoft Graph client. The authentication provider handles acquiring access tokens for the application. The different application providers support different client scenarios. For details about which provider and options are appropriate for your scenario, see Choose an Authentication Provider.

C#Copy

```csharp
var scopes = new[] { "User.Read" };

// Multi-tenant apps can use "common",
// single-tenant apps must use the tenant ID from the Azure portal
var tenantId = "common";

// Value from app registration
var clientId = "YOUR_CLIENT_ID";

// using Azure.Identity;
var options = new TokenCredentialOptions
{
    AuthorityHost = AzureAuthorityHosts.AzurePublicCloud
};

// Callback function that receives the user prompt
// Prompt contains the generated device code that you must
// enter during the auth process in the browser
Func<DeviceCodeInfo, CancellationToken, Task> callback = (code, cancellation) => {
    Console.WriteLine(code.Message);
    return Task.FromResult(0);
};

// https://learn.microsoft.com/dotnet/api/azure.identity.devicecodecredential
var deviceCodeCredential = new DeviceCodeCredential(
    callback, tenantId, clientId, options);

var graphClient = new GraphServiceClient(deviceCodeCredential, scopes);
```

Read information from Microsoft Graph

To read information from Microsoft Graph, you first need to create a request object and then run the `GET` method on the request.

C#Copy

```
// GET https://graph.microsoft.com/v1.0/me

var user = await graphClient.Me
    .Request()
    .GetAsync();
```

Retrieve a list of entities

Retrieving a list of entities is similar to retrieving a single entity except there are other options for configuring the request. The `$filter` query parameter can be used to reduce the result set to only those rows that match the provided condition. The `$orderBy` query parameter requests that the server provides the list of entities sorted by the specified properties.

C#Copy

```
// GET
https://graph.microsoft.com/v1.0/me/messages?$select=subject,sender&$filter=<some
condition>&orderBy=receivedDateTime

var messages = await graphClient.Me.Messages
    .Request()
    .Select(m => new {
        m.Subject,
        m.Sender
    })
    .Filter("<filter condition>")
    .OrderBy("receivedDateTime")
    .GetAsync();
```

Delete an entity

Delete requests are constructed in the same way as requests to retrieve an entity, but use a `DELETE` request instead of a `GET`.

C#Copy

```
// DELETE https://graph.microsoft.com/v1.0/me/messages/{message-id}

string messageId = "AQMkAGUy...";
var message = await graphClient.Me.Messages[messageId]
    .Request()
    .DeleteAsync();
```

Create a new entity

For SDKs that support a fluent style, new items can be added to collections with an `Add` method. For template-based SDKs, the request object exposes a `post` method.

C#Copy

```
// POST https://graph.microsoft.com/v1.0/me/calendars

var calendar = new Calendar
{
    Name = "Volunteer"
};

var newCalendar = await graphClient.Me.Calendars
    .Request()
    .AddAsync(calendar);
```

Other resources

- Microsoft Graph REST API v1.0 reference

Apply best practices to Microsoft Graph

Completed100 XP

- 3 minutes

This unit describes best practices that you can apply to help your applications get the most out of Microsoft Graph and make your application more reliable for end users.

Authentication

To access the data in Microsoft Graph, your application needs to acquire an OAuth 2.0 access token, and present it to Microsoft Graph in either of the following methods:

- The HTTP *Authorization* request header, as a *Bearer* token
- The graph client constructor, when using a Microsoft Graph client library

Use the Microsoft Authentication Library API, MSAL to acquire the access token to Microsoft Graph.

Consent and authorization

Apply the following best practices for consent and authorization in your app:

- **Use least privilege**. Only request permissions that are necessary, and only when you need them. For the APIs, your application calls check the permissions section in the method topics. For example, see creating a user and choose the least privileged permissions.
- **Use the correct permission type based on scenarios**. If you're building an interactive application where a signed in user is present, your application should use *delegated* permissions. If, however, your application runs without a signed-in user, such as a background service or daemon, your application should use application permissions.

  Caution

  Using application permissions for interactive scenarios can put your application at compliance and security risk. Be sure to check user's privileges to ensure they don't have undesired access to information, or are circumnavigating policies configured by an administrator.

- **Consider the end user and admin experience**. This will directly affect end user and admin experiences. For example:
  - Consider who will be consenting to your application, either end users or administrators, and configure your application to request permissions appropriately.
  - Ensure that you understand the difference between static, dynamic and incremental consent.
- **Consider multi-tenant applications**. Expect customers to have various application and consent controls in different states. For example:
  - Tenant administrators can disable the ability for end users to consent to applications. In this case, an administrator would need to consent on behalf of their users.
  - Tenant administrators can set custom authorization policies such as blocking users from reading other user's profiles, or limiting self-service group creation to a limited set of users. In this case, your application should expect to handle 403 error response when acting on behalf of a user.

Handle responses effectively

Depending on the requests you make to Microsoft Graph, your applications should be prepared to handle different types of responses. The following are some of the most important practices to follow to ensure that your application behaves reliably and predictably for your end users. For example:

- **Pagination**: When querying resource collections, you should expect that Microsoft Graph will return the result set in multiple pages, due to server-side page size limits. Your application should **always** handle the possibility that the responses are paged in nature, and use the `@odata.nextLink` property to obtain the next paged set of results, until all pages of the result set have been read. The final page won't contain an `@odata.nextLink` property. For more information, visit paging.
- **Evolvable enumerations**: Adding members to existing enumerations can break applications already using these enums. Evolvable enums are a mechanism that Microsoft Graph API uses to add new members to existing enumerations without causing a breaking change for applications. By default, a GET operation returns only known members for properties of evolvable enum types and your application needs to handle only the known members. If you design your application to handle unknown members as well, you can opt in to receive those members by using an HTTP `Prefer` request header.

Storing data locally

Your application should ideally make calls to Microsoft Graph to retrieve data in real time as necessary. You should only cache or store data locally necessary for a specific scenario, and if that use case is covered by your terms of use and privacy policy, and doesn't violate the Microsoft APIs Terms of Use. Your application should also implement proper retention and deletion policies.

Implement Azure Key Vault

Introduction

Completed100 XP

- 3 minutes

Azure Key Vault is a cloud service for securely storing and accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, certificates, or cryptographic keys.

After completing this module, you'll be able to:

- Describe the benefits of using Azure Key Vault
- Explain how to authenticate to Azure Key Vault
- Set and retrieve a secret from Azure Key Vault by using the Azure CLI

Explore Azure Key Vault

Completed100 XP

- 3 minutes

The Azure Key Vault service supports two types of containers: vaults and managed hardware security module(HSM) pools. Vaults support storing software and HSM-backed keys, secrets, and certificates. Managed HSM pools only support HSM-backed keys.

Azure Key Vault helps solve the following problems:

- **Secrets Management:** Azure Key Vault can be used to Securely store and tightly control access to tokens, passwords, certificates, API keys, and other secrets
- **Key Management:** Azure Key Vault can also be used as a Key Management solution. Azure Key Vault makes it easy to create and control the encryption keys used to encrypt your data.
- **Certificate Management:** Azure Key Vault is also a service that lets you easily provision, manage, and deploy public and private Secure Sockets Layer/Transport Layer Security (SSL/TLS) certificates for use with Azure and your internal connected resources.

Azure Key Vault has two service tiers: Standard, which encrypts with a software key, and a Premium tier, which includes hardware security module(HSM)-protected keys. To see a comparison between the Standard and Premium tiers, see the Azure Key Vault pricing page.

Key benefits of using Azure Key Vault

- **Centralized application secrets:** Centralizing storage of application secrets in Azure Key Vault allows you to control their distribution. For example, instead of storing the connection string in the app's code you can store it securely in Key Vault. Your applications can securely access the information they need by using URIs. These URIs allow the applications to retrieve specific versions of a secret.
- **Securely store secrets and keys:** Access to a key vault requires proper authentication and authorization before a caller (user or application) can get access. Authentication is done via Azure Active Directory. Authorization may be done via Azure role-based access control (Azure RBAC) or Key Vault access policy. Azure RBAC is used when dealing with the management of the vaults and key vault access policy is used when attempting to access data stored in a vault. Azure Key Vaults may be either software-protected or, with the Azure Key Vault Premium tier, hardware-protected by hardware security modules (HSMs).
- **Monitor access and use:** You can monitor activity by enabling logging for your vaults. You have control over your logs and you may secure them by restricting access and you may also delete logs that you no longer need. Azure Key Vault can be configured to:
    - Archive to a storage account.
    - Stream to an event hub.
    - Send the logs to Azure Monitor logs.
- **Simplified administration of application secrets:** Security information must be secured, it must follow a life cycle, and it must be highly available. Azure Key Vault simplifies the process of meeting these requirements by:
    - Removing the need for in-house knowledge of Hardware Security Modules
    - Scaling up on short notice to meet your organization's usage spikes.
    - Replicating the contents of your Key Vault within a region and to a secondary region. Data replication ensures high availability and takes away the need of any action from the administrator to trigger the failover.
    - Providing standard Azure administration options via the portal, Azure CLI and PowerShell.

    - Automating certain tasks on certificates that you purchase from Public CAs, such as enrollment and renewal.

Discover Azure Key Vault best practices

Completed100 XP

- 3 minutes

Azure Key Vault is a tool for securely storing and accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, or certificates. A vault is logical group of secrets.

Authentication

To do any operations with Key Vault, you first need to authenticate to it. There are three ways to authenticate to Key Vault:

- **Managed identities for Azure resources**: When you deploy an app on a virtual machine in Azure, you can assign an identity to your virtual machine that has access to Key Vault. You can also assign identities to other Azure resources. The benefit of this approach is that the app or service isn't managing the rotation of the first secret. Azure automatically rotates the service principal client secret associated with the identity. We recommend this approach as a best practice.
- **Service principal and certificate**: You can use a service principal and an associated certificate that has access to Key Vault. We don't recommend this approach because the application owner or developer must rotate the certificate.
- **Service principal and secret**: Although you can use a service principal and a secret to authenticate to Key Vault, we don't recommend it. It's hard to automatically rotate the bootstrap secret that's used to authenticate to Key Vault.

Encryption of data in transit

Azure Key Vault enforces Transport Layer Security (TLS) protocol to protect data when it’s traveling between Azure Key Vault and clients. Clients negotiate a TLS connection with Azure Key Vault. TLS provides strong authentication, message privacy, and integrity (enabling detection of message tampering, interception, and forgery), interoperability, algorithm flexibility, and ease of deployment and use.

Perfect Forward Secrecy (PFS) protects connections between customers' client systems and Microsoft cloud services by unique keys. Connections also use RSA-based 2,048-bit encryption key lengths. This combination makes it difficult for someone to intercept and access data that is in transit.

Azure Key Vault best practices

- **Use separate key vaults:** Recommended using a vault per application per environment (Development, Pre-Production and Production). This pattern helps you not share secrets across environments and also reduces the threat if there is a breach.
- **Control access to your vault:** Key Vault data is sensitive and business critical, you need to secure access to your key vaults by allowing only authorized applications and users.
- **Backup:** Create regular back ups of your vault on update/delete/create of objects within a Vault.
- **Logging:** Be sure to turn on logging and alerts.
- **Recovery options:** Turn on soft-delete and purge protection if you want to guard against force deletion of the secret.

Authenticate to Azure Key Vault

Completed100 XP

- 3 minutes

Authentication with Key Vault works with Azure Active Directory, which is responsible for authenticating the identity of any given security principal.

For applications, there are two ways to obtain a service principal:

- Enable a system-assigned **managed identity** for the application. With managed identity, Azure internally manages the application's service principal and automatically authenticates the application with other Azure services. Managed identity is available for applications deployed to various services.
- If you can't use managed identity, you instead register the application with your Azure AD tenant. Registration also creates a second application object that identifies the app across all tenants.

Note

It is recommended to use a system-assigned managed identity.

The following is information on authenticating to Key Vault without using a managed identity.

Authentication to Key Vault in application code

Key Vault SDK is using Azure Identity client library, which allows seamless authentication to Key Vault across environments with same code. The table below provides information on the Azure Identity client libraries:

.NET	Python	Java	JavaScript
Azure Identity SDK .NET	Azure Identity SDK Python	Azure Identity SDK Java	Azure Identity SD

Authentication to Key Vault with REST

Access tokens must be sent to the service using the HTTP Authorization header:

HTTPCopy

```
PUT /keys/MYKEY?api-version=<api_version>  HTTP/1.1
Authorization: Bearer [Removed]
```

When an access token isn't supplied, or when a token isn't accepted by the service, an `HTTP 401` error is returned to the client and will include the `WWW-Authenticate` header, for example:

HTTPCopy

```
401 Not Authorized
WWW-Authenticate: Bearer authorization="…", resource="…"
```

The parameters on the `WWW-Authenticate` header are:

- authorization: The address of the OAuth2 authorization service that may be used to obtain an access token for the request.
- resource: The name of the resource (`https://vault.azure.net`) to use in the authorization request.

Other resources

- Azure Key Vault developer's guide
- Azure Key Vault availability and redundancy

Exercise: Set and retrieve a secret from Azure Key Vault by using Azure CLI

Completed 100 XP

- 3 minutes

In this exercise you learn how to perform the following actions by using the Azure CLI:

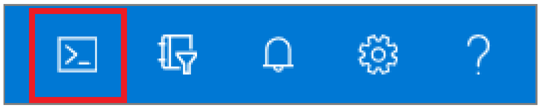
- Create a Key Vault
- Add and retrieve a secret

Prerequisites

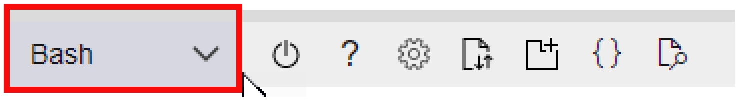
-
  - An **Azure account** with an active subscription. If you don't already have one, you can sign up for a free trial at https://azure.com/free

Sign in to Azure and start the Cloud Shell

1. Sign in to the Azure portal and open the Cloud Shell.



2. When the shell opens be sure to select the **Bash** environment.



Create a Key Vault

1. Let's set some variables for the CLI commands to use to reduce the amount of retyping. Replace the `<myLocation>` variable string with a region that

makes sense for you. The Key Vault name needs to be a globally unique name, and the following script generates a random string.

BashCopy

```
myKeyVault=az204vault-$RANDOM
myLocation=<myLocation>
```

2. Create a resource group.

   Azure CLICopy

   ```
   az group create --name az204-vault-rg --location $myLocation
   ```

3. Create a Key Vault by using the `az keyvault create` command.

   Azure CLICopy

   ```
   az keyvault create --name $myKeyVault --resource-group az204-vault-rg --
   location $myLocation
   ```

   Note

   This can take a few minutes to run.

Add and retrieve a secret

To add a secret to the vault, you just need to take a couple of extra steps.

1. Create a secret. Let's add a password that could be used by an app. The password is called **ExamplePassword** and will store the value of **hVFkk965BuUv** in it.

   Azure CLICopy

   ```
   az keyvault secret set --vault-name $myKeyVault --name "ExamplePassword"
   --value "hVFkk965BuUv"
   ```

2. Use the `az keyvault secret show` command to retrieve the secret.

   Azure CLICopy

   ```
   az keyvault secret show --name "ExamplePassword" --vault-name $myKeyVault
   ```

   This command returns some JSON. The last line contains the password in plain text.

JSONCopy

```
"value": "hVFkk965BuUv"
```

You've created a Key Vault, stored a secret, and retrieved it.

Clean up resources

If you no longer need the resources in this exercise use the following command to delete the resource group and associated Key Vault.

Azure CLICopy

```
az group delete --name az204-vault-rg --no-wait
```

Implement Azure App Configuration

Azure App Configuration provides a service to centrally manage application settings and feature flags.

After completing this module, you'll be able to:

- Explain the benefits of using Azure App Configuration
- Describe how Azure App Configuration stores information
- Implement feature management
- Securely access your app configuration information

Explore the Azure App Configuration service

Completed100 XP

- 3 minutes

Azure App Configuration provides a service to centrally manage application settings and feature flags. Modern programs, especially programs running in a cloud, generally have many components that are distributed in nature. Spreading configuration settings across these components can lead to hard-to-troubleshoot errors during an application deployment. Use App Configuration to store all the settings for your application and secure their accesses in one place.

App Configuration offers the following benefits:

- A fully managed service that can be set up in minutes
- Flexible key representations and mappings
- Tagging with labels
- Point-in-time replay of settings
- Dedicated UI for feature flag management
- Comparison of two sets of configurations on custom-defined dimensions
- Enhanced security through Azure-managed identities
- Encryption of sensitive information at rest and in transit
- Native integration with popular frameworks

App Configuration complements Azure Key Vault, which is used to store application secrets. App Configuration makes it easier to implement the following scenarios:

- Centralize management and distribution of hierarchical configuration data for different environments and geographies
- Dynamically change application settings without the need to redeploy or restart an application
- Control feature availability in real-time

Use App Configuration

The easiest way to add an App Configuration store to your application is through a client library that Microsoft provides. Based on the programming language and framework, the following best methods are available to you.

Programming language and framework	How to connect
.NET Core and ASP.NET Core	App Configuration provider for .NET Core
.NET Framework and ASP.NET	App Configuration builder for .NET
Java Spring	App Configuration client for Spring Cloud
JavaScript/Node.js	App Configuration client for JavaScript
Python	App Configuration client for Python
Others	App Configuration REST API

Create paired keys and values

Completed100 XP

- 3 minutes

Azure App Configuration stores configuration data as key-value pairs.

Keys

Keys serve as the name for key-value pairs and are used to store and retrieve corresponding values. It's a common practice to organize keys into a hierarchical namespace by using a character delimiter, such as / or :. Use a convention that's best suited for your application. App Configuration treats keys as a whole. It doesn't parse keys to figure out how their names are structured or enforce any rule on them.

Keys stored in App Configuration are case-sensitive, unicode-based strings. The keys *app1* and *App1* are distinct in an App Configuration store. Keep this in mind when you use configuration settings within an application because some frameworks handle configuration keys case-insensitively.

You can use any unicode character in key names entered into App Configuration except for `*`, `,`, and `\`. These characters are reserved. If you need to include a reserved character, you must escape it by using `\{Reserved Character}`. There's a combined size limit of 10,000 characters on a key-value pair. This limit includes all characters in the key, its value, and all associated optional attributes. Within this limit, you can have many hierarchical levels for keys.

Design key namespaces

There are two general approaches to naming keys used for configuration data: flat or hierarchical. These methods are similar from an application usage standpoint, but hierarchical naming offers many advantages:

- Easier to read. Instead of one long sequence of characters, delimiters in a hierarchical key name function as spaces in a sentence.
- Easier to manage. A key name hierarchy represents logical groups of configuration data.
- Easier to use. It's simpler to write a query that pattern-matches keys in a hierarchical structure and retrieves only a portion of configuration data.

Following are some examples of how you can structure your key names into a hierarchy:

- Based on component services

  Copy

  ```
  AppName:Service1:ApiEndpoint
  AppName:Service2:ApiEndpoint
  ```

- Based on deployment regions

  Copy

  ```
  AppName:Region1:DbEndpoint
  AppName:Region2:DbEndpoint
  ```

Label keys

Key values in App Configuration can optionally have a label attribute. Labels are used to differentiate key values with the same key. A key *app1* with labels *A* and *B* forms two separate keys in an App Configuration store. By default, the label for a key value is empty, or `null`.

Label provides a convenient way to create variants of a key. A common use of labels is to specify multiple environments for the same key:

Copy

```
Key = AppName:DbEndpoint & Label = Test
Key = AppName:DbEndpoint & Label = Staging
Key = AppName:DbEndpoint & Label = Production
```

Version key values

App Configuration doesn't version key values automatically as they're modified. Use labels as a way to create multiple versions of a key value. For example, you can input an application version number or a Git commit ID in labels to identify key values associated with a particular software build.

Query key values

Each key value is uniquely identified by its key plus a label that can be `null`. You query an App Configuration store for key values by specifying a pattern. The App Configuration store returns all key values that match the pattern and their corresponding values and attributes.

Values

Values assigned to keys are also unicode strings. You can use all unicode characters for values. There's an optional user-defined content type associated with each value. Use this attribute to store information, for example an encoding scheme, about a value that helps your application to process it properly.

Configuration data stored in an App Configuration store, which includes all keys and values, is encrypted at rest and in transit. App Configuration isn't a replacement solution for Azure Key Vault. Don't store application secrets in it.

Manage application features

Completed100 XP

- 3 minutes

Feature management is a modern software-development practice that decouples feature release from code deployment and enables quick changes to feature availability on demand. It uses a technique called feature flags (also known as feature toggles, feature switches, and so on) to dynamically administer a feature's lifecycle.

Basic concepts

Here are several new terms related to feature management:

- **Feature flag**: A feature flag is a variable with a binary state of *on* or *off*. The feature flag also has an associated code block. The state of the feature flag triggers whether the code block runs or not.
- **Feature manager**: A feature manager is an application package that handles the lifecycle of all the feature flags in an application. The feature manager typically provides extra functionality, such as caching feature flags and updating their states.
- **Filter**: A filter is a rule for evaluating the state of a feature flag. A user group, a device or browser type, a geographic location, and a time window are all examples of what a filter can represent.

An effective implementation of feature management consists of at least two components working in concert:

- An application that makes use of feature flags.
- A separate repository that stores the feature flags and their current states.

How these components interact is illustrated in the following examples.

Feature flag usage in code

The basic pattern for implementing feature flags in an application is simple. You can think of a feature flag as a Boolean state variable used with an `if` conditional statement in your code:

C#Copy

```csharp
if (featureFlag) {
    // Run the following code
}
```

In this case, if `featureFlag` is set to `True`, the enclosed code block is executed; otherwise, it's skipped. You can set the value of `featureFlag` statically, as in the following code example:

C#Copy

```
bool featureFlag = true;
```

You can also evaluate the flag's state based on certain rules:

C#Copy

```
bool featureFlag = isBetaUser();
```

A slightly more complicated feature flag pattern includes an `else` statement as well:

C#Copy

```
if (featureFlag) {
    // This following code will run if the featureFlag value is true
} else {
    // This following code will run if the featureFlag value is false
}
```

Feature flag declaration

Each feature flag has two parts: a name and a list of one or more filters that are used to evaluate if a feature's state is *on* (that is, when its value is `True`). A filter defines a use case for when a feature should be turned on.

When a feature flag has multiple filters, the filter list is traversed in order until one of the filters determines the feature should be enabled. At that point, the feature flag is *on*, and any remaining filter results are skipped. If no filter indicates the feature should be enabled, the feature flag is *off*.

The feature manager supports *appsettings.json* as a configuration source for feature flags. The following example shows how to set up feature flags in a JSON file:

JSONCopy

```
"FeatureManagement": {
    "FeatureA": true, // Feature flag set to on
    "FeatureB": false, // Feature flag set to off
    "FeatureC": {
        "EnabledFor": [
            {
                "Name": "Percentage",
                "Parameters": {
                    "Value": 50
                }
            }
        ]
    }
}
```

Feature flag repository

To use feature flags effectively, you need to externalize all the feature flags used in an application. This approach allows you to change feature flag states without modifying and redeploying the application itself.

Azure App Configuration is designed to be a centralized repository for feature flags. You can use it to define different kinds of feature flags and manipulate their states quickly and confidently. You can then use the App Configuration libraries for various programming language frameworks to easily access these feature flags from your application.

Secure app configuration data

Completed100 XP

- 3 minutes

In this unit you learn how to secure your apps configuration data by using:

- Customer-managed keys
- Private endpoints
- Managed identities

Encrypt configuration data by using customer-managed keys

Azure App Configuration encrypts sensitive information at rest using a 256-bit AES encryption key provided by Microsoft. Every App Configuration instance has its own encryption key managed by the service and used to encrypt sensitive information. Sensitive information includes the values found in key-value pairs. When customer-managed key capability is enabled, App Configuration uses a managed identity assigned to the App Configuration instance to authenticate with Azure Active Directory. The managed identity then calls Azure Key Vault and wraps the App Configuration instance's encryption key. The wrapped encryption key is then stored and the unwrapped encryption key is cached within App Configuration for one hour. App Configuration refreshes the unwrapped version of the App Configuration instance's encryption key hourly. This ensures availability under normal operating conditions.

Enable customer-managed key capability

The following components are required to successfully enable the customer-managed key capability for Azure App Configuration:

- Standard tier Azure App Configuration instance
- Azure Key Vault with soft-delete and purge-protection features enabled
- An RSA or RSA-HSM key within the Key Vault: The key must not be expired, it must be enabled, and it must have both wrap and unwrap capabilities enabled

Once these resources are configured, two steps remain to allow Azure App Configuration to use the Key Vault key:

1. Assign a managed identity to the Azure App Configuration instance
2. Grant the identity `GET`, `WRAP`, and `UNWRAP` permissions in the target Key Vault's access policy.

Use private endpoints for Azure App Configuration

You can use private endpoints for Azure App Configuration to allow clients on a virtual network (VNet) to securely access data over a private link. The private endpoint uses an IP address from the VNet address space for your App Configuration store. Network traffic between the clients on the VNet and the App Configuration store traverses over the VNet using a private link on the Microsoft backbone network, eliminating exposure to the public internet.

Using private endpoints for your App Configuration store enables you to:

- Secure your application configuration details by configuring the firewall to block all connections to App Configuration on the public endpoint.
- Increase security for the virtual network (VNet) ensuring data doesn't escape from the VNet.
- Securely connect to the App Configuration store from on-premises networks that connect to the VNet using VPN or ExpressRoutes with private-peering.

Managed identities

A managed identity from Azure Active Directory (Azure AD) allows Azure App Configuration to easily access other AAD-protected resources, such as Azure Key Vault. The identity is managed by the Azure platform. It doesn't require you to provision or rotate any secrets.

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to your configuration store. It's deleted if your configuration store is deleted. A configuration store can only have one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your configuration store. A configuration store can have multiple user-assigned identities.

Add a system-assigned identity

To set up a managed identity using the Azure CLI, use the `az appconfig identity assign` command against an existing configuration store. The following Azure CLI example creates a system-assigned identity for an Azure App Configuration store named `myTestAppConfigStore`.

BashCopy

```
az appconfig identity assign \
    --name myTestAppConfigStore \
    --resource-group myResourceGroup
```

Add a user-assigned identity

Creating an App Configuration store with a user-assigned identity requires that you create the identity and then assign its resource identifier to your store. The following Azure CLI examples create a user-assigned identity called `myUserAssignedIdentity` and assign it to an Azure App Configuration store named `myTestAppConfigStore`.

Create an identity using the `az identity create` command:

BashCopy

```
az identity create --resource-group myResourceGroup --name myUserAssignedIdentity
```

Assign the new user-assigned identity to the `myTestAppConfigStore` configuration store:

BashCopy

```
az appconfig identity assign --name myTestAppConfigStore \
    --resource-group myResourceGroup \
    --identities /subscriptions/[subscription
id]/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedId
entities/myUserAssignedIdentity
```

Monitor app performance

Introduction

Completed100 XP

- 3 minutes

Instrumenting and monitoring, your apps helps you maximize their availability and performance.

After completing this module, you'll be able to:

- Describe how Application Insights works and how it collects events and metrics.
- Instrument an app for monitoring, perform availability tests, and use Application Map to help you monitor performance and troubleshoot issues.

Explore Application Insights

Completed100 XP

- 3 minutes

Application Insights is an extension of Azure Monitor and provides Application Performance Monitoring (also known as “APM”) features. APM tools are useful to monitor applications from development, through test, and into production in the following ways:

- Proactively understand how an application is performing.
- Reactively review application execution data to determine the cause of an incident.

In addition to collecting metrics and application telemetry data, which describe application activities and health, Application Insights can also be used to collect and store application trace logging data.

The log trace is associated with other telemetry to give a detailed view of the activity. Adding trace logging to existing apps only requires providing a destination for the logs; the logging framework rarely needs to be changed.

Application Insights feature overview

Features include, but not limited to:

Feature	Description
Live Metrics	Observe activity from your deployed application in real time with no effect on the host envir
Availability	Also known as “Synthetic Transaction Monitoring”, probe your applications external endpoi availability and responsiveness over time.
GitHub or Azure DevOps integration	Create GitHub or Azure DevOps work items in context of Application Insights data.
Usage	Understand which features are popular with users and how users interact and use your appl
Smart Detection	Automatic failure and anomaly detection through proactive telemetry analysis.
Application Map	A high level top-down view of the application architecture and at-a-glance visual references and responsiveness.
Distributed Tracing	Search and visualize an end-to-end flow of a given execution or transaction.

What Application Insights monitors

Application Insights collects Metrics and application Telemetry data, which describe application activities and health, as well as trace logging data.

- **Request rates, response times, and failure rates** - Find out which pages are most popular, at what times of day, and where your users are. See which pages perform best. If your response times and failure rates go high when there are more requests, then perhaps you have a resourcing problem.
- **Dependency rates, response times, and failure rates** - Find out whether external services are slowing you down.
- **Exceptions** - Analyze the aggregated statistics, or pick specific instances and drill into the stack trace and related requests. Both server and browser exceptions are reported.
- **Page views and load performance** - reported by your users' browsers.
- **AJAX calls** from web pages - rates, response times, and failure rates.
- **User and session counts**.
- **Performance counters** from your Windows or Linux server machines, such as CPU, memory, and network usage.
- **Host diagnostics** from Docker or Azure.
- **Diagnostic trace logs** from your app - so that you can correlate trace events with requests.
- **Custom events and metrics** that you write yourself in the client or server code, to track business events such as items sold or games won.

Getting started with Application Insights

Application Insights is one of the many services hosted within Microsoft Azure, and telemetry is sent there for analysis and presentation. It's free to sign up, and if you choose the basic pricing plan of Application Insights, there's no charge until your application has grown to have substantial usage.

There are several ways to get started monitoring and analyzing app performance:

- **At run time:** instrument your web app on the server. Ideal for applications already deployed. Avoids any update to the code.
- **At development time:** add Application Insights to your code. Allows you to customize telemetry collection and send more telemetry.
- **Instrument your web pages** for page view, AJAX, and other client-side telemetry.
- **Analyze mobile app usage** by integrating with Visual Studio App Center.
- **Availability tests** - ping your website regularly from our servers.

Discover log-based metrics

Completed100 XP

- 3 minutes

Application Insights log-based metrics let you analyze the health of your monitored apps, create powerful dashboards, and configure alerts. There are two kinds of metrics:

- **Log-based metrics** behind the scene are translated into Kusto queries from stored events.
- **Standard metrics** are stored as pre-aggregated time series.

Since *standard metrics* are pre-aggregated during collection, they have better performance at query time. Standard metrics are a better choice for dashboarding and in real-time alerting. The *log-based metrics* have more dimensions, which makes them the superior option for data analysis and ad-hoc diagnostics. Use the namespace selector to switch between log-based and standard metrics in metrics explorer.

Log-based metrics

Developers can use the SDK to send events manually (by writing code that explicitly invokes the SDK) or they can rely on the automatic collection of events from auto-instrumentation. In either case, the Application Insights backend stores all collected events as logs, and the Application Insights blades in the Azure portal act as an analytical and diagnostic tool for visualizing event-based data from logs.

Using logs to retain a complete set of events can bring great analytical and diagnostic value. For example, you can get an exact count of requests to a particular URL with the number of distinct users who made these calls. Or you can get detailed diagnostic traces, including exceptions and dependency calls for any user session. Having this type of information can significantly improve visibility into the application health and usage, allowing to cut down the time necessary to diagnose issues with an app.

At the same time, collecting a complete set of events may be impractical (or even impossible) for applications that generate a large volume of telemetry. For situations when the volume of events is too high, Application Insights implements several telemetry volume reduction techniques, such as sampling and filtering that reduces the number of collected and stored events. Unfortunately, lowering the number of stored events also lowers the accuracy of the metrics that, behind the scenes, must perform query-time aggregations of the events stored in logs.

Pre-aggregated metrics

The pre-aggregated metrics aren't stored as individual events with lots of properties. Instead, they're stored as pre-aggregated time series, and only with key dimensions. This makes the new metrics superior at query time: retrieving data happens faster and requires less compute power. This enables new scenarios such as near real-time alerting on dimensions of metrics, more responsive dashboards, and more.

Important

Both, log-based and pre-aggregated metrics coexist in Application Insights. To differentiate the two, in the Application Insights UX the pre-aggregated metrics are now called "Standard metrics (preview)", while the traditional metrics from the events were renamed to "Log-based metrics".

The newer SDKs (Application Insights 2.7 SDK or later for .NET) pre-aggregate metrics during collection. This applies to standard metrics sent by default so the accuracy isn't affected by sampling or filtering. It also applies to custom metrics sent using GetMetric resulting in less data ingestion and lower cost.

For the SDKs that don't implement pre-aggregation the Application Insights backend still populates the new metrics by aggregating the events received by the Application Insights event collection endpoint. While you don't benefit from the reduced volume of data transmitted over the wire, you can still use the pre-aggregated metrics and experience better performance and support of the near real-time dimensional alerting with SDKs that don't pre-aggregate metrics during collection.

It's worth mentioning that the collection endpoint pre-aggregates events before ingestion sampling, which means that ingestion sampling will never impact the accuracy of pre-aggregated metrics, regardless of the SDK version you use with your application.

Instrument an app for monitoring

Completed100 XP

- 3 minutes

Application Insights is enabled through either Auto-Instrumentation (agent) or by adding the Application Insights SDK to your application code.

Auto-instrumentation

Auto-instrumentation is the preferred instrumentation method. It requires no developer investment and eliminates future overhead related to updating the SDK. It's also the only way to instrument an application in which you don't have access to the source code.

In essence, all you have to do is enable and - in some cases - configure the agent, which collects the telemetry automatically.

The list of services supported by auto-instrumentation changes rapidly, visit this page for a list of what is currently supported.

Enabling via Application Insights SDKs

You only need to install the Application Insights SDK in the following circumstances:

- You require custom events and metrics
- You require control over the flow of telemetry
- Auto-Instrumentation isn't available (typically due to language or platform limitations)

To use the SDK, you install a small instrumentation package in your app and then instrument the web app, any background components, and JavaScript within the web pages. The app and its components don't have to be hosted in Azure. The instrumentation monitors your app and directs the telemetry data to an Application Insights resource by using a unique token.

The Application Insights SDKs for .NET, .NET Core, Java, Node.js, and JavaScript all support distributed tracing natively.

Additionally, any technology can be tracked manually with a call to `TrackDependency` on the `TelemetryClient`.

Enable via OpenCensus

In addition to the Application Insights SDKs, Application Insights also supports distributed tracing through OpenCensus. OpenCensus is an open source, vendor-agnostic, single distribution of libraries to provide metrics collection and distributed tracing for services. It also enables the open source community to enable distributed tracing with popular technologies like Redis, Memcached, or MongoDB.

Select an availability test

Completed100 XP

- 3 minutes

After you've deployed your web app or website, you can set up recurring tests to monitor availability and responsiveness. Application Insights sends web requests to your application at regular intervals from points around the world. It can alert you if your application isn't responding or responds too slowly.

You can set up availability tests for any HTTP or HTTPS endpoint that's accessible from the public internet. You don't have to make any changes to the website you're testing. In fact, it doesn't even have to be a site that you own. You can test the availability of a REST API that your service depends on.

You can create up to 100 availability tests per Application Insights resource, and there are three types of availability tests:

- URL ping test (classic): You can create this test through the portal to validate whether an endpoint is responding and measure performance associated with that response. You can also set custom success criteria coupled with more advanced features, like parsing dependent requests and allowing for retries.
- Standard test (Preview): This single request test is similar to the URL ping test. It includes SSL certificate validity, proactive lifetime check, HTTP request verb (for

example `GET`, `HEAD`, or `POST`), custom headers, and custom data associated with your HTTP request.

- Custom TrackAvailability test: If you decide to create a custom application to run availability tests, you can use the TrackAvailability() method to send the results to Application Insights.

Note

Multi-step test is a fourth type of availability test, however that is only available through Visual Studio 2019. **Custom TrackAvailability test** is the long term supported solution for multi request or authentication test scenarios.

Important

The **URL ping test** relies on the DNS infrastructure of the public internet to resolve the domain names of the tested endpoints. If you're using private DNS, you must ensure that the public domain name servers can resolve every domain name of your test. When that's not possible, you can use custom **TrackAvailability tests** instead.

Visit the troubleshooting article for guidance on diagnosing availability issues.

Troubleshoot app performance by using Application Map

Completed100 XP

- 3 minutes

Application Map helps you spot performance bottlenecks or failure hotspots across all components of your distributed application. Each node on the map represents an application component or its dependencies; and has health KPI and alerts status. You can click through from any component to more detailed diagnostics, such as Application Insights events. If your app uses Azure services, you can also click through to Azure diagnostics, such as SQL Database Advisor recommendations.

Components are independently deployable parts of your distributed/microservices application. Developers and operations teams have code-level visibility or access to telemetry generated by these application components.

- Components are different from "observed" external dependencies such as SQL, Event Hubs, etc. which your team/organization may not have access to (code or telemetry).
- Components run on any number of server/role/container instances.

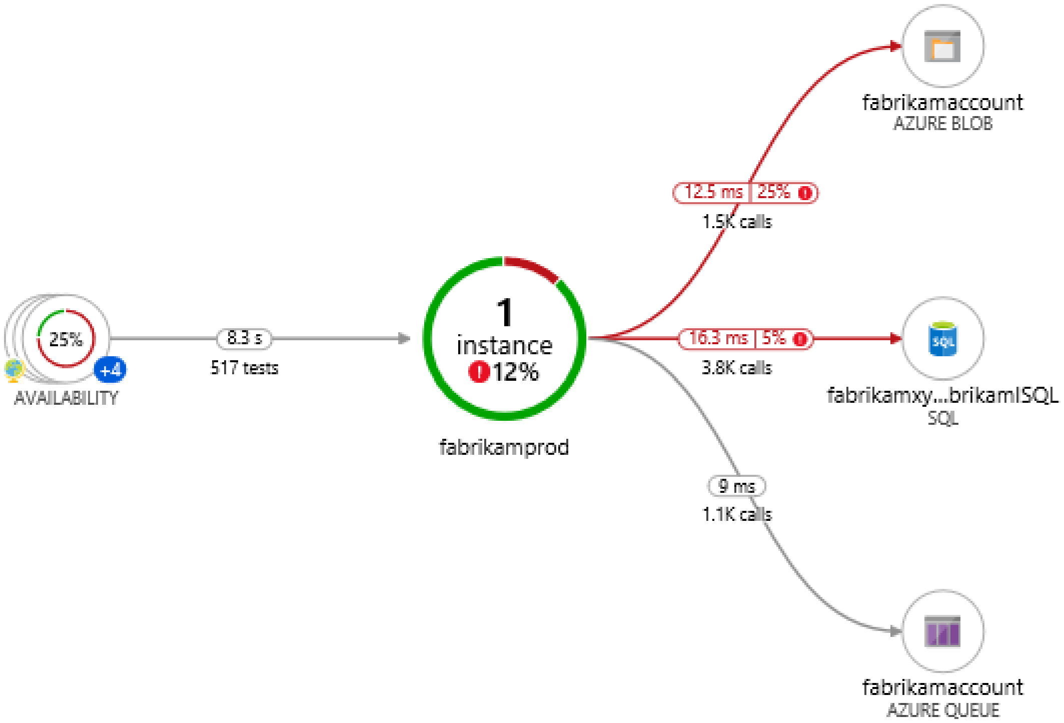
- Components can be separate Application Insights instrumentation keys (even if subscriptions are different) or different roles reporting to a single Application Insights instrumentation key. The preview map experience shows the components regardless of how they're set up.

You can see the full application topology across multiple levels of related application components. Components could be different Application Insights resources, or different roles in a single resource. The app map finds components by following HTTP dependency calls made between servers with the Application Insights SDK installed.

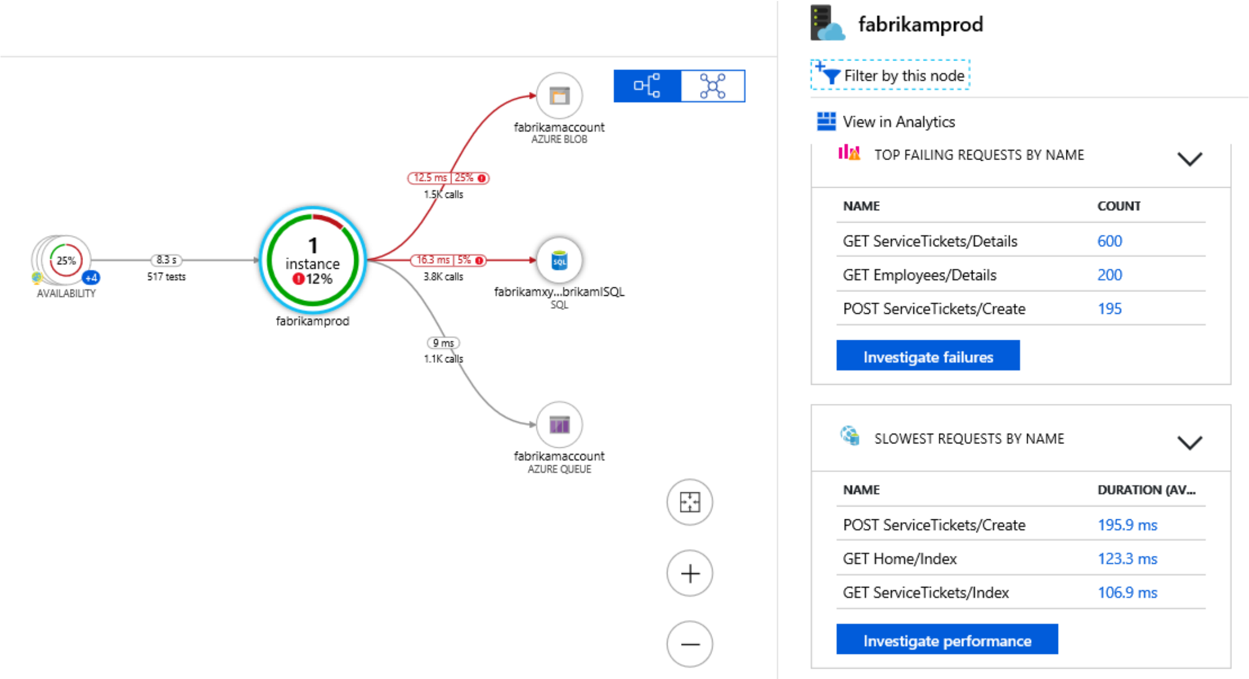
This experience starts with progressive discovery of the components. When you first load the application map, a set of queries is triggered to discover the components related to this component. A button at the top-left corner updates with the number of components in your application as they're discovered.

On clicking "Update map components", the map is refreshed with all components discovered until that point. Depending on the complexity of your application, this may take a minute to load.

If all of the components are roles within a single Application Insights resource, then this discovery step isn't required. The initial load for such an application has all its components.



One of the key objectives with this experience is to be able to visualize complex topologies with hundreds of components. Click on any component to see related insights and go to the performance and failure triage experience for that component.



fabrikamprod

TOP FAILING REQUESTS BY NAME

NAME	COUNT
GET ServiceTickets/Details	600
GET Employees/Details	200
POST ServiceTickets/Create	195

SLOWEST REQUESTS BY NAME

NAME	DURATION (AV...
POST ServiceTickets/Create	195.9 ms
GET Home/Index	123.3 ms
GET ServiceTickets/Index	106.9 ms

Application Map uses the cloud role name property to identify the components on the map. You can manually set or override the cloud role name and change what gets displayed on the Application Map.

Explore Azure Container Apps

Completed100 XP

- 3 minutes

Azure Container Apps enables you to run microservices and containerized applications on a serverless platform that runs on top of Azure Kubernetes Service. Common uses of Azure Container Apps include:

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

Applications built on Azure Container Apps can dynamically scale based on: HTTP traffic, event-driven processing, CPU or memory load, and any KEDA-supported scaler.

With Azure Container Apps, you can:

- Run multiple container revisions and manage the container app's application lifecycle.
- Autoscale your apps based on any KEDA-supported scale trigger. Most applications can scale to zero. (Applications that scale on CPU or memory load can't scale to zero.)
- Enable HTTPS ingress without having to manage other Azure infrastructure.
- Split traffic across multiple versions of an application for Blue/Green deployments and A/B testing scenarios.
- Use internal ingress and service discovery for secure internal-only endpoints with built-in DNS-based service discovery.
- Build microservices with Dapr and access its rich set of APIs.
- Run containers from any registry, public or private, including Docker Hub and Azure Container Registry (ACR).
- Use the Azure CLI extension, Azure portal or ARM templates to manage your applications.
- Provide an existing virtual network when creating an environment for your container apps.
- Securely manage secrets directly in your application.
- Monitor logs using Azure Log Analytics.

Azure Container Apps environments

Individual container apps are deployed to a single Container Apps environment, which acts as a secure boundary around groups of container apps. Container Apps in the same environment are deployed in the same virtual network and write logs to the same Log Analytics workspace. You may provide an existing virtual network when you create an environment.

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

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

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

- Two applications never share the same compute resources
- Two Dapr applications can't communicate via the Dapr service invocation API

Microservices with Azure Container Apps

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

- Independent scaling, versioning, and upgrades
- Service discovery
- Native Dapr integration

Dapr integration

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

Explore containers in Azure Container Apps

Completed100 XP

- 5 minutes

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

Azure Container Apps supports any Linux-based x86-64 (`linux/amd64`) container image. There's no required base container image, and if a container crashes it automatically restarts.

Configuration

The following code is an example of the `containers` array in the `properties.template` section of a container app resource template. The excerpt shows some of the available configuration options when setting up a container when using Azure Resource Manager (ARM) templates. Changes to the template ARM configuration section trigger a new container app revision.

JSONCopy

```json
"containers": [
  {
       "name": "main",
       "image": "[parameters('container_image')]",
    "env": [
      {
        "name": "HTTP_PORT",
        "value": "80"
      },
      {
        "name": "SECRET_VAL",
        "secretRef": "mysecret"
      }
    ],
    "resources": {
      "cpu": 0.5,
      "memory": "1Gi"
    },
    "volumeMounts": [
      {
        "mountPath": "/myfiles",
        "volumeName": "azure-files-volume"
      }
    ]
    "probes":[
        {
            "type":"liveness",
            "httpGet":{
            "path":"/health",
            "port":8080,
            "httpHeaders":[
                {
                    "name":"Custom-Header",
                    "value":"liveness probe"
                }]
            },
            "initialDelaySeconds":7,
            "periodSeconds":3
// file is truncated for brevity
```

Multiple containers

You can define multiple containers in a single container app to implement the sidecar pattern. The containers in a container app share hard disk and network resources and experience the same application lifecycle.

Examples of sidecar containers include:

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

Note

Running multiple containers in a single container app is an advanced use case. In most situations where you want to run multiple containers, such as when implementing a microservice architecture, deploy each service as a separate container app.

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

Container registries

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

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

JSONCopy

```json
{
  ...
  "registries": [{
    "server": "docker.io",
    "username": "my-registry-user-name",
    "passwordSecretRef": "my-password-secret-name"
  }]
}
```

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

Limitations

Azure Container Apps has the following limitations:

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

Implement authentication and authorization in Azure Container Apps

Completed100 XP

- 5 minutes

Azure Container Apps provides built-in authentication and authorization features to secure your external ingress-enabled container app with minimal or no code. The built-in authentication feature for Container Apps can save you time and effort by providing out-of-the-box authentication with federated identity providers, allowing you to focus on the rest of your application.

- Azure Container Apps provides access to various built-in authentication providers.
- The built-in auth features don't require any particular language, SDK, security expertise, or even any code that you have to write.

This feature should only be used with HTTPS. Ensure `allowInsecure` is disabled on your container app's ingress configuration. You can configure your container app for authentication with or without restricting access to your site content and APIs.

- To restrict app access only to authenticated users, set its *Restrict access* setting to **Require authentication**.
- To authenticate but not restrict access, set its *Restrict access* setting to **Allow unauthenticated** access.

Identity providers

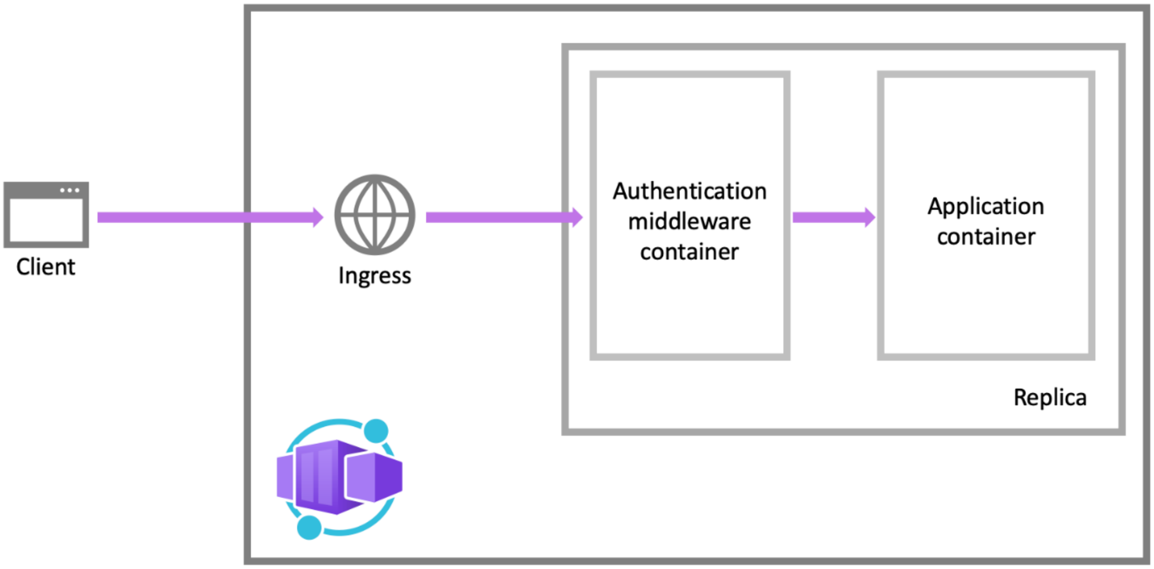
Container Apps uses federated identity, in which a third-party identity provider manages the user identities and authentication flow for you. The following identity providers are available by default:

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

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

Feature architecture

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



The platform middleware handles several things for your app:

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

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

Authentication flow

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

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

Manage revisions and secrets in Azure Container Apps

Completed100 XP

- 5 minutes

Azure Container Apps implements container app versioning by creating revisions. A revision is an immutable snapshot of a container app version. You can use revisions to release a new version of your app, or quickly revert to an earlier version of your app. New revisions are created when you update your application with revision-scope changes. You can also update your container app based on a specific revision.

You can control which revisions are active, and the external traffic that is routed to each active revision. Revision names are used to identify a revision, and in the revision's URL. You can customize the revision name by setting the revision suffix.

By default, Container Apps creates a unique revision name with a suffix consisting of a semi-random string of alphanumeric characters. For example, for a container app named *album-api*, setting the revision suffix name to *1st-revision* would create a revision with the name *album-api--1st-revision*. You can set the revision suffix in the ARM template, through the Azure CLI `az containerapp create` and `az containerapp update` commands, or when creating a revision via the Azure portal.

Updating your container app

With the `az containerapp update` command you can modify environment variables, compute resources, scale parameters, and deploy a different image. If your container app update includes revision-scope changes, a new revision is generated.

BashCopy

```
az containerapp update \
  --name <APPLICATION_NAME> \
  --resource-group <RESOURCE_GROUP_NAME> \
  --image <IMAGE_NAME>
```

You can list all revisions associated with your container app with the `az containerapp revision list` command.

BashCopy

```
az containerapp revision list \
  --name <APPLICATION_NAME> \
  --resource-group <RESOURCE_GROUP_NAME> \
  -o table
```

For more information about Container Apps commands, visit the az containerapp reference.

Manage secrets in Azure Container Apps

Azure Container Apps allows your application to securely store sensitive configuration values. Once secrets are defined at the application level, secured values are available to container apps. Specifically, you can reference secured values inside scale rules.

- Secrets are scoped to an application, outside of any specific revision of an application.
- Adding, removing, or changing secrets doesn't generate new revisions.
- Each application revision can reference one or more secrets.
- Multiple revisions can reference the same secret(s).

An updated or deleted secret doesn't automatically affect existing revisions in your app. When a secret is updated or deleted, you can respond to changes in one of two ways:

1. Deploy a new revision.
2. Restart an existing revision.

Before you delete a secret, deploy a new revision that no longer references the old secret. Then deactivate all revisions that reference the secret.

Note

Container Apps doesn't support Azure Key Vault integration. Instead, enable managed identity in the container app and use the Key Vault SDK in your app to access secrets.

Defining secrets

When you create a container app, secrets are defined using the `--secrets` parameter.

- The parameter accepts a space-delimited set of name/value pairs.
- Each pair is delimited by an equals sign (=).

In the example below a connection string to a queue storage account is declared in the `--secrets` parameter. The value for queue-connection-string comes from an environment variable named `$CONNECTION_STRING`.

BashCopy

```
az containerapp create \
  --resource-group "my-resource-group" \
  --name queuereader \
  --environment "my-environment-name" \
  --image demos/queuereader:v1 \
  --secrets "queue-connection-string=$CONNECTION_STRING"
```

After declaring secrets at the application level, you can reference them in environment variables when you create a new revision in your container app. When an environment variable references a secret, its value is populated with the value defined in the secret. To reference a secret in an environment variable in the Azure CLI, set its value to `secretref:`, followed by the name of the secret.

The following example shows an application that declares a connection string at the application level. This connection is referenced in a container environment variable.

BashCopy

```
az containerapp create \
  --resource-group "my-resource-group" \
  --name myQueueApp \
  --environment "my-environment-name" \
  --image demos/myQueueApp:v1 \
  --secrets "queue-connection-string=$CONNECTIONSTRING" \
  --env-vars "QueueName=myqueue" "ConnectionString=secretref:queue-connection-string"
```

Explore Dapr integration with Azure Container Apps

Completed100 XP

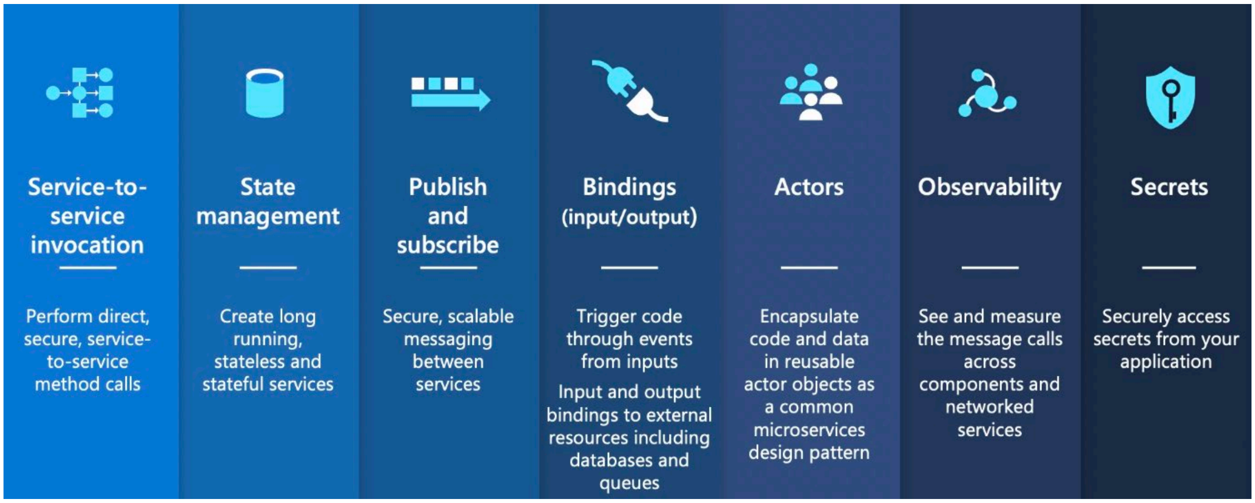
- 5 minutes

The Distributed Application Runtime (Dapr) is a set of incrementally adoptable features that simplify the authoring of distributed, microservice-based applications. Dapr provides capabilities for enabling application intercommunication through messaging via pub/sub or reliable and secure service-to-service calls.

Dapr is an open source, Cloud Native Computing Foundation (CNCF) project. The CNCF is part of the Linux Foundation and provides support, oversight, and direction for fast-growing, cloud native projects. As an alternative to deploying and managing the Dapr OSS project yourself, the Container Apps platform:

- Provides a managed and supported Dapr integration
- Handles Dapr version upgrades seamlessly
- Exposes a simplified Dapr interaction model to increase developer productivity

Dapr APIs



Service-to-service invocation

Perform direct, secure, service-to-service method calls

State management

Create long running, stateless and stateful services

Publish and subscribe

Secure, scalable messaging between services

Bindings (input/output)

Trigger code through events from inputs

Input and output bindings to external resources including databases and queues

Actors

Encapsulate code and data in reusable actor objects as a common microservices design pattern

Observability

See and measure the message calls across components and networked services

Secrets

Securely access secrets from your application

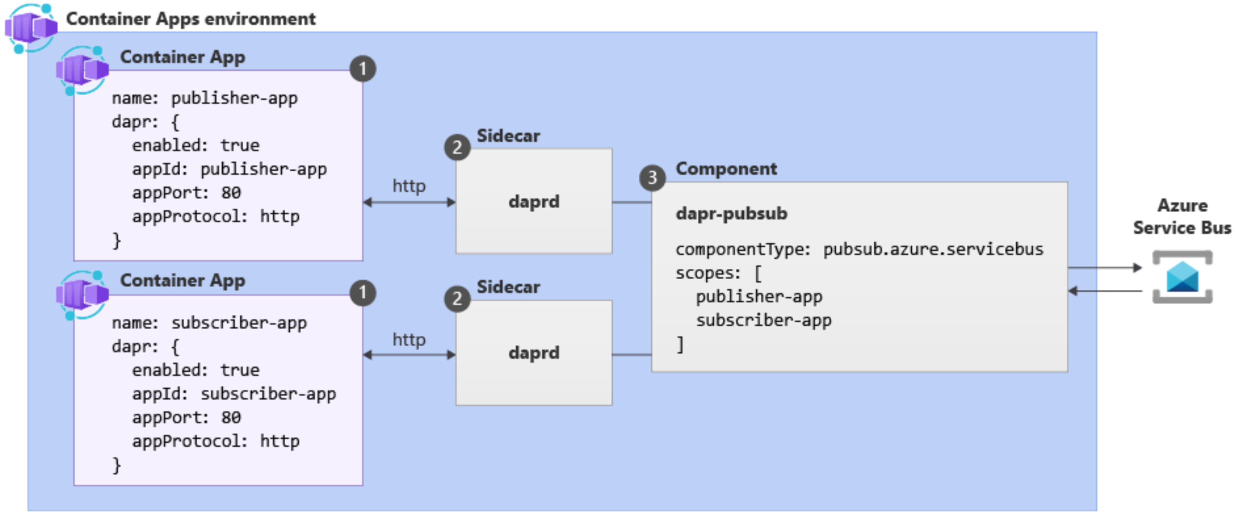
Dapr API	Description
Service-to-service invocation	Discover services and perform reliable, direct service-to-service calls with automatic mTLS auth encryption.
State management	Provides state management capabilities for transactions and CRUD operations.
Pub/sub	Allows publisher and subscriber container apps to intercommunicate via an intermediary mess
Bindings	Trigger your applications based on events
Actors	Dapr actors are message-driven, single-threaded, units of work designed to quickly scale. For e workload situations.
Observability	Send tracing information to an Application Insights backend.
Secrets	Access secrets from your application code or reference secure values in your Dapr components

Note

The table covers stable Dapr APIs. To learn more about using alpha APIs and features, **visit limitations**.

Dapr core concepts

The following example based on the Pub/sub API is used to illustrate core concepts related to Dapr in Azure Container Apps.



```
name: publisher-app
dapr: {
  enabled: true
  appId: publisher-app
  appPort: 80
  appProtocol: http
}
```

```
name: subscriber-app
dapr: {
  enabled: true
  appId: subscriber-app
  appPort: 80
  appProtocol: http
}
```

```
componentType: pubsub.azure.servicebus
scopes: [
  publisher-app
  subscriber-app
]
```

Label	Dapr settings	Description
1	Container Apps with Dapr enabled	Dapr is enabled at the container app level by configuring a set of Dapr arguments. These valu of a given container app when running in multiple revisions mode.
2	Dapr	The fully managed Dapr APIs are exposed to each container app through a Dapr sidecar. The D invoked from your container app via HTTP or gRPC. The Dapr sidecar runs on HTTP port 3500
3	Dapr component configuration	Dapr uses a modular design where functionality is delivered as a component. Dapr componer multiple container apps. The Dapr app identifiers provided in the scopes array dictate which c apps load a given component at runtime.

Dapr enablement

You can configure Dapr using various arguments and annotations based on the runtime context. Azure Container Apps provides three channels through which you can configure Dapr:

- Container Apps CLI
- Infrastructure as Code (IaC) templates, as in Bicep or Azure Resource Manager (ARM) templates
- The Azure portal

Dapr components and scopes

Dapr uses a modular design where functionality is delivered as a component. The use of Dapr components is optional and dictated exclusively by the needs of your application.

Dapr components in container apps are environment-level resources that:

- Can provide a pluggable abstraction model for connecting to supporting external services.
- Can be shared across container apps or scoped to specific container apps.
- Can use Dapr secrets to securely retrieve configuration metadata.

By default, all Dapr-enabled container apps within the same environment load the full set of deployed components. To ensure components are loaded at runtime by only the appropriate container apps, application scopes should be used.

Check your knowledge

1.

Which of the following options is true about the built-in authentication feature in Azure Container Apps?

○

It can only be configured to restrict access to authenticated users.

○

It allows for out-of-the-box authentication with federated identity providers.

Correct. Azure Container Apps provides built-in authentication and authorization features to secure your external ingress-enabled container app with minimal or no code.

○

It requires the use of a specific language or SDK.

2.

What is a revision in Azure Container Apps?

○

A dynamic snapshot of a container app version.

○

A version of a container app that is actively being used.

○

An immutable snapshot of a container app version.

Correct. A revision is an immutable snapshot of a container app version.

Manage container images in Azure Container Registry

Discover the Azure Container Registry

Completed100 XP

- 3 minutes

Use the Azure Container Registry (ACR) service with your existing container development and deployment pipelines, or use Azure Container Registry Tasks to build container images in Azure. Build on demand, or fully automate builds with triggers such as source code commits and base image updates.

Use cases

Pull images from an Azure container registry to various deployment targets:

- **Scalable orchestration systems** that manage containerized applications across clusters of hosts, including Kubernetes, DC/OS, and Docker Swarm.
- **Azure services** that support building and running applications at scale, including Azure Kubernetes Service (AKS), App Service, Batch, Service Fabric, and others.

Developers can also push to a container registry as part of a container development workflow. For example, target a container registry from a continuous integration and delivery tool such as Azure Pipelines or Jenkins.

Configure ACR Tasks to automatically rebuild application images when their base images are updated, or automate image builds when your team commits code to a Git repository. Create multi-step tasks to automate building, testing, and patching multiple container images in parallel in the cloud.

Azure Container Registry service tiers

Azure Container Registry is available in multiple service tiers. These tiers provide predictable pricing and several options for aligning to the capacity and usage patterns of your private Docker registry in Azure.

Tier	Description
Basic	A cost-optimized entry point for developers learning about Azure Container Registry. Basic registries have the same programmatic capabilities as Standard and Premium (such as Azure Active Directory authentication integration, image deletion, and webhooks). However, the included storage and image throughput are most appropriate for lower usage scenarios.
Standard	Standard registries offer the same capabilities as Basic, with increased included storage and image throughput. Standard registries should satisfy the needs of most production scenarios.
Premium	Premium registries provide the highest amount of included storage and concurrent operations, enabling high-volume scenarios. In addition to higher image throughput, Premium adds features such as geo-replication for managing a single registry across multiple regions, content trust for image tag signing, and private link with private endpoints to restrict access to the registry.

Supported images and artifacts

Grouped in a repository, each image is a read-only snapshot of a Docker-compatible container. Azure container registries can include both Windows and Linux images. In addition to Docker container images, Azure Container Registry stores related content formats such as Helm charts and images built to the Open Container Initiative (OCI) Image Format Specification.

Automated image builds

Use Azure Container Registry Tasks (ACR Tasks) to streamline building, testing, pushing, and deploying images in Azure. Configure build tasks to automate your container OS and framework patching pipeline, and build images automatically when your team commits code to source control.

Explore storage capabilities

Completed 100 XP

- 3 minutes

Basic, Standard, and Premium Azure container registry tiers benefit from advanced Azure storage features like encryption-at-rest for image data security and geo-redundancy for image data protection.

- **Encryption-at-rest:** All container images in your registry are encrypted at rest. Azure automatically encrypts an image before storing it, and decrypts it on-the-fly when you or your applications and services pull the image.
- **Regional storage:** Azure Container Registry stores data in the region where the registry is created, to help customers meet data residency and compliance requirements. In all regions except Brazil South and Southeast Asia, Azure may also store registry data in a paired region in the same geography. In the Brazil South and Southeast Asia regions, registry data is always confined to the region, to accommodate data residency requirements for those regions.

  If a regional outage occurs, the registry data may become unavailable and isn't automatically recovered. Customers who wish to have their registry data stored in multiple regions for better performance across different geographies or who wish to have resiliency in the event of a regional outage should enable geo-replication.

- **Zone redundancy:** A feature of the Premium service tier, zone redundancy uses Azure availability zones to replicate your registry to a minimum of three separate zones in each enabled region.
- **Scalable storage:** Azure Container Registry allows you to create as many repositories, images, layers, or tags as you need, up to the registry storage limit.

  High numbers of repositories and tags can impact the performance of your registry. Periodically delete unused repositories, tags, and images as part of your registry maintenance routine. Deleted registry resources like repositories, images, and tags *can't* be recovered after deletion.

Build and manage containers with tasks

Completed100 XP

- 3 minutes

ACR Tasks is a suite of features within Azure Container Registry. It provides cloud-based container image building for platforms including Linux, Windows, and Azure Resource Manager, and can automate OS and framework patching for your Docker containers. ACR Tasks enables automated builds triggered by source code updates, updates to a container's base image, or timers.

Task scenarios

ACR Tasks supports several scenarios to build and maintain container images and other artifacts.

- **Quick task** - Build and push a single container image to a container registry on-demand, in Azure, without needing a local Docker Engine installation. Think `docker build`, `docker push` in the cloud.
- **Automatically triggered tasks** - Enable one or more *triggers* to build an image:
  - Trigger on source code update
  - Trigger on base image update
  - Trigger on a schedule
- **Multi-step task** - Extend the single image build-and-push capability of ACR Tasks with multi-step, multi-container-based workflows.

Each ACR Task has an associated source code context - the location of a set of source files used to build a container image or other artifact. Example contexts include a Git repository or a local filesystem.

Quick task

Before you commit your first line of code, ACR Tasks's quick task feature can provide an integrated development experience by offloading your container image builds to Azure. With quick tasks, you can verify your automated build definitions and catch potential problems prior to committing your code.

Using the familiar `docker build` format, the az acr build command in the Azure CLI takes a context (the set of files to build), sends it to ACR Tasks and, by default, pushes the built image to its registry upon completion.

Trigger task on source code update

Trigger a container image build or multi-step task when code is committed, or a pull request is made or updated, to a Git repository in GitHub or Azure DevOps Services. For example, configure a build task with the Azure CLI command `az acr task create` by specifying a Git repository and optionally a branch and Dockerfile. When your team updates code in the repository, an ACR Tasks-created webhook triggers a build of the container image defined in the repo.

Trigger on base image update

You can set up an ACR task to track a dependency on a base image when it builds an application image. When the updated base image is pushed to your registry, or a base image is updated in a public repo such as in Docker Hub, ACR Tasks can automatically build any application images based on it.

Schedule a task

Optionally schedule a task by setting up one or more timer triggers when you create or update the task. Scheduling a task is useful for running container workloads on a defined schedule, or running maintenance operations or tests on images pushed regularly to your registry.

Multi-step tasks

Multi-step tasks, defined in a YAML file specify individual build and push operations for container images or other artifacts. They can also define the execution of one or more containers, with each step using the container as its execution environment. For example, you can create a multi-step task that automates the following:

1. Build a web application image
2. Run the web application container
3. Build a web application test image
4. Run the web application test container, which performs tests against the running application container
5. If the tests pass, build a Helm chart archive package
6. Perform a `helm upgrade` using the new Helm chart archive package

Image platforms

By default, ACR Tasks builds images for the Linux OS and the amd64 architecture. Specify the `--platform` tag to build Windows images or Linux images for other architectures. Specify the OS and optionally a supported architecture in OS/architecture format (for example, `--platform Linux/arm`). For ARM architectures, optionally specify a variant in OS/architecture/variant format (for example, `--platform Linux/arm64/v8`):

OS	Architecture
Linux	amd64 arm arm64 386
Windows	amd64

Explore elements of a Dockerfile

Completed100 XP

- 3 minutes

A Dockerfile is a script that contains a series of instructions that are used to build a Docker image. Dockerfiles typically include the following information:

- The base or parent image we use to create the new image
- Commands to update the base OS and install other software
- Build artifacts to include, such as a developed application
- Services to expose, such a storage and network configuration
- Command to run when the container is launched

Create a Dockerfile

The first step in creating a Dockerfile is choosing a base image that serves as the foundation for your application. For example, if you're building a .NET application, you might choose a Microsoft .NET image as your base.

DockerfileCopy

```dockerfile
# Use the .NET 6 runtime as a base image
FROM mcr.microsoft.com/dotnet/runtime:6.0

# Set the working directory to /app
WORKDIR /app

# Copy the contents of the published app to the container's /app directory
COPY bin/Release/net6.0/publish/ .

# Expose port 80 to the outside world
EXPOSE 80
```

```
# Set the command to run when the container starts
CMD ["dotnet", "MyApp.dll"]
```

Let's go through each line to see what it does:

- **`FROM mcr.microsoft.com/dotnet/runtime:6.0`**: This command sets the base image to the .NET 6 runtime, which is needed to run .NET 6 apps.
- **`WORKDIR /app`**: Sets the working directory inside the container to `/app`, which is where app files will be copied.
- **`COPY bin/Release/net6.0/publish/ .`**: Copies the contents of the app's publish folder to the container's `/app` directory. We assume that the .NET 6 app has already been built and published to the `bin/Release/net6.0/publish` directory.
- **`EXPOSE 80`**: Exposes port 80, which is the default HTTP port, to the outside world. Change this line accordingly if your app listens on a different port.
- **`CMD ["dotnet", "MyApp.dll"]`**: The command to run when the container starts. In this case, we're running the dotnet command with the name of our app's DLL file (`MyApp.dll`). Change this line to match your apps name and entry point.

We're not going to cover the Dockerfile file specification, visit the Dockerfile reference for more information. Each of these steps creates a cached container image as we build the final container image. These temporary images are layered on top of the previous and presented as single image once all steps complete.

Resources

- Docker run reference (CLI)
    - https://docs.docker.com/engine/reference/run/
- Docker build reference
    - https://docs.docker.com/engine/reference/commandline/build/

Check your knowledge

1.

Which of the following Azure Container Registry options support geo-replication to manage a single registry across multiple regions?

○

Basic

○

Standard

○

Premium

Correct. The premium tier adds geo-replication as a feature.

2.

Which Azure container registry tiers benefit from encryption-at-rest?

○

Basic, Standard, and Premium

Correct. Encryption-at-rest is supported in all three tiers.

○

Basic and Standard only

○

Premium only

Run container images in Azure Container Instances

Azure Container Instances (ACI) offers the fastest and simplest way to run a container in Azure, without having to manage any virtual machines and without having to adopt a higher-level service.

Explore Azure Container Instances

Completed 100 XP

- 3 minutes

Azure Container Instances (ACI) is a great solution for any scenario that can operate in isolated containers, including simple applications, task automation, and build jobs. Here are some of the benefits:

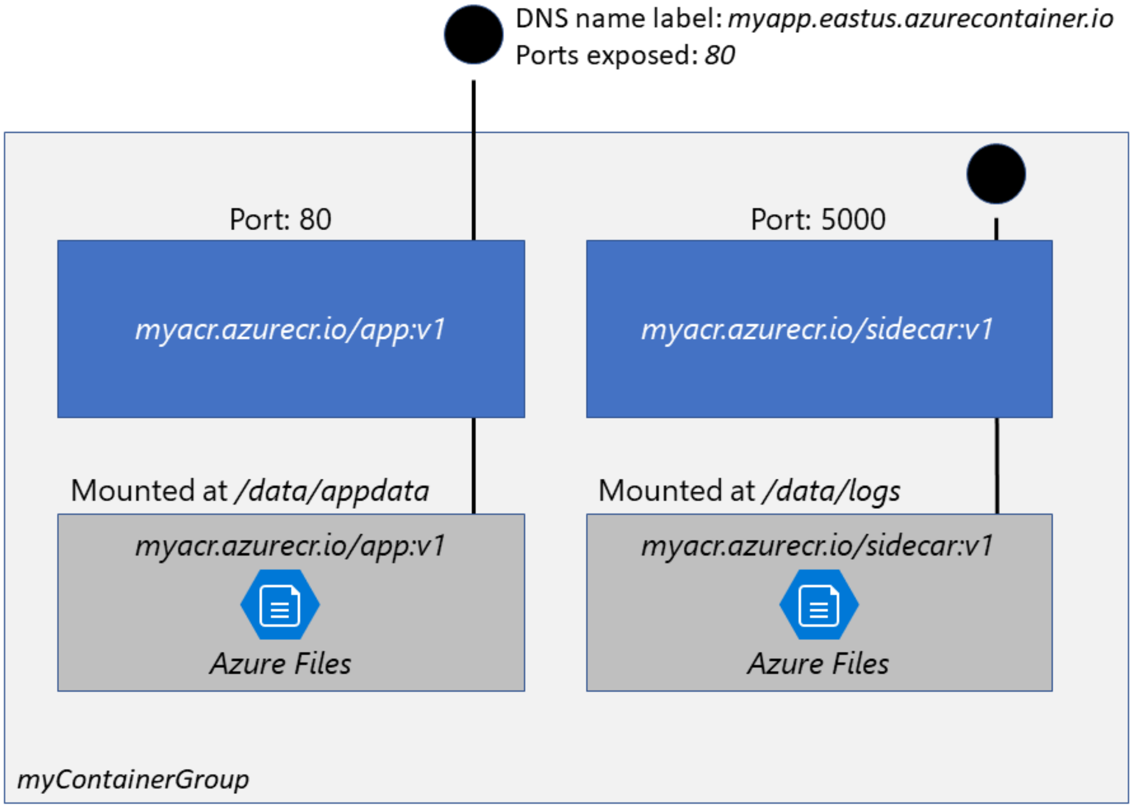
- **Fast startup**: ACI can start containers in Azure in seconds, without the need to provision and manage VMs
- **Container access**: ACI enables exposing your container groups directly to the internet with an IP address and a fully qualified domain name (FQDN)
- **Hypervisor-level security**: Isolate your application as completely as it would be in a VM
- **Customer data**: The ACI service stores the minimum customer data required to ensure your container groups are running as expected
- **Custom sizes**: ACI provides optimum utilization by allowing exact specifications of CPU cores and memory
- **Persistent storage**: Mount Azure Files shares directly to a container to retrieve and persist state
- **Linux and Windows**: Schedule both Windows and Linux containers using the same API.

For scenarios where you need full container orchestration, including service discovery across multiple containers, automatic scaling, and coordinated application upgrades, we recommend Azure Kubernetes Service (AKS).

Container groups

The top-level resource in Azure Container Instances is the *container group*. A container group is a collection of containers that get scheduled on the same host machine. The containers in a container group share a lifecycle, resources, local network, and storage volumes. It's similar in concept to a *pod* in Kubernetes.

The following diagram shows an example of a container group that includes multiple containers:


DNS name label: *myapp.eastus.azurecontainer.io*
Ports exposed: *80*
Port: 80
myacr.azurecr.io/app:v1
Port: 5000
myacr.azurecr.io/sidecar:v1
Mounted at */data/appdata*
myacr.azurecr.io/app:v1
Azure Files
Mounted at */data/logs*
myacr.azurecr.io/sidecar:v1
Azure Files
myContainerGroup

This example container group:

- Is scheduled on a single host machine.
- Is assigned a DNS name label.
- Exposes a single public IP address, with one exposed port.
- Consists of two containers. One container listens on port 80, while the other listens on port 5000.
- Includes two Azure file shares as volume mounts, and each container mounts one of the shares locally.

Note

Multi-container groups currently support only Linux containers. For Windows containers, Azure Container Instances only supports deployment of a single instance.

Deployment

There are two common ways to deploy a multi-container group: use a Resource Manager template or a YAML file. A Resource Manager template is recommended when you need to deploy additional Azure service resources (for example, an Azure Files share) when you deploy the container instances. Due to the YAML format's more concise nature, a YAML file is recommended when your deployment includes only container instances.

Resource allocation

Azure Container Instances allocates resources such as CPUs, memory, and optionally GPUs (preview) to a container group by adding the resource requests of the instances in the group. Using CPU resources as an example, if you create a container group with two instances, each requesting one CPU, then the container group is allocated two CPUs.

Networking

Container groups share an IP address and a port namespace on that IP address. To enable external clients to reach a container within the group, you must expose the port on the IP address and from the container. Because containers within the group share a port namespace, port mapping isn't supported. Containers within a group can reach each other via localhost on the ports that they've exposed, even if those ports aren't exposed externally on the group's IP address.

Storage

You can specify external volumes to mount within a container group. You can map those volumes into specific paths within the individual containers in a group. Supported volumes include:

- Azure file share
- Secret
- Empty directory
- Cloned git repo

Common scenarios

Multi-container groups are useful in cases where you want to divide a single functional task into a few container images. These images can then be delivered by different teams and have separate resource requirements.

Example usage could include:

- A container serving a web application and a container pulling the latest content from source control.
- An application container and a logging container. The logging container collects the logs and metrics output by the main application and writes them to long-term storage.
- An application container and a monitoring container. The monitoring container periodically makes a request to the application to ensure that it's running and responding correctly, and raises an alert if it's not.
- A front-end container and a back-end container. The front end might serve a web application, with the back end running a service to retrieve data.

Exercise - Deploy a container instance by using the Azure CLI

Completed100 XP

- 10 minutes

In this exercise you learn how to perform the following actions:

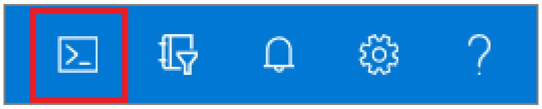
- Create a resource group for the container
- Create a container
- Verify the container is running

Prerequisites

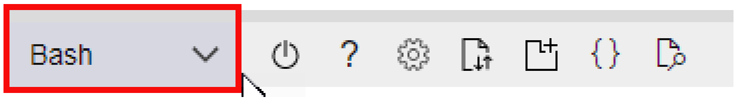
- An **Azure account** with an active subscription. If you don't already have one, you can sign up for a free trial at https://azure.com/free

Sign-in to Azure and create the resource group

1. Sign-in to the Azure portal and open the Cloud Shell.



2. When the shell opens be sure to select the **Bash** environment.



3. Create a new resource group with the name **az204-aci-rg** so that it's easier to clean up these resources when you're finished with the module. Replace `<myLocation>` with a region near you.

   BashCopy

   ```
   az group create --name az204-aci-rg --location <myLocation>
   ```

Create a container

You create a container by providing a name, a Docker image, and an Azure resource group to the `az container create` command. You expose the container to the Internet by specifying a DNS name label.

1. Create a DNS name to expose your container to the Internet. Your DNS name must be unique, run this command from Cloud Shell to create a variable that holds a unique name.

   BashCopy

   ```
   DNS_NAME_LABEL=aci-example-$RANDOM
   ```

2. Run the following `az container create` command to start a container instance. Be sure to replace the `<myLocation>` with the region you specified earlier. It takes a few minutes for the operation to complete.

   BashCopy

   ```
   az container create --resource-group az204-aci-rg
       --name mycontainer
       --image mcr.microsoft.com/azuredocs/aci-helloworld
       --ports 80
       --dns-name-label $DNS_NAME_LABEL --location <myLocation>
   ```

In the previous command, `$DNS_NAME_LABEL` specifies your DNS name. The image name, `mcr.microsoft.com/azuredocs/aci-helloworld`, refers to a Docker image that runs a basic Node.js web application.

Verify the container is running

1. When the `az container create` command completes, run `az container show` to check its status.

   BashCopy

   ```
   az container show --resource-group az204-aci-rg
       --name mycontainer
       --query "{FQDN:ipAddress.fqdn,ProvisioningState:provisioningState}"
       --out table
   ```

   You see your container's fully qualified domain name (FQDN) and its provisioning state. Here's an example.

   Copy

   ```
   FQDN                                    ProvisioningState
   --------------------------------------  -------------------
   aci-wt.eastus.azurecontainer.io         Succeeded
   ```

   Note

   If your container is in the **Creating** state, wait a few moments and run the command again until you see the **Succeeded** state.

2. From a browser, navigate to your container's FQDN to see it running. You may get a warning that the site isn't safe.

Clean up resources

When no longer needed, you can use the `az group delete` command to remove the resource group, the container registry, and the container images stored there.

BashCopy

```
az group delete --name az204-aci-rg --no-wait
```

Run containerized tasks with restart policies

Completed100 XP

- 3 minutes

The ease and speed of deploying containers in Azure Container Instances provides a compelling platform for executing run-once tasks like build, test, and image rendering in a container instance.

With a configurable restart policy, you can specify that your containers are stopped when their processes have completed. Because container instances are billed by the second, you're charged only for the compute resources used while the container executing your task is running.

Container restart policy

When you create a container group in Azure Container Instances, you can specify one of three restart policy settings.

Restart policy	Description
`Always`	Containers in the container group are always restarted. This is the **default** setting applied when no restart policy is specified at container creation.
`Never`	Containers in the container group are never restarted. The containers run at most once.
`OnFailure`	Containers in the container group are restarted only when the process executed in the container fails (when it terminates with a nonzero exit code). The containers are run at least once.

Specify a restart policy

Specify the `--restart-policy` parameter when you call `az container create`.

BashCopy

```
az container create \
    --resource-group myResourceGroup \
    --name mycontainer \
    --image mycontainerimage \
    --restart-policy OnFailure
```

Run to completion

Azure Container Instances starts the container, and then stops it when its application, or script, exits. When Azure Container Instances stops a container whose restart policy is `Never` or `OnFailure`, the container's status is set to **Terminated**.

Set environment variables in container instances

Completed100 XP

- 3 minutes

Setting environment variables in your container instances allows you to provide dynamic configuration of the application or script run by the container. These environment variables are similar to the `--env` command-line argument to `docker run`.

If you need to pass secrets as environment variables, Azure Container Instances supports secure values for both Windows and Linux containers.

In the following example, two variables are passed to the container when it's created. The example is assuming you're running the CLI in a Bash shell or Cloud Shell, if you use the Windows Command Prompt, specify the variables with double-quotes, such as `--environment-variables "NumWords"="5" "MinLength"="8"`.

BashCopy

```
az container create \
    --resource-group myResourceGroup \
    --name mycontainer2 \
    --image mcr.microsoft.com/azuredocs/aci-wordcount:latest
    --restart-policy OnFailure \
    --environment-variables 'NumWords'='5' 'MinLength'='8'\
```

Secure values

Objects with secure values are intended to hold sensitive information like passwords or keys for your application. Using secure values for environment variables is both safer and more flexible than including it in your container's image.

Environment variables with secure values aren't visible in your container's properties. Their values can be accessed only from within the container. For example, container properties viewed in the Azure portal or Azure CLI display only a secure variable's name, not its value.

Set a secure environment variable by specifying the `secureValue` property instead of the regular `value` for the variable's type. The two variables defined in the following YAML demonstrate the two variable types.

YAMLCopy

```yaml
apiVersion: 2018-10-01
location: eastus
name: securetest
properties:
  containers:
  - name: mycontainer
    properties:
      environmentVariables:
        - name: 'NOTSECRET'
          value: 'my-exposed-value'
        - name: 'SECRET'
          secureValue: 'my-secret-value'
      image: nginx
      ports: []
      resources:
        requests:
          cpu: 1.0
          memoryInGB: 1.5
  osType: Linux
  restartPolicy: Always
tags: null
type: Microsoft.ContainerInstance/containerGroups
```

You would run the following command to deploy the container group with YAML:

BashCopy

```bash
az container create --resource-group myResourceGroup \
    --file secure-env.yaml \
```

Mount an Azure file share in Azure Container Instances

Completed100 XP

- 3 minutes

By default, Azure Container Instances are stateless. If the container crashes or stops, all of its state is lost. To persist state beyond the lifetime of the container, you must mount a volume from an external store. As shown in this unit, Azure Container Instances can mount an Azure file share created with Azure Files. Azure Files offers fully managed file shares in the cloud that are accessible via the industry standard Server Message Block (SMB) protocol. Using an Azure file share with Azure Container Instances provides file-sharing features similar to using an Azure file share with Azure virtual machines.

Limitations

- You can only mount Azure Files shares to Linux containers.
- Azure file share volume mount requires the Linux container run as *root*.
- Azure File share volume mounts are limited to CIFS support.

Deploy container and mount volume

To mount an Azure file share as a volume in a container by using the Azure CLI, specify the share and volume mount point when you create the container with `az container create`. Following is an example of the command:

Azure CLICopy

```
az container create \
    --resource-group $ACI_PERS_RESOURCE_GROUP \
    --name hellofiles \
    --image mcr.microsoft.com/azuredocs/aci-hellofiles \
    --dns-name-label aci-demo \
    --ports 80 \
    --azure-file-volume-account-name $ACI_PERS_STORAGE_ACCOUNT_NAME \
    --azure-file-volume-account-key $STORAGE_KEY \
    --azure-file-volume-share-name $ACI_PERS_SHARE_NAME \
    --azure-file-volume-mount-path /aci/logs/
```

The `--dns-name-label` value must be unique within the Azure region where you create the container instance. Update the value in the preceding command if you receive a **DNS name label** error message when you execute the command.

Deploy container and mount volume - YAML

You can also deploy a container group and mount a volume in a container with the Azure CLI and a YAML template. Deploying by YAML template is the preferred method when deploying container groups consisting of multiple containers.

The following YAML template defines a container group with one container created with the `aci-hellofiles` image. The container mounts the Azure file share *acishare* created previously as a volume. Following is an example YAML file.

YAMLCopy

```yaml
apiVersion: '2019-12-01'
location: eastus
name: file-share-demo
properties:
  containers:
  - name: hellofiles
    properties:
      environmentVariables: []
      image: mcr.microsoft.com/azuredocs/aci-hellofiles
      ports:
      - port: 80
      resources:
        requests:
          cpu: 1.0
          memoryInGB: 1.5
      volumeMounts:
      - mountPath: /aci/logs/
        name: filesharevolume
  osType: Linux
  restartPolicy: Always
  ipAddress:
    type: Public
    ports:
      - port: 80
    dnsNameLabel: aci-demo
  volumes:
  - name: filesharevolume
    azureFile:
      sharename: acishare
      storageAccountName: <Storage account name>
      storageAccountKey: <Storage account key>
tags: {}
type: Microsoft.ContainerInstance/containerGroups
```

Mount multiple volumes

To mount multiple volumes in a container instance, you must deploy using an Azure Resource Manager template or a YAML file. To use a template or YAML file, provide the share details and define the volumes by populating the `volumes` array in the `properties` section of the template.

For example, if you created two Azure Files shares named *share1* and *share2* in storage account *myStorageAccount*, the `volumes` array in a Resource Manager template would appear similar to the following:

JSONCopy

```json
"volumes": [{
  "name": "myvolume1",
  "azureFile": {
    "shareName": "share1",
    "storageAccountName": "myStorageAccount",
    "storageAccountKey": "<storage-account-key>"
  }
},
{
  "name": "myvolume2",
  "azureFile": {
    "shareName": "share2",
    "storageAccountName": "myStorageAccount",
    "storageAccountKey": "<storage-account-key>"
  }
}]
```

Next, for each container in the container group in which you'd like to mount the volumes, populate the `volumeMounts` array in the `properties` section of the container definition. For example, this mounts the two volumes, *myvolume1* and *myvolume2*, previously defined:

JSONCopy

```json
"volumeMounts": [{
  "name": "myvolume1",
  "mountPath": "/mnt/share1/"
},
{
  "name": "myvolume2",
  "mountPath": "/mnt/share2/"
}]
```

Check your knowledge

1.

Which of the following methods is recommended when deploying a multi-container group that includes only containers?

○

Azure Resource Management template

Incorrect. Due to the YAML format's more concise nature, a YAML file is recommended when your deployment includes only container instances.

○

YAML file

Correct. Due to the YAML format's more concise nature, a YAML file is recommended when your deployment includes only container instances.

○

`az container creates` command

Work with Azure Cosmos DB

Explore Microsoft .NET SDK v3 for Azure Cosmos DB

Completed100 XP

- 3 minutes

This **unit** focuses on Azure Cosmos DB .NET SDK v3 for API for NoSQL. (**Microsoft.Azure.Cosmos** NuGet package.) If you're familiar with the previous version of the .NET SDK, you may be used to the terms collection and document.

The azure-cosmos-dotnet-v3 GitHub repository includes the latest .NET sample solutions. You use these solutions to perform CRUD (create, read, update, and delete) and other common operations on Azure Cosmos DB resources.

Because Azure Cosmos DB supports multiple API models, version 3 of the .NET SDK uses the generic terms "container" and "item". A ***container*** can be a collection, graph, or table. An ***item*** can be a document, edge/vertex, or row, and is the content inside a container.

Below are examples showing some of the key operations you should be familiar with. For more examples, please visit the GitHub link shown earlier. The examples below all use the async version of the methods.

CosmosClient

Creates a new `CosmosClient` with a connection string. `CosmosClient` is thread-safe. It's recommended to maintain a single instance of `CosmosClient` per lifetime of the application that enables efficient connection management and performance.

C#Copy

```
CosmosClient client = new CosmosClient(endpoint, key);
```

Database examples

Create a database

The `CosmosClient.CreateDatabaseIfNotExistsAsync` checks if a database exists, and if it doesn't, creates it. Only the database `id` is used to verify if there's an existing database.

C#Copy

```
// An object containing relevant information about the response
DatabaseResponse databaseResponse = await
client.CreateDatabaseIfNotExistsAsync(databaseId, 10000);
```

Read a database by ID

Reads a database from the Azure Cosmos DB service as an asynchronous operation.

C#Copy

```
DatabaseResponse readResponse = await database.ReadAsync();
```

Delete a database

Delete a Database as an asynchronous operation.

C#Copy

```
await database.DeleteAsync();
```

Container examples

Create a container

The `Database.CreateContainerIfNotExistsAsync` method checks if a container exists, and if it doesn't, it creates it. Only the container `id` is used to verify if there's an existing container.

C#Copy

```
// Set throughput to the minimum value of 400 RU/s
ContainerResponse simpleContainer = await database.CreateContainerIfNotExistsAsync(
    id: containerId,
    partitionKeyPath: partitionKey,
    throughput: 400);
```

Get a container by ID

C#Copy

```
Container container = database.GetContainer(containerId);
ContainerProperties containerProperties = await container.ReadContainerAsync();
```

Delete a container

Delete a Container as an asynchronous operation.

C#Copy

```
await database.GetContainer(containerId).DeleteContainerAsync();
```

Item examples

Create an item

Use the `Container.CreateItemAsync` method to create an item. The method requires a JSON serializable object that must contain an `id` property, and a `partitionKey`.

C#Copy

```
ItemResponse<SalesOrder> response = await container.CreateItemAsync(salesOrder, new
PartitionKey(salesOrder.AccountNumber));
```

Read an item

Use the `Container.ReadItemAsync` method to read an item. The method requires type to serialize the item to along with an `id` property, and a `partitionKey`.

C#Copy

```
string id = "[id]";
string accountNumber = "[partition-key]";
ItemResponse<SalesOrder> response = await container.ReadItemAsync(id, new
PartitionKey(accountNumber));
```

Query an item

The `Container.GetItemQueryIterator` method creates a query for items under a container in an Azure Cosmos database using a SQL statement with parameterized values. It returns a `FeedIterator`.

C#Copy

```
QueryDefinition query = new QueryDefinition(
    "select * from sales s where s.AccountNumber = @AccountInput ")
    .WithParameter("@AccountInput", "Account1");

FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<SalesOrder>(
    query,
    requestOptions: new QueryRequestOptions()
    {
        PartitionKey = new PartitionKey("Account1"),
        MaxItemCount = 1
    });
```

Other resources

- The azure-cosmos-dotnet-v3 GitHub repository includes the latest .NET sample solutions to perform CRUD and other common operations on Azure Cosmos DB resources.
- Visit this article Azure Coaz gsmos DB.NET V3 SDK (Microsoft.Azure.Cosmos) examples for the SQL API for direct links to specific examples in the GitHub repository.

Exercise: Create resources by using the Microsoft .NET SDK v3

Completed100 XP

- 15 minutes

In this exercise you create a console app to perform the following operations in Azure Cosmos DB:

- Connect to an Azure Cosmos DB account
- Create a database
- Create a container

Prerequisites

- An Azure account with an active subscription. If you don't already have one, you can sign up for a free trial at https://azure.com/free.
- Visual Studio Code on one of the supported platforms.
- .NET 6 is the target framework for the exercise.

- The C# extension for Visual Studio Code.

Setting up

Perform the following actions to prepare Azure, and your local environment, for the exercise.

Connecting to Azure

1. Start Visual Studio Code and open a terminal window by selecting **Terminal** from the top application bar, then choosing **New Terminal**.
2. Sign in to Azure by using the following command. A browser window should open letting you choose which account to sign in with.

   Copy
   ```
   az login
   ```

Create resources in Azure

1. Create a resource group for the resources needed for this exercise. Replace `<myLocation>` with a region near you.

   Copy
   ```
   az group create --location <myLocation> --name az204-cosmos-rg
   ```

2. Create the Azure Cosmos DB account. Replace `<myCosmosDBacct>` with a *unique* name to identify your Azure Cosmos DB account. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length. *This command takes a few minutes to complete.*

   Copy
   ```
   az cosmosdb create --name <myCosmosDBacct> --resource-group az204-cosmos-rg
   ```

   Record the `documentEndpoint` shown in the JSON response, it's used later in the exercise.

3. Retrieve the primary key for the account by using the following command. Record the `primaryMasterKey` from the command results it will be used in the code.

   Copy
   ```
   # Retrieve the primary key
   az cosmosdb keys list --name <myCosmosDBacct> --resource-group az204-
   cosmos-rg
   ```

Set up the console application

Now that the needed resources are deployed to Azure the next step is to set up the console application using the same terminal window in Visual Studio Code.

1. Create a folder for the project and change in to the folder.

   Copy
   ```
   md az204-cosmos
   cd az204-cosmos
   ```

2. Create the .NET console app.

   Copy
   ```
   dotnet new console
   ```

3. Open the current folder in Visual Studio Code using the following command. The `-r` option opens the folder without launching a new Visual Studio Code window.

   Copy
   ```
   code . -r
   ```

4. Select the *Program.cs* file in the **Explorer** pane to open the file in the editor.

Build the console app

It's time to start adding the packages and code to the project.

Add packages and using statements

1. Open the terminal in Visual Studio Code and use the following command to add the `Microsoft.Azure.Cosmos` package to the project.

   Copy
   ```
   dotnet add package Microsoft.Azure.Cosmos
   ```

2. Delete any existing code in the `Program.cs` file and add the `using Microsoft.Azure.Cosmos` statement.

   C#Copy
   ```
   using Microsoft.Azure.Cosmos;
   ```

Add code to connect to an Azure Cosmos DB account

1. Add the following code snippet after the `using` statement. The code snippet adds constants and variables into the class and adds some error checking. Be sure to replace the placeholder values for `EndpointUri` and `PrimaryKey` following the directions in the code comments.

   C#Copy
   ```
   public class Program
   {
       // Replace <documentEndpoint> with the information created earlier
       private static readonly string EndpointUri = "<documentEndpoint>";

       // Set variable to the Primary Key from earlier.
       private static readonly string PrimaryKey = "<your primary key>";

       // The Cosmos client instance
       private CosmosClient cosmosClient;

       // The database we will create
       private Database database;

       // The container we will create.
       private Container container;

       // The names of the database and container we will create
       private string databaseId = "az204Database";
       private string containerId = "az204Container";

       public static async Task Main(string[] args)
       {
           try
           {
               Console.WriteLine("Beginning operations...\n");
               Program p = new Program();
   ```

```
            await p.CosmosAsync();

        }
        catch (CosmosException de)
        {
            Exception baseException = de.GetBaseException();
            Console.WriteLine("{0} error occurred: {1}", de.StatusCode,
de);
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: {0}", e);
        }
        finally
        {
            Console.WriteLine("End of program, press any key to exit.");
            Console.ReadKey();
        }
    }
    //The sample code below gets added below this line
}
```

2. Below the `Main` method, add a new asynchronous task called `CosmosAsync`, which instantiates our new `CosmosClient` and adds code to call the methods you add later to create a database and a container.

   C#Copy

   ```
   public async Task CosmosAsync()
   {
       // Create a new instance of the Cosmos Client
       this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);

       // Runs the CreateDatabaseAsync method
       await this.CreateDatabaseAsync();

       // Run the CreateContainerAsync method
       await this.CreateContainerAsync();
   }
   ```

Create a database

Copy and paste the `CreateDatabaseAsync` method after the `CosmosAsync` method. `CreateDatabaseAsync` creates a new database with ID `az204Database` if it doesn't already exist.

C#Copy

```
private async Task CreateDatabaseAsync()
{
    // Create a new database using the cosmosClient
```

```
    this.database = await
this.cosmosClient.CreateDatabaseIfNotExistsAsync(databaseId);
    Console.WriteLine("Created Database: {0}\n", this.database.Id);
}
```

Create a container

Copy and paste the `CreateContainerAsync` method below the `CreateDatabaseAsync` method.

C#Copy

```
private async Task CreateContainerAsync()
{
    // Create a new container
    this.container = await this.database.CreateContainerIfNotExistsAsync(containerId,
"/LastName");
    Console.WriteLine("Created Container: {0}\n", this.container.Id);
}
```

Run the application

1. Save your work and, in a terminal in Visual Studio Code, run the `dotnet build` command to check for any errors. If the build is successful run the `dotnet run` command. The console displays the following messages.

   Copy

   ```
   Beginning operations...

   Created Database: az204Database

   Created Container: az204Container

   End of program, press any key to exit.
   ```

2. Verify the results by opening the Azure portal, navigating to your Azure Cosmos DB resource, and use the **Data Explorer** to view the database and container.

Clean up Azure resources

You can now safely delete the *az204-cosmos-rg* resource group from your account by running the following command.

Copy

```
az group delete --name az204-cosmos-rg --no-wait
```

Create stored procedures

Completed100 XP

- 3 minutes

Azure Cosmos DB provides language-integrated, transactional execution of JavaScript that lets you write **stored procedures**, **triggers**, and **user-defined functions (UDFs)**. To call a stored procedure, trigger, or user-defined function, you need to register it. For more information, see How to work with stored procedures, triggers, user-defined functions in Azure Cosmos DB.

Note

This unit focuses on stored procedures, the following unit covers triggers and user-defined functions.

Writing stored procedures

Stored procedures can create, update, read, query, and delete items inside an Azure Cosmos container. Stored procedures are registered per collection, and can operate on any document or an attachment present in that collection.

Here's a simple stored procedure that returns a "Hello World" response.

JavaScriptCopy

```
var helloWorldStoredProc = {
    id: "helloWorld",
    serverScript: function () {
        var context = getContext();
        var response = context.getResponse();

        response.setBody("Hello, World");
    }
}
```

The context object provides access to all operations that can be performed in Azure Cosmos DB, and access to the request and response objects. In this case, you use the response object to set the body of the response to be sent back to the client.

Create an item using stored procedure

When you create an item by using stored procedure, it's inserted into the Azure Cosmos container and an ID for the newly created item is returned. Creating an item is an asynchronous operation and depends on the JavaScript callback functions. The callback function has two parameters:

- The error object in case the operation fails
- A return value

Inside the callback, you can either handle the exception or throw an error. In case a callback isn't provided and there's an error, the Azure Cosmos DB runtime throws an error.

The stored procedure also includes a parameter to set the description, it's a boolean value. When the parameter is set to true and the description is missing, the stored procedure throws an exception. Otherwise, the rest of the stored procedure continues to run.

This stored procedure takes as input `documentToCreate`, the body of a document to be created in the current collection. All such operations are asynchronous and depend on JavaScript function callbacks. The callback function has two parameters, one for the error object in case the operation fails, and one for the created object. Inside the callback, users can either handle the exception or throw an error. In case a callback isn't provided and there's an error, the DocumentDB runtime throws an error.

JavaScriptCopy

```
var createDocumentStoredProc = {
    id: "createMyDocument",
    body: function createMyDocument(documentToCreate) {
        var context = getContext();
        var collection = context.getCollection();
        var accepted = collection.createDocument(collection.getSelfLink(),
              documentToCreate,
              function (err, documentCreated) {
                  if (err) throw new Error('Error' + err.message);
                  context.getResponse().setBody(documentCreated.id)
              });
        if (!accepted) return;
    }
}
```

Arrays as input parameters for stored procedures

When defining a stored procedure in the Azure portal, input parameters are always sent as a string to the stored procedure. Even if you pass an array of strings as an input, the array is converted to string and sent to the stored procedure. To work around this, you can define a function within your stored procedure to parse the string as an array. The following code shows how to parse a string input parameter as an array:

JavaScriptCopy

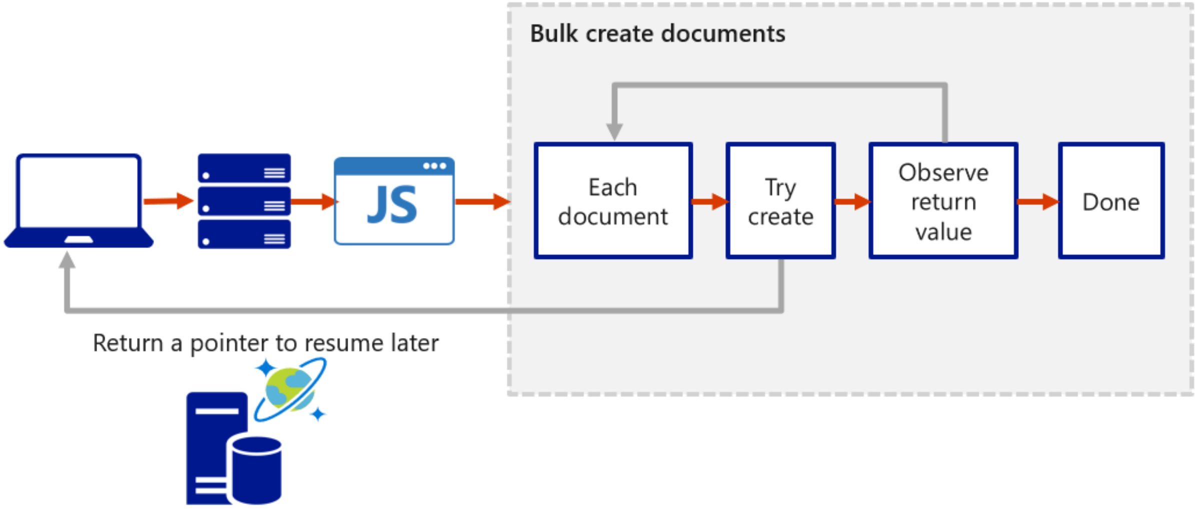
```javascript
function sample(arr) {
    if (typeof arr === "string") arr = JSON.parse(arr);

    arr.forEach(function(a) {
        // do something here
        console.log(a);
    });
}
```

Bounded execution

All Azure Cosmos DB operations must complete within a limited amount of time. Stored procedures have a limited amount of time to run on the server. All collection functions return a Boolean value that represents whether that operation completes or not

Transactions within stored procedures

You can implement transactions on items within a container by using a stored procedure. JavaScript functions can implement a continuation-based model to batch or resume execution. The continuation value can be any value of your choice and your applications can then use this value to resume a transaction from a new starting point. The diagram below depicts how the transaction continuation model can be used to repeat a server-side function until the function finishes its entire processing workload.



Create triggers and user-defined functions

Completed100 XP

- 3 minutes

Azure Cosmos DB supports pretriggers and post-triggers. Pretriggers are executed before modifying a database item and post-triggers are executed after modifying a database item. Triggers aren't automatically executed, they must be specified for each database operation where you want them to execute. After you define a trigger, you should register it by using the Azure Cosmos DB SDKs.

For examples of how to register and call a trigger, see pretriggers and post-triggers.

Pretriggers

The following example shows how a pretrigger is used to validate the properties of an Azure Cosmos item that is being created, it adds a timestamp property to a newly added item if it doesn't contain one.

JavaScriptCopy

```
function validateToDoItemTimestamp() {
    var context = getContext();
    var request = context.getRequest();

    // item to be created in the current operation
    var itemToCreate = request.getBody();
    // validate properties
    if (!("timestamp" in itemToCreate)) {
        var ts = new Date();
        itemToCreate["timestamp"] = ts.getTime();
    }

    // update the item that will be created
    request.setBody(itemToCreate);
}
```

Pretriggers can't have any input parameters. The request object in the trigger is used to manipulate the request message associated with the operation. In the previous example, the pretrigger is run when creating an Azure Cosmos item, and the request message body contains the item to be created in JSON format.

When triggers are registered, you can specify the operations that it can run with. This trigger should be created with a `TriggerOperation` value of `TriggerOperation.Create`, which means using the trigger in a replace operation isn't permitted.

For examples of how to register and call a pretrigger, visit the pretriggers article.

Post-triggers

The following example shows a post-trigger. This trigger queries for the metadata item and updates it with details about the newly created item.

JavaScriptCopy

```
function updateMetadata() {
var context = getContext();
var container = context.getCollection();
var response = context.getResponse();

// item that was created
var createdItem = response.getBody();

// query for metadata document
var filterQuery = 'SELECT * FROM root r WHERE r.id = "_metadata"';
var accept = container.queryDocuments(container.getSelfLink(), filterQuery,
    updateMetadataCallback);
if(!accept) throw "Unable to update metadata, abort";

function updateMetadataCallback(err, items, responseOptions) {
    if(err) throw new Error("Error" + err.message);
        if(items.length != 1) throw 'Unable to find metadata document';

        var metadataItem = items[0];
        // update metadata
        metadataItem.createdItems += 1;
        metadataItem.createdNames += " " + createdItem.id;
        var accept = container.replaceDocument(metadataItem._self,
            metadataItem, function(err, itemReplaced) {
                    if(err) throw "Unable to update metadata, abort";
            });
        if(!accept) throw "Unable to update metadata, abort";
        return;
    }
}
```

One thing that is important to note is the transactional execution of triggers in Azure Cosmos DB. The post-trigger runs as part of the same transaction for the underlying item itself. An exception during the post-trigger execution fails the whole transaction. Anything committed is rolled back and an exception returned.

User-defined functions

The following sample creates a UDF to calculate income tax for various income brackets. This user-defined function would then be used inside a query. For the purposes of this example assume there's a container called "Incomes" with properties as follows:

JSONCopy

```
{
   "name": "User One",
   "country": "USA",
   "income": 70000
}
```

The following is a function definition to calculate income tax for various income brackets:

JavaScriptCopy

```
function tax(income) {

        if(income == undefined)
            throw 'no input';

        if (income < 1000)
            return income * 0.1;
        else if (income < 10000)
            return income * 0.2;
        else
            return income * 0.4;
    }
```

Explore change feed in Azure Cosmos DB

Completed100 XP

- 5 minutes

Change feed in Azure Cosmos DB is a persistent record of changes to a container in the order they occur. Change feed support in Azure Cosmos DB works by listening to an Azure Cosmos DB container for any changes. It then outputs the sorted list of documents that were changed in the order in which they were modified. The persisted changes can be processed asynchronously and incrementally, and the output can be distributed across one or more consumers for parallel processing.

Change feed and different operations

Today, you see all inserts and updates in the change feed. You can't filter the change feed for a specific type of operation. Currently change feed doesn't log delete operations. As a workaround, you can add a soft marker on the items that are being deleted. For example, you can add an attribute in the item called "deleted," set its value to "true," and then set a time-to-live (TTL) value on the item. Setting the TTL ensures that the item is automatically deleted.

Reading Azure Cosmos DB change feed

You can work with the Azure Cosmos DB change feed using either a push model or a pull model. With a push model, the change feed processor pushes work to a client that has business logic for processing this work. However, the complexity in checking for work and storing state for the last processed work is handled within the change feed processor.

With a pull model, the client has to pull the work from the server. The client, in this case, not only has business logic for processing work but also storing state for the last processed work, handling load balancing across multiple clients processing work in parallel, and handling errors.

Note

It is recommended to use the push model because you won't need to worry about polling the change feed for future changes, storing state for the last processed change, and other benefits.

Most scenarios that use the Azure Cosmos DB change feed use one of the push model options. However, there are some scenarios where you might want the additional low level control of the pull model. These include:

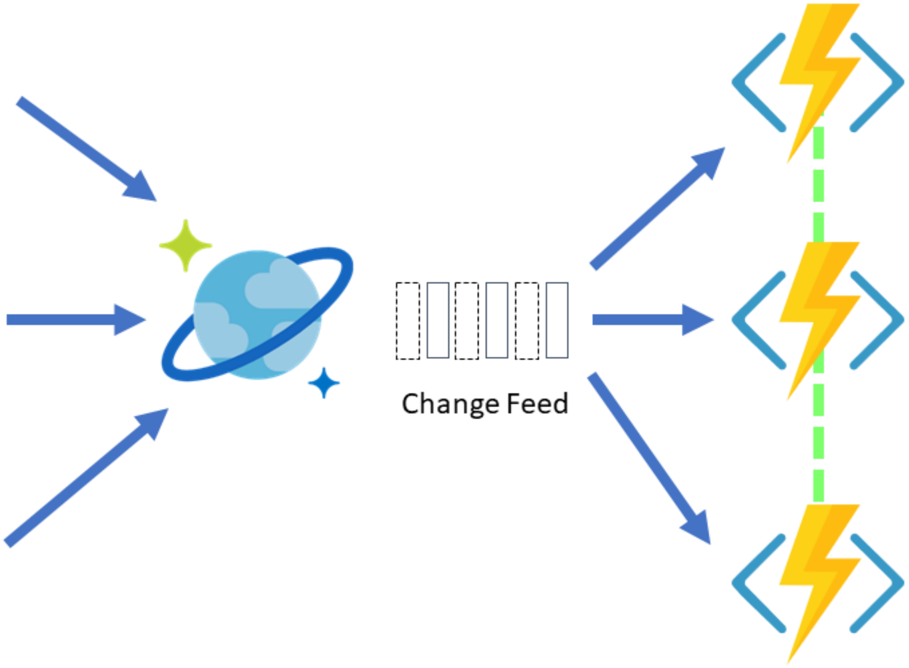
- Reading changes from a particular partition key
- Controlling the pace at which your client receives changes for processing
- Doing a one-time read of the existing data in the change feed (for example, to do a data migration)

Reading change feed with a push model

There are two ways you can read from the change feed with a push model: Azure Functions Azure Cosmos DB triggers, and the change feed processor library. Azure Functions uses the change feed processor behind the scenes, so these are both similar ways to read the change feed. Think of Azure Functions as simply a hosting platform for the change feed processor, not an entirely different way of reading the change feed. Azure Functions uses the change feed processor behind the scenes, it automatically parallelizes change processing across your container's partitions.

Azure Functions

You can create small reactive Azure Functions that will be automatically triggered on each new event in your Azure Cosmos DB container's change feed. With the Azure Functions trigger for Azure Cosmos DB, you can use the Change Feed Processor's scaling and reliable event detection functionality without the need to maintain any worker infrastructure.



Change feed processor

The change feed processor is part of the Azure Cosmos DB .NET V3 and Java V4 SDKs. It simplifies the process of reading the change feed and distributes the event processing across multiple consumers effectively.

There are four main components of implementing the change feed processor:

1. **The monitored container**: The monitored container has the data from which the change feed is generated. Any inserts and updates to the monitored container are reflected in the change feed of the container.
2. **The lease container**: The lease container acts as a state storage and coordinates processing the change feed across multiple workers. The lease container can be stored in the same account as the monitored container or in a separate account.
3. **The compute instance**: A compute instance hosts the change feed processor to listen for changes. Depending on the platform, it could be represented by a VM, a kubernetes pod, an Azure App Service instance, an actual physical machine. It has a unique identifier referenced as the instance name throughout this article.

4. **The delegate**: The delegate is the code that defines what you, the developer, want to do with each batch of changes that the change feed processor reads.

When implementing the change feed processor the point of entry is always the monitored container, from a `Container` instance you call `GetChangeFeedProcessorBuilder`:

C#Copy

```
/// <summary>
/// Start the Change Feed Processor to listen for changes and process them with the
HandleChangesAsync implementation.
/// </summary>
private static async Task<ChangeFeedProcessor> StartChangeFeedProcessorAsync(
    CosmosClient cosmosClient,
    IConfiguration configuration)
{
    string databaseName = configuration["SourceDatabaseName"];
    string sourceContainerName = configuration["SourceContainerName"];
    string leaseContainerName = configuration["LeasesContainerName"];

    Container leaseContainer = cosmosClient.GetContainer(databaseName,
leaseContainerName);
    ChangeFeedProcessor changeFeedProcessor = cosmosClient.GetContainer(databaseName,
sourceContainerName)
        .GetChangeFeedProcessorBuilder<ToDoItem>(processorName: "changeFeedSample",
onChangesDelegate: HandleChangesAsync)
            .WithInstanceName("consoleHost")
            .WithLeaseContainer(leaseContainer)
            .Build();

    Console.WriteLine("Starting Change Feed Processor...");
    await changeFeedProcessor.StartAsync();
    Console.WriteLine("Change Feed Processor started.");
    return changeFeedProcessor;
}
```

Where the first parameter is a distinct name that describes the goal of this processor and the second name is the delegate implementation that will handle changes. Following is an example of a delegate:

C#Copy

```
/// <summary>
/// The delegate receives batches of changes as they are generated in the change feed
and can process them.
/// </summary>
static async Task HandleChangesAsync(
    ChangeFeedProcessorContext context,
    IReadOnlyCollection<ToDoItem> changes,
    CancellationToken cancellationToken)
{
    Console.WriteLine($"Started handling changes for lease {context.LeaseToken}...");
    Console.WriteLine($"Change Feed request consumed {context.Headers.RequestCharge}
RU.");
    // SessionToken if needed to enforce Session consistency on another client
instance
    Console.WriteLine($"SessionToken ${context.Headers.Session}");

    // We may want to track any operation's Diagnostics that took longer than some
threshold
    if (context.Diagnostics.GetClientElapsedTime() > TimeSpan.FromSeconds(1))
    {
        Console.WriteLine($"Change Feed request took longer than expected.
Diagnostics:" + context.Diagnostics.ToString());
    }

    foreach (ToDoItem item in changes)
    {
        Console.WriteLine($"Detected operation for item with id {item.id}, created at
{item.creationTime}.");
        // Simulate some asynchronous operation
        await Task.Delay(10);
    }

    Console.WriteLine("Finished handling changes.");
}
```

Afterwards, you define the compute instance name or unique identifier with `WithInstanceName`, this should be unique and different in each compute instance you're deploying, and finally, which is the container to maintain the lease state with `WithLeaseContainer`.

Calling `Build` gives you the processor instance that you can start by calling `StartAsync`.

The normal life cycle of a host instance is:

1. Read the change feed.
2. If there are no changes, sleep for a predefined amount of time (customizable with `WithPollInterval` in the `Builder`) and go to #1.
3. If there are changes, send them to the delegate.
4. When the delegate finishes processing the changes successfully, update the lease store with the latest processed point in time and go to #1.

Check your knowledge

1.

When defining a stored procedure in the Azure portal input parameters are always sent as what type to the stored procedure?

○

String

Correct. When defining a stored procedure in Azure portal, input parameters are always sent as a string to the stored procedure.

○

Integer

○

Boolean

2.

Which of the following would one use to validate properties of an item being created?

○

Pretrigger

Correct. Pretriggers can be used to conform data before it's added to the container.

○

Post-trigger

○

User-defined function

Consume an Azure Cosmos DB for NoSQL change feed using the SDK

Understand change feed features in the SDK

Completed100 XP

- 1 minute

The .NET SDK for Azure Cosmos DB for NoSQL ships with a change feed processor that simplifies the task of reading changes from the feed. The change feed processor also natively supports distributed scenarios where event processing responsibilities are shared across multiple consumer client applications in an efficient manner.

The change feed processor includes four core components:

Component	Description
Monitored container	This container is monitored for any insert or update operations. These changes are then reflected in the feed.
Lease container	The lease container serves as a storage mechanism to manage state across multiple change feed consumers (clients).
Host	The host is a client application instance that listens for and reacts to changes from the change feed.
Delegate	The delegate is code within the client application that will implement business logic for each batch of changes.

Prior to using the change feed processor, you should create a **lease** container that you will reference when configuring the processor.

Implement a delegate for the change feed processor

Completed100 XP

- 5 minutes

In C#, a delegate is a special type of variable or member that references a method with a specific parameter list and return type.

For the change feed processor, the library expects a delegate of type **ChangesHandler<>** that takes in a generic type to represent your serialized individual items. The delegate includes two parameters; a read-only list of changes and a cancellation token.

You could create a method named **HandleChangesAsync** with the same method signature as the delegate in a verbose syntax.

C#Copy
```
static async Task HandleChangesAsync(
    IReadOnlyCollection<Product> changes,
    CancellationToken cancellationToken
)
{
    // Do something with the batch of changes
}
```

Once that is created, you can create a new variable of type **ChangesHandler<>**, and then reference the earlier method.

C#Copy
```
ChangesHandler<Product> changeHandlerDelegate = HandleChangesAsync;
```

An even more concise syntax that accomplishes the same thing would use an anonymous function instead of a named method member.

C#Copy
```
ChangesHandler<Product> changeHandlerDelegate = async (
    IReadOnlyCollection<Product> changes,
    CancellationToken cancellationToken
) => {
    // Do something with the batch of changes
};
```

Within the delegate, you can iterate over the list of changes and then implement any business logic that makes sense for your application. In this example, you can think of each change as a “snapshot” of the item at some point in time that is delivered at-least-once to the host client application.

A foreach loop is used to iterate through the current batch of changes, and then each item is printed to the console window.

C#Copy

```
ChangesHandler<Product> changeHandlerDelegate = async (
    IReadOnlyCollection<Product> changes,
    CancellationToken cancellationToken
) => {
    foreach(Product product in changes)
    {
        await Console.Out.WriteLineAsync($"Detected
Operation:\t[{product.id}]\t{product.name}");
        // Do something with each change
    }
};
```

Implement the change feed processor

Completed100 XP

- 5 minutes

The change feed processor is created in a few steps:

1. Get the processor builder from the monitored (source) container variable
2. Use the builder to build-out the processor by specifying the delegate, processor name, lease container, and host instance name
3. Start the processor

First, you should create an instance of **Microsoft.Azure.Cosmos.Container** for both the source and lease container.

C#Copy

```
Container sourceContainer = client.GetContainer("cosmicworks", "products");

Container leaseContainer = client.GetContainer("cosmicworks", "productslease");
```

Next, you can use the **GetChangeFeedProcessorBuilder** method from a container instance to create a builder. At this point, you should specify the name of the processor and the delegate to handle each batch of changes.

C#Copy

```
var builder = sourceContainer.GetChangeFeedProcessorBuilder<Product>(
    processorName: "productItemProcessor",
    onChangesDelegate: changeHandlerDelegate
);
```

The builder ships with a series of fluent methods to configure the processor that includes, but is not limited to:

Method	Description
WithInstanceName	Name of host instance
WithStartTime	Set the pointer (in time) to start looking for changes after
WithLeaseContainer	Configures the lease container
WithErrorNotification	Assigns a delegate to handle errors during execution
WithMaxItems	Quantifies the max number of items in each batch
WithPollInterval	Sets the delay when the processor will poll the change feed for new changes

A simple example would be to configure the instance name using **WithInstanceName**, the lease container using **WithLeaseContainer**, and then **Build** the change feed processor.

C#Copy

```
ChangeFeedProcessor processor = builder
    .WithInstanceName("desktopApplication")
    .WithLeaseContainer(leaseContainer)
    .Build();
```

The resulting change feed processor includes both **StartAsync** and **StopAsync** methods to run and terminate the processor asynchronously.

C#Copy

```
await processor.StartAsync();

// Wait while processor handles items

await processor.StopAsync();
```

Implement the change feed estimator

Completed100 XP

- 4 minutes

As the change feed processor handles changes, it functions as a time-based pointer. The pointer moves forward in time across the change feed and sends batches of changes to the delegate to run business logic.

The change feed processor can potentially be constrained by the physical resources of the host application. If so, it's almost immediately assumed that the change feed processor should be scaled out across multiple hosts, all reading from the change feed concurrently.

However, identifying if your change feed solution needs to scale out can be difficult and requires an estimator feature. The change feed estimator is a sidecar feature to the processor that measures the number of changes that are pending to be read by the processor at any point in time.

To implement the estimator, you must first analyze the processor. In this example, a processor is created with a specific lease container and a delegate to handle changes.

C#Copy

```
Container sourceContainer = client.GetContainer("cosmicworks", "products");

Container leaseContainer = client.GetContainer("cosmicworks", "productslease");

ChangeFeedProcessor processor =
sourceContainer.GetChangeFeedProcessorBuilder<Product>(
    processorName: "productItemProcessor",
    onChangesDelegate: changeHandlerDelegate)
    .WithInstanceName("desktopApplication")
    .WithLeaseContainer(leaseContainer)
    .Build();
```

Next, you should implement a delegate to using the type **ChangesEstimationHandler** to handle each time the estimator polls the change feed to see how many changes have not been processed yet.

C#Copy

```
ChangesEstimationHandler changeEstimationDelegate = async (
    long estimation,
    CancellationToken cancellationToken
) => {
    // Do something with the estimation
};
```

Finally, you build the estimator in a manner similar to the processor reusing the same lease container.

C#Copy

```
ChangeFeedProcessor estimator = sourceContainer.GetChangeFeedEstimatorBuilder(
    processorName: "productItemEstimator",
    estimationDelegate: changeEstimationDelegate)
    .WithLeaseContainer(leaseContainer)
    .Build();
```

1.

Which method of the Container class is used to create a new change feed estimator?

○

`GetChangeFeedProcessorBuilder<>`

○

`GetChangeFeedEstimatorBuilder`

That's correct. The `GetChangeFeedEstimatorBuilder` method constructs an estimator that will run side by side with a processor.

○

`GetChangeFeedIterator<>`

2.

Which method of the ChangeFeedProcessor class should you invoke to start consuming changes from the change feed?

○

`GetChangeFeedProcessorBuilder<>`

○

`StartAsync`

That's correct. This method is a member of the ChangeFeedProcessor class and is invoked to start consuming changes from the change feed.

○

`Build`

Explore Azure Event Grid

Introduction

Completed100 XP

- 3 minutes

Azure Event Grid is deeply integrated with Azure services and can be integrated with third-party services. It simplifies event consumption and lowers costs by eliminating the need for constant polling. Event Grid efficiently and reliably routes events from Azure and non-Azure resources, and distributes the events to registered subscriber endpoints.

After completing this module, you'll be able to:

- Describe how Event Grid operates and how it connects to services and event handlers.
- Explain how Event Grid delivers events and how it handles errors.
- Implement authentication and authorization.
- Route custom events to web endpoint by using Azure CLI.

Explore Azure Event Grid

Completed100 XP

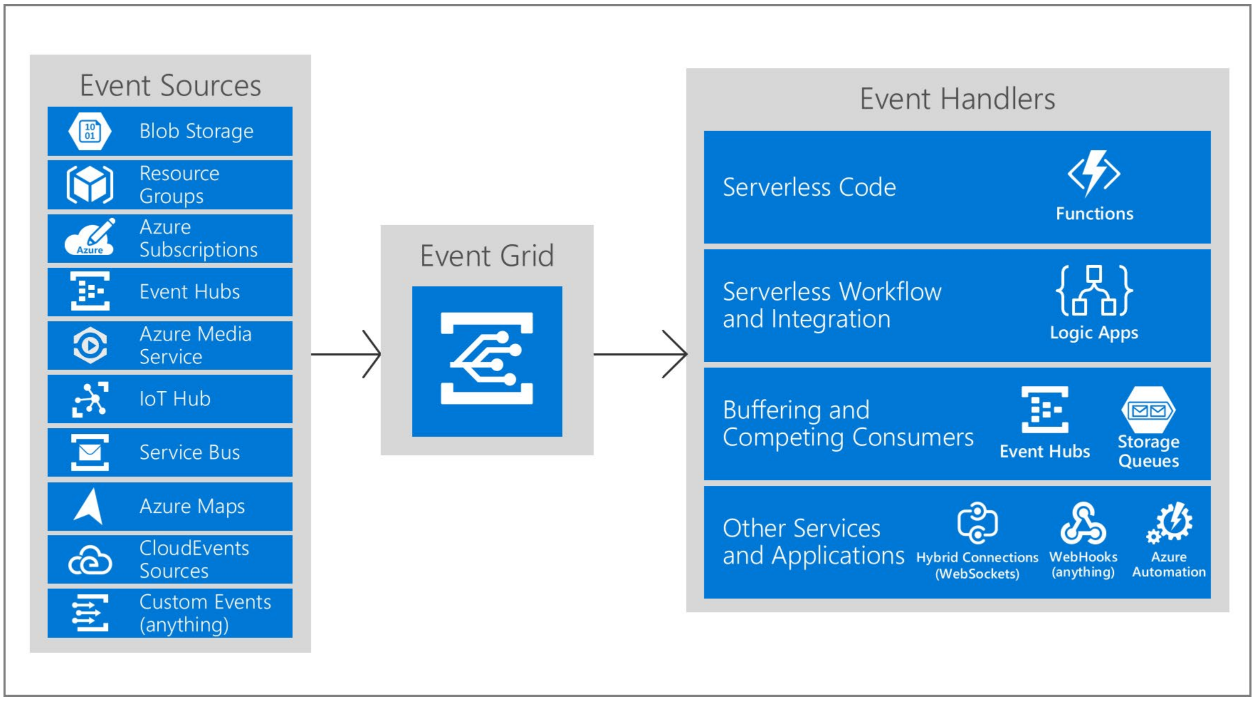
- 3 minutes

Azure Event Grid is a serverless event broker that you can use to integrate applications using events. Events are delivered by Event Grid to subscriber destinations such as applications, Azure services, or any endpoint to which Event Grid has network access. The source of those events can be other applications, SaaS services and Azure services. Publishers emit events, but have no expectation about how the events are handled. Subscribers decide on which events they want to handle.

Event Grid allows you to easily build applications with event-based architectures. Event Grid has built-in support for events coming from Azure services, like storage blobs and resource groups. Event Grid also has support for your own events, using custom topics.

You can use filters to route specific events to different endpoints, multicast to multiple endpoints, and make sure your events are reliably delivered.

The following image shows how Event Grid connects sources and handlers, and isn't a comprehensive list of supported integrations.



Concepts in Azure Event Grid

There are five concepts in Azure Event Grid you need to understand to help you get started:

- **Events** - What happened.
- **Event sources** - Where the event took place.
- **Topics** - The endpoint where publishers send events.
- **Event subscriptions** - The endpoint or built-in mechanism to route events, sometimes to more than one handler. Subscriptions are also used by handlers to intelligently filter incoming events.
- **Event handlers** - The app or service reacting to the event.

Events

An event is the smallest amount of information that fully describes something that happened in the system. Every event has common information like: source of the event, time the event took place, and unique identifier. Every event also has specific information that is only relevant to the specific type of event. For example, an event about a new file being created in Azure Storage has details about the file, such as the `lastTimeModified` value. Or, an Event Hubs event has the URL of the Capture file.

An event of size up to 64 KB is covered by General Availability (GA) Service Level Agreement (SLA). The support for an event of size up to 1 MB is currently in preview. Events over 64 KB are charged in 64-KB increments.

Event sources

An event source is where the event happens. Each event source is related to one or more event types. For example, Azure Storage is the event source for blob created events. IoT Hub is the event source for device created events. Your application is the event source for custom events that you define. Event sources are responsible for sending events to Event Grid.

Topics

The Event Grid topic provides an endpoint where the source sends events. The publisher creates the Event Grid topic, and decides whether an event source needs one topic or more than one topic. A topic is used for a collection of related events. To respond to certain types of events, subscribers decide which topics to subscribe to.

System topics are built-in topics provided by Azure services. You don't see system topics in your Azure subscription because the publisher owns the topics, but you can subscribe to them. To subscribe, you provide information about the resource you want to receive events from. As long as you have access to the resource, you can subscribe to its events.

Custom topics are application and third-party topics. When you create or are assigned access to a custom topic, you see that custom topic in your subscription.

Event subscriptions

A subscription tells Event Grid which events on a topic you're interested in receiving. When creating the subscription, you provide an endpoint for handling the event. You can filter the events that are sent to the endpoint. You can filter by event type, or subject pattern. Set an expiration for event subscriptions that are only needed for a limited time and you don't want to worry about cleaning up those subscriptions.

Event handlers

From an Event Grid perspective, an event handler is the place where the event is sent. The handler takes some further action to process the event. Event Grid supports several handler types. You can use a supported Azure service or your own webhook as the handler. Depending on the type of handler, Event Grid follows different mechanisms to guarantee the delivery of the event. For HTTP webhook event handlers, the event is retried until the handler returns a status code of `200 – OK`. For Azure Storage Queue, the events are retried until the Queue service successfully processes the message push into the queue.

Discover event schemas

Completed100 XP

- 3 minutes

Azure Event Grid supports two types of event schemas: Event Grid event schema and Cloud event schema. Events consist of a set of four required string properties. The properties are common to all events from any publisher.

The data object has properties that are specific to each publisher. For system topics, these properties are specific to the resource provider, such as Azure Storage or Azure Event Hubs.

Event sources send events to Azure Event Grid in an array, which can have several event objects. When posting events to an Event Grid topic, the array can have a total size of up to 1 MB. Each event in the array is limited to 1 MB. If an event or the array is greater than the size limits, you receive the response `413 Payload Too Large`. Operations are charged in 64 KB increments though. So, events over 64 KB incur operations charges as though they were multiple events. For example, an event that is 130 KB would incur charges as though it were three separate events.

Event Grid sends the events to subscribers in an array that has a single event. You can find the JSON schema for the Event Grid event and each Azure publisher's data payload in the Event Schema store.

Event schema

The following example shows the properties that are used by all event publishers:

JSONCopy

```
[
  {
    "topic": string,
    "subject": string,
    "id": string,
    "eventType": string,
    "eventTime": string,
    "data":{
      object-unique-to-each-publisher
    },
    "dataVersion": string,
    "metadataVersion": string
  }
]
```

Event properties

All events have the same following top-level data:

Property	**Type**	**Required**	**Description**
topic	string	No. If not included, Event Grid stamps onto the event. If included, it must match the Event Grid topic Azure Resource Manager ID exactly.	Full resource path to the ev isn't writeable. Event Grid p
subject	string	Yes	Publisher-defined path to t
eventType	string	Yes	One of the registered even source.
eventTime	string	Yes	The time the event is gener provider's UTC time.
id	string	Yes	Unique identifier for the ev
data	object	No	Event data specific to the r
dataVersion	string	No. If not included, it is stamped with an empty value.	The schema version of the publisher defines the schen
metadataVersion	string	No. If not included, Event Grid will stamp onto the event. If included, must match the Event Grid Schema `metadataVersion` exactly (currently, only `1`).	The schema version of the Grid defines the schema of properties. Event Grid prov

For custom topics, the event publisher determines the data object. The top-level data should have the same fields as standard resource-defined events.

When publishing events to custom topics, create subjects for your events that make it easy for subscribers to know whether they're interested in the event. Subscribers use the subject to filter and route events. Consider providing the path for where the event happened, so subscribers can filter by segments of that path. The path enables subscribers to narrowly or broadly filter events. For example, if you provide a three segment path like `/A/B/C` in the subject, subscribers can filter by the first segment `/A` to get a broad set of events. Those subscribers get events with subjects like `/A/B/C` or `/A/D/E`. Other subscribers can filter by `/A/B` to get a narrower set of events.

Sometimes your subject needs more detail about what happened. For example, the **Storage Accounts** publisher provides the subject `/blobServices/default/containers/<container-name>/blobs/<file>` when a file is added to a container. A subscriber could filter by the path `/blobServices/default/containers/testcontainer` to get all events for that container but not other containers in the storage account. A subscriber could also filter or route by the suffix `.txt` to only work with text files.

Cloud events schema

In addition to its default event schema, Azure Event Grid natively supports events in the JSON implementation of CloudEvents v1.0 and HTTP protocol binding. CloudEvents is an open specification for describing event data.

CloudEvents simplifies interoperability by providing a common event schema for publishing, and consuming cloud based events. This schema allows for uniform tooling, standard ways of routing & handling events, and universal ways of deserializing the outer event schema. With a common schema, you can more easily integrate work across platforms.

Here is an example of an Azure Blob Storage event in CloudEvents format:

JSONCopy

```
{
    "specversion": "1.0",
    "type": "Microsoft.Storage.BlobCreated",
    "source": "/subscriptions/{subscription-id}/resourceGroups/{resource-
group}/providers/Microsoft.Storage/storageAccounts/{storage-account}",
    "id": "9aeb0fdf-c01e-0131-0922-9eb54906e209",
    "time": "2019-11-18T15:13:39.4589254Z",
    "subject": "blobServices/default/containers/{storage-container}/blobs/{new-
file}",
    "dataschema": "#",
    "data": {
        "api": "PutBlockList",
        "clientRequestId": "4c5dd7fb-2c48-4a27-bb30-5361b5de920a",
        "requestId": "9aeb0fdf-c01e-0131-0922-9eb549000000",
        "eTag": "0x8D76C39E4407333",
        "contentType": "image/png",
        "contentLength": 30699,
        "blobType": "BlockBlob",
        "url": "https://gridtesting.blob.core.windows.net/testcontainer/{new-file}",
        "sequencer": "00000000000000000000000000099240000000000c41c18",
        "storageDiagnostics": {
            "batchId": "681fe319-3006-00a8-0022-9e7cde000000"
        }
    }
}
```

A detailed description of the available fields, their types, and definitions in CloudEvents v1.0 is available here.

The headers values for events delivered in the CloudEvents schema and the Event Grid schema are the same except for `content-type`. For CloudEvents schema, that header value is `"content-type":"application/cloudevents+json; charset=utf-8"`. For Event Grid schema, that header value is `"content-type":"application/json; charset=utf-8"`.

You can use Event Grid for both input and output of events in CloudEvents schema. You can use CloudEvents for system events, like Blob Storage events and IoT Hub events, and custom events. It can also transform those events on the wire back and forth.

Explore event delivery durability

Completed 100 XP

- 3 minutes

Event Grid provides durable delivery. It tries to deliver each event at least once for each matching subscription immediately. If a subscriber's endpoint doesn't acknowledge receipt of an event or if there's a failure, Event Grid retries delivery based on a fixed retry schedule and retry policy. By default, Event Grid delivers one event at a time to the subscriber, and the payload is an array with a single event.

Note

Event Grid doesn't guarantee order for event delivery, so subscribers may receive them out of order.

Retry schedule

When Event Grid receives an error for an event delivery attempt, Event Grid decides whether it should retry the delivery, dead-letter the event, or drop the event based on the type of the error.

If the error returned by the subscribed endpoint is a configuration-related error that can't be fixed with retries (for example, if the endpoint is deleted), Event Grid will either dead-letter the event or drop the event if dead-letter isn't configured.

The following table describes the types of endpoints and errors for which retry doesn't happen:

Endpoint Type	Error codes
Azure Resources	400 Bad Request, 413 Request Entity Too Large, 403 Forbidden
Webhook	400 Bad Request, 413 Request Entity Too Large, 403 Forbidden, 404 Not Found, 401 Unauthorized

Important

If Dead-Letter isn't configured for an endpoint, events will be dropped when the above errors happen. Consider configuring Dead-Letter if you don't want these kinds of events to be dropped.

If the error returned by the subscribed endpoint isn't among the above list, Event Grid waits 30 seconds for a response after delivering a message. After 30 seconds, if the endpoint hasn’t responded, the message is queued for retry. Event Grid uses an exponential backoff retry policy for event delivery.

If the endpoint responds within 3 minutes, Event Grid attempts to remove the event from the retry queue on a best effort basis but duplicates may still be received. Event Grid adds a small randomization to all retry steps and may opportunistically skip certain retries if an endpoint is consistently unhealthy, down for a long period, or appears to be overwhelmed.

Retry policy

You can customize the retry policy when creating an event subscription by using the following two configurations. An event is dropped if either of the limits of the retry policy is reached.

- **Maximum number of attempts** - The value must be an integer between 1 and 30. The default value is 30.
- **Event time-to-live (TTL)** - The value must be an integer between 1 and 1440. The default value is 1440 minutes

The following example shows setting the maximum number of attempts to 18 by using the Azure CLI.

BashCopy

```
az eventgrid event-subscription create \
  -g gridResourceGroup \
  --topic-name <topic_name> \
  --name <event_subscription_name> \
  --endpoint <endpoint_URL> \
  --max-delivery-attempts 18
```

Output batching

You can configure Event Grid to batch events for delivery for improved HTTP performance in high-throughput scenarios. Batching is turned off by default and can be turned on per-subscription via the portal, CLI, PowerShell, or SDKs.

Batched delivery has two settings:

- **Max events per batch** - Maximum number of events Event Grid delivers per batch. This number won't be exceeded, however fewer events may be delivered if no other events are available at the time of publish. Event Grid doesn't delay events to create a batch if fewer events are available. Must be between 1 and 5,000.
- **Preferred batch size in kilobytes** - Target ceiling for batch size in kilobytes. Similar to max events, the batch size may be smaller if more events aren't available at the time of publish. It's possible that a batch is larger than the preferred batch size *if* a single event is larger than the preferred size. For example, if the preferred size is 4 KB and a 10-KB event is pushed to Event Grid, the 10-KB event will still be delivered in its own batch rather than being dropped.

Delayed delivery

As an endpoint experiences delivery failures, Event Grid begins to delay the delivery and retry of events to that endpoint. For example, if the first 10 events published to an endpoint fail, Event Grid assumes that the endpoint is experiencing issues and will delay all subsequent retries, and new deliveries, for some time - in some cases up to several hours.

The functional purpose of delayed delivery is to protect unhealthy endpoints and the Event Grid system. Without back-off and delay of delivery to unhealthy endpoints, Event Grid's retry policy and volume capabilities can easily overwhelm a system.

Dead-letter events

When Event Grid can't deliver an event within a certain time period or after trying to deliver the event a number of times, it can send the undelivered event to a storage account. This process is known as **dead-lettering**. Event Grid dead-letters an event when **one of the following** conditions is met.

- Event isn't delivered within the **time-to-live** period.
- The **number of tries** to deliver the event exceeds the limit.

If either of the conditions is met, the event is dropped or dead-lettered. By default, Event Grid doesn't turn on dead-lettering. To enable it, you must specify a storage account to hold undelivered events when creating the event subscription. You pull events from this storage account to resolve deliveries.

If Event Grid receives a 400 (Bad Request) or 413 (Request Entity Too Large) response code, it immediately schedules the event for dead-lettering. These response codes indicate delivery of the event will never succeed.

There's a five-minute delay between the last attempt to deliver an event and when it's delivered to the dead-letter location. This delay is intended to reduce the number of Blob storage operations. If the dead-letter location is unavailable for four hours, the event is dropped.

Custom delivery properties

Event subscriptions allow you to set up HTTP headers that are included in delivered events. This capability allows you to set custom headers that are required by a destination. You can set up to 10 headers when creating an event subscription. Each header value shouldn't be greater than 4,096 bytes. You can set custom headers on the events that are delivered to the following destinations:

- Webhooks
- Azure Service Bus topics and queues
- Azure Event Hubs
- Relay Hybrid Connections

Before setting the dead-letter location, you must have a storage account with a container. You provide the endpoint for this container when creating the event subscription.

Control access to events

Completed100 XP

- 3 minutes

Azure Event Grid allows you to control the level of access given to different users to do various management operations such as list event subscriptions, create new ones, and generate keys. Event Grid uses Azure role-based access control (Azure RBAC).

Built-in roles

Event Grid provides the following built-in roles:

Role	**Description**
Event Grid Subscription Reader	Lets you read Event Grid event subscriptions.
Event Grid Subscription Contributor	Lets you manage Event Grid event subscription operations.
Event Grid Contributor	Lets you create and manage Event Grid resources.
Event Grid Data Sender	Lets you send events to Event Grid topics.

The Event Grid Subscription Reader and Event Grid Subscription Contributor roles are for managing event subscriptions. They're important when implementing event domains because they give users the permissions they need to subscribe to topics in your event domain. These roles are focused on event subscriptions and don't grant access for actions such as creating topics.

The Event Grid Contributor role allows you to create and manage Event Grid resources.

Permissions for event subscriptions

If you're using an event handler that isn't a WebHook (such as an event hub or queue storage), you need write access to that resource. This permissions check prevents an unauthorized user from sending events to your resource.

You must have the **Microsoft.EventGrid/EventSubscriptions/Write** permission on the resource that is the event source. You need this permission because you're writing a new subscription at the scope of the resource. The required resource differs based on whether you're subscribing to a system topic or custom topic. Both types are described in this section.

Topic Type	**Description**
System topics	Need permission to write a new event subscription at the scope of the resource publishing the event. The form is: `/subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/{resource-provide type}/{resource-name}`
Custom topics	Need permission to write a new event subscription at the scope of the event grid topic. The format of the resou is: `/subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/Microsoft.EventGr name}`

Receive events by using webhooks

Completed100 XP

- 3 minutes

Webhooks are one of the many ways to receive events from Azure Event Grid. When a new event is ready, Event Grid service POSTs an HTTP request to the configured endpoint with the event in the request body.

Like many other services that support webhooks, Event Grid requires you to prove ownership of your Webhook endpoint before it starts delivering events to that endpoint. This requirement prevents a malicious user from flooding your endpoint with events.

When you use any of the following three Azure services, the Azure infrastructure automatically handles this validation:

- Azure Logic Apps with Event Grid Connector
- Azure Automation via webhook
- Azure Functions with Event Grid Trigger

Endpoint validation with Event Grid events

If you're using any other type of endpoint, such as an HTTP trigger based Azure function, your endpoint code needs to participate in a validation handshake with Event Grid. Event Grid supports two ways of validating the subscription.

- **Synchronous handshake**: At the time of event subscription creation, Event Grid sends a subscription validation event to your endpoint. The schema of this event is similar to any other Event Grid event. The data portion of this event includes a `validationCode` property. Your application verifies that the validation request is for an expected event subscription, and returns the validation code in the response synchronously. This handshake mechanism is supported in all Event Grid versions.
- **Asynchronous handshake**: In certain cases, you can't return the ValidationCode in response synchronously. For example, if you use a third-party service (like Zapier or IFTTT), you can't programmatically respond with the validation code.

Starting with version 2018-05-01-preview, Event Grid supports a manual validation handshake. If you're creating an event subscription with an SDK or tool that uses API version 2018-05-01-preview or later, Event Grid sends a `validationUrl` property in the data portion of the subscription validation event. To complete the handshake, find that URL in the event data and do a GET request to it. You can use either a REST client or your web browser.

The provided URL is valid for **5 minutes**. During that time, the provisioning state of the event subscription is `AwaitingManualAction`. If you don't complete the manual validation within 5 minutes, the provisioning state is set to `Failed`. You have to create the event subscription again before starting the manual validation.

This authentication mechanism also requires the webhook endpoint to return an HTTP status code of 200 so that it knows that the POST for the validation event was accepted before it can be put in the manual validation mode. In other words, if the endpoint returns 200 but doesn't return back a validation response synchronously, the mode is transitioned to the manual validation mode. If there's a GET on the validation URL within 5 minutes, the validation handshake is considered to be successful.

Note

Using self-signed certificates for validation isn't supported. Use a signed certificate from a commercial certificate authority (CA) instead.

Filter events

Completed100 XP

- 3 minutes

When creating an event subscription, you have three options for filtering:

- Event types
- Subject begins with or ends with
- Advanced fields and operators

Event type filtering

By default, all event types for the event source are sent to the endpoint. You can decide to send only certain event types to your endpoint. For example, you can get notified of updates to your resources, but not notified for other operations like deletions. In that case, filter by the `Microsoft.Resources.ResourceWriteSuccess` event type. Provide an array with the event types, or specify `All` to get all event types for the event source.

The JSON syntax for filtering by event type is:

JSONCopy

```json
"filter": {
  "includedEventTypes": [
    "Microsoft.Resources.ResourceWriteFailure",
    "Microsoft.Resources.ResourceWriteSuccess"
  ]
}
```

Subject filtering

For simple filtering by subject, specify a starting or ending value for the subject. For example, you can specify the subject ends with `.txt` to only get events related to uploading a text file to storage account. Or, you can filter the subject begins with `/blobServices/default/containers/testcontainer` to get all events for that container but not other containers in the storage account.

The JSON syntax for filtering by subject is:

JSONCopy

```json
"filter": {
  "subjectBeginsWith": "/blobServices/default/containers/mycontainer/log",
  "subjectEndsWith": ".jpg"
}
```

Advanced filtering

To filter by values in the data fields and specify the comparison operator, use the advanced filtering option. In advanced filtering, you specify the:

- operator type - The type of comparison.
- key - The field in the event data that you're using for filtering. It can be a number, boolean, or string.
- value or values - The value or values to compare to the key.

The JSON syntax for using advanced filters is:

JSONCopy

```
"filter": {
  "advancedFilters": [
    {
      "operatorType": "NumberGreaterThanOrEquals",
      "key": "Data.Key1",
      "value": 5
    },
    {
      "operatorType": "StringContains",
      "key": "Subject",
      "values": ["container1", "container2"]
    }
  ]
}
```

Exercise - Route custom events to web endpoint by using Azure CLI

Completed100 XP

- 3 minutes

In this exercise you learn how to:

- Enable an Event Grid resource provider
- Create a custom topic
- Create a message endpoint
- Subscribe to a custom topic
- Send an event to a custom topic

Prerequisites

- An **Azure account** with an active subscription. If you don't already have one, you can sign up for a free trial at https://azure.com/free.

Create a resource group

In this section, you open your terminal and create some variables that are used throughout the rest of the exercise to make command entry, and unique resource name creation, a bit easier.

1. Launch the Cloud Shell: https://shell.azure.com
2. Select **Bash** as the shell.
3. Run the following commands to create the variables. Replace `<myLocation>` with a region near you.

   BashCopy

   ```
   let rNum=$RANDOM*$RANDOM
   myLocation=<myLocation>
   myTopicName="az204-egtopic-${rNum}"
   mySiteName="az204-egsite-${rNum}"
   mySiteURL="https://${mySiteName}.azurewebsites.net"
   ```

4. Create a resource group for the new resources you're creating.

   BashCopy

   ```
   az group create --name az204-evgrid-rg --location $myLocation
   ```

Enable an Event Grid resource provider

Note

This step is only needed on subscriptions that haven't previously used Event Grid.

Register the Event Grid resource provider by using the `az provider register` command.

BashCopy

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

It can take a few minutes for the registration to complete. To check the status run the following command.

BashCopy

```
az provider show --namespace Microsoft.EventGrid --query "registrationState"
```

Create a custom topic

Create a custom topic by using the `az eventgrid topic create` command. The name must be unique because it's part of the DNS entry.

BashCopy

```
az eventgrid topic create --name $myTopicName \
    --location $myLocation \
    --resource-group az204-evgrid-rg
```

Create a message endpoint

Before subscribing to the custom topic, we need to create the endpoint for the event message. Typically, the endpoint takes actions based on the event data. The following script uses a prebuilt web app that displays the event messages. The deployed solution includes an App Service plan, an App Service web app, and source code from GitHub. It also generates a unique name for the site.

1. Create a message endpoint. The deployment may take a few minutes to complete.

   BashCopy

   ```
   az deployment group create \
       --resource-group az204-evgrid-rg \
       --template-uri "https://raw.githubusercontent.com/Azure-
   Samples/azure-event-grid-viewer/main/azuredeploy.json" \
       --parameters siteName=$mySiteName hostingPlanName=viewerhost

   echo "Your web app URL: ${mySiteURL}"
   ```

   Note

   This command may take a few minutes to complete.

2. In a new tab, navigate to the URL generated at the end of the previous script to ensure the web app is running. You should see the site with no messages currently displayed.

   Tip

Leave the browser running, it is used to show updates.

Subscribe to a custom topic

You subscribe to an Event Grid topic to tell Event Grid which events you want to track and where to send those events.

1. Subscribe to a custom topic by using the `az eventgrid event-subscription create` command. The following script grabs the needed subscription ID from your account and use in the creation of the event subscription.

   BashCopy

   ```
   endpoint="${mySiteURL}/api/updates"
   subId=$(az account show --subscription "" | jq -r '.id')

   az eventgrid event-subscription create \
       --source-resource-id "/subscriptions/$subId/resourceGroups/az204-evgrid-rg/providers/Microsoft.EventGrid/topics/$myTopicName" \
       --name az204ViewerSub \
       --endpoint $endpoint
   ```

2. View your web app again, and notice that a subscription validation event has been sent to it. Select the eye icon to expand the event data. Event Grid sends the validation event so the endpoint can verify that it wants to receive event data. The web app includes code to validate the subscription.

Send an event to your custom topic

Trigger an event to see how Event Grid distributes the message to your endpoint.

1. Retrieve URL and key for the custom topic.

   BashCopy

   ```
   topicEndpoint=$(az eventgrid topic show --name $myTopicName -g az204-evgrid-rg --query "endpoint" --output tsv)
   key=$(az eventgrid topic key list --name $myTopicName -g az204-evgrid-rg --query "key1" --output tsv)
   ```

2. Create event data to send. Typically, an application or Azure service would send the event data, we're creating data for the purposes of the exercise.

   BashCopy

```
event='[ {"id": "'"$RANDOM"'", "eventType": "recordInserted", "subject":
"myapp/vehicles/motorcycles", "eventTime": "'`date +%Y-%m-
%dT%H:%M:%S%z`'", "data":{ "make": "Contoso", "model":
"Monster"},"dataVersion": "1.0"} ]'
```

3. Use `curl` to send the event to the topic.

   BashCopy

   ```
   curl -X POST -H "aeg-sas-key: $key" -d "$event" $topicEndpoint
   ```

4. View your web app to see the event you just sent. Select the eye icon to expand the event data.

   JSONCopy

   ```
   {
   "id": "29078",
   "eventType": "recordInserted",
   "subject": "myapp/vehicles/motorcycles",
   "eventTime": "2019-12-02T22:23:03+00:00",
   "data": {
       "make": "Contoso",
       "model": "Northwind"
   },
   "dataVersion": "1.0",
   "metadataVersion": "1",
   "topic": "/subscriptions/{subscription-id}/resourceGroups/az204-evgrid-
   rg/providers/Microsoft.EventGrid/topics/az204-egtopic-589377852"
   }
   ```

Clean up resources

When you no longer need the resources in this exercise use the following command to delete the resource group and associated resources.

BashCopy

```
az group delete --name az204-evgrid-rg --no-wait
```

Explore Azure Event Hubs

Introduction

Completed100 XP

- 3 minutes

Azure Event Hubs is a big data streaming platform and event ingestion service. It can receive and process millions of events per second. Data sent to an event hub can be transformed and stored by using any real-time analytics provider or batching/storage adapters.

After completing this module, you'll be able to:

- Describe the benefits of using Event Hubs and how it captures streaming data.
- Explain how to process events.
- Perform common operations with the Event Hubs client library.

Discover Azure Event Hubs

Completed100 XP

- 3 minutes

Azure Event Hubs represents the "front door" for an event pipeline, often called an *event ingestor* in solution architectures. An event ingestor is a component or service that sits between event publishers and event consumers to decouple the production of an event stream from the consumption of those events. Event Hubs provides a unified streaming platform with time retention buffer, decoupling event producers from event consumers.

The following table highlights key features of the Azure Event Hubs service:

Feature	**Description**
Fully managed PaaS	Event Hubs is a fully managed Platform-as-a-Service (PaaS) with little configuration or management ov on your business solutions. Event Hubs for Apache Kafka ecosystems give you the PaaS Kafka experien manage, configure, or run your clusters.
Real-time and batch processing	Event Hubs uses a partitioned consumer model, enabling multiple applications to process the stream you control the speed of processing.

Feature	Description
Capture event data	Capture your data in near-real time in Azure Blob storage or Azure Data Lake Storage for long-term ret processing.
Scalable	Scaling options, like Auto-inflate, scale the number of throughput units to meet your usage needs.
Rich ecosystem	Event Hubs for Apache Kafka ecosystems enables Apache Kafka (1.0 and later) clients and applications You don't need to set up, configure, and manage your own Kafka clusters.

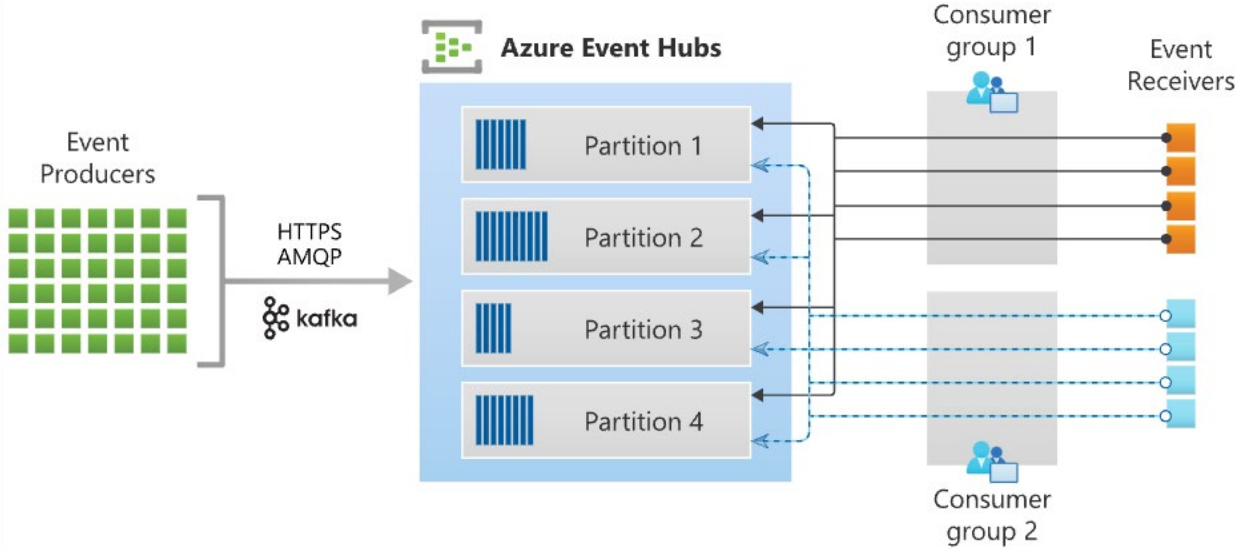
Key concepts

Event Hubs contains the following key components:

- An **Event Hubs client** is the primary interface for developers interacting with the Event Hubs client library. There are several different Event Hubs clients, each dedicated to a specific use of Event Hubs, such as publishing or consuming events.
- An **Event Hubs producer** is a type of client that serves as a source of telemetry data, diagnostics information, usage logs, or other log data, as part of an embedded device solution, a mobile device application, a game title running on a console or other device, some client or server based business solution, or a web site.
- An **Event Hubs consumer** is a type of client that reads information from the Event Hubs and allows processing of it. Processing may involve aggregation, complex computation and filtering. Processing may also involve distribution or storage of the information in a raw or transformed fashion. Event Hubs consumers are often robust and high-scale platform infrastructure parts with built-in analytics capabilities, like Azure Stream Analytics, Apache Spark.
- A **partition** is an ordered sequence of events that is held in an Event Hubs. Partitions are a means of data organization associated with the parallelism required by event consumers. Azure Event Hubs provides message streaming through a partitioned consumer pattern in which each consumer only reads a specific subset, or partition, of the message stream. As newer events arrive, they're added to the end of this sequence. The number of partitions is specified at the time an Event Hubs is created and can't be changed.
- A **consumer group** is a view of an entire Event Hubs. Consumer groups enable multiple consuming applications to each have a separate view of the event stream, and to read the stream independently at their own pace and from their own position. There can be at most five concurrent readers on a partition per consumer group; however it's recommended that there's only one active consumer for a given partition and consumer group pairing. Each active reader receives all of the events from its partition; if there are multiple readers on the same partition, then they'll receive duplicate events.

- **Event receivers**: Any entity that reads event data from an Event Hubs. All Event Hubs consumers connect via the AMQP 1.0 session. The Event Hubs service delivers events through a session as they become available. All Kafka consumers connect via the Kafka protocol 1.0 and later.
- **Throughput units** or **processing units**: Prepurchased units of capacity that control the throughput capacity of Event Hubs.

The following figure shows the Event Hubs stream processing architecture:

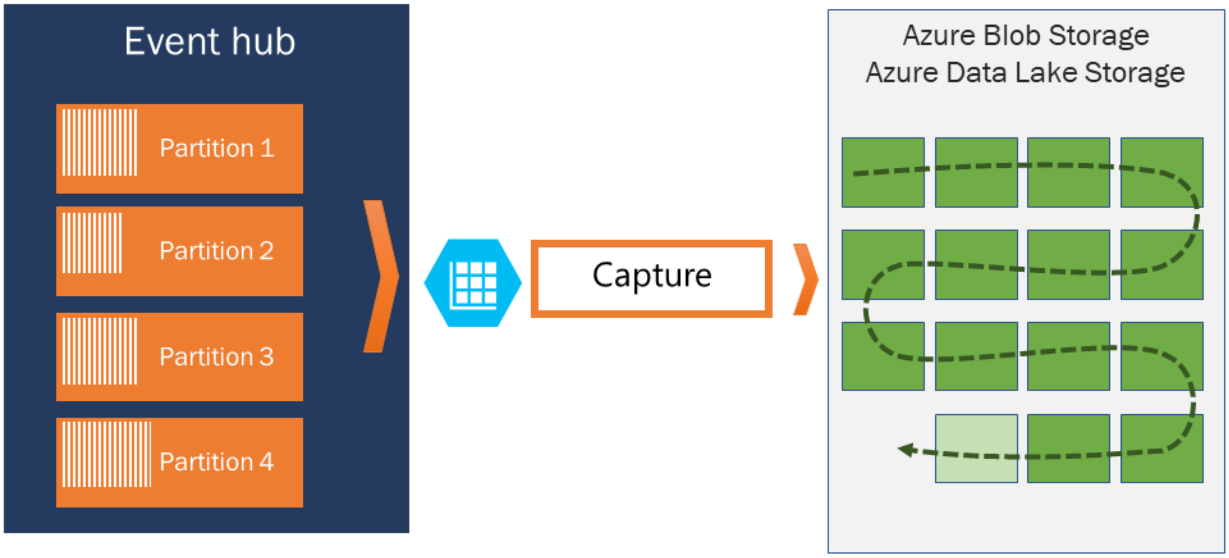


Explore Event Hubs Capture

Completed100 XP

- 3 minutes

Azure Event Hubs enables you to automatically capture the streaming data in Event Hubs in an Azure Blob storage or Azure Data Lake Storage account of your choice, with the added flexibility of specifying a time or size interval. Setting up Capture is fast, there are no administrative costs to run it, and it scales automatically with Event Hubs throughput units in the standard tier or processing units in the premium tier.



Event Hubs Capture enables you to process real-time and batch-based pipelines on the same stream. This means you can build solutions that grow with your needs over time.

How Event Hubs Capture works

Event Hubs is a time-retention durable buffer for telemetry ingress, similar to a distributed log. The key to scaling in Event Hubs is the partitioned consumer model. Each partition is an independent segment of data and is consumed independently. Over time this data ages off, based on the configurable retention period. As a result, a given event hub never gets "too full."

Event Hubs Capture enables you to specify your own Azure Blob storage account and container, or Azure Data Lake Store account, which are used to store the captured data. These accounts can be in the same region as your event hub or in another region, adding to the flexibility of the Event Hubs Capture feature.

Captured data is written in Apache Avro format: a compact, fast, binary format that provides rich data structures with inline schema. This format is widely used in the Hadoop ecosystem, Stream Analytics, and Azure Data Factory. More information about working with Avro is available later in this article.

Capture windowing

Event Hubs Capture enables you to set up a window to control capturing. This window is a minimum size and time configuration with a "first wins policy," meaning that the first trigger encountered causes a capture operation. Each partition captures independently and writes a completed block blob at the time of capture, named for the time at which the capture interval was encountered. The storage naming convention is as follows:

Copy

```
{Namespace}/{EventHub}/{PartitionId}/{Year}/{Month}/{Day}/{Hour}/{Minute}/{Second}
```

Note the date values are padded with zeroes; an example filename might be:

Copy

```
https://mystorageaccount.blob.core.windows.net/mycontainer/mynamespace/myeventhub/0/2
017/12/08/03/03/17.avro
```

Scaling to throughput units

Event Hubs traffic is controlled by throughput units. A single throughput unit allows 1 MB per second or 1000 events per second of ingress and twice that amount of egress. Standard Event Hubs can be configured with 1-20 throughput units, and you can purchase more with a quota increase support request. Usage beyond your purchased throughput units is throttled. Event Hubs Capture copies data directly from the internal Event Hubs storage, bypassing throughput unit egress quotas and saving your egress for other processing readers, such as Stream Analytics or Spark.

Once configured, Event Hubs Capture runs automatically when you send your first event, and continues running. To make it easier for your downstream processing to know that the process is working, Event Hubs writes empty files when there's no data. This process provides a predictable cadence and marker that can feed your batch processors.

Scale your processing application

Completed100 XP

- 3 minutes

To scale your event processing application, you can run multiple instances of the application and have it balance the load among themselves. In the older versions, **EventProcessorHost** allowed you to balance the load between multiple instances of your program and checkpoint events when receiving. In the newer versions (5.0 onwards), **EventProcessorClient** (.NET and Java), or **EventHubConsumerClient** (Python and JavaScript) allows you to do the same.

Note

The key to scale for Event Hubs is the idea of partitioned consumers. In contrast to the competing consumers pattern, the partitioned consumer pattern enables high scale by removing the contention bottleneck and facilitating end to end parallelism.

Example scenario

As an example scenario, consider a home security company that monitors 100,000 homes. Every minute, it gets data from various sensors such as a motion detector, door/window open sensor, glass break detector, and so on, installed in each home. The company provides a web site for residents to monitor the activity of their home in near real time.

Each sensor pushes data to an event hub. The event hub is configured with 16 partitions. On the consuming end, you need a mechanism that can read these events, consolidate them, and dump the aggregate to a storage blob, which is then projected to a user-friendly web page.

When designing the consumer in a distributed environment, the scenario must handle the following requirements:

- **Scale:** Create multiple consumers, with each consumer taking ownership of reading from a few Event Hubs partitions.
- **Load balance:** Increase or reduce the consumers dynamically. For example, when a new sensor type (for example, a carbon monoxide detector) is added to each home, the number of events increases. In that case, the operator (a human) increases the number of consumer instances. Then, the pool of consumers can rebalance the number of partitions they own, to share the load with the newly added consumers.
- **Seamless resume on failures:** If a consumer (**consumer A**) fails (for example, the virtual machine hosting the consumer suddenly crashes), then other consumers can pick up the partitions owned by **consumer A** and continue. Also, the continuation point, called a *checkpoint* or *offset*, should be at the exact point at which **consumer A** failed, or slightly before that.
- **Consume events:** While the previous three points deal with the management of the consumer, there must be code to consume the events and do something useful with it. For example, aggregate it and upload it to blob storage.

Event processor or consumer client

You don't need to build your own solution to meet these requirements. The Azure Event Hubs SDKs provide this functionality. In .NET or Java SDKs, you use an event processor client (`EventProcessorClient`), and in Python and JavaScript SDKs, you use `EventHubConsumerClient`.

For most production scenarios, we recommend that you use the event processor client for reading and processing events. Event processor clients can work cooperatively within the context of a consumer group for a given event hub. Clients will automatically manage distribution and balancing of work as instances become available or unavailable for the group.

Partition ownership tracking

An event processor instance typically owns and processes events from one or more partitions. Ownership of partitions is evenly distributed among all the active event processor instances associated with an event hub and consumer group combination.

Each event processor is given a unique identifier and claims ownership of partitions by adding or updating an entry in a checkpoint store. All event processor instances communicate with this store periodically to update its own processing state and to learn about other active instances. This data is then used to balance the load among the active processors.

Receive messages

When you create an event processor, you specify the functions that process events and errors. Each call to the function that processes events delivers a single event from a specific partition. It's your responsibility to handle this event. If you want to make sure the consumer processes every message at least once, you need to write your own code with retry logic. But be cautious about poisoned messages.

We recommend that you do things relatively fast. That is, do as little processing as possible. If you need to write to storage and do some routing, it's better to use two consumer groups and have two event processors.

Checkpointing

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 and occurs on a per-partition basis within a consumer group.

If an event processor disconnects from a partition, another instance can resume processing the partition at the checkpoint that was previously committed by the last processor of that partition in that consumer group. When the processor connects, it passes the offset to the event hub to specify the location at which to start reading. In this way, you can use checkpointing to both mark events as "complete" by downstream applications and to provide resiliency when an event processor goes down. It's possible to return to older data by specifying a lower offset from this checkpointing process.

Thread safety and processor instances

By default, the function that processes the events is called sequentially for a given partition. Subsequent events and calls to this function from the same partition queue up behind the scenes as the event pump continues to run in the background on other threads. Events from different partitions can be processed concurrently and any shared state that is accessed across partitions have to be synchronized.

Control access to events

Completed100 XP

- 3 minutes

Azure Event Hubs supports both Azure Active Directory and shared access signatures (SAS) to handle both authentication and authorization. Azure provides the following Azure built-in roles for authorizing access to Event Hubs data using Azure Active Directory and OAuth:

- Azure Event Hubs Data Owner: Use this role to give *complete access* to Event Hubs resources.
- Azure Event Hubs Data Sender: Use this role to give *send access* to Event Hubs resources.
- Azure Event Hubs Data Receiver: Use this role to give *receiving access* to Event Hubs resources.

Authorize access with managed identities

To authorize a request to Event Hubs service from a managed identity in your application, you need to configure Azure role-based access control settings for that managed identity. Azure Event Hubs defines Azure roles that encompass permissions for sending and reading from Event Hubs. When the Azure role is assigned to a managed identity, the managed identity is granted access to Event Hubs data at the appropriate scope.

Authorize access with Microsoft Identity Platform

A key advantage of using Azure AD with Event Hubs is that your credentials no longer need to be stored in your code. Instead, you can request an OAuth 2.0 access token from Microsoft identity platform. Azure AD authenticates the security principal (a user, a group, or service principal) running the application. If authentication succeeds, Azure AD returns the access token to the application, and the application can then use the access token to authorize requests to Azure Event Hubs.

Authorize access to Event Hubs publishers with shared access signatures

An event publisher defines a virtual endpoint for an Event Hubs. The publisher can only be used to send messages to an event hub and not receive messages. Typically, an event hub employs one publisher per client. All messages that are sent to any of the publishers of an event hub are enqueued within that event hub. Publishers enable fine-grained access control.

Each Event Hubs client is assigned a unique token that is uploaded to the client. A client that holds a token can only send to one publisher, and no other publisher. If multiple clients share the same token, then each of them shares the publisher.

All tokens are assigned with shared access signature keys. Typically, all tokens are signed with the same key. Clients aren't aware of the key, which prevents clients from manufacturing tokens. Clients operate on the same tokens until they expire.

Authorize access to Event Hubs consumers with shared access signatures

To authenticate back-end applications that consume from the data generated by Event Hubs producers, Event Hubs token authentication requires its clients to either have the **manage** rights or the **listen** privileges assigned to its Event Hubs namespace or event hub instance or topic. Data is consumed from Event Hubs using consumer groups. While SAS policy gives you granular scope, this scope is defined only at the entity level and not at the consumer level. It means that the privileges defined at the namespace level or the event hub instance or topic level are to the consumer groups of that entity.

Perform common operations with the Event Hubs client library

Completed100 XP

- 3 minutes

This unit contains examples of common operations you can perform with the Event Hubs client library (`Azure.Messaging.EventHubs`) to interact with an Event Hubs.

Inspect Event Hubs

Many Event Hubs operations take place within the scope of a specific partition. Because partitions are owned by the Event Hubs, their names are assigned at the time of creation. To understand what partitions are available, you query the Event Hubs using one of the Event Hubs clients. For illustration, the `EventHubProducerClient` is demonstrated in these examples, but the concept and form are common across clients.

C#Copy

```csharp
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

await using (var producer = new EventHubProducerClient(connectionString,
eventHubName))
{
    string[] partitionIds = await producer.GetPartitionIdsAsync();
}
```

Publish events to Event Hubs

In order to publish events, you need to create an `EventHubProducerClient`. Producers publish events in batches and may request a specific partition, or allow the Event Hubs service to decide which partition events should be published to. We recommended using automatic routing when the publishing of events needs to be highly available or when event data should be distributed evenly among the partitions. Our example takes advantage of automatic routing.

C#Copy

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

await using (var producer = new EventHubProducerClient(connectionString,
eventHubName))
{
    using EventDataBatch eventBatch = await producer.CreateBatchAsync();
    eventBatch.TryAdd(new EventData(new BinaryData("First")));
    eventBatch.TryAdd(new EventData(new BinaryData("Second")));

    await producer.SendAsync(eventBatch);
}
```

Read events from an Event Hubs

In order to read events from an Event Hubs, you need to create an `EventHubConsumerClient` for a given consumer group. When an Event Hubs is created, it provides a default consumer group that can be used to get started with exploring Event Hubs. In our example, we'll focus on reading all events that have been published to the Event Hubs using an iterator.

Note

It is important to note that this approach to consuming is intended to improve the experience of exploring the Event Hubs client library and prototyping. It is recommended that it not be used in production scenarios. For production use, we recommend using the **Event Processor Client**, as it provides a more robust and performant experience.

C#Copy

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;

await using (var consumer = new EventHubConsumerClient(consumerGroup,
connectionString, eventHubName))
{
    using var cancellationSource = new CancellationTokenSource();
    cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

    await foreach (PartitionEvent receivedEvent in
consumer.ReadEventsAsync(cancellationSource.Token))
    {
        // At this point, the loop will wait for events to be available in the Event
Hub.  When an event
        // is available, the loop will iterate with the event that was received.
Because we did not
        // specify a maximum wait time, the loop will wait forever unless
cancellation is requested using
        // the cancellation token.
    }
}
```

Read events from an Event Hubs partition

To read from a specific partition, the consumer needs to specify where in the event stream to begin receiving events; in our example, we focus on reading all published events for the first partition of the Event Hubs.

C#Copy

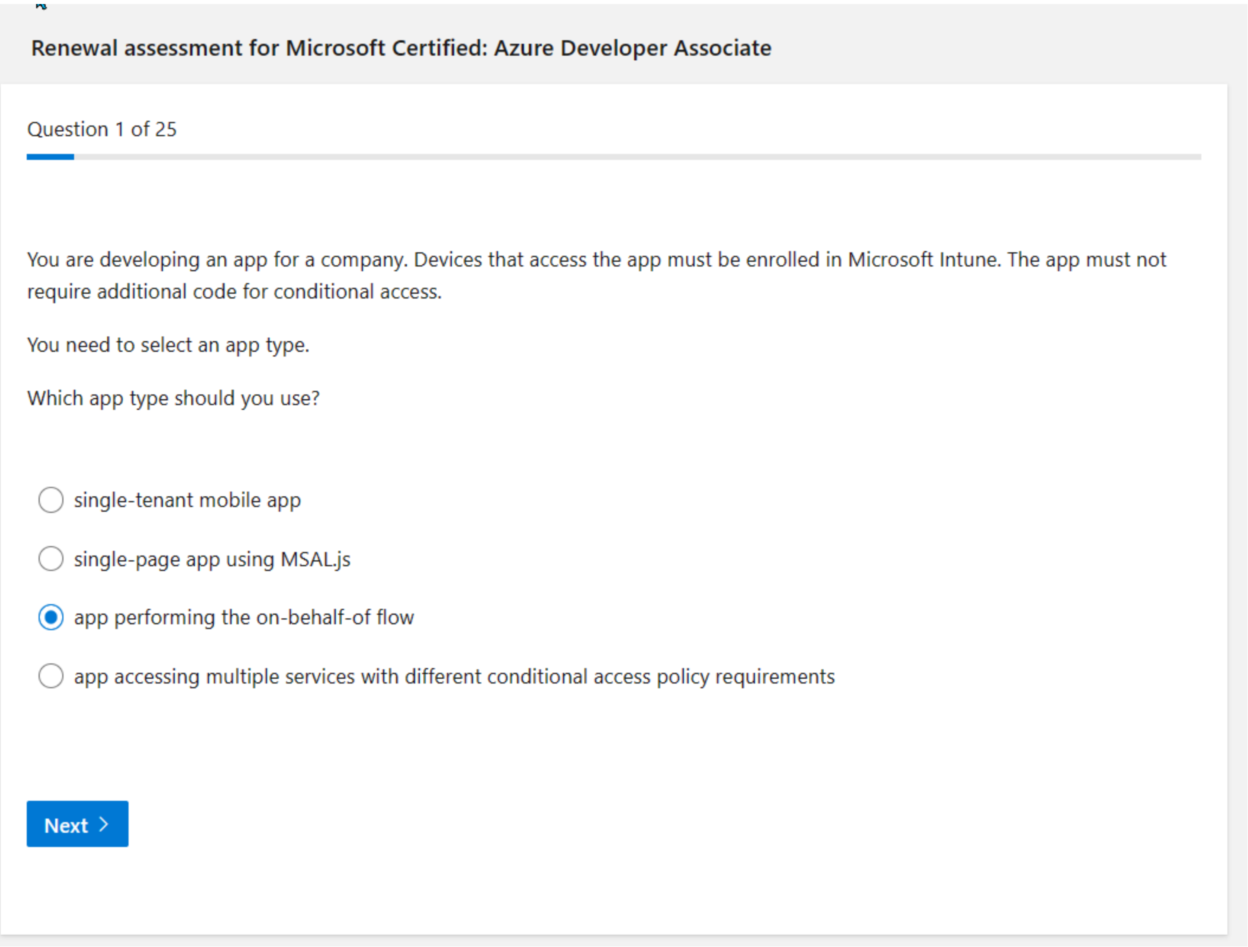
```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;

await using (var consumer = new EventHubConsumerClient(consumerGroup,
connectionString, eventHubName))
{
    EventPosition startingPosition = EventPosition.Earliest;
    string partitionId = (await consumer.GetPartitionIdsAsync()).First();

    using var cancellationSource = new CancellationTokenSource();
    cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

    await foreach (PartitionEvent receivedEvent in
consumer.ReadEventsFromPartitionAsync(partitionId, startingPosition,
cancellationSource.Token))
    {
        // At this point, the loop will wait for events to be available in the
partition.  When an event
        // is available, the loop will iterate with the event that was received.
Because we did not
        // specify a maximum wait time, the loop will wait forever unless
cancellation is requested using
        // the cancellation token.
    }
}
```

Process events using an Event Processor client

For most production scenarios, it's recommended that the `EventProcessorClient` be used for reading and processing events. Since the `EventProcessorClient` has a dependency on Azure Storage blobs for persistence of its state, you need to provide a `BlobContainerClient` for the processor, which has been configured for the storage account and container that should be used.

C#Copy

```csharp
var cancellationSource = new CancellationTokenSource();
cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

var storageConnectionString = "<< CONNECTION STRING FOR THE STORAGE ACCOUNT >>";
var blobContainerName = "<< NAME OF THE BLOB CONTAINER >>";

var eventHubsConnectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE
>>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";
var consumerGroup = "<< NAME OF THE EVENT HUB CONSUMER GROUP >>";

Task processEventHandler(ProcessEventArgs eventArgs) => Task.CompletedTask;
Task processErrorHandler(ProcessErrorEventArgs eventArgs) => Task.CompletedTask;

var storageClient = new BlobContainerClient(storageConnectionString,
blobContainerName);
var processor = new EventProcessorClient(storageClient, consumerGroup,
eventHubsConnectionString, eventHubName);

processor.ProcessEventAsync += processEventHandler;
processor.ProcessErrorAsync += processErrorHandler;

await processor.StartProcessingAsync();

try
{
    // The processor performs its work in the background; block until cancellation
    // to allow processing to take place.

    await Task.Delay(Timeout.Infinite, cancellationSource.Token);
}
catch (TaskCanceledException)
{
    // This is expected when the delay is canceled.
}

try
{
    await processor.StopProcessingAsync();
}
finally
{
    // To prevent leaks, the handlers should be removed when processing is complete.

    processor.ProcessEventAsync -= processEventHandler;
    processor.ProcessErrorAsync -= processErrorHandler;
}
```



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 1 of 25

You are developing an app for a company. Devices that access the app must be enrolled in Microsoft Intune. The app must not require additional code for conditional access.

You need to select an app type.

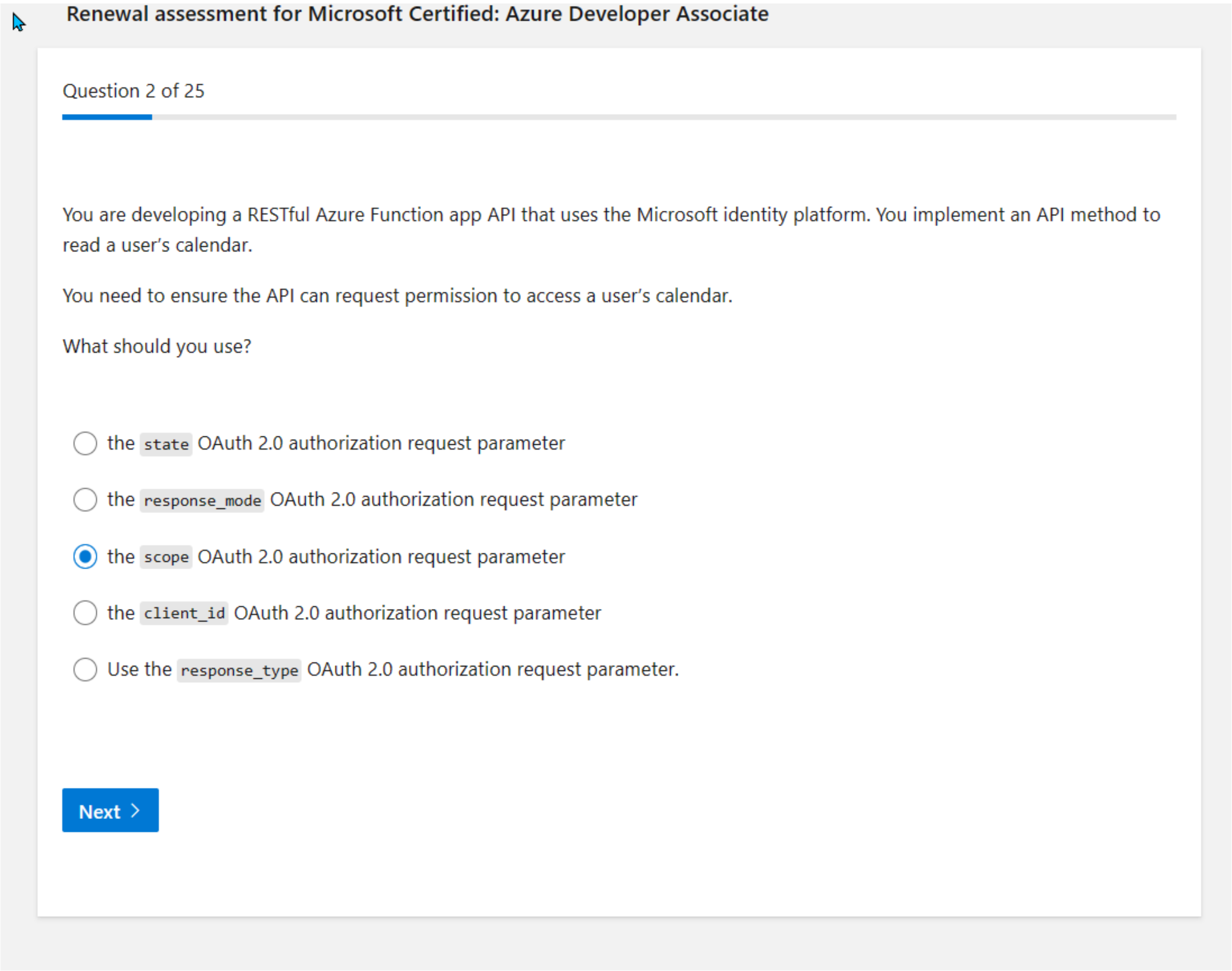
Which app type should you use?

○ single-tenant mobile app

○ single-page app using MSAL.js

● app performing the on-behalf-of flow

○ app accessing multiple services with different conditional access policy requirements



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 2 of 25

You are developing a RESTful Azure Function app API that uses the Microsoft identity platform. You implement an API method to read a user's calendar.

You need to ensure the API can request permission to access a user's calendar.

What should you use?

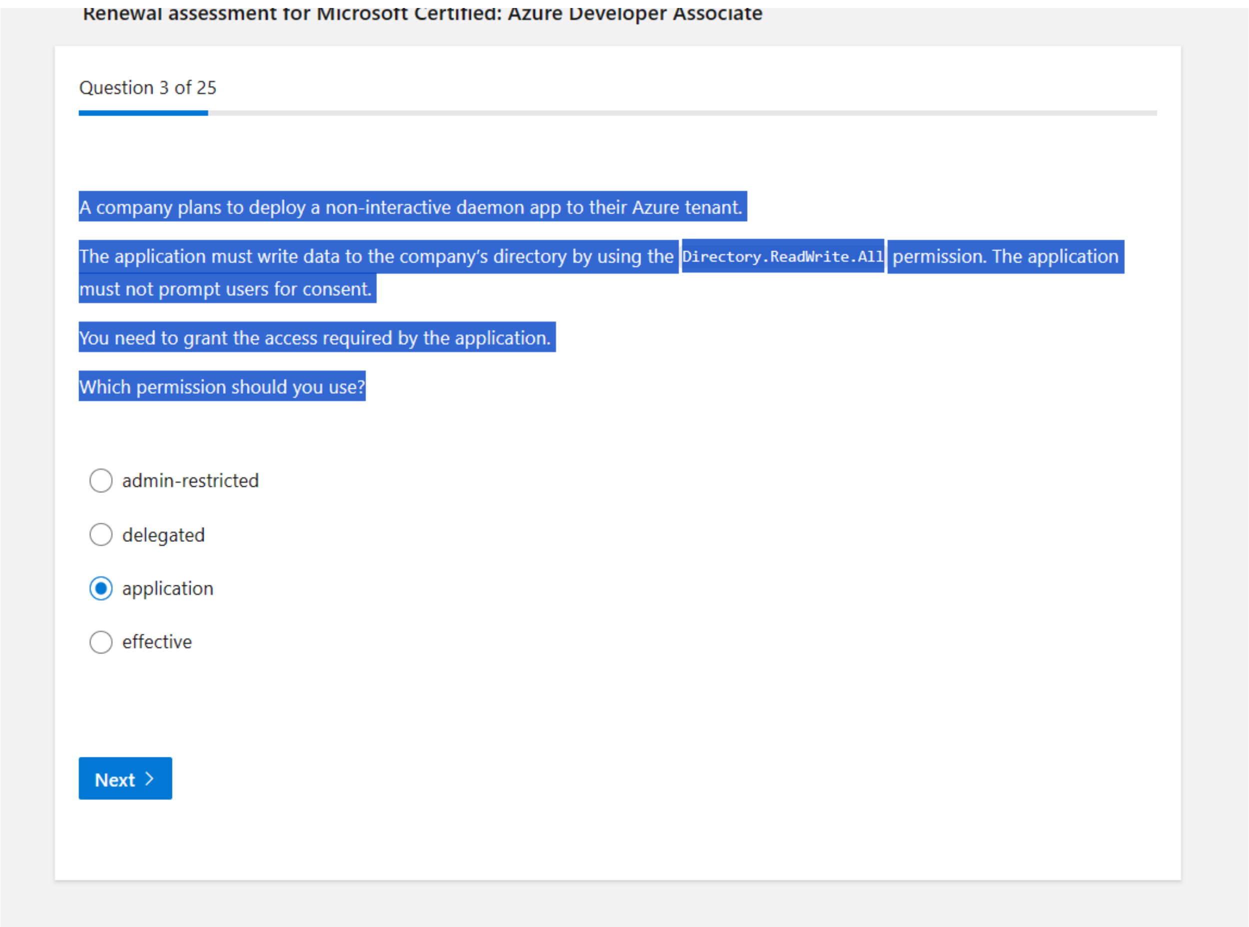
○ the `state` OAuth 2.0 authorization request parameter

○ the `response_mode` OAuth 2.0 authorization request parameter

● the `scope` OAuth 2.0 authorization request parameter

○ the `client_id` OAuth 2.0 authorization request parameter

○ Use the `response_type` OAuth 2.0 authorization request parameter.



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 3 of 25

A company plans to deploy a non-interactive daemon app to their Azure tenant.

The application must write data to the company's directory by using the `Directory.ReadWrite.All` permission. The application must not prompt users for consent.

You need to grant the access required by the application.

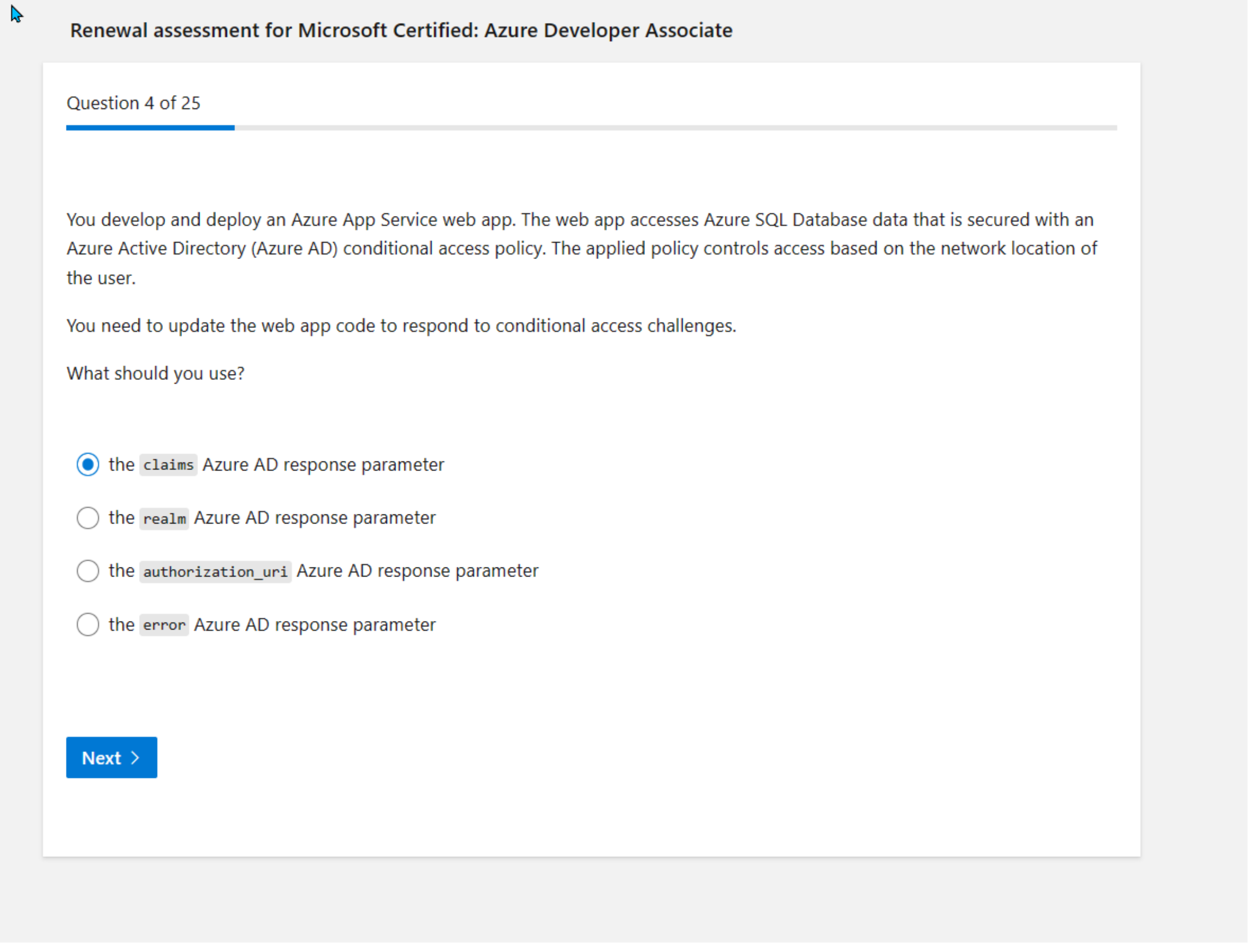
Which permission should you use?

○ admin-restricted

○ delegated

● application

○ effective



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 4 of 25

You develop and deploy an Azure App Service web app. The web app accesses Azure SQL Database data that is secured with an Azure Active Directory (Azure AD) conditional access policy. The applied policy controls access based on the network location of the user.

You need to update the web app code to respond to conditional access challenges.

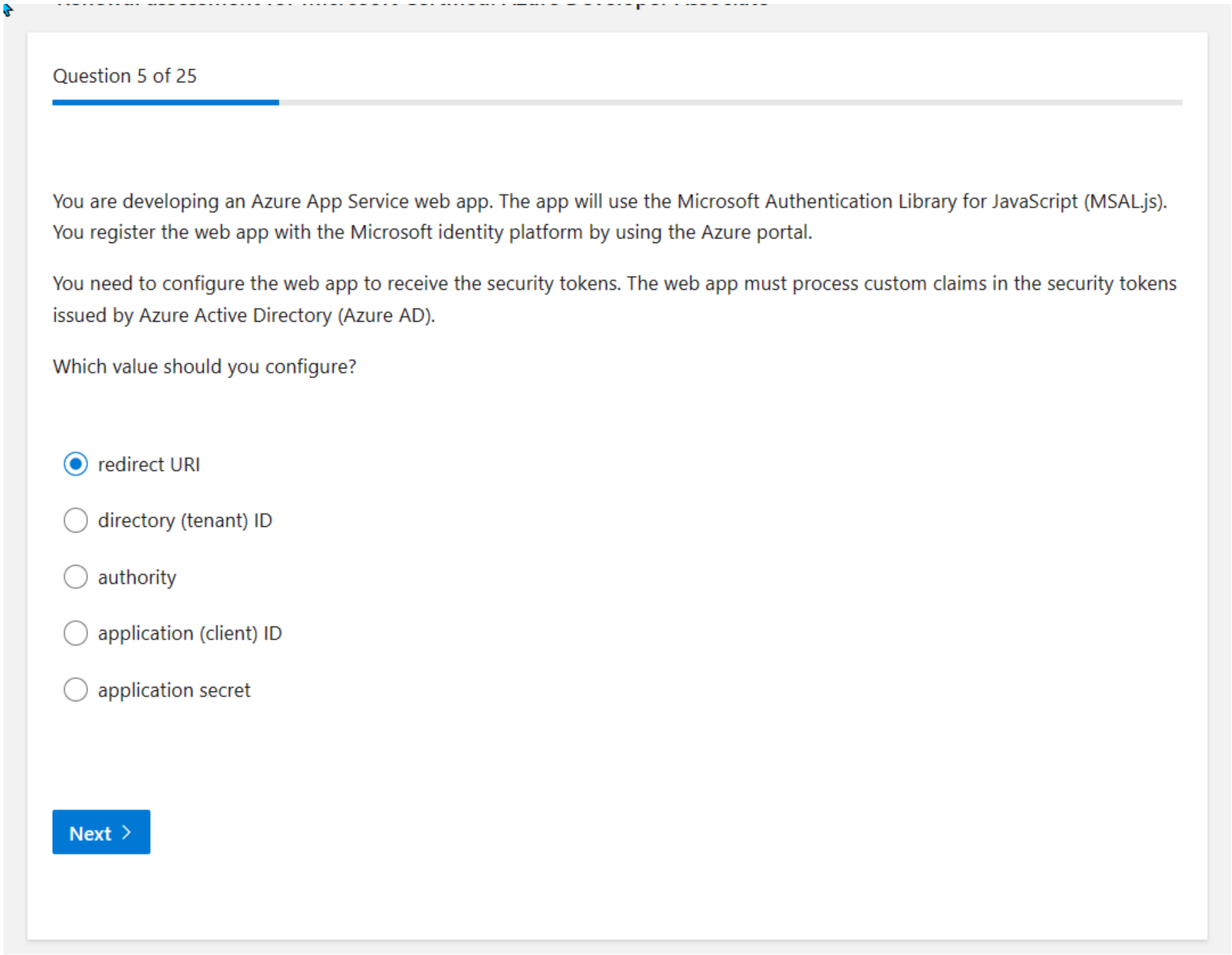
What should you use?

● the `claims` Azure AD response parameter

○ the `realm` Azure AD response parameter

○ the `authorization_uri` Azure AD response parameter

○ the `error` Azure AD response parameter



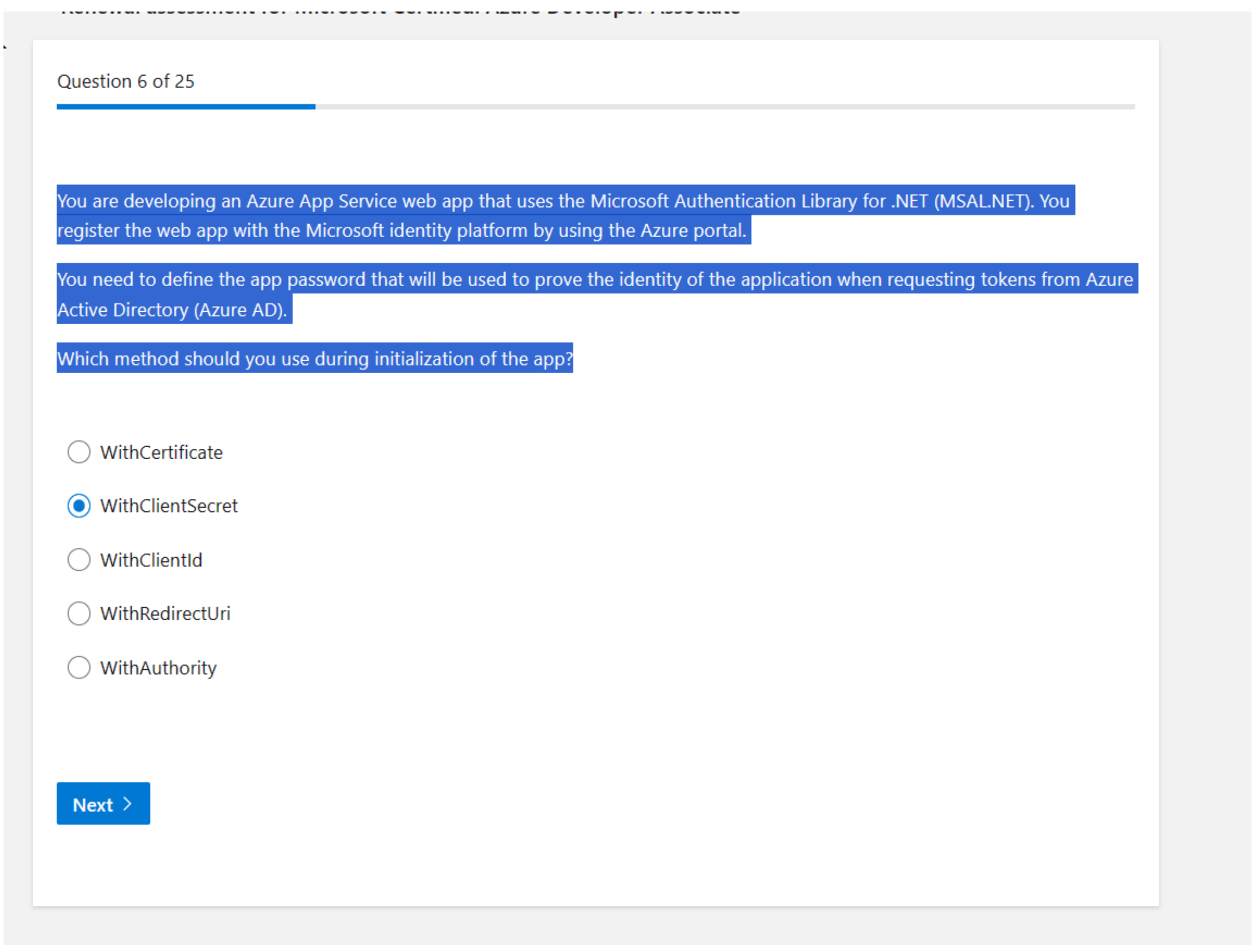
Question 5 of 25

You are developing an Azure App Service web app. The app will use the Microsoft Authentication Library for JavaScript (MSAL.js). You register the web app with the Microsoft identity platform by using the Azure portal.

You need to configure the web app to receive the security tokens. The web app must process custom claims in the security tokens issued by Azure Active Directory (Azure AD).

Which value should you configure?

- ● redirect URI
- ○ directory (tenant) ID
- ○ authority
- ○ application (client) ID
- ○ application secret



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 6 of 25

You are developing an Azure App Service web app that uses the Microsoft Authentication Library for .NET (MSAL.NET). You register the web app with the Microsoft identity platform by using the Azure portal.

You need to define the app password that will be used to prove the identity of the application when requesting tokens from Azure Active Directory (Azure AD).

Which method should you use during initialization of the app?

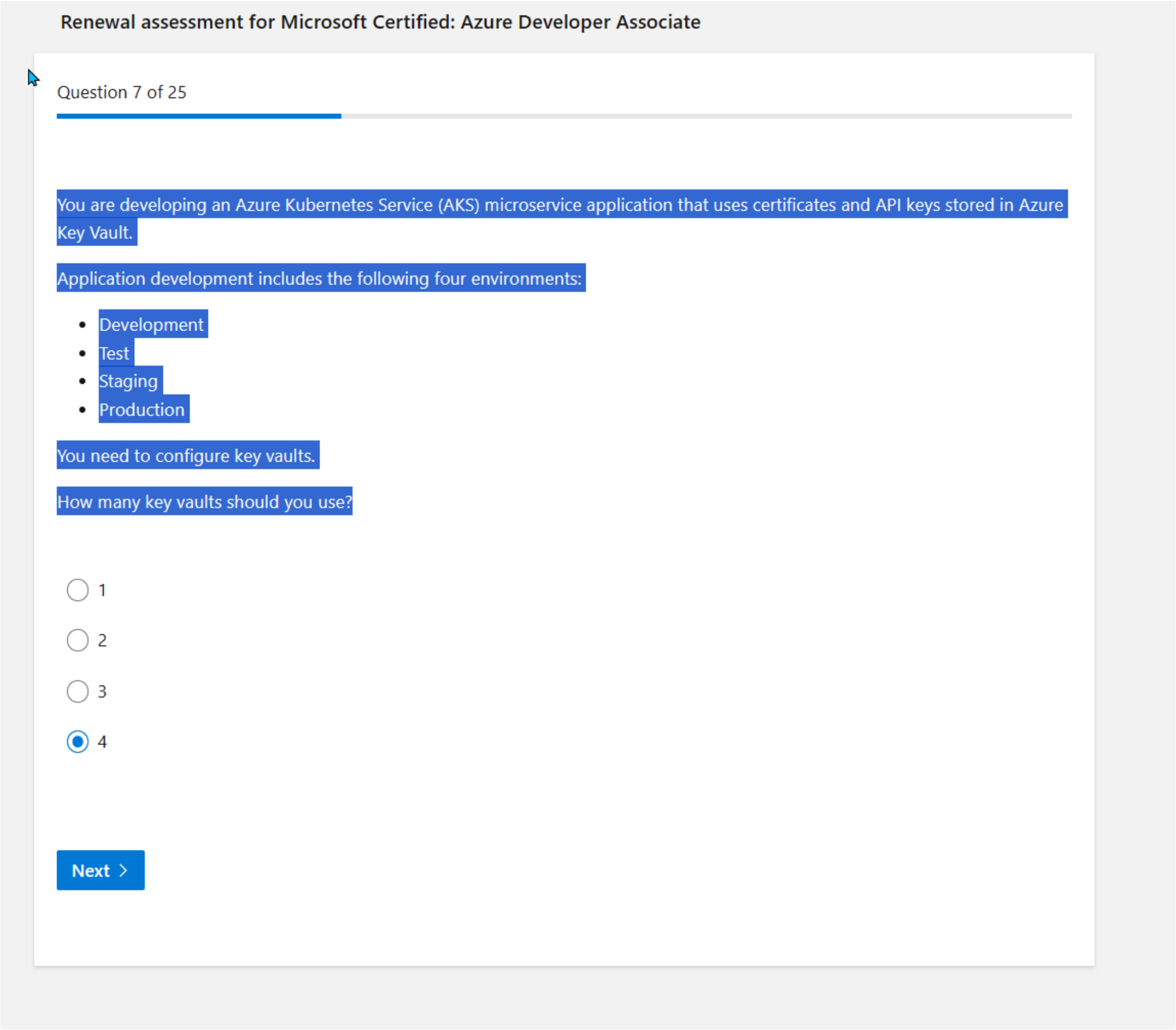
○ WithCertificate

● WithClientSecret

○ WithClientId

○ WithRedirectUri

○ WithAuthority



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 7 of 25

You are developing an Azure Kubernetes Service (AKS) microservice application that uses certificates and API keys stored in Azure Key Vault.

Application development includes the following four environments:

- Development
- Test
- Staging
- Production

You need to configure key vaults.

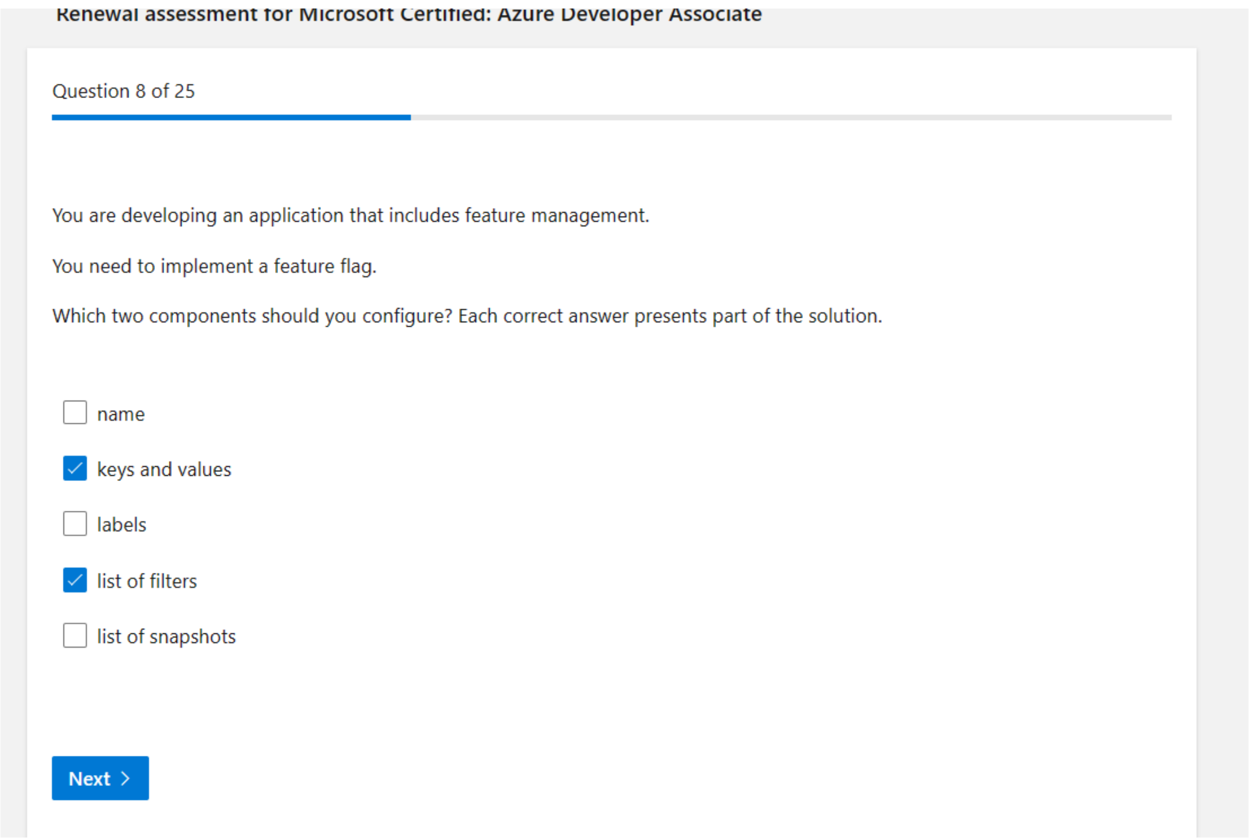
How many key vaults should you use?

○ 1

○ 2

○ 3

● 4



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 8 of 25

You are developing an application that includes feature management.

You need to implement a feature flag.

Which two components should you configure? Each correct answer presents part of the solution.

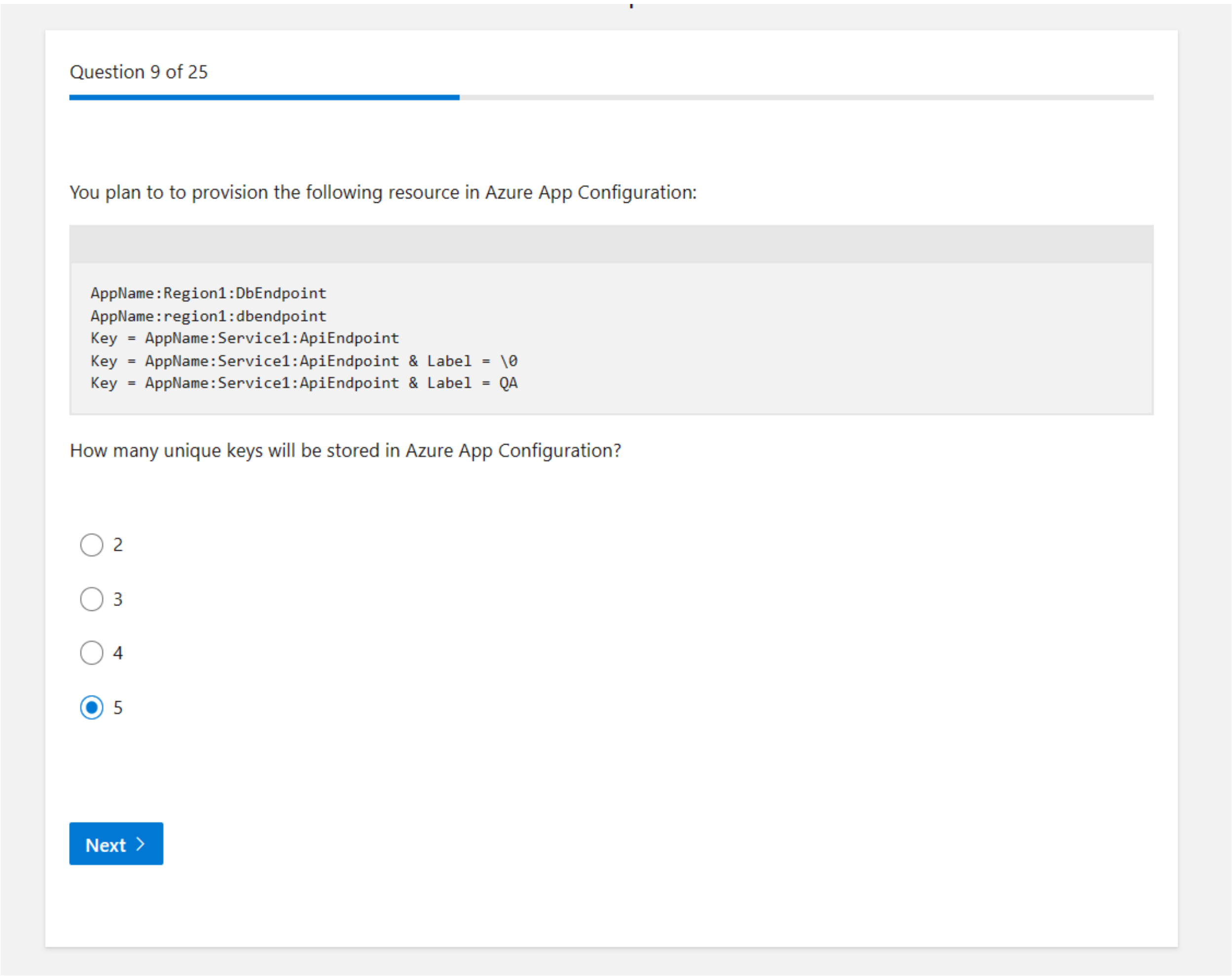
☐ name

☑ keys and values

☐ labels

☑ list of filters

☐ list of snapshots



Question 9 of 25

You plan to to provision the following resource in Azure App Configuration:

```
AppName:Region1:DbEndpoint
AppName:region1:dbendpoint
Key = AppName:Service1:ApiEndpoint
Key = AppName:Service1:ApiEndpoint & Label = \0
Key = AppName:Service1:ApiEndpoint & Label = QA
```

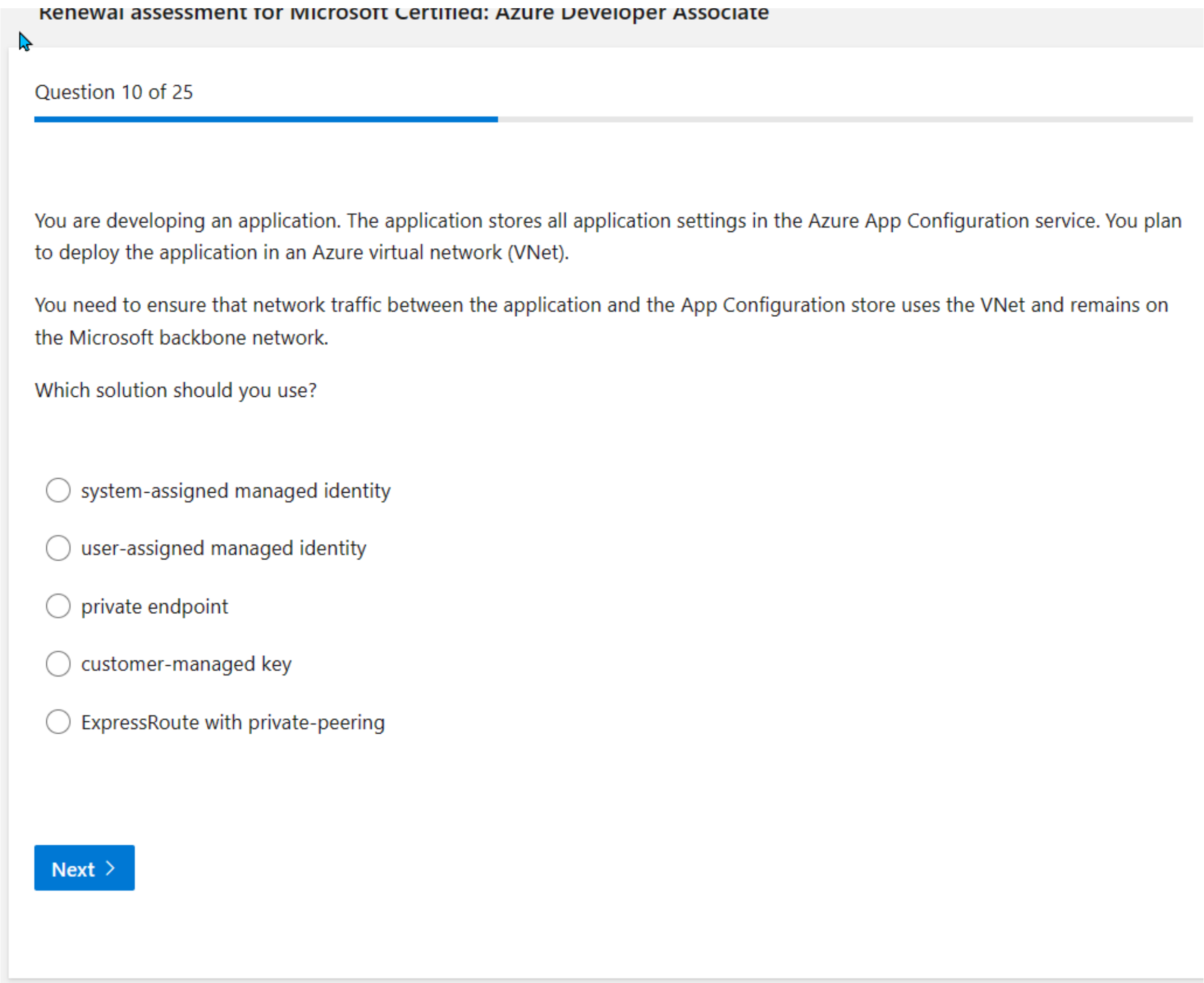
How many unique keys will be stored in Azure App Configuration?

○ 2

○ 3

○ 4

● 5



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 10 of 25

You are developing an application. The application stores all application settings in the Azure App Configuration service. You plan to deploy the application in an Azure virtual network (VNet).

You need to ensure that network traffic between the application and the App Configuration store uses the VNet and remains on the Microsoft backbone network.

Which solution should you use?

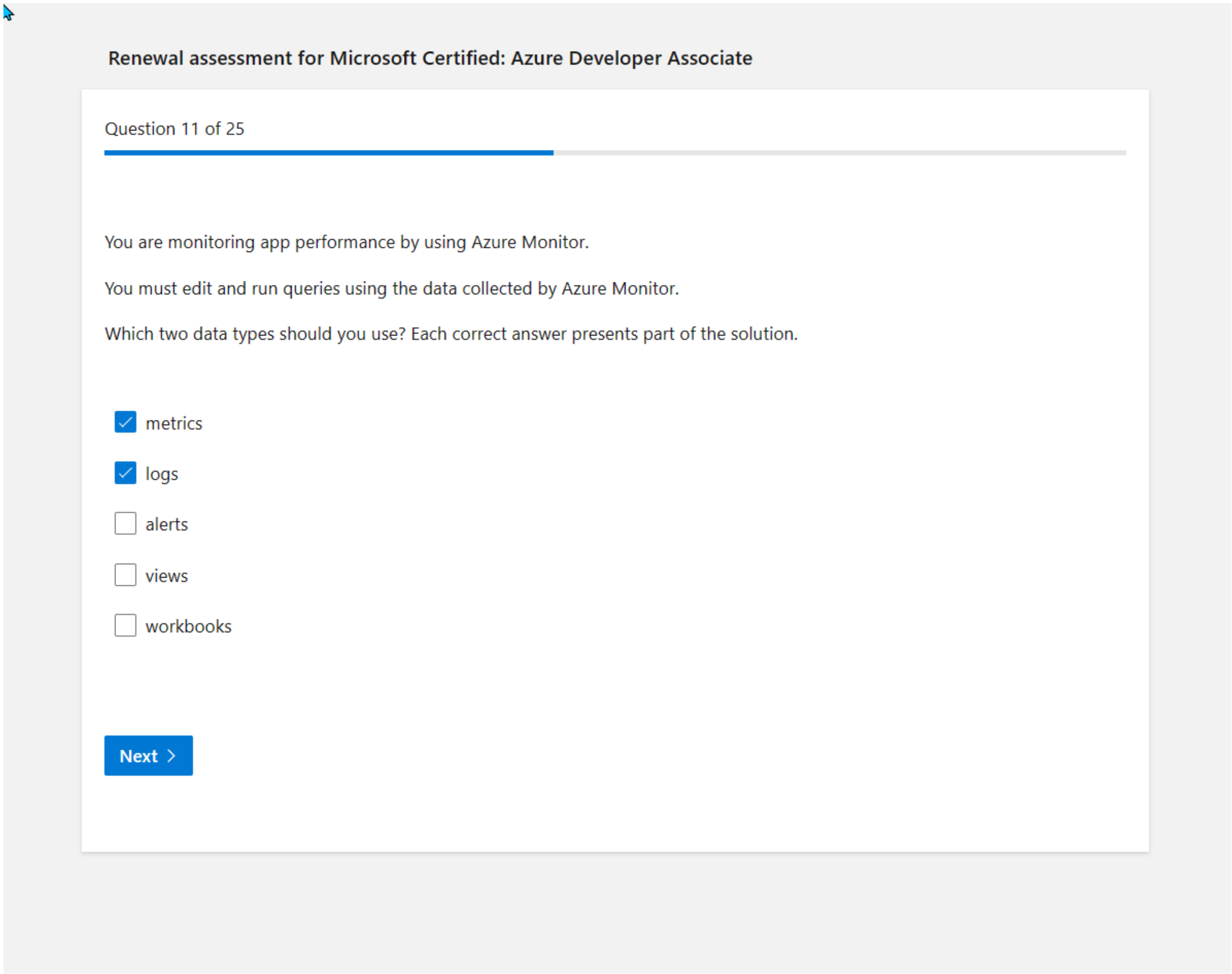
○ system-assigned managed identity

○ user-assigned managed identity

○ private endpoint

○ customer-managed key

○ ExpressRoute with private-peering



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 11 of 25

You are monitoring app performance by using Azure Monitor.

You must edit and run queries using the data collected by Azure Monitor.

Which two data types should you use? Each correct answer presents part of the solution.

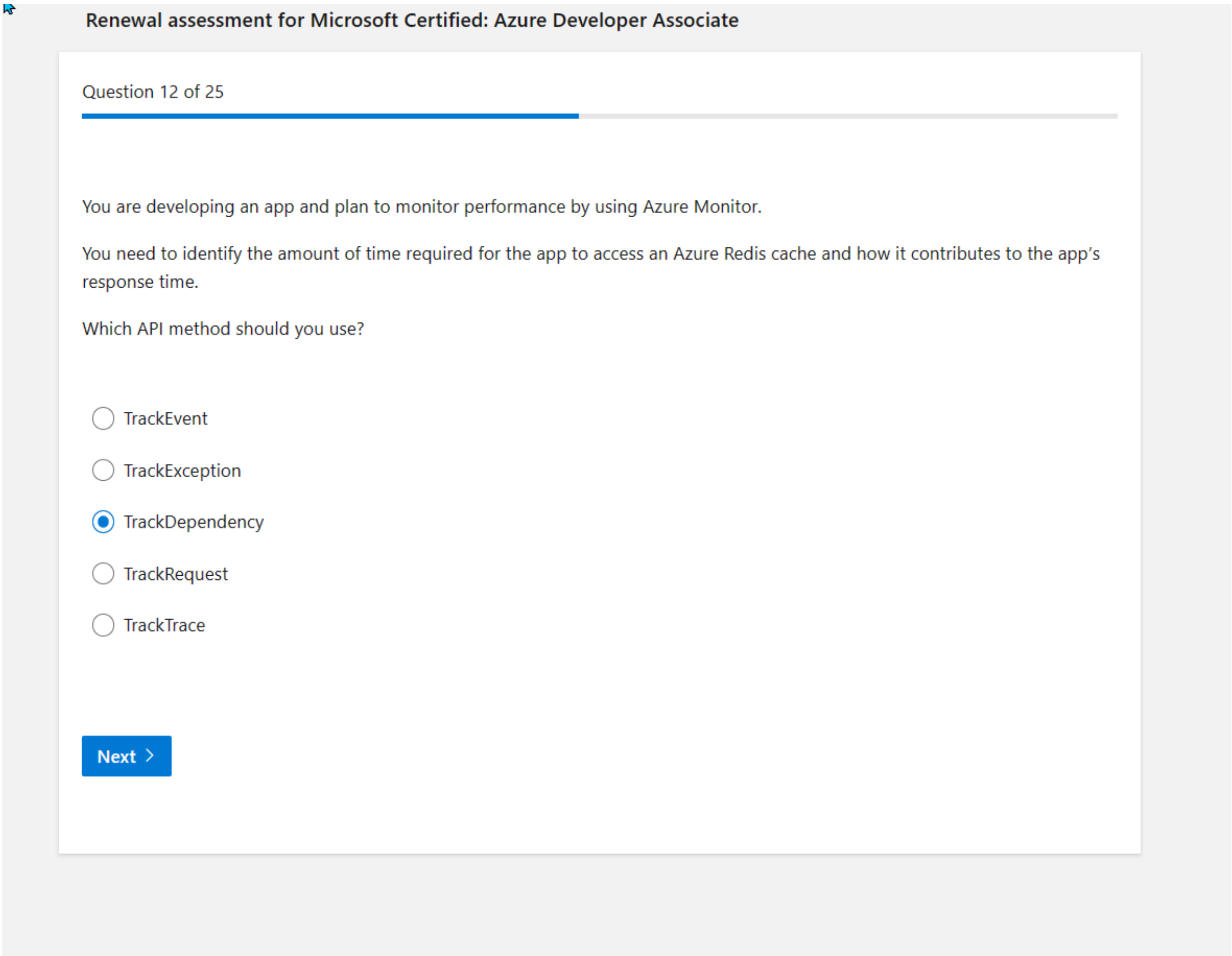
☑ metrics

☑ logs

□ alerts

□ views

□ workbooks



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 12 of 25

You are developing an app and plan to monitor performance by using Azure Monitor.

You need to identify the amount of time required for the app to access an Azure Redis cache and how it contributes to the app's response time.

Which API method should you use?

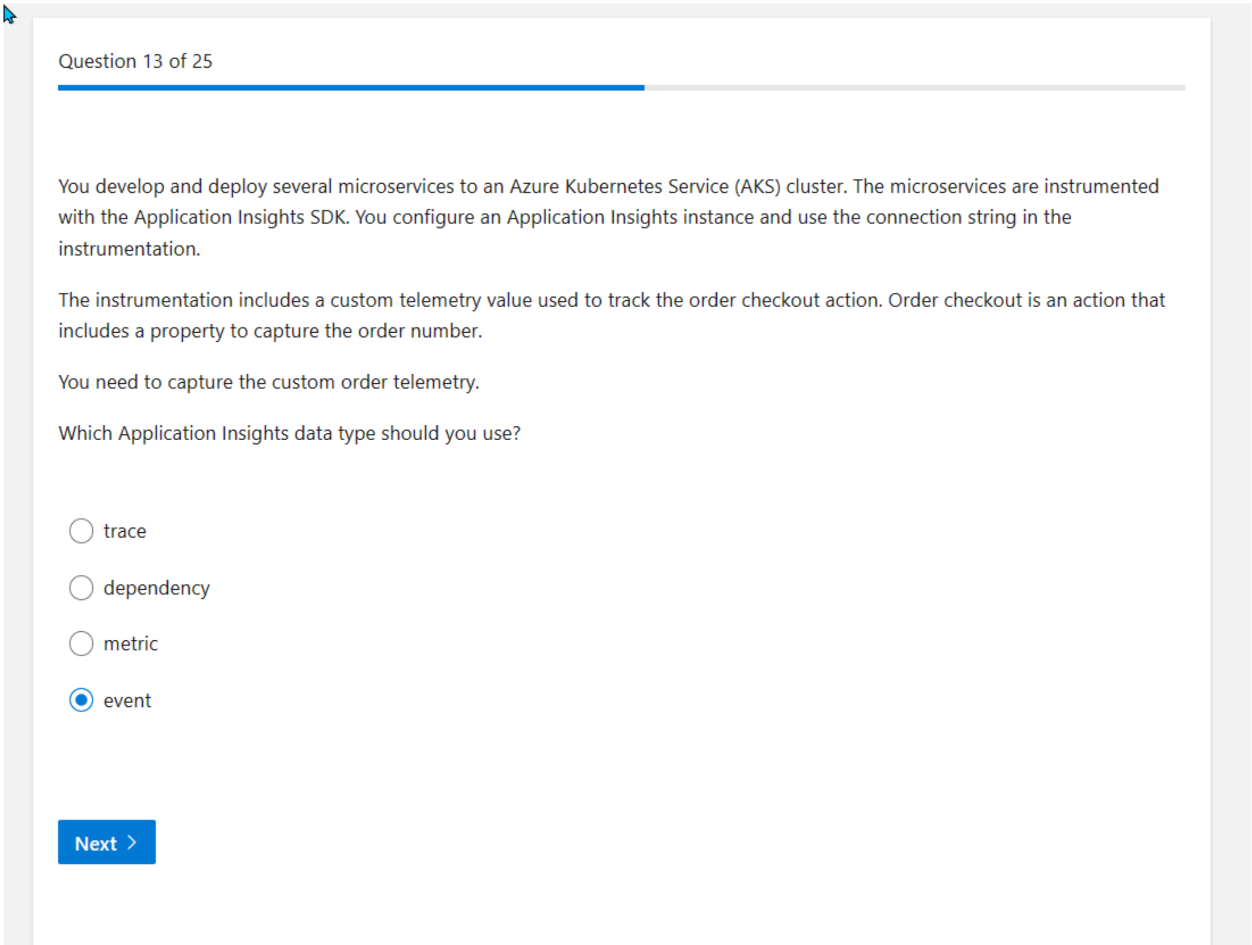
○ TrackEvent

○ TrackException

● TrackDependency

○ TrackRequest

○ TrackTrace



Question 13 of 25

You develop and deploy several microservices to an Azure Kubernetes Service (AKS) cluster. The microservices are instrumented with the Application Insights SDK. You configure an Application Insights instance and use the connection string in the instrumentation.

The instrumentation includes a custom telemetry value used to track the order checkout action. Order checkout is an action that includes a property to capture the order number.

You need to capture the custom order telemetry.

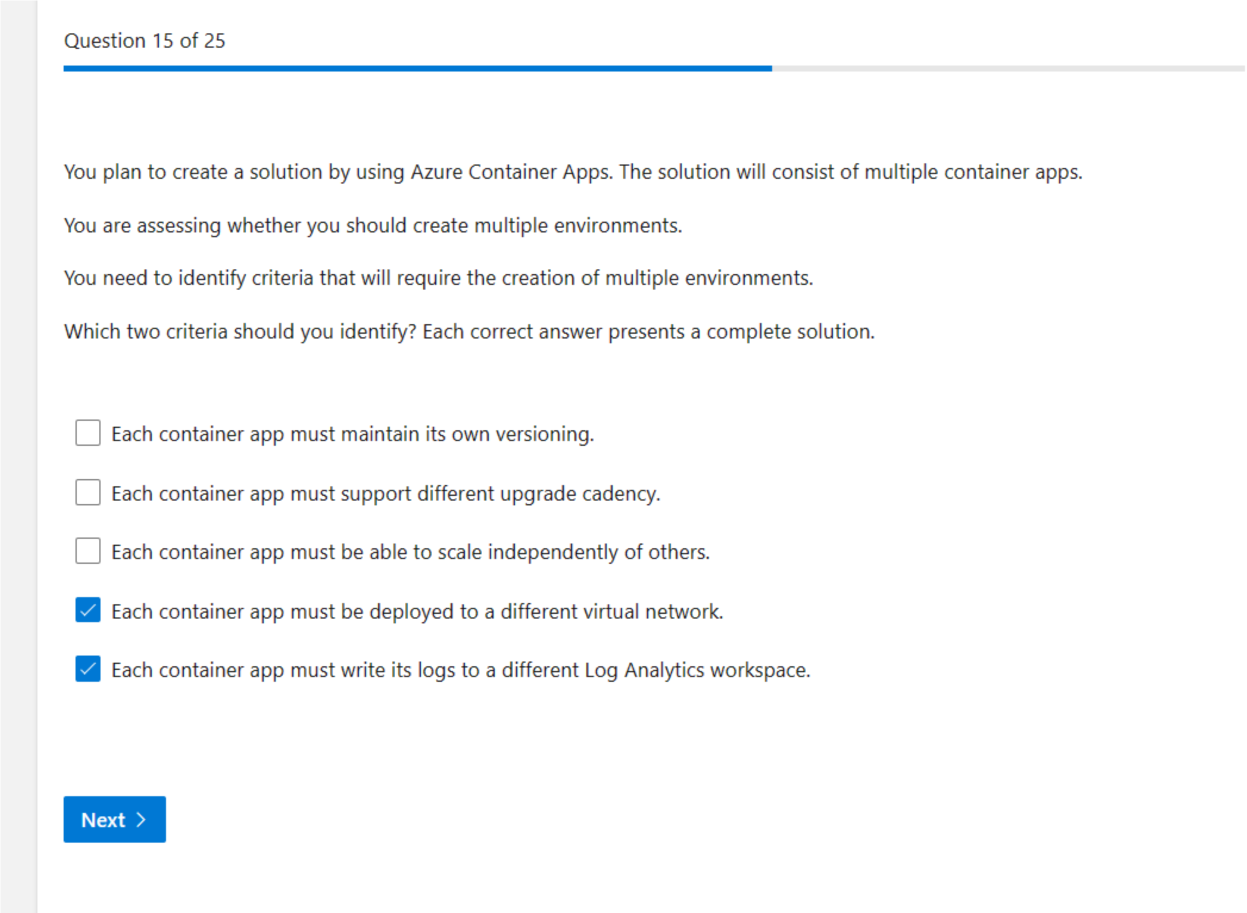
Which Application Insights data type should you use?

○ trace

○ dependency

○ metric

● event



Question 15 of 25

You plan to create a solution by using Azure Container Apps. The solution will consist of multiple container apps.

You are assessing whether you should create multiple environments.

You need to identify criteria that will require the creation of multiple environments.

Which two criteria should you identify? Each correct answer presents a complete solution.

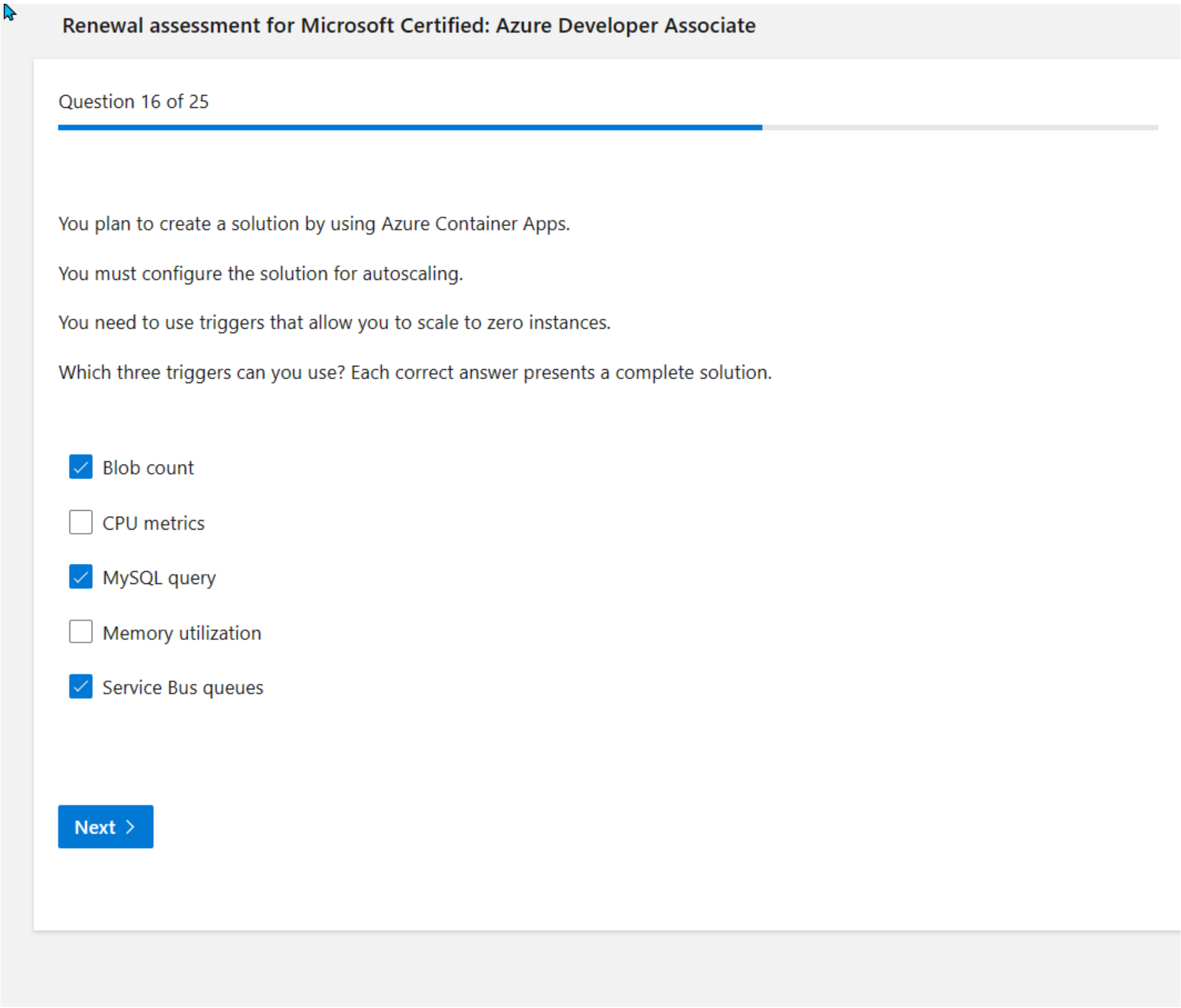
☐ Each container app must maintain its own versioning.

☐ Each container app must support different upgrade cadency.

☐ Each container app must be able to scale independently of others.

☑ Each container app must be deployed to a different virtual network.

☑ Each container app must write its logs to a different Log Analytics workspace.



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 16 of 25

You plan to create a solution by using Azure Container Apps.

You must configure the solution for autoscaling.

You need to use triggers that allow you to scale to zero instances.

Which three triggers can you use? Each correct answer presents a complete solution.

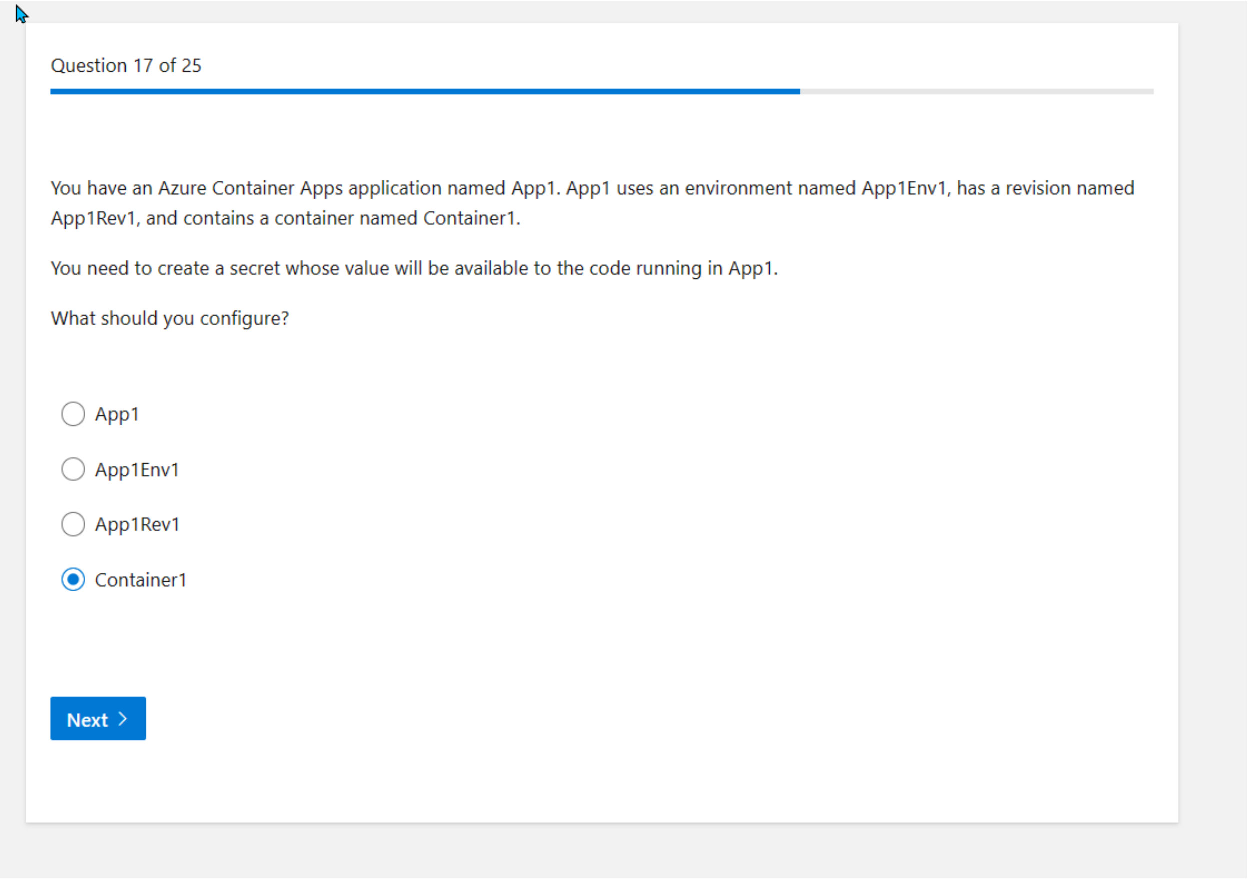
☑ Blob count

□ CPU metrics

☑ MySQL query

□ Memory utilization

☑ Service Bus queues



Question 17 of 25

You have an Azure Container Apps application named App1. App1 uses an environment named App1Env1, has a revision named App1Rev1, and contains a container named Container1.

You need to create a secret whose value will be available to the code running in App1.

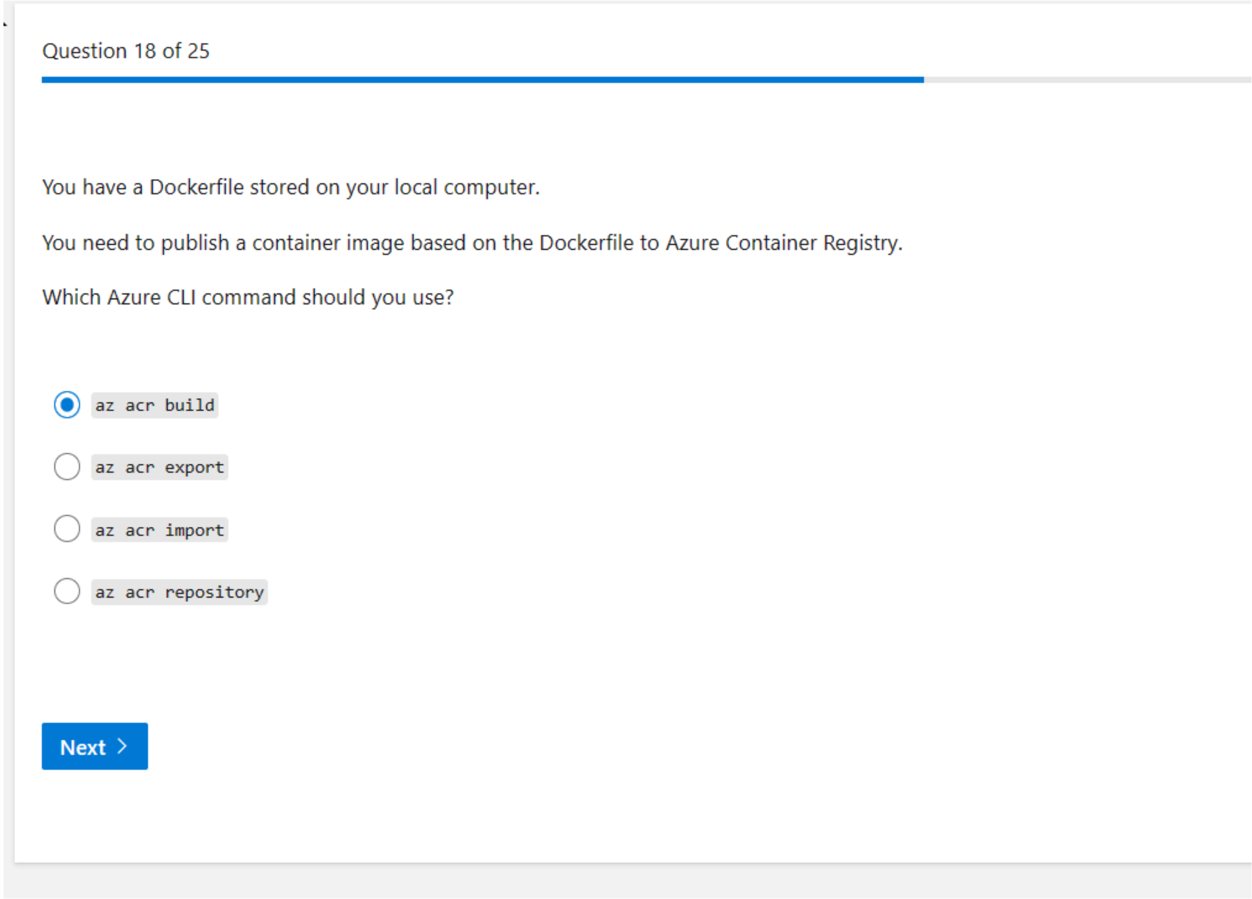
What should you configure?

○ App1

○ App1Env1

○ App1Rev1

● Container1



Question 18 of 25

You have a Dockerfile stored on your local computer.

You need to publish a container image based on the Dockerfile to Azure Container Registry.

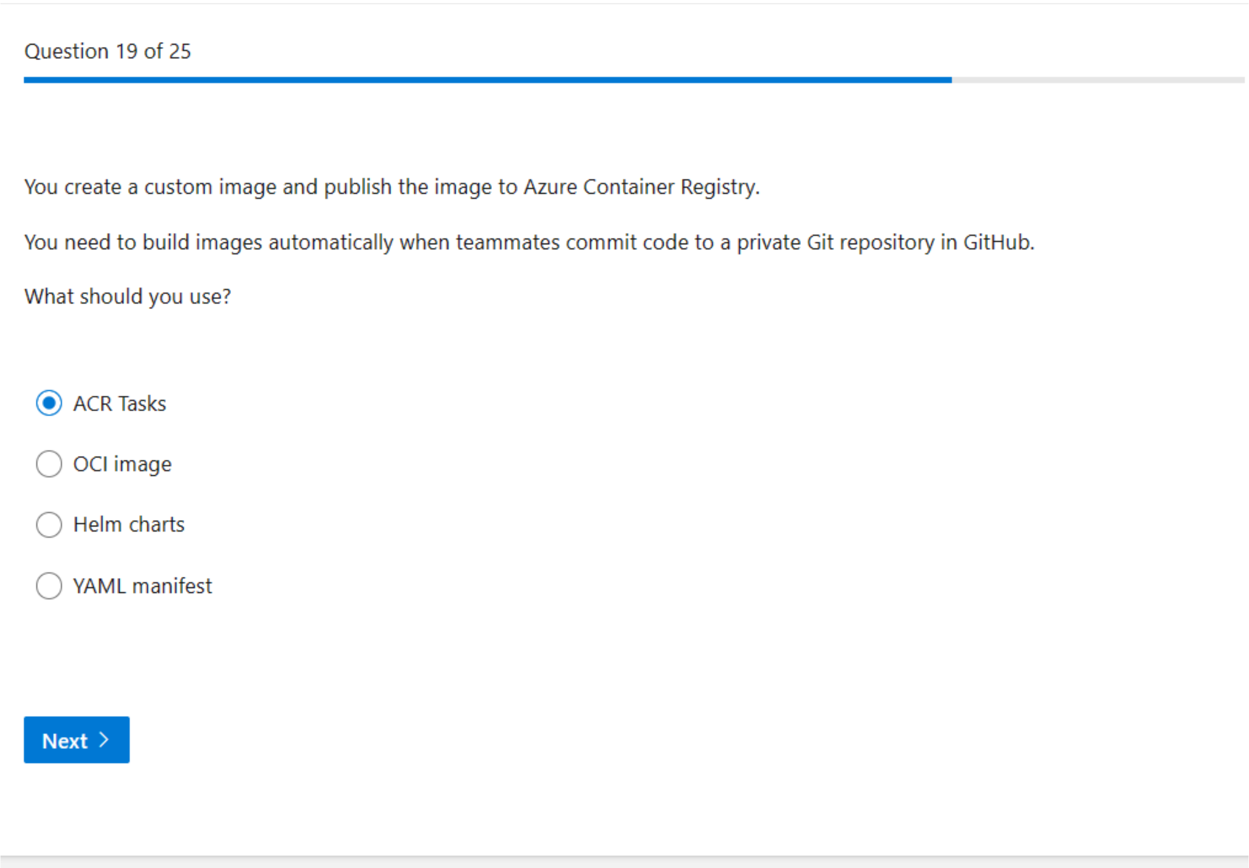
Which Azure CLI command should you use?

● `az acr build`

○ `az acr export`

○ `az acr import`

○ `az acr repository`



Question 19 of 25

You create a custom image and publish the image to Azure Container Registry.

You need to build images automatically when teammates commit code to a private Git repository in GitHub.

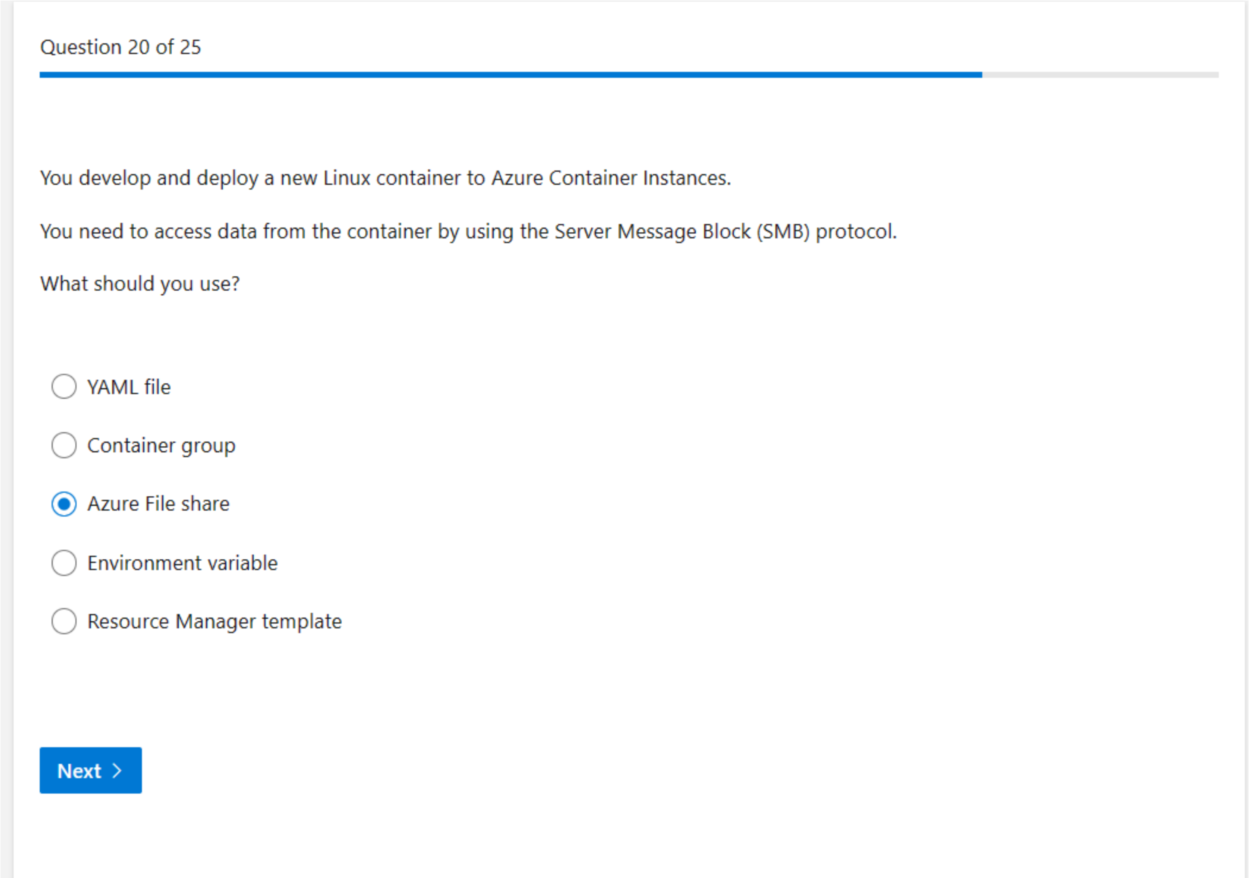
What should you use?

● ACR Tasks

○ OCI image

○ Helm charts

○ YAML manifest



Question 20 of 25

You develop and deploy a new Linux container to Azure Container Instances.

You need to access data from the container by using the Server Message Block (SMB) protocol.

What should you use?

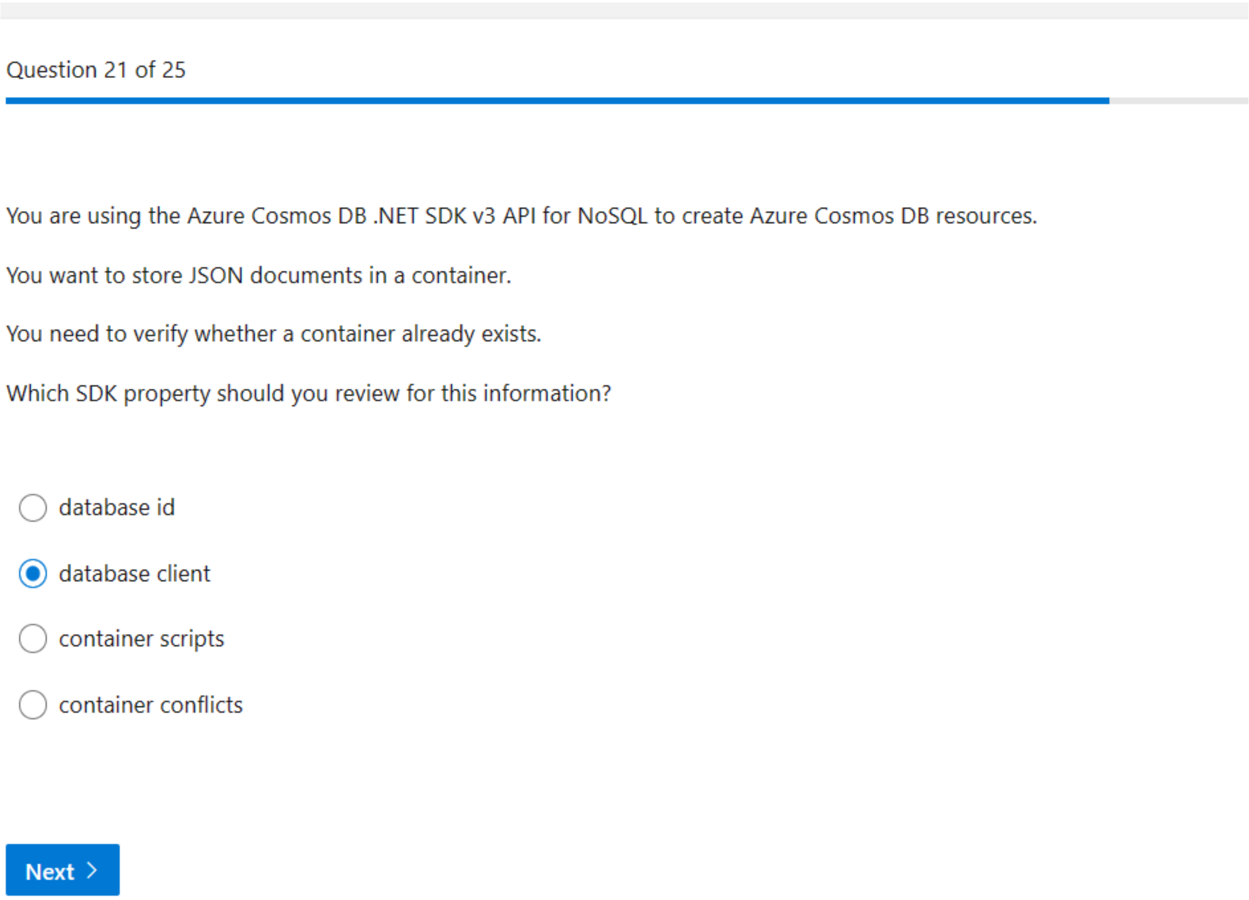
○ YAML file

○ Container group

● Azure File share

○ Environment variable

○ Resource Manager template



Question 21 of 25

You are using the Azure Cosmos DB .NET SDK v3 API for NoSQL to create Azure Cosmos DB resources.

You want to store JSON documents in a container.

You need to verify whether a container already exists.

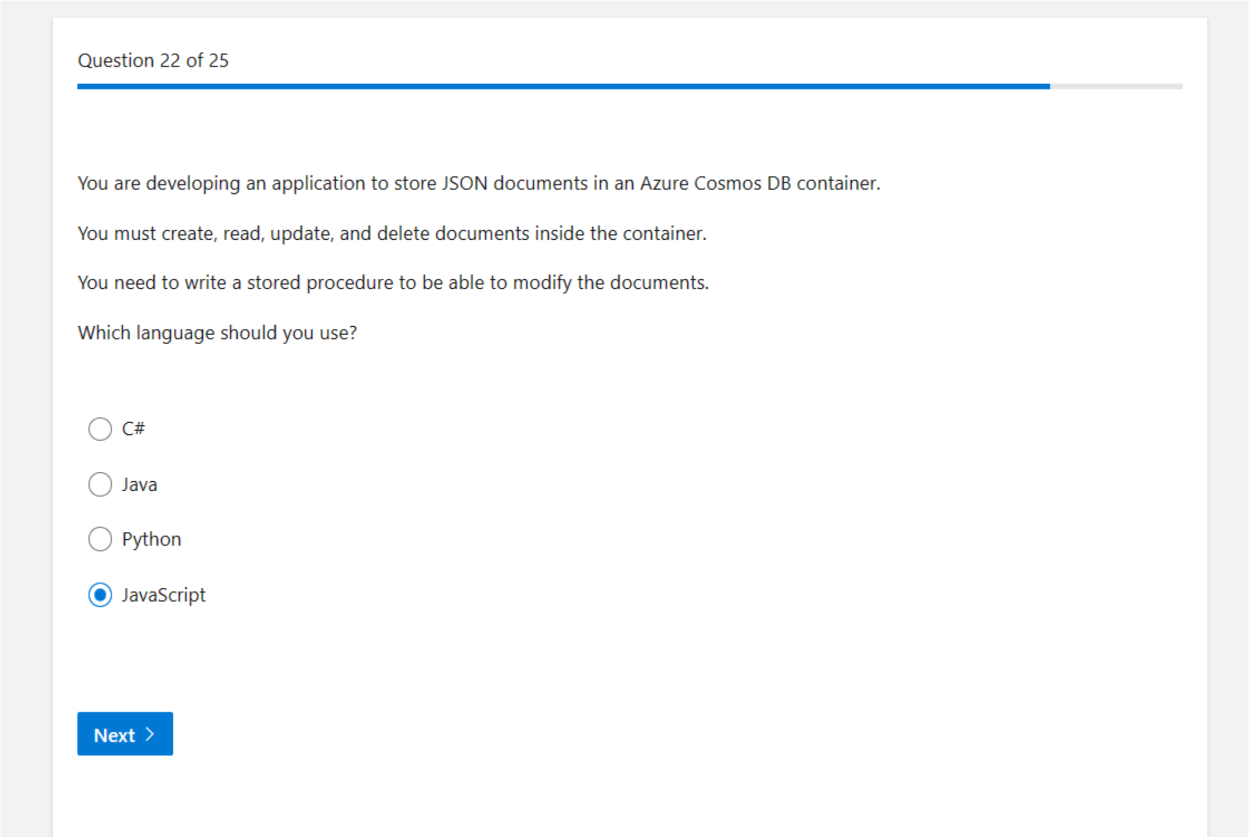
Which SDK property should you review for this information?

○ database id

● database client

○ container scripts

○ container conflicts



Question 22 of 25

You are developing an application to store JSON documents in an Azure Cosmos DB container.

You must create, read, update, and delete documents inside the container.

You need to write a stored procedure to be able to modify the documents.

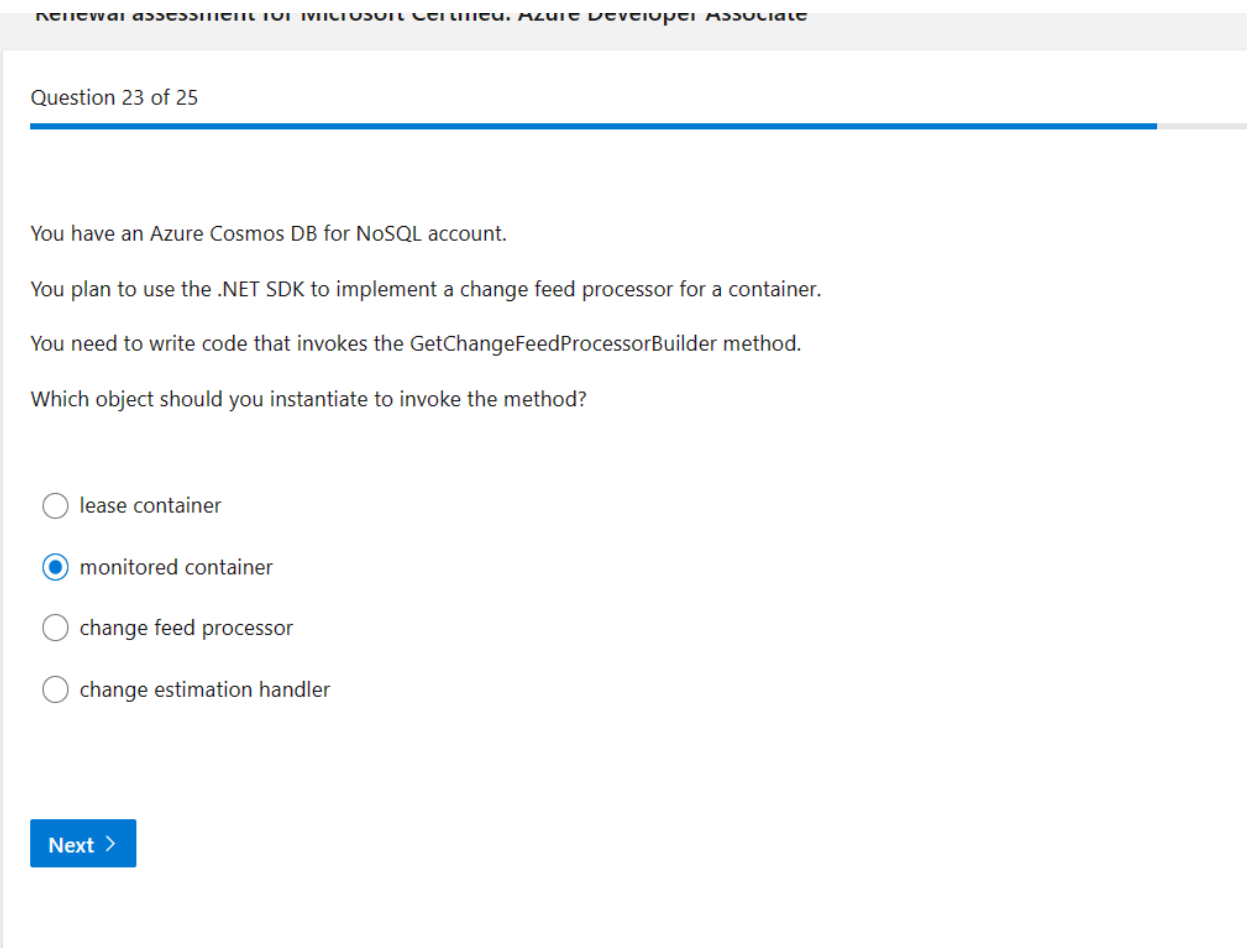
Which language should you use?

○ C#

○ Java

○ Python

● JavaScript



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 23 of 25

You have an Azure Cosmos DB for NoSQL account.

You plan to use the .NET SDK to implement a change feed processor for a container.

You need to write code that invokes the GetChangeFeedProcessorBuilder method.

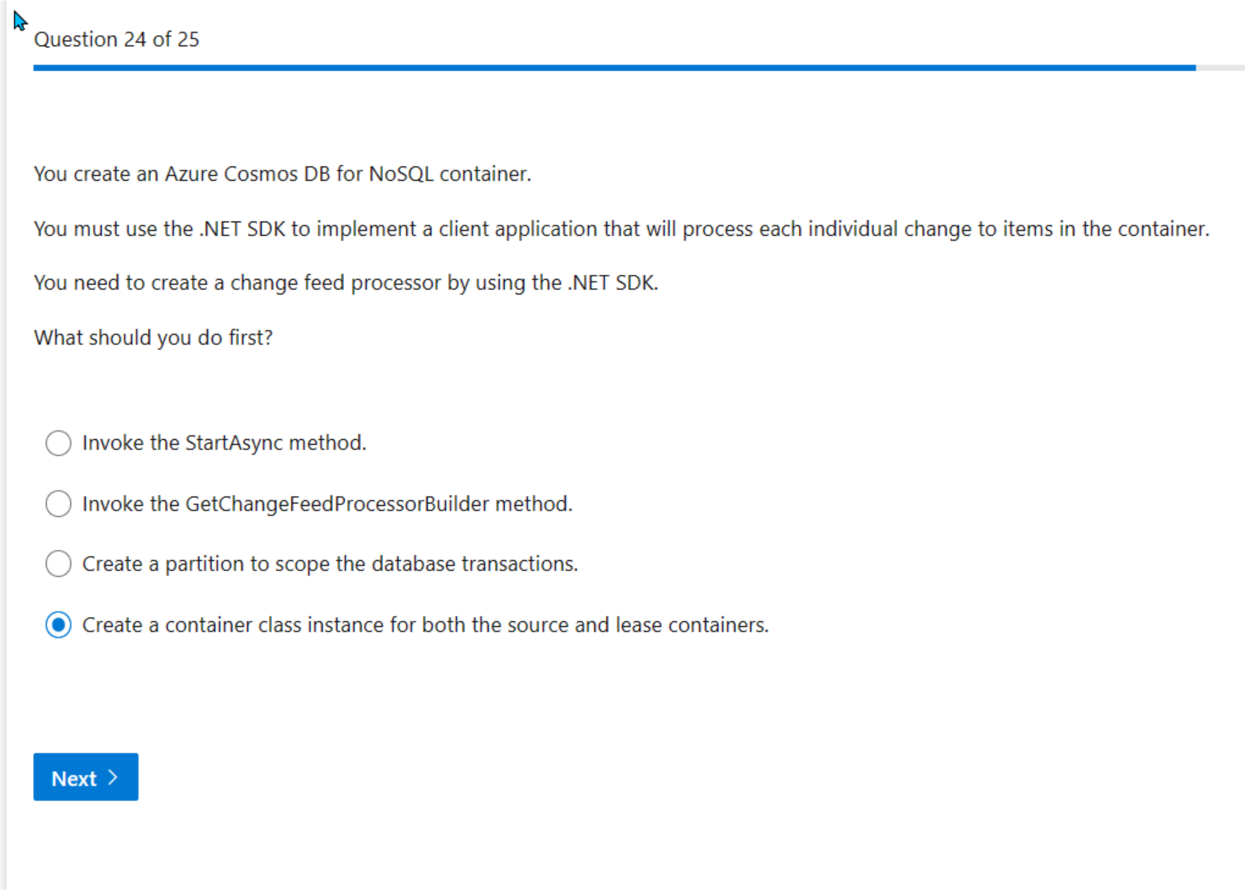
Which object should you instantiate to invoke the method?

○ lease container

● monitored container

○ change feed processor

○ change estimation handler



Question 24 of 25

You create an Azure Cosmos DB for NoSQL container.

You must use the .NET SDK to implement a client application that will process each individual change to items in the container.

You need to create a change feed processor by using the .NET SDK.

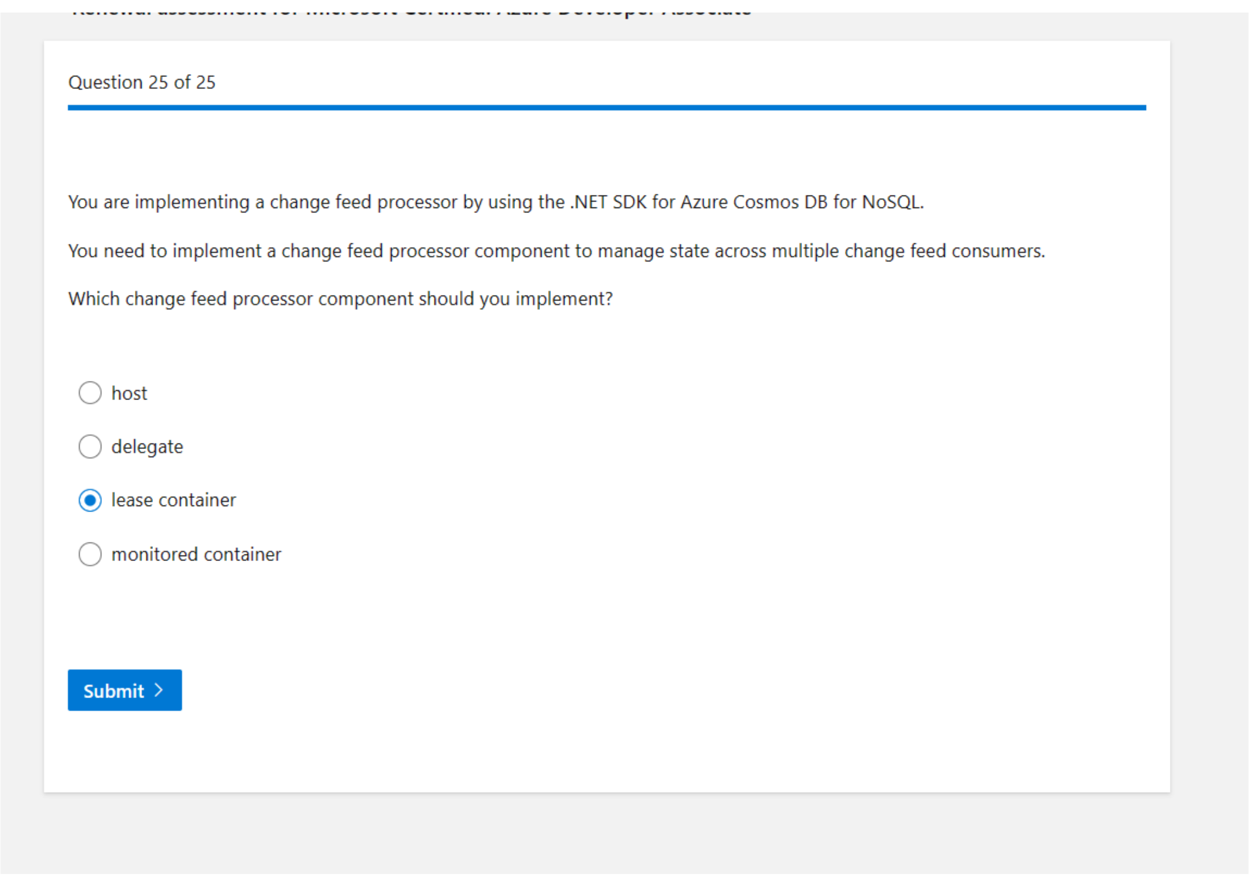
What should you do first?

○ Invoke the StartAsync method.

○ Invoke the GetChangeFeedProcessorBuilder method.

○ Create a partition to scope the database transactions.

● Create a container class instance for both the source and lease containers.



Renewal assessment for Microsoft Certified: Azure Developer Associate

Question 25 of 25

You are implementing a change feed processor by using the .NET SDK for Azure Cosmos DB for NoSQL.

You need to implement a change feed processor component to manage state across multiple change feed consumers.

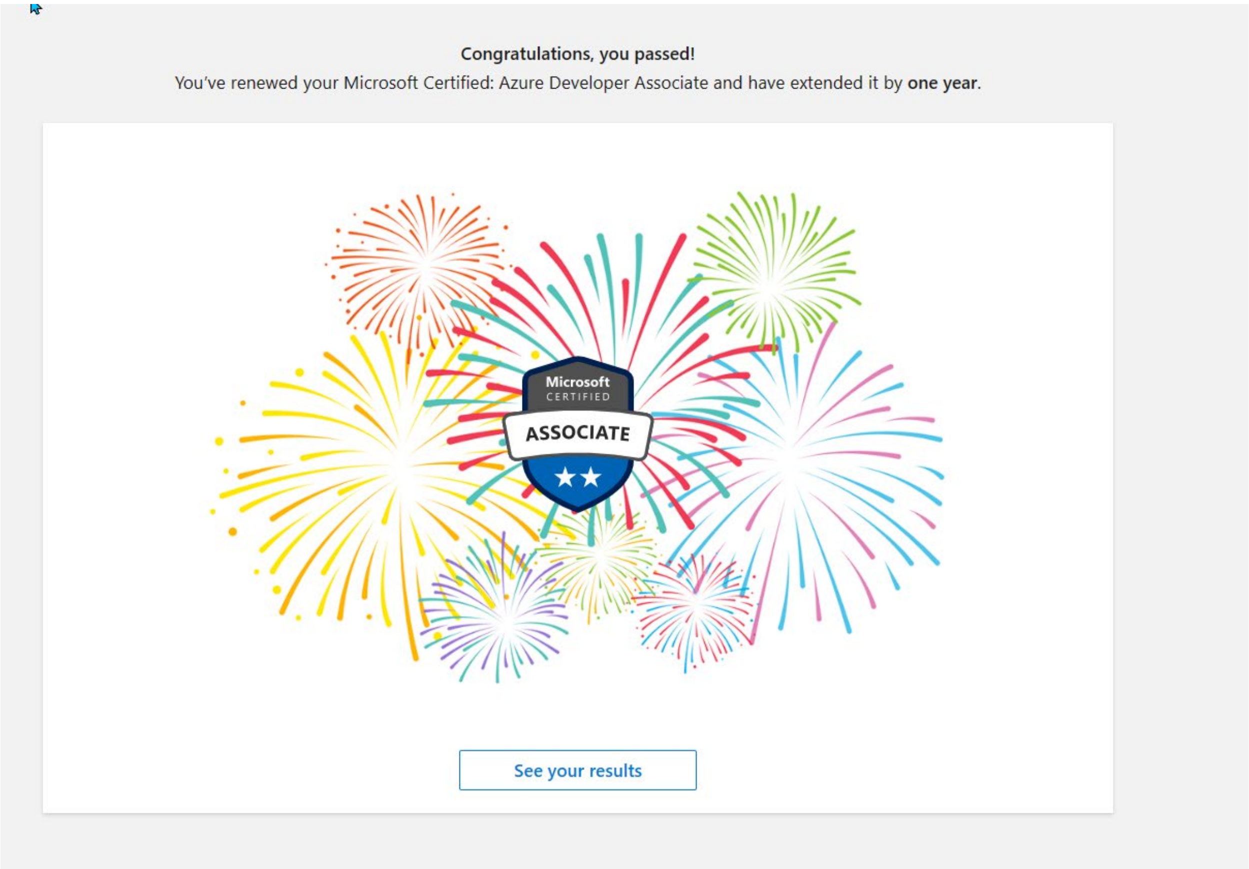
Which change feed processor component should you implement?

○ host

○ delegate

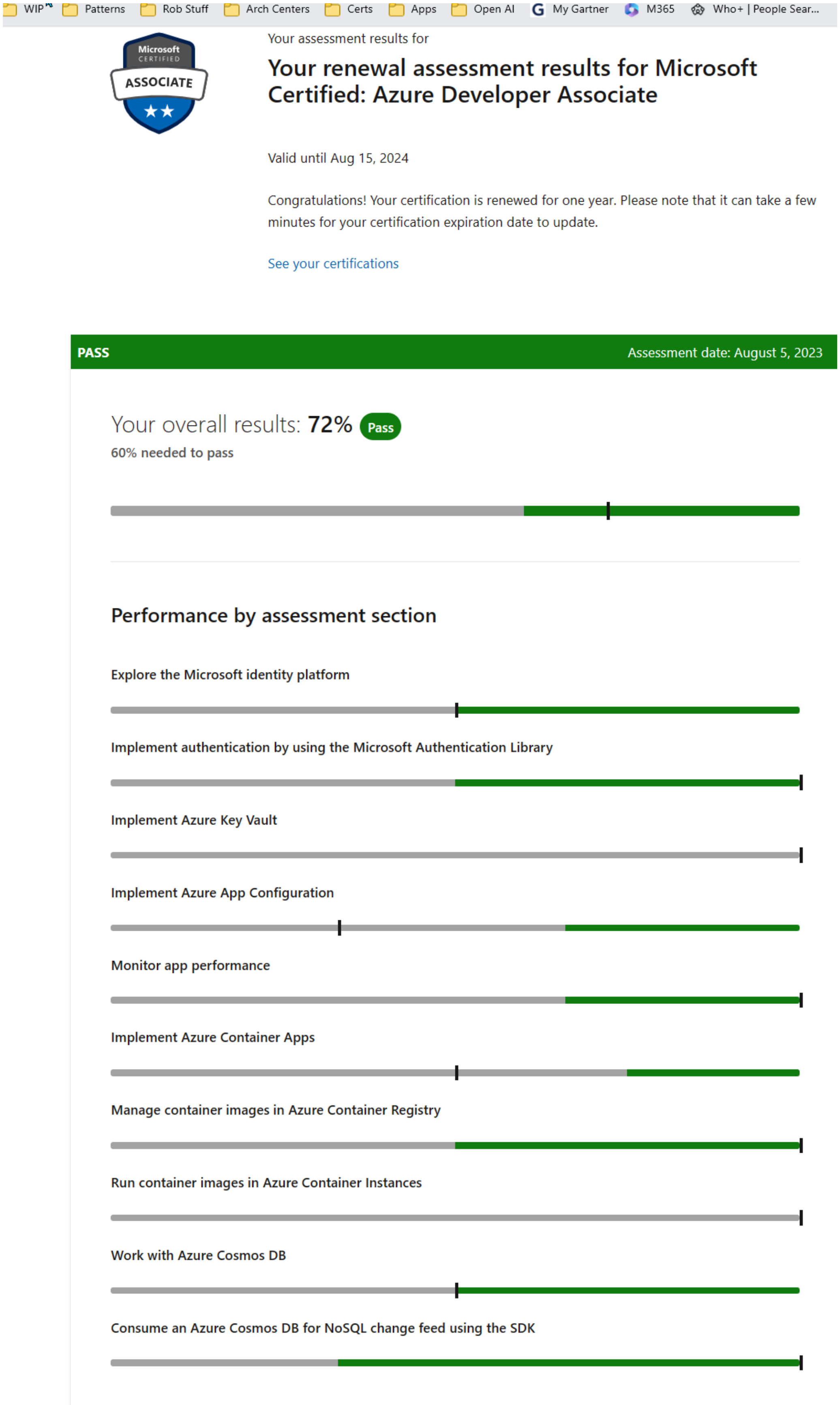
● lease container

○ monitored container



Congratulations, you passed!

You've renewed your Microsoft Certified: Azure Developer Associate and have extended it by **one year**.



Your assessment results for

Your renewal assessment results for Microsoft Certified: Azure Developer Associate

Valid until Aug 15, 2024

Congratulations! Your certification is renewed for one year. Please note that it can take a few minutes for your certification expiration date to update.

See your certifications

PASS Assessment date: August 5, 2023

Your overall results: **72%** Pass

60% needed to pass

Performance by assessment section

Explore the Microsoft identity platform

Implement authentication by using the Microsoft Authentication Library

Implement Azure Key Vault

Implement Azure App Configuration

Monitor app performance

Implement Azure Container Apps

Manage container images in Azure Container Registry

Run container images in Azure Container Instances

Work with Azure Cosmos DB

Consume an Azure Cosmos DB for NoSQL change feed using the SDK