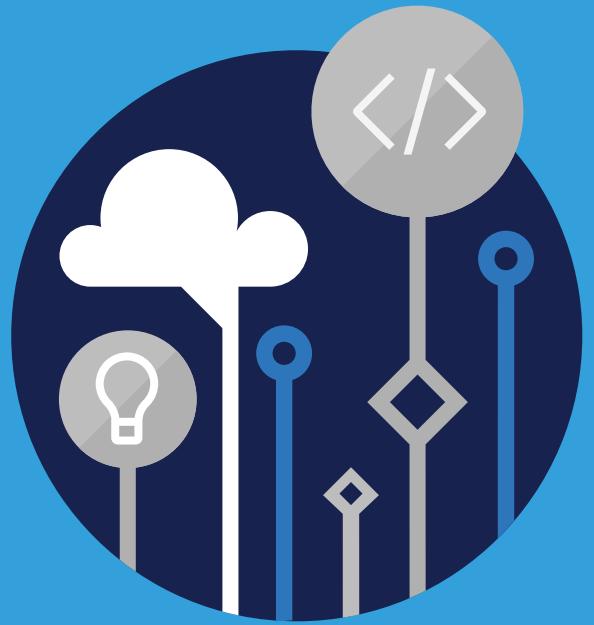


Microsoft
Official
Course



AZ-204T00

Developing Solutions for
Microsoft Azure

AZ-204T00
Developing Solutions for
Microsoft Azure

II Disclaimer

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

The names of manufacturers, products, or URLs are provided for informational purposes only and Microsoft makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of Microsoft of the manufacturer or product. Links may be provided to third party sites. Such sites are not under the control of Microsoft and Microsoft is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. Microsoft is not responsible for webcasting or any other form of transmission received from any linked site. Microsoft is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Microsoft of the site or the products contained therein.

© 2019 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at <http://www.microsoft.com/trademarks>¹ are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

¹ <http://www.microsoft.com/trademarks>

MICROSOFT LICENSE TERMS

MICROSOFT INSTRUCTOR-LED COURSEWARE

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to your use of the content accompanying this agreement which includes the media on which you received it, if any. These license terms also apply to Trainer Content and any updates and supplements for the Licensed Content unless other terms accompany those items. If so, those terms apply.

**BY ACCESSING, DOWNLOADING OR USING THE LICENSED CONTENT, YOU ACCEPT THESE TERMS.
IF YOU DO NOT ACCEPT THEM, DO NOT ACCESS, DOWNLOAD OR USE THE LICENSED CONTENT.**

If you comply with these license terms, you have the rights below for each license you acquire.

1. DEFINITIONS.

1. "Authorized Learning Center" means a Microsoft Imagine Academy (MSIA) Program Member, Microsoft Learning Competency Member, or such other entity as Microsoft may designate from time to time.
2. "Authorized Training Session" means the instructor-led training class using Microsoft Instructor-Led Courseware conducted by a Trainer at or through an Authorized Learning Center.
3. "Classroom Device" means one (1) dedicated, secure computer that an Authorized Learning Center owns or controls that is located at an Authorized Learning Center's training facilities that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
4. "End User" means an individual who is (i) duly enrolled in and attending an Authorized Training Session or Private Training Session, (ii) an employee of an MPN Member (defined below), or (iii) a Microsoft full-time employee, a Microsoft Imagine Academy (MSIA) Program Member, or a Microsoft Learn for Educators – Validated Educator.
5. "Licensed Content" means the content accompanying this agreement which may include the Microsoft Instructor-Led Courseware or Trainer Content.
6. "Microsoft Certified Trainer" or "MCT" means an individual who is (i) engaged to teach a training session to End Users on behalf of an Authorized Learning Center or MPN Member, and (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program.
7. "Microsoft Instructor-Led Courseware" means the Microsoft-branded instructor-led training course that educates IT professionals, developers, students at an academic institution, and other learners on Microsoft technologies. A Microsoft Instructor-Led Courseware title may be branded as MOC, Microsoft Dynamics, or Microsoft Business Group courseware.
8. "Microsoft Imagine Academy (MSIA) Program Member" means an active member of the Microsoft Imagine Academy Program.
9. "Microsoft Learn for Educators – Validated Educator" means an educator who has been validated through the Microsoft Learn for Educators program as an active educator at a college, university, community college, polytechnic or K-12 institution.
10. "Microsoft Learning Competency Member" means an active member of the Microsoft Partner Network program in good standing that currently holds the Learning Competency status.
11. "MOC" means the "Official Microsoft Learning Product" instructor-led courseware known as Microsoft Official Course that educates IT professionals, developers, students at an academic institution, and other learners on Microsoft technologies.
12. "MPN Member" means an active Microsoft Partner Network program member in good standing.

13. "Personal Device" means one (1) personal computer, device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
 14. "Private Training Session" means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective using Microsoft Instructor-Led Courseware. These classes are not advertised or promoted to the general public and class attendance is restricted to individuals employed by or contracted by the corporate customer.
 15. "Trainer" means (i) an academically accredited educator engaged by a Microsoft Imagine Academy Program Member to teach an Authorized Training Session, (ii) an academically accredited educator validated as a Microsoft Learn for Educators – Validated Educator, and/or (iii) a MCT.
 16. "Trainer Content" means the trainer version of the Microsoft Instructor-Led Courseware and additional supplemental content designated solely for Trainers' use to teach a training session using the Microsoft Instructor-Led Courseware. Trainer Content may include Microsoft PowerPoint presentations, trainer preparation guide, train the trainer materials, Microsoft One Note packs, classroom setup guide and Pre-release course feedback form. To clarify, Trainer Content does not include any software, virtual hard disks or virtual machines.
2. **USE RIGHTS.** The Licensed Content is licensed, not sold. The Licensed Content is licensed on a **one copy per user basis**, such that you must acquire a license for each individual that accesses or uses the Licensed Content.
- 2.1 Below are five separate sets of use rights. Only one set of rights apply to you.
 1. **If you are a Microsoft Imagine Academy (MSIA) Program Member:**
 1. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
 2. For each license you acquire on behalf of an End User or Trainer, you may either:
 1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User who is enrolled in the Authorized Training Session, and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
 2. provide one (1) End User with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
 3. provide one (1) Trainer with the unique redemption code and instructions on how they can access one (1) Trainer Content.
 3. For each license you acquire, you must comply with the following:
 1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
 2. you will ensure each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
 3. you will ensure that each End User provided with the hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End

User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,

4. you will ensure that each Trainer teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
5. you will only use qualified Trainers who have in-depth knowledge of and experience with the Microsoft technology that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Authorized Training Sessions,
6. you will only deliver a maximum of 15 hours of training per week for each Authorized Training Session that uses a MOC title, and
7. you acknowledge that Trainers that are not MCTs will not have access to all of the trainer resources for the Microsoft Instructor-Led Courseware.

2. If you are a Microsoft Learning Competency Member:

1. Each license acquire may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
2. For each license you acquire on behalf of an End User or MCT, you may either:
 1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Authorized Training Session and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware provided, **or**
 2. provide one (1) End User attending the Authorized Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
 3. you will provide one (1) MCT with the unique redemption code and instructions on how they can access one (1) Trainer Content.
3. For each license you acquire, you must comply with the following:
 1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
 2. you will ensure that each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
 3. you will ensure that each End User provided with a hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,

4. you will ensure that each MCT teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
5. you will only use qualified MCTs who also hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Authorized Training Sessions using MOC,
6. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
7. you will only provide access to the Trainer Content to MCTs.

3. If you are a MPN Member:

1. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
2. For each license you acquire on behalf of an End User or Trainer, you may either:
 1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Private Training Session, and only immediately prior to the commencement of the Private Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
 2. provide one (1) End User who is attending the Private Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
 3. you will provide one (1) Trainer who is teaching the Private Training Session with the unique redemption code and instructions on how they can access one (1) Trainer Content.
3. For each license you acquire, you must comply with the following:
 1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
 2. you will ensure that each End User attending an Private Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Private Training Session,
 3. you will ensure that each End User provided with a hard copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,
 4. you will ensure that each Trainer teaching an Private Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Private Training Session,

5. you will only use qualified Trainers who hold the applicable Microsoft Certification credential that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Private Training Sessions,
6. you will only use qualified MCTs who hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Private Training Sessions using MOC,
7. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
8. you will only provide access to the Trainer Content to Trainers.

4. If you are an End User:

For each license you acquire, you may use the Microsoft Instructor-Led Courseware solely for your personal training use. If the Microsoft Instructor-Led Courseware is in digital format, you may access the Microsoft Instructor-Led Courseware online using the unique redemption code provided to you by the training provider and install and use one (1) copy of the Microsoft Instructor-Led Courseware on up to three (3) Personal Devices. You may also print one (1) copy of the Microsoft Instructor-Led Courseware. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.

5. If you are a Trainer.

1. For each license you acquire, you may install and use one (1) copy of the Trainer Content in the form provided to you on one (1) Personal Device solely to prepare and deliver an Authorized Training Session or Private Training Session, and install one (1) additional copy on another Personal Device as a backup copy, which may be used only to reinstall the Trainer Content. You may not install or use a copy of the Trainer Content on a device you do not own or control. You may also print one (1) copy of the Trainer Content solely to prepare for and deliver an Authorized Training Session or Private Training Session.

2. If you are an MCT, you may customize the written portions of the Trainer Content that are logically associated with instruction of a training session in accordance with the most recent version of the MCT agreement.
3. If you elect to exercise the foregoing rights, you agree to comply with the following: (i) customizations may only be used for teaching Authorized Training Sessions and Private Training Sessions, and (ii) all customizations will comply with this agreement. For clarity, any use of "customize" refers only to changing the order of slides and content, and/or not using all the slides or content, it does not mean changing or modifying any slide or content.

- 2.2 **Separation of Components.** The Licensed Content is licensed as a single unit and you may not separate their components and install them on different devices.
- 2.3 **Redistribution of Licensed Content.** Except as expressly provided in the use rights above, you may not distribute any Licensed Content or any portion thereof (including any permitted modifications) to any third parties without the express written permission of Microsoft.
- 2.4 **Third Party Notices.** The Licensed Content may include third party code that Microsoft, not the third party, licenses to you under this agreement. Notices, if any, for the third party code are included for your information only.
- 2.5 **Additional Terms.** Some Licensed Content may contain components with additional terms, conditions, and licenses regarding its use. Any non-conflicting terms in those conditions and licenses also apply to your use of that respective component and supplements the terms described in this agreement.

3. **LICENSED CONTENT BASED ON PRE-RELEASE TECHNOLOGY.** If the Licensed Content's subject matter is based on a pre-release version of Microsoft technology ("Pre-release"), then in addition to the other provisions in this agreement, these terms also apply:
 1. **Pre-Release Licensed Content.** This Licensed Content subject matter is on the Pre-release version of the Microsoft technology. The technology may not work the way a final version of the technology will and we may change the technology for the final version. We also may not release a final version. Licensed Content based on the final version of the technology may not contain the same information as the Licensed Content based on the Pre-release version. Microsoft is under no obligation to provide you with any further content, including any Licensed Content based on the final version of the technology.
 2. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, either directly or through its third party designee, you give to Microsoft without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft technology, Microsoft product, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its technology, technologies, or products to third parties because we include your feedback in them. These rights survive this agreement.
 3. **Pre-release Term.** If you are an Microsoft Imagine Academy Program Member, Microsoft Learning Competency Member, MPN Member, Microsoft Learn for Educators – Validated Educator, or Trainer, you will cease using all copies of the Licensed Content on the Pre-release technology upon (i) the date which Microsoft informs you is the end date for using the Licensed Content on the Pre-release technology, or (ii) sixty (60) days after the commercial release of the technology that is the subject of the Licensed Content, whichever is earliest ("Pre-release term"). Upon expiration or termination of the Pre-release term, you will irretrievably delete and destroy all copies of the Licensed Content in your possession or under your control.
 4. **SCOPE OF LICENSE.** The Licensed Content is licensed, not sold. This agreement only gives you some rights to use the Licensed Content. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the Licensed Content only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the Licensed Content that only allows you to use it in certain ways. Except as expressly permitted in this agreement, you may not:
 - access or allow any individual to access the Licensed Content if they have not acquired a valid license for the Licensed Content,
 - alter, remove or obscure any copyright or other protective notices (including watermarks), branding or identifications contained in the Licensed Content,
 - modify or create a derivative work of any Licensed Content,
 - publicly display, or make the Licensed Content available for others to access or use,
 - copy, print, install, sell, publish, transmit, lend, adapt, reuse, link to or post, make available or distribute the Licensed Content to any third party,
 - work around any technical limitations in the Licensed Content, or
 - reverse engineer, decompile, remove or otherwise thwart any protections or disassemble the Licensed Content except and only to the extent that applicable law expressly permits, despite this limitation.
 5. **RESERVATION OF RIGHTS AND OWNERSHIP.** Microsoft reserves all rights not expressly granted to you in this agreement. The Licensed Content is protected by copyright and other intellectual property

laws and treaties. Microsoft or its suppliers own the title, copyright, and other intellectual property rights in the Licensed Content.

6. **EXPORT RESTRICTIONS.** The Licensed Content is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the Licensed Content. These laws include restrictions on destinations, end users and end use. For additional information, see www.microsoft.com/exporting.
7. **SUPPORT SERVICES.** Because the Licensed Content is provided "as is", we are not obligated to provide support services for it.
8. **TERMINATION.** Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of this agreement. Upon termination of this agreement for any reason, you will immediately stop all use of and delete and destroy all copies of the Licensed Content in your possession or under your control.
9. **LINKS TO THIRD PARTY SITES.** You may link to third party sites through the use of the Licensed Content. The third party sites are not under the control of Microsoft, and Microsoft is not responsible for the contents of any third party sites, any links contained in third party sites, or any changes or updates to third party sites. Microsoft is not responsible for webcasting or any other form of transmission received from any third party sites. Microsoft is providing these links to third party sites to you only as a convenience, and the inclusion of any link does not imply an endorsement by Microsoft of the third party site.
10. **ENTIRE AGREEMENT.** This agreement, and any additional terms for the Trainer Content, updates and supplements are the entire agreement for the Licensed Content, updates and supplements.
11. **APPLICABLE LAW.**
 1. United States. If you acquired the Licensed Content in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.
 2. Outside the United States. If you acquired the Licensed Content in any other country, the laws of that country apply.
12. **LEGAL EFFECT.** This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the Licensed Content. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.
13. **DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS" AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. MICROSOFT AND ITS RESPECTIVE AFFILIATES GIVES NO EXPRESS WARRANTIES, GUARANTEES, OR CONDITIONS. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT AND ITS RESPECTIVE AFFILIATES EXCLUDES ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.**
14. **LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT, ITS RESPECTIVE AFFILIATES AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO US\$5.00. YOU CANNOT RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES.**

This limitation applies to

- anything related to the Licensed Content, services, content (including code) on third party Internet sites or third-party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential, or other damages.

Please note: As this Licensed Content is distributed in Quebec, Canada, some of the clauses in this agreement are provided below in French.

Remarque : Ce le contenu sous licence étant distribué au Québec, Canada, certaines des clauses dans ce contrat sont fournies ci-dessous en français.

EXONÉRATION DE GARANTIE. Le contenu sous licence visé par une licence est offert « tel quel ». Toute utilisation de ce contenu sous licence est à votre seule risque et péril. Microsoft n'accorde aucune autre garantie expresse. Vous pouvez bénéficier de droits additionnels en vertu du droit local sur la protection dues consommateurs, que ce contrat ne peut modifier. La ou elles sont permises par le droit locale, les garanties implicites de qualité marchande, d'adéquation à un usage particulier et d'absence de contrefaçon sont exclues.

LIMITATION DES DOMMAGES-INTÉRÊTS ET EXCLUSION DE RESPONSABILITÉ POUR LES DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs uniquement à hauteur de 5,00 \$ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.

Cette limitation concerne:

- tout ce qui est relié au le contenu sous licence, aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers; et.
- les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également, même si Microsoft connaissait ou devrait connaître l'éventualité d'un tel dommage. Si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit, il se peut que la limitation ou l'exclusion ci-dessus ne s'appliquera pas à votre égard.

EFFET JURIDIQUE. Le présent contrat décrit certains droits juridiques. Vous pourriez avoir d'autres droits prévus par les lois de votre pays. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre pays si celles-ci ne le permettent pas.

Revised April 2019



Contents

■	Module 0 Course introduction	1
	About this course	1
■	Module 1 Explore Azure App Service	7
	Explore Azure App Service	7
	Configure web app settings	18
	Scale apps in Azure App Service	31
	Explore Azure App Service deployment slots	43
■	Module 2 Implement Azure Functions	53
	Explore Azure Functions	53
	Develop Azure Functions	59
	Implement Durable Functions	71
■	Module 3 Develop solutions that use Blob storage	87
	Explore Azure Blob storage	87
	Manage the Azure Blob storage lifecycle	95
	Work with Azure Blob storage	103
■	Module 4 Develop solutions that use Azure Cosmos DB	117
	Explore Azure Cosmos DB	117
	Implement partitioning in Azure Cosmos DB	128
	Work with Azure Cosmos DB	133
■	Module 5 Implement infrastructure as a service solutions	149
	Provision virtual machines in Azure	149
	Create and deploy Azure Resource Manager templates	157
	Manage container images in Azure Container Registry	170
	Run container images in Azure Container Instances	179
■	Module 6 Implement user authentication and authorization	191
	Explore the Microsoft identity platform	191
	Implement authentication by using the Microsoft Authentication Library	198
	Implement shared access signatures	207
	Explore Microsoft Graph	212
■	Module 7 Implement secure cloud solutions	223
	Implement Azure Key Vault	223
	Implement managed identities	229

Implement Azure App Configuration	238
Module 8 Implement API Management	249
Explore API Management	249
Module 9 Develop event-based solutions	269
Explore Azure Event Grid	269
Explore Azure Event Hubs	284
Module 10 Develop message-based solutions	297
Discover Azure message queues	297
Module 11 Instrument solutions to support monitoring and logging	317
Monitor app performance	317
Module 12 Integrate caching and content delivery within solutions	329
Develop for Azure Cache for Redis	329
Develop for storage on CDNs	342

Module 0 Course introduction

About this course

About this course

Welcome to the **Developing Solutions for Microsoft Azure** course. This course teaches developers how to create solutions that are hosted in, and utilize, Azure services.

Level: Intermediate

Audience

This course is for Azure Developers. They design and build cloud solutions such as applications and services. They participate in all phases of development, from solution design, to development and deployment, to testing and maintenance. They partner with cloud solution architects, cloud DBAs, cloud administrators, and clients to implement the solution.

Prerequisites

This course assumes you have already acquired the following skills and experience:

- At least one year of experience developing scalable solutions through all phases of software development.
- Be skilled in at least one cloud-supported programming language. Much of the course focuses on C#, .NET Framework, HTML, and using REST in applications.
- Have a base understanding of Azure and cloud concepts, services, and the Azure Portal. If you need to ramp up you can start with the **Azure Fundamentals**¹ course which is freely available.

¹ <https://docs.microsoft.com/learn/patterns/azure-fundamentals/>

Labs and exercises

The labs and demonstrations in this course are designed to be performed in the lab environments provided to the students as part of the course.

Many of the exercises include links to tools you may need to install if you want to practice them on your own.

Course syllabus

The course content includes a mix of content, exercises, hands-on labs, and reference links.

Module 01: Create Azure App Service web apps

In this module you will learn how Azure App Service functions and how to create and update an app. You will also explore App Service authentication and authorization, how to configure app settings, scale apps, and how to use deployment slots. This module includes:

- Explore Azure App Service
- Configure web app settings
- Scale apps in Azure App Service
- Explore Azure App Service deployment slots
- Lab 01: Build a web application on Azure platform as a service offerings

Module 02: Implement Azure Functions

In this module you will learn how to create and deploy Azure Functions. Explore hosting options, bindings, triggers, and how to use Durable Functions to define stateful workflows. This module includes:

- Explore Azure Functions
- Develop Azure Functions
- Implement Durable Functions
- Lab 02: Implement task processing logic by using Azure Functions

Module 03: Develop solutions that use Blob storage

In this module you will learn how to create Azure Blob storage resources, manage data through the blob storage lifecycle, and work with containers and items by using the Azure Blob storage client library V12 for .NET. This module includes:

- Explore Azure Blob storage
- Manage the Azure Blob storage lifecycle
- Work with Azure Blob storage
- Lab 03: Retrieve Azure Storage resources and metadata by using the Azure Storage SDK for .NET

Module 04: Develop solutions that use Azure Cosmos DB

In this module you will learn how to create Azure Cosmos DB resources with the appropriate consistency levels, choose and create a partition key, and perform data operations by using the .NET SDK V3 for Azure Cosmos DB. This module includes:

- Explore Azure Cosmos DB
- Implement partitioning in Azure Cosmos DB
- Work with Azure Cosmos DB
- Lab 04: Construct a polyglot data solution

Module 05: Implement infrastructure as a service solutions

In this module you will learn how to create and deploy virtual machine, deploy resources using Azure Resource Manager templates, and manage and deploy containers. This module includes:

- Provision virtual machine in Azure
- Create and deploy Azure Resource Manager templates
- Manage container images in Azure Container Registry
- Run container images in Azure Container Instances
- Lab 05: Deploy compute workloads by using images and containers

Module 06: Implement user authentication and authorization

In this module you will learn how to implement authentication and authorization to resources by using the Microsoft identity platform, Microsoft Authentication Library, shared access signatures, and use Microsoft Graph. This module includes:

- Explore the Microsoft identity platform
- Implement authentication by using the Microsoft Authentication Library
- Implement shared access signatures
- Explore Microsoft Graph
- Lab 06: Authenticate by using OpenID Connect, MSAL, and .NET SDKs

Module 07: Implement secure cloud solutions

In this module you will learn how to more securely deploy apps in Azure by using Azure Key Vault, managed identities, and Azure App Configuration. This module includes:

- Implement Azure Key Vault
- Implement managed identities
- Implement Azure App Configuration
- Lab 07: Access resource secrets more securely across services

Module 08: Implement API Management

In this module you will learn how the API Management service functions, how to transform and secure APIs, and how to create a backend API. This module includes:

- Explore API Management
- Lab 08: Create a multi-tier solution by using Azure services

Module 09: Develop event-based solutions

In this module you will learn how to build applications with event-based architectures by integrating Azure Event Grid and Azure Event Hubs in to your solution. This module includes:

- Explore Azure Event Grid
- Explore Azure Event Hubs
- Lab 09: Publish and subscribe to Event Grid events

Module 10: Develop message-based solutions

In this module you will learn how to build applications with message-based architectures by integrating Azure Service Bus and Azure Queue Storage in to your solution. This module includes:

- Discover Azure message queues
- Lab 10: Asynchronously process messages by using Azure Service Bus Queues

Module 11: Instrument solutions to support monitoring and logging

In this module you will learn how to instrument apps to enable Application Insights to monitor performance and help troubleshoot issues. This module includes:

- Monitor app performance
- Lab 11: Monitor services that are deployed to Azure

Module 12: Integrate caching and content delivery within solutions

In this module you will learn how to improve the performance and scalability of your applications by integrating Azure Cache for Redis and Azure Content Delivery Network in to your solution. This module includes:

- Develop for Azure Cache for Redis
- Develop for storage on CDNs
- Lab 12: Enhance a web application by using the Azure Content Delivery Network

Certification exam preparation

This course helps you prepare for the **AZ-204: Developing Solutions for Microsoft Azure²** certification exam.

AZ-204 includes six study areas, as shown in the table. The percentages indicate the relative weight of each area on the exam. The higher the percentage, the more questions you are likely to see in that area.

AZ-204 Study Areas	Weight
Develop Azure compute solutions	25-30%
Develop for Azure storage	15-20%
Implement Azure security	20-25%
Monitor, troubleshoot, and optimize Azure solutions	15-20%
Connect to and consume Azure and third-party services	15-20%

Note: The relative weightings are subject to change. For the latest information visit the [exam page³](#) and review the Skills measured section.

Passing the exam will earn you the **Microsoft Certified: Azure Developer Associate⁴** certification.

The modules in the course are mapped to the objectives listed in each study area on the Skills Measured section of the exam page to make it easier for you to focus on areas of the exam you choose to revisit.

Course resources

There are a lot of resources to help you learn about Azure. We recommend you bookmark these pages.

- [Azure Community Support⁵](#)
- [Azure Documentation⁶](#)
- [Microsoft Azure Blog⁷](#)
- [Microsoft Learn⁸](#)
- [Microsoft Learn Blog⁹](#)

² <https://docs.microsoft.com/en-us/learn/certifications/exams/az-204>

³ <https://docs.microsoft.com/learn/certifications/exams/az-204>

⁴ <https://docs.microsoft.com/learn/certifications/azure-developer>

⁵ <https://azure.microsoft.com/support/community/>

⁶ <https://docs.microsoft.com/azure/>

⁷ <https://azure.microsoft.com/blog/>

⁸ <https://docs.microsoft.com/learn/>

⁹ <https://techcommunity.microsoft.com/t5/microsoft-learn-blog/bg-p/MicrosoftLearnBlog>

Module 1 Explore Azure App Service

Explore Azure App Service

Introduction

Your company is considering moving their web apps to the cloud and has asked you to evaluate Azure App Service.

Learning objectives

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

- Describe Azure App Service key components and value.
- Explain how Azure App Service manages authentication and authorization.
- Identify methods to control inbound and outbound traffic to your web app.
- Deploy an app to App Service using Azure CLI commands.

Examine Azure App Service

Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends. You can develop in your favorite language, be it .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python. Applications run and scale with ease on both Windows and Linux-based environments.

Built-in auto scale support

Baked into Azure App Service is the ability to scale up/down or scale out/in. Depending on the usage of the web app, you can scale your app up/down the resources of the underlying machine that is hosting your web app. Resources include the number of cores or the amount of RAM available. Scaling out/in is the ability to increase, or decrease, the number of machine instances that are running your web app.

Continuous integration/deployment support

The Azure portal provides out-of-the-box continuous integration and deployment with Azure DevOps, GitHub, Bitbucket, FTP, or a local Git repository on your development machine. Connect your web app with any of the above sources and App Service will do the rest for you by auto-syncing code and any future changes on the code into the web app.

Deployment slots

Using the Azure portal, or command-line tools, you can easily add deployment slots to an App Service web app. For instance, you can create a staging deployment slot where you can push your code to test on Azure. Once you are happy with your code, you can easily swap the staging deployment slot with the production slot. You do all this with a few simple mouse clicks in the Azure portal.

Note: Deployment slots are only available in the Standard and Premium plan tiers.

App Service on Linux

App Service can also host web apps natively on Linux for supported application stacks. It can also run custom Linux containers (also known as Web App for Containers). App Service on Linux supports a number of language specific built-in images. Just deploy your code. Supported languages include: Node.js, Java (JRE 8 & JRE 11), PHP, Python, .NET Core, and Ruby. If the runtime your application requires is not supported in the built-in images, you can deploy it with a custom container.

The languages, and their supported versions, are updated on a regular basis. You can retrieve the current list by using the following command in the Cloud Shell.

```
az webapp list-runtimes --linux
```

Limitations

App Service on Linux does have some limitations:

- App Service on Linux is not supported on Shared pricing tier.
- You can't mix Windows and Linux apps in the same App Service plan.
- Historically, you could not mix Windows and Linux apps in the same resource group. However, all resource groups created on or after January 21, 2021 do support this scenario. Support for resource groups created before January 21, 2021 will be rolled out across Azure regions (including National cloud regions) soon.
- The Azure portal shows only features that currently work for Linux apps. As features are enabled, they're activated on the portal.

Examine Azure App Service plans

In App Service, an app (Web Apps, API Apps, or Mobile Apps) always runs in an *App Service plan*. An App Service plan defines a set of compute resources for a web app to run. One or more apps can be configured to run on the same computing resources (or in the same App Service plan). In addition, Azure Functions also has the option of running in an App Service plan.

When you create an App Service plan in a certain region (for example, West Europe), a set of compute resources is created for that plan in that region. Whatever apps you put into this App Service plan run on these compute resources as defined by your App Service plan. Each App Service plan defines:

- Region (West US, East US, etc.)
- Number of VM instances
- Size of VM instances (Small, Medium, Large)
- Pricing tier (Free, Shared, Basic, Standard, Premium, PremiumV2, PremiumV3, Isolated)

The *pricing tier* of an App Service plan determines what App Service features you get and how much you pay for the plan. There are a few categories of pricing tiers:

- **Shared compute:** Both **Free** and **Shared** share the resource pools of your apps with the apps of other customers. These tiers allocate CPU quotas to each app that runs on the shared resources, and the resources can't scale out.
- **Dedicated compute:** The **Basic**, **Standard**, **Premium**, **PremiumV2**, and **PremiumV3** tiers run apps on dedicated Azure VMs. Only apps in the same App Service plan share the same compute resources. The higher the tier, the more VM instances are available to you for scale-out.
- **Isolated:** This tier runs dedicated Azure VMs on dedicated Azure Virtual Networks. It provides network isolation on top of compute isolation to your apps. It provides the maximum scale-out capabilities.
- **Consumption:** This tier is only available to *function apps*. It scales the functions dynamically depending on workload.

Note: App Service Free and Shared (preview) hosting plans are base tiers that run on the same Azure virtual machines as other App Service apps. Some apps might belong to other customers. These tiers are intended to be used only for development and testing purposes.

How does my app run and scale?

In the **Free** and **Shared** tiers, an app receives CPU minutes on a shared VM instance and can't scale out. In other tiers, an app runs and scales as follows:

- An app runs on all the VM instances configured in the App Service plan.
- If multiple apps are in the same App Service plan, they all share the same VM instances.
- If you have multiple deployment slots for an app, all deployment slots also run on the same VM instances.
- If you enable diagnostic logs, perform backups, or run WebJobs, they also use CPU cycles and memory on these VM instances.

In this way, the App Service plan is the **scale unit** of the App Service apps. If the plan is configured to run five VM instances, then all apps in the plan run on all five instances. If the plan is configured for autoscaling, then all apps in the plan are scaled out together based on the autoscale settings.

What if my app needs more capabilities or features?

Your App Service plan can be scaled up and down at any time. It is as simple as changing the pricing tier of the plan. If your app is in the same App Service plan with other apps, you may want to improve the app's performance by isolating the compute resources. You can do it by moving the app into a separate App Service plan.

You can potentially save money by putting multiple apps into one App Service plan. However, since apps in the same App Service plan all share the same compute resources you need to understand the capacity of the existing App Service plan and the expected load for the new app.

Isolate your app into a new App Service plan when:

- The app is resource-intensive.
- You want to scale the app independently from the other apps in the existing plan.
- The app needs resource in a different geographical region.

This way you can allocate a new set of resources for your app and gain greater control of your apps.

Deploy to App Service

Every development team has unique requirements that can make implementing an efficient deployment pipeline difficult on any cloud service. App Service supports both automated and manual deployment.

Automated deployment

Automated deployment, or continuous integration, is a process used to push out new features and bug fixes in a fast and repetitive pattern with minimal impact on end users.

Azure supports automated deployment directly from several sources. The following options are available:

- **Azure DevOps:** You can push your code to Azure DevOps, build your code in the cloud, run the tests, generate a release from the code, and finally, push your code to an Azure Web App.
- **GitHub:** Azure supports automated deployment directly from GitHub. When you connect your GitHub repository to Azure for automated deployment, any changes you push to your production branch on GitHub will be automatically deployed for you.
- **Bitbucket:** With its similarities to GitHub, you can configure an automated deployment with Bitbucket.

Manual deployment

There are a few options that you can use to manually push your code to Azure:

- **Git:** App Service web apps feature a Git URL that you can add as a remote repository. Pushing to the remote repository will deploy your app.
- **CLI:** `webapp up` is a feature of the `az` command-line interface that packages your app and deploys it. Unlike other deployment methods, `az webapp up` can create a new App Service web app for you if you haven't already created one.
- **Zip deploy:** Use `curl` or a similar HTTP utility to send a ZIP of your application files to App Service.
- **FTP/S:** FTP or FTPS is a traditional way of pushing your code to many hosting environments, including App Service.

Use deployment slots

Whenever possible, use deployment slots when deploying a new production build. When using a Standard App Service Plan tier or better, you can deploy your app to a staging environment and then swap your staging and production slots. The swap operation warms up the necessary worker instances to match your production scale, thus eliminating downtime.

Explore authentication and authorization in App Service

Azure App Service provides built-in authentication and authorization support, so you can sign in users and access data by writing minimal or no code in your web app, API, and mobile back end, and also Azure Functions.

Why use the built-in authentication?

You're not required to use App Service for authentication and authorization. Many web frameworks are bundled with security features, and you can use them if you like. If you need more flexibility than App Service provides, you can also write your own utilities.

The built-in authentication feature for App Service and Azure Functions 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 App Service allows you to integrate a variety of auth capabilities into your web app or API without implementing them yourself.
- It's built directly into the platform and doesn't require any particular language, SDK, security expertise, or even any code to utilize.
- You can integrate with multiple login providers. For example, Azure AD, Facebook, Google, Twitter.

Identity providers

App Service 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	App Service Microsoft Identity Platform login (https://docs.microsoft.com/azure/app-service/configure-authentication-provider-aad)
Facebook	/auth/login/facebook	App Service Facebook login (https://docs.microsoft.com/azure/app-service/configure-authentication-provider-facebook)
Google	/auth/login/google	App Service Google login (https://docs.microsoft.com/azure/app-service/configure-authentication-provider-google)
Twitter	/auth/login/twitter	App Service Twitter login (https://docs.microsoft.com/azure/app-service/configure-authentication-provider-twitter)

Provider	Sign-in endpoint	How-To guidance
Any OpenID Connect provider	/.auth/login/<provider-Name>	App Service OpenID Connect login (https://docs.microsoft.com/azure/app-service/configure-authentication-provider-openid-connect)

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

How it works

The authentication and authorization module runs in the same sandbox as your application code. When it's enabled, every incoming HTTP request passes through it before being handled by your application code. This module handles several things for your app:

- Authenticates users with the specified provider
- Validates, stores, and refreshes tokens
- Manages the authenticated session
- Injects identity information into request headers

The module runs separately from your application code and is configured using app settings. No SDKs, specific languages, or changes to your application code are required.

Note: In Linux and containers the authentication and authorization module runs in a separate container, isolated from your application code. Because it does not run in-process, no direct integration with specific language frameworks is possible.

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: The application delegates federated sign-in to App Service. This is typically the case with browser apps, which can present the provider's login page to the user. The server code manages the sign-in process, so it is also called *server-directed flow* or *server flow*.
- With provider SDK: The application signs users in to the provider manually and then submits the authentication token to App Service for validation. This is typically the case with browser-less apps, which can't present the provider's sign-in page to the user. The application code manages the sign-in process, so it is also called *client-directed flow* or *client flow*. This applies to REST APIs, Azure Functions, JavaScript browser clients, and native mobile apps that sign users in using the provider's SDK.

The table below shows the steps of the authentication flow.

Step	Without provider SDK	With provider SDK
Sign user in	Redirects client to /.auth/login/<provider>.	Client code signs user in directly with provider's SDK and receives an authentication token. For information, see the provider's documentation.

Step	Without provider SDK	With provider SDK
Post-authentication	Provider redirects client to <code>/auth/login/<provider>/callback</code> .	Client code posts token from provider to <code>/auth/login/<provider></code> for validation.
Establish authenticated session	App Service adds authenticated cookie to response.	App Service returns its own authentication token to client code.
Serve authenticated content	Client includes authentication cookie in subsequent requests (automatically handled by browser).	Client code presents authentication token in <code>X-ZUMO-AUTH</code> header (automatically handled by Mobile Apps client SDKs).

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

Authorization behavior

In the Azure portal, you can configure App Service with a number of behaviors when an incoming request is not authenticated.

- **Allow unauthenticated requests:** This option defers authorization of unauthenticated traffic to your application code. For authenticated requests, App Service also passes along authentication information in the HTTP headers. This option provides more flexibility in handling anonymous requests. It lets you present multiple sign-in providers to your users.
- **Require authentication:** This option will reject any unauthenticated traffic to your application. This rejection can be a redirect action to one of the configured identity providers. In these cases, a browser client is redirected to `/auth/login/<provider>` for the provider you choose. If the anonymous request comes from a native mobile app, the returned response is an `HTTP 401 Unauthorized`. You can also configure the rejection to be an `HTTP 401 Unauthorized` or `HTTP 403 Forbidden` for all requests.

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

Discover App Service networking features

By default, apps hosted in App Service are accessible directly through the internet and can reach only internet-hosted endpoints. But for many applications, you need to control the inbound and outbound network traffic.

There are two main deployment types for Azure App Service. The multitenant public service hosts App Service plans in the Free, Shared, Basic, Standard, Premium, PremiumV2, and PremiumV3 pricing SKUs. There is also the single-tenant App Service Environment (ASE) hosts Isolated SKU App Service plans directly in your Azure virtual network.

Multitenant App Service networking features

Azure App Service is a distributed system. The roles that handle incoming HTTP or HTTPS requests are called *front ends*. The roles that host the customer workload are called *workers*. All the roles in an App

Service deployment exist in a multitenant network. Because there are many different customers in the same App Service scale unit, you can't connect the App Service network directly to your network.

Instead of connecting the networks, you need features to handle the various aspects of application communication. The features that handle requests to your app can't be used to solve problems when you're making calls from your app. Likewise, the features that solve problems for calls from your app can't be used to solve problems to your app.

Inbound features	Outbound features
App-assigned address	Hybrid Connections
Access restrictions	Gateway-required VNet Integration
Service endpoints	VNet Integration
Private endpoints	

You can mix the features to solve your problems with a few exceptions. The following inbound use cases are examples of how to use App Service networking features to control traffic inbound to your app.

Inbound use case	Feature
Support IP-based SSL needs for your app	App-assigned address
Support unshared dedicated inbound address for your app	App-assigned address
Restrict access to your app from a set of well-defined addresses	Access restrictions

Default networking behavior

Azure App Service scale units support many customers in each deployment. The Free and Shared SKU plans host customer workloads on multitenant workers. The Basic and higher plans host customer workloads that are dedicated to only one App Service plan. If you have a Standard App Service plan, all the apps in that plan will run on the same worker. If you scale out the worker, all the apps in that App Service plan will be replicated on a new worker for each instance in your App Service plan.

Outbound addresses

The worker VMs are broken down in large part by the App Service plans. The Free, Shared, Basic, Standard, and Premium plans all use the same worker VM type. The PremiumV2 plan uses another VM type. PremiumV3 uses yet another VM type. When you change the VM family, you get a different set of outbound addresses. If you scale from Standard to PremiumV2, your outbound addresses will change. If you scale from PremiumV2 to PremiumV3, your outbound addresses will change. In some older scale units, both the inbound and outbound addresses will change when you scale from Standard to PremiumV2.

There are a number of addresses that are used for outbound calls. The outbound addresses used by your app for making outbound calls are listed in the properties for your app. These addresses are shared by all the apps running on the same worker VM family in the App Service deployment. If you want to see all the addresses that your app might use in a scale unit, there's a property called `possibleOutboundAddresses` that will list them.

Find outbound IPs

To find the outbound IP addresses currently used by your app in the Azure portal, click **Properties** in your app's left-hand navigation.

You can find the same information by running the following command in the Cloud Shell. They are listed in the **Additional Outbound IP Addresses** field.

```
az webapp show \
--resource-group <group_name> \
--name <app_name> \
--query outboundIpAddresses \
--output tsv
```

To find all possible outbound IP addresses for your app, regardless of pricing tiers, run the following command in the Cloud Shell.

```
az webapp show \
--resource-group <group_name> \
--name <app_name> \
--query possibleOutboundIpAddresses \
--output tsv
```

Exercise: Create a static HTML web app by using Azure Cloud Shell

After gathering information about App Service you've decided to create and update a simple web app to try it out. In this exercise you'll deploy a basic HTML+CSS site to Azure App Service by using the Azure CLI `az webapp up` command. You will then update the code and redeploy it by using the same command.

The `az webapp up` command makes it easy to create and update web apps. When executed it performs the following actions:

- Create a default resource group.
- Create a default app service plan.
- Create an app with the specified name.
- Zip deploy files from the current working directory to the web app.

Prerequisites

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

- 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>.

Login to Azure and download the sample app

1. Login to the **Azure portal**¹ and open the Cloud Shell.



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

¹ <https://portal.azure.com>



3. In the Cloud Shell, create a directory and then navigate to it.

```
mkdir htmlapp
```

```
cd htmlapp
```

4. Run the following `git` command to clone the sample app repository to your `htmlapp` directory.

```
git clone https://github.com/Azure-Samples/html-docs-hello-world.git
```

Create the web app

1. Change to the directory that contains the sample code and run the `az webapp up` command. In the following example, replace `<myAppName>` with a unique app name, and `<myLocation>` with a region near you.

```
cd html-docs-hello-world
```

```
az webapp up --location <myLocation> --name <myAppName> --html
```

This command may take a few minutes to run. While running, it displays information similar to the example below. Make a note of the `resourceGroup` value. You need it for the *Clean up resources* section later.

```
{
  "app_url": "https://<myAppName>.azurewebsites.net",
  "location": "westeurope",
  "name": "<app_name>",
  "os": "Windows",
  "resourcegroup": "<resource_group_name>",
  "serverfarm": "appsvc_asp_Windows_westeurope",
  "sku": "FREE",
  "src_path": "/home/<username>/demoHTML/html-docs-hello-world",
  < JSON data removed for brevity. >
}
```

2. Open a browser and navigate to the app URL (`http://<myAppName>.azurewebsites.net`) and verify the app is running - take note of the title at the top of the page. Leave the browser open on the app for the next section.

Update and redeploy the app

1. In the Cloud Shell, type `code index.html` to open the editor. In the `<h1>` heading tag, change *Azure App Service - Sample Static HTML Site* to *Azure App Service Updated* - or to anything else that you'd like.
2. Use the commands **ctrl-s** to save and **ctrl-q** to exit.

3. Redeploy the app with the same `az webapp up` command. Be sure to use the same values for `<myLocation>` and `<myAppName>` as you used earlier.

```
az webapp up --location <myLocation> --name <myAppName> --html
```

Tip: You can use the up arrow on your keyboard to scroll through previous commands.

4. Once deployment is completed switch back to the browser from step 2 in the "Create the web app" section above and refresh the page.

Clean up resources

If you no longer need the resources you created in this exercise you can delete the resource group using the `az group delete` command below. Replace `<resource_group>` with resource group name you noted in step 1 of the "Create the web app" section above.

```
az group delete --name <resource_group> --no-wait
```

Knowledge check

Multiple choice

Which of the following App Service plans supports only function apps?

- Dedicated
- Isolated
- Consumption

Multiple choice

Which of the following networking features of App Service can be used to control outbound network traffic?

- App-assigned address
- Hybrid Connections
- Service endpoints

Summary

In this module, you learned how to:

- Describe Azure App Service key components and value.
- Explain how Azure App Service manages authentication and authorization.
- Identify methods to control inbound and outbound traffic to your web app.
- Deploy an app to App Service using Azure CLI commands.

Configure web app settings

Introduction

In App Service, app settings are variables passed as environment variables to the application code.

Learning objectives

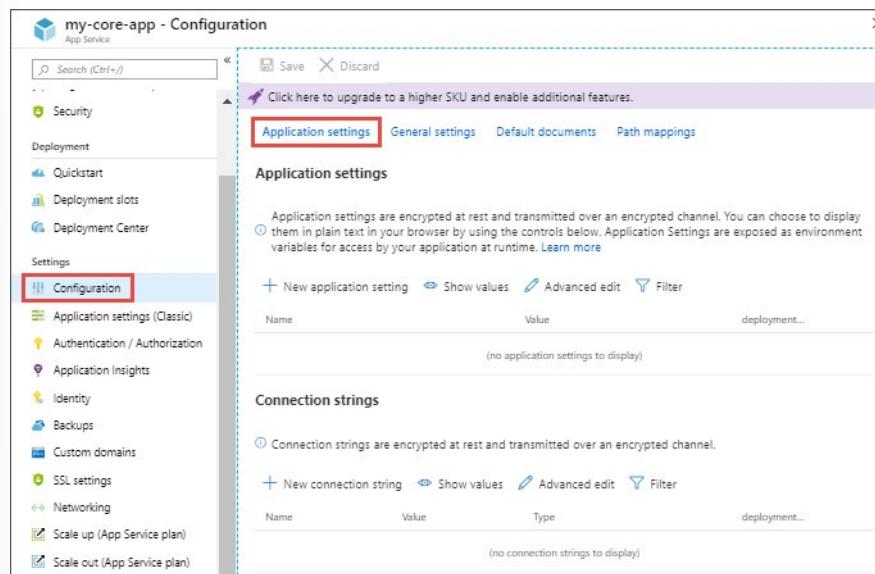
After completing this module, you'll be able to:

- Create application settings that are bound to deployment slots.
- Explain the options for installing SSL/TLS certificates for your app.
- Enable diagnostic logging for your app to aid in monitoring and debugging.
- Create virtual app to directory mappings.

Configure application settings

In App Service, app settings are variables passed as environment variables to the application code. For Linux apps and custom containers, App Service passes app settings to the container using the `--env` flag to set the environment variable in the container.

Application settings can be accessed by navigating to your app's management page and selecting **Configuration > Application Settings**.

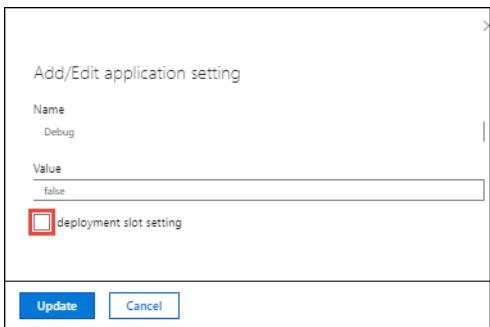


For ASP.NET and ASP.NET Core developers, setting app settings in App Service are like setting them in `<appSettings>` in `Web.config` or `appsettings.json`, but the values in App Service override the ones in `Web.config` or `appsettings.json`. You can keep development settings (for example, local MySQL password) in `Web.config` or `appsettings.json`, but production secrets (for example, Azure MySQL database password) safe in App Service. The same code uses your development settings when you debug locally, and it uses your production secrets when deployed to Azure.

App settings are always encrypted when stored (encrypted-at-rest).

Adding and editing settings

To add a new app setting, click **New application setting**. If you are using deployment slots you can specify if your setting is swappable or not. In the dialog, you can stick the setting to the current slot.



To edit a setting, click the **Edit** button on the right side.

When finished, click **Update**. Don't forget to click **Save** back in the **Configuration** page.

Note: In a default, or custom, Linux container any nested JSON key structure in the app setting name like ApplicationInsights:InstrumentationKey needs to be configured in App Service as ApplicationInsights__InstrumentationKey for the key name. In other words, any : should be replaced by __ (double underscore).

Editing application settings in bulk

To add or edit app settings in bulk, click the **Advanced** edit button. When finished, click **Update**. App settings have the following JSON formatting:

```
[  
  {  
    "name": "<key-1>",  
    "value": "<value-1>",  
    "slotSetting": false  
  },  
  {  
    "name": "<key-2>",  
    "value": "<value-2>",  
    "slotSetting": false  
  },  
  ...  
]
```

Configure connection strings

For ASP.NET and ASP.NET Core developers the values you set in App Service override the ones in *Web.config*. For other language stacks, it's better to use app settings instead, because connection strings require special formatting in the variable keys in order to access the values. Connection strings are always encrypted when stored (encrypted-at-rest).

Tip: There is one case where you may want to use connection strings instead of app settings for non-.NET languages: certain Azure database types are backed up along with the app only if you configure a connection string for the database in your App Service app.

Adding and editing connection strings follow the same principles as other app settings and they can also be tied to deployment slots. Below is an example of connection strings in JSON formatting that you would use for bulk adding or editing.

```
[  
  {  
    "name": "name-1",  
    "value": "conn-string-1",  
    "type": "SQLServer",  
    "slotSetting": false  
  },  
  {  
    "name": "name-2",  
    "value": "conn-string-2",  
    "type": "PostgreSQL",  
    "slotSetting": false  
  },  
  ...  
]
```

Configure general settings

In the **Configuration > General settings** section you can configure some common settings for your app. Some settings require you to scale up to higher pricing tiers.

Below is a list of the currently available settings:

- **Stack settings:** The software stack to run the app, including the language and SDK versions. For Linux apps and custom container apps, you can also set an optional start-up command or file.



- **Platform settings:** Lets you configure settings for the hosting platform, including:
 - **Bitness:** 32-bit or 64-bit.
 - **WebSocket protocol:** For ASP.NET SignalR or socket.io, for example.
 - **Always On:** Keep the app loaded even when there's no traffic. By default, **Always On** is not enabled and the app is unloaded after 20 minutes without any incoming requests. It's required for continuous WebJobs or for WebJobs that are triggered using a CRON expression.

- **Managed pipeline version:** The IIS pipeline mode. Set it to **Classic** if you have a legacy app that requires an older version of IIS.
- **HTTP version:** Set to 2.0 to enable support for HTTPS/2 protocol.
- **ARR affinity:** In a multi-instance deployment, ensure that the client is routed to the same instance for the life of the session. You can set this option to **Off** for stateless applications.
- **Debugging:** Enable remote debugging for ASP.NET, ASP.NET Core, or Node.js apps. This option turns off automatically after 48 hours.
- **Incoming client certificates:** require client certificates in mutual authentication. TLS mutual authentication is used to restrict access to your app by enabling different types of authentication for it.

Configure path mappings

In the **Configuration > Path mappings** section you can configure handler mappings, and virtual application and directory mappings. The **Path mappings** page will display different options based on the OS type.

Windows apps (uncontainerized)

For Windows apps, you can customize the IIS handler mappings and virtual applications and directories.

Handler mappings let you add custom script processors to handle requests for specific file extensions. To add a custom handler, select **New handler**. Configure the handler as follows:

- **Extension:** The file extension you want to handle, such as `*.php` or `handler.fcgi`.
- **Script processor:** The absolute path of the script processor. Requests to files that match the file extension are processed by the script processor. Use the path `D:\home\site\wwwroot` to refer to your app's root directory.
- **Arguments:** Optional command-line arguments for the script processor.

Each app has the default root path `(/)` mapped to `D:\home\site\wwwroot`, where your code is deployed by default. If your app root is in a different folder, or if your repository has more than one application, you can edit or add virtual applications and directories.

You can configure virtual applications and directories by specifying each virtual directory and its corresponding physical path relative to the website root (`D:\home`). To mark a virtual directory as a web application, clear the **Directory** check box.

Linux and containerized apps

You can add custom storage for your containerized app. Containerized apps include all Linux apps and also the Windows and Linux custom containers running on App Service. Click **New Azure Storage Mount** and configure your custom storage as follows:

- **Name:** The display name.
- **Configuration options:** Basic or Advanced.
- **Storage accounts:** The storage account with the container you want.
- **Storage type:** **Azure Blobs** or **Azure Files**. Windows container apps only support Azure Files.
- **Storage container:** For basic configuration, the container you want.
- **Share name:** For advanced configuration, the file share name.

- **Access key:** For advanced configuration, the access key.
- **Mount path:** The absolute path in your container to mount the custom storage.

Enable diagnostic logging

There are built-in diagnostics to assist with debugging an App Service app. In this lesson, you will learn how to enable diagnostic logging and add instrumentation to your application, as well as how to access the information logged by Azure.

The table below shows the types of logging, the platforms supported, and where the logs can be stored and located for accessing the information.

Type	Platform	Location	Description
Application logging	Windows, Linux	App Service file system and/or Azure Storage blobs	Logs messages generated by your application code. The messages can be generated by the web framework you choose, or from your application code directly using the standard logging pattern of your language. Each message is assigned one of the following categories: Critical, Error, Warning, Info, Debug, and Trace .
Web server logging	Windows	App Service file system or Azure Storage blobs	Raw HTTP request data in the W3C extended log file format. Each log message includes data like the HTTP method, resource URI, client IP, client port, user agent, response code, and so on.
Detailed error logging	Windows	App Service file system	Copies of the .htm error pages that would have been sent to the client browser. For security reasons, detailed error pages shouldn't be sent to clients in production, but App Service can save the error page each time an application error occurs that has HTTP code 400 or greater.

Type	Platform	Location	Description
Failed request tracing	Windows	App Service file system	Detailed tracing information on failed requests, including a trace of the IIS components used to process the request and the time taken in each component. One folder is generated for each failed request, which contains the XML log file, and the XSL stylesheet to view the log file with.
Deployment logging	Windows, Linux	App Service file system	Helps determine why a deployment failed. Deployment logging happens automatically and there are no configurable settings for deployment logging.

Enable application logging (Windows)

1. To enable application logging for Windows apps in the Azure portal, navigate to your app and select **App Service logs**.
2. Select **On** for either **Application Logging (Filesystem)** or **Application Logging (Blob)**, or both. The **Filesystem** option is for temporary debugging purposes, and turns itself off in 12 hours. The **Blob** option is for long-term logging, and needs a blob storage container to write logs to.
3. You can also set the **Level** of details included in the log as shown in the table below.

Level	Included categories
Disabled	None
Error	Error, Critical
Warning	Warning, Error, Critical
Information	Info, Warning, Error, Critical
Verbose	Trace, Debug, Info, Warning, Error, Critical (all categories)

4. When finished, select **Save**.

Enable application logging (Linux/Container)

1. In **App Service logs** set the **Application logging** option to **File System**.
2. In **Quota (MB)**, specify the disk quota for the application logs. In **Retention Period (Days)**, set the number of days the logs should be retained.
3. When finished, select **Save**.

Enable web server logging

1. For **Web server logging**, select **Storage** to store logs on blob storage, or **File System** to store logs on the App Service file system.
2. In **Retention Period (Days)**, set the number of days the logs should be retained.
3. When finished, select **Save**.

Add log messages in code

In your application code, you use the usual logging facilities to send log messages to the application logs. For example:

- ASP.NET applications can use the `System.Diagnostics.Trace` class to log information to the application diagnostics log. For example:

```
System.Diagnostics.Trace.TraceError("If you're seeing this, something bad happened");
```

- By default, ASP.NET Core uses the `Microsoft.Extensions.Logging.AzureAppServices` logging provider.

Stream logs

Before you stream logs in real time, enable the log type that you want. Any information written to files ending in .txt, .log, or .htm that are stored in the `/LogFiles` directory (`d:/home/logfiles`) is streamed by App Service.

Note: Some types of logging buffer write to the log file, which can result in out of order events in the stream. For example, an application log entry that occurs when a user visits a page may be displayed in the stream before the corresponding HTTP log entry for the page request.

- Azure portal - To stream logs in the Azure portal, navigate to your app and select **Log stream**.
- Azure CLI - To stream logs live in Cloud Shell, use the following command:
`az webapp log tail --name appname --resource-group myResourceGroup`
- Local console - To stream logs in the local console, install Azure CLI and sign in to your account. Once signed in, follow the instructions for Azure CLI above.

Access log files

If you configure the Azure Storage blobs option for a log type, you need a client tool that works with Azure Storage.

For logs stored in the App Service file system, the easiest way is to download the ZIP file in the browser at:

- Linux/container apps: `https://<app-name>.scm.azurewebsites.net/api/logs/docker/zip`
- Windows apps: `https://<app-name>.scm.azurewebsites.net/api/dump`

For Linux/container apps, the ZIP file contains console output logs for both the docker host and the docker container. For a scaled-out app, the ZIP file contains one set of logs for each instance. In the App Service file system, these log files are the contents of the `/home/LogFiles` directory.

Configure security certificates

You have been asked to help secure information being transmitted between your company's app and the customer. Azure App Service has tools that let you create, upload, or import a private certificate or a public certificate into App Service.

A certificate uploaded into an app is stored in a deployment unit that is bound to the app service plan's resource group and region combination (internally called a *webspace*). This makes the certificate accessible to other apps in the same resource group and region combination.

The table below details the options you have for adding certificates in App Service:

Option	Description
Create a free App Service managed certificate	A private certificate that's free of charge and easy to use if you just need to secure your custom domain in App Service.
Purchase an App Service certificate	A private certificate that's managed by Azure. It combines the simplicity of automated certificate management and the flexibility of renewal and export options.
Import a certificate from Key Vault	Useful if you use Azure Key Vault to manage your certificates.
Upload a private certificate	If you already have a private certificate from a third-party provider, you can upload it.
Upload a public certificate	Public certificates are not used to secure custom domains, but you can load them into your code if you need them to access remote resources.

Private certificate requirements

The free **App Service managed certificate** and the **App Service certificate** already satisfy the requirements of App Service. If you want to use a private certificate in App Service, your certificate must meet the following requirements:

- Exported as a password-protected PFX file, encrypted using triple DES.
- Contains private key at least 2048 bits long
- Contains all intermediate certificates in the certificate chain

To secure a custom domain in a TLS binding, the certificate has additional requirements:

- Contains an Extended Key Usage for server authentication (OID = 1.3.6.1.5.5.7.3.1)
- Signed by a trusted certificate authority

Creating a free managed certificate

To create custom TLS/SSL bindings or enable client certificates for your App Service app, your App Service plan must be in the **Basic**, **Standard**, **Premium**, or **Isolated** tier. Custom SSL is not supported in the **F1** or **D1** tier.

The free App Service managed certificate is a turn-key solution for securing your custom DNS name in App Service. It's a TLS/SSL server certificate that's fully managed by App Service and renewed continu-

ously and automatically in six-month increments, 45 days before expiration. You create the certificate and bind it to a custom domain, and let App Service do the rest.

The free certificate comes with the following limitations:

- Does not support wildcard certificates.
- Does not support usage as a client certificate by certificate thumbprint.
- Is not exportable.
- Is not supported on App Service Environment (ASE).
- Is not supported with root domains that are integrated with Traffic Manager.
- If a certificate is for a CNAME-mapped domain, the CNAME must be mapped directly to <app-name>.azurewebsites.net.

Import an App Service Certificate

If you purchase an App Service Certificate from Azure, Azure manages the following tasks:

- Takes care of the purchase process from GoDaddy.
- Performs domain verification of the certificate.
- Maintains the certificate in Azure Key Vault.
- Manages certificate renewal.
- Synchronizes the certificate automatically with the imported copies in App Service apps.

If you already have a working App Service certificate, you can:

- Import the certificate into App Service.
- Manage the certificate, such as renew, rekey, and export it.

Note: App Service Certificates are not supported in Azure National Clouds at this time.

Upload a private certificate

If your certificate authority gives you multiple certificates in the certificate chain, you need to merge the certificates in order. Then you can Export your merged TLS/SSL certificate with the private key that your certificate request was generated with.

If you generated your certificate request using OpenSSL, then you have created a private key file. To export your certificate to PFX, run the following command. Replace the placeholders <private-key-file> and <merged-certificate-file> with the paths to your private key and your merged certificate file.

```
openssl pkcs12 -export -out myserver.pfx -inkey <private-key-file> -in  
<merged-certificate-file>
```

When prompted, define an export password. You'll use this password when uploading your TLS/SSL certificate to App Service.

Enforce HTTPS

By default, anyone can still access your app using HTTP. You can redirect all HTTP requests to the HTTPS port by navigating to your app page and, in the left navigation, select **TLS/SSL settings**. Then, in **HTTPS Only**, select **On**.

The screenshot shows the 'TLS/SSL settings' blade in the Azure portal. The left sidebar has a red box around 'TLS/SSL settings'. The top navigation bar has a red box around 'Bindings'. In the 'Protocol Settings' section, the 'HTTPS Only' switch is highlighted with a red box and set to 'On'. Below it, the 'Minimum TLS Version' dropdown shows '1.0' selected. In the 'TLS/SSL bindings' section, there is a note about specifying certificates for specific hostnames, a 'Learn more' link, and a 'Add TLS/SSL Binding' button. A note at the bottom says 'No TLS/SSL bindings configured for the app.'

Manage app features

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 additional 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:

```
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:

```
bool featureFlag = true;
```

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

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

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

```
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:

```
"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.

Knowledge check

Multiple choice

In which of the app configuration settings categories below would you set the language and SDK version?

- Application settings
- Path mappings
- General settings

Multiple choice

Which of the following types of application logging is supported on the Linux platform?

- Web server logging
- Failed request tracing
- Deployment logging

Multiple choice

Which of the following choices correctly lists the two parts of a feature flag?

- Name, App Settings
- Name, one or more filters
- Feature manager, one or more filters

Summary

In this module, you learned how to:

- Create application settings that are bound to deployment slots.
- Explain the options for installing SSL/TLS certificates for your app.

- Enable diagnostic logging for your app to aid in monitoring and debugging.
- Create virtual app to directory mappings.

Scale apps in Azure App Service

Introduction

Autoscaling enables a system to adjust the resources required to meet the varying demand from users, while controlling the costs associated with these resources. You can use autoscaling with many Azure services, including web applications. Autoscaling requires you to configure autoscale rules that specify the conditions under which resources should be added or removed.

Learning objectives

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

- Identify scenarios for which autoscaling is an appropriate solution
- Create autoscaling rules for a web app
- Monitor the effects of autoscaling

Examine autoscale factors

Autoscaling can be triggered according to a schedule, or by assessing whether the system is running short on resources. For example, autoscaling could be triggered if CPU utilization grows, memory occupancy increases, the number of incoming requests to a service appears to be surging, or some combination of factors.

What is autoscaling?

Autoscaling is a cloud system or process that adjusts available resources based on the current demand. Autoscaling performs scaling *in and out*, as opposed to scaling *up and down*.

Azure App Service Autoscaling

Autoscaling in Azure App Service monitors the resource metrics of a web app as it runs. It detects situations where additional resources are required to handle an increasing workload, and ensures those resources are available before the system becomes overloaded.

Autoscaling responds to changes in the environment by adding or removing web servers and balancing the load between them. Autoscaling doesn't have any effect on the CPU power, memory, or storage capacity of the web servers powering the app, it only changes the number of web servers.

Autoscaling rules

Autoscaling makes its decisions based on rules that you define. A rule specifies the threshold for a metric, and triggers an autoscale event when this threshold is crossed. Autoscaling can also deallocate resources when the workload has diminished.

Define your autoscaling rules carefully. For example, a Denial of Service attack will likely result in a large-scale influx of incoming traffic. Trying to handle a surge in requests caused by a DoS attack would be fruitless and expensive. These requests aren't genuine, and should be discarded rather than processed. A better solution is to implement detection and filtering of requests that occur during such an attack before they reach your service.

When should you consider autoscaling?

Autoscaling provides elasticity for your services. It's a suitable solution when hosting any application when you can't easily predict the workload in advance, or when the workload is likely to vary by date or time. For example, you might expect increased/reduced activity for a business app during holidays.

Autoscaling improves availability and fault tolerance. It can help ensure that client requests to a service won't be denied because an instance is either not able to acknowledge the request in a timely manner, or because an overloaded instance has crashed.

Autoscaling works by adding or removing web servers. If your web apps perform resource-intensive processing as part of each request, then autoscaling might not be an effective approach. In these situations, manually scaling up may be necessary. For example, if a request sent to a web app involves performing complex processing over a large dataset, depending on the instance size, this single request could exhaust the processing and memory capacity of the instance.

Autoscaling isn't the best approach to handling long-term growth. You might have a web app that starts with a small number of users, but increases in popularity over time. Autoscaling has an overhead associated with monitoring resources and determining whether to trigger a scaling event. In this scenario, if you can anticipate the rate of growth, manually scaling the system over time may be a more cost effective approach.

The number of instances of a service is also a factor. You might expect to run only a few instances of a service most of the time. However, in this situation, your service will always be susceptible to downtime or lack of availability whether autoscaling is enabled or not. The fewer the number of instances initially, the less capacity you have to handle an increasing workload while autoscaling spins up additional instances.

Identify autoscale factors

Autoscaling enables you to specify the conditions under which a web app should be scaled out, and back in again. Effective autoscaling ensures sufficient resources are available to handle large volumes of requests at peak times, while managing costs when the demand drops.

You can configure autoscaling to detect when to scale in and out according to a combination of factors, based on resource usage. You can also configure autoscaling to occur according to a schedule.

In this unit, you'll learn how to specify the factors that can be used to autoscale a service.

Autoscaling and the App Service Plan

Autoscaling is a feature of the App Service Plan used by the web app. When the web app scales out, Azure starts new instances of the hardware defined by the App Service Plan to the app.

To prevent runaway autoscaling, an App Service Plan has an instance limit. Plans in more expensive pricing tiers have a higher limit. Autoscaling cannot create more instances than this limit.

Note: Not all App Service Plan pricing tiers support autoscaling.

Autoscale conditions

You indicate how to autoscale by creating autoscale conditions. Azure provides two options for autoscaling:

- Scale based on a metric, such as the length of the disk queue, or the number of HTTP requests awaiting processing.

- Scale to a specific instance count according to a schedule. For example, you can arrange to scale out at a particular time of day, or on a specific date or day of the week. You also specify an end date, and the system will scale back in at this time.

Scaling to a specific instance count only enables you to scale out to a defined number of instances. If you need to scale out incrementally, you can combine metric and schedule-based autoscaling in the same autoscale condition. So, you could arrange for the system to scale out if the number of HTTP requests exceeds some threshold, but only between certain hours of the day.

You can create multiple autoscale conditions to handle different schedules and metrics. Azure will autoscale your service when any of these conditions apply. An App Service Plan also has a default condition that will be used if none of the other conditions are applicable. This condition is always active and doesn't have a schedule.

Metrics for autoscale rules

Autoscaling by metric requires that you define one or more autoscale rules. An autoscale rule specifies a metric to monitor, and how autoscaling should respond when this metric crosses a defined threshold. The metrics you can monitor for a web app are:

- **CPU Percentage.** This metric is an indication of the CPU utilization across all instances. A high value shows that instances are becoming CPU-bound, which could cause delays in processing client requests.
- **Memory Percentage.** This metric captures the memory occupancy of the application across all instances. A high value indicates that free memory could be running low, and could cause one or more instances to fail.
- **Disk Queue Length.** This metric is a measure of the number of outstanding I/O requests across all instances. A high value means that disk contention could be occurring.
- **Http Queue Length.** This metric shows how many client requests are waiting for processing by the web app. If this number is large, client requests might fail with HTTP 408 (Timeout) errors.
- **Data In.** This metric is the number of bytes received across all instances.
- **Data Out.** This metric is the number of bytes sent by all instances.

You can also scale based on metrics for other Azure services. For example, if the web app processes requests received from a Service Bus Queue, you might want to spin up additional instances of a web app if the number of items held in an Azure Service Bus Queue exceeds a critical length.

How an autoscale rule analyzes metrics

Autoscaling works by analyzing trends in metric values over time across all instances. Analysis is a multi-step process.

In the first step, an autoscale rule aggregates the values retrieved for a metric for all instances across a period of time known as the *time grain*. Each metric has its own intrinsic time grain, but in most cases this period is 1 minute. The aggregated value is known as the *time aggregation*. The options available are *Average*, *Minimum*, *Maximum*, *Total*, *Last*, and *Count*.

An interval of one minute is a very short interval in which to determine whether any change in metric is long-lasting enough to make autoscaling worthwhile. So, an autoscale rule performs a second step that performs a further aggregation of the value calculated by the *time aggregation* over a longer, user-specified period, known as the *Duration*. The minimum *Duration* is 5 minutes. If the *Duration* is set to 10 minutes for example, the autoscale rule will aggregate the 10 values calculated for the *time grain*.

The aggregation calculation for the *Duration* can be different for that of the *time grain*. For example, if the *time aggregation* is *Average* and the statistic gathered is *CPU Percentage* across a one-minute *time grain*, each minute the average CPU percentage utilization across all instances for that minute will be calculated. If the *time grain statistic* is set to *Maximum*, and the *Duration* of the rule is set to 10 minutes, the maximum of the 10 average values for the CPU percentage utilization will be used to determine whether the rule threshold has been crossed.

Autoscale actions

When an autoscale rule detects that a metric has crossed a threshold, it can perform an autoscale action. An autoscale action can be *scale-out* or *scale-in*. A scale-out action increases the number of instances, and a scale-in action reduces the instance count. An autoscale action uses an operator (such as *less than*, *greater than*, *equal to*, and so on) to determine how to react to the threshold. Scale-out actions typically use the *greater than* operator to compare the metric value to the threshold. Scale-in actions tend to compare the metric value to the threshold with the *less than* operator. An autoscale action can also set the instance count to a specific level, rather than incrementing or decrementing the number available.

An autoscale action has a *cool down* period, specified in minutes. During this interval, the scale rule won't be triggered again. This is to allow the system to stabilize between autoscale events. Remember that it takes time to start up or shut down instances, and so any metrics gathered might not show any significant changes for several minutes. The minimum cool down period is five minutes.

Pairing autoscale rules

You should plan for scaling-in when a workload decreases. Consider defining autoscale rules in pairs in the same autoscale condition. One autoscale rule should indicate how to scale the system out when a metric exceeds an upper threshold. Then other rule should define how to scale the system back in again when the same metric drops below a lower threshold.

Combining autoscale rules

A single autoscale condition can contain several autoscale rules (for example, a scale-out rule and the corresponding scale-in rule). However, the autoscale rules in an autoscale condition don't have to be directly related. You could define the following four rules in the same autoscale condition:

- If the HTTP queue length exceeds 10, scale out by 1
- If the CPU utilization exceeds 70%, scale out by 1
- If the HTTP queue length is zero, scale in by 1
- If the CPU utilization drops below 50%, scale in by 1

When determining whether to scale out, the autoscale action will be performed if **any** of the scale-out rules are met (HTTP queue length exceeds 10 **or** CPU utilization exceeds 70%). When scaling in, the autoscale action will run **only if all** of the scale-in rules are met (HTTP queue length drops to zero **and** CPU utilization falls below 50%). If you need to scale in if only one the scale-in rules are met, you must define the rules in separate autoscale conditions.

Enable autoscale in App Service

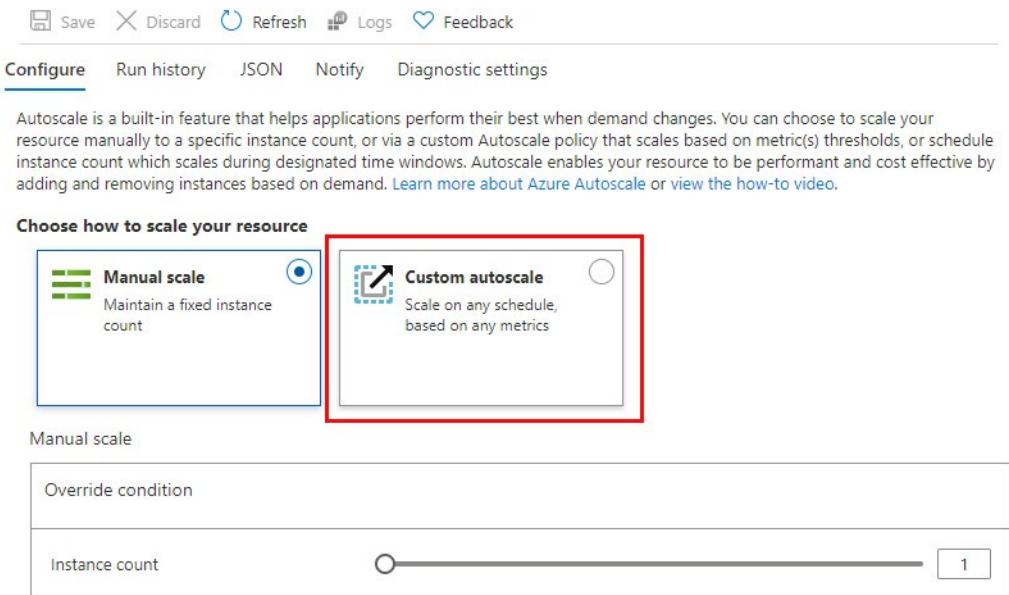
In this unit, you will learn how to enable autoscaling, create autoscale rules, and monitor autoscaling activity

Enable autoscaling

To get started with autoscaling navigate to your App Service plan in the Azure portal and select **Scale out (App Service plan)** in the **Settings** group in the left navigation pane.

Note: Not all pricing tiers support autoscaling. The development pricing tiers are either limited to a single instance (the **F1** and **D1** tiers), or they only provide manual scaling (the **B1** tier). If you've selected one of these tiers, you must first scale up to the **S1** or any of the **P** level production tiers.

By default, an App Service Plan only implements manual scaling. Selecting **Custom autoscale** reveals condition groups you can use to manage your scale settings.



Add scale conditions

Once you enable autoscaling, you can edit the automatically created default scale condition, and you can add your own custom scale conditions. Remember that each scale condition can either scale based on a metric, or scale to a specific instance count.

The Default scale condition is executed when none of the other scale conditions are active.

The screenshot shows the AWS Lambda Metrics-based Scale Condition configuration page. It displays two scale conditions:

- Default***: Auto created default scale condition. This condition is set to "Scale to a specific instance count" with a value of 1. A note below states: "This scale condition is executed when none of the other scale condition(s) match".
- Auto created scale condition 1**: This condition is set to "Scale based on a metric". It includes a note: "No metric rules defined; click Add a rule to scale out and scale in your instances based on rules. For example: 'Add a rule that increases instance count by 1 when CPU percentage is above 70%'. If you save the setting without any rules defined, no scaling will occur." Below this note is a "+ Add a rule" link. The "Instance limits" section shows values: Minimum 1, Maximum 2, and Default 1. The "Schedule" section shows "Specify start/end dates" selected, with a dropdown for "Timezone" set to "(UTC-08:00) Pacific Time (US & Canada)". The "Start date" is 07/17/2021 at 12:00:00 AM, and the "End date" is 07/17/2021 at 11:59:00 PM.

A metric-based scale condition can also specify the minimum and maximum number of instances to create. The maximum number can't exceed the limits defined by the pricing tier. Additionally, all scale conditions other than the default may include a schedule indicating when the condition should be applied.

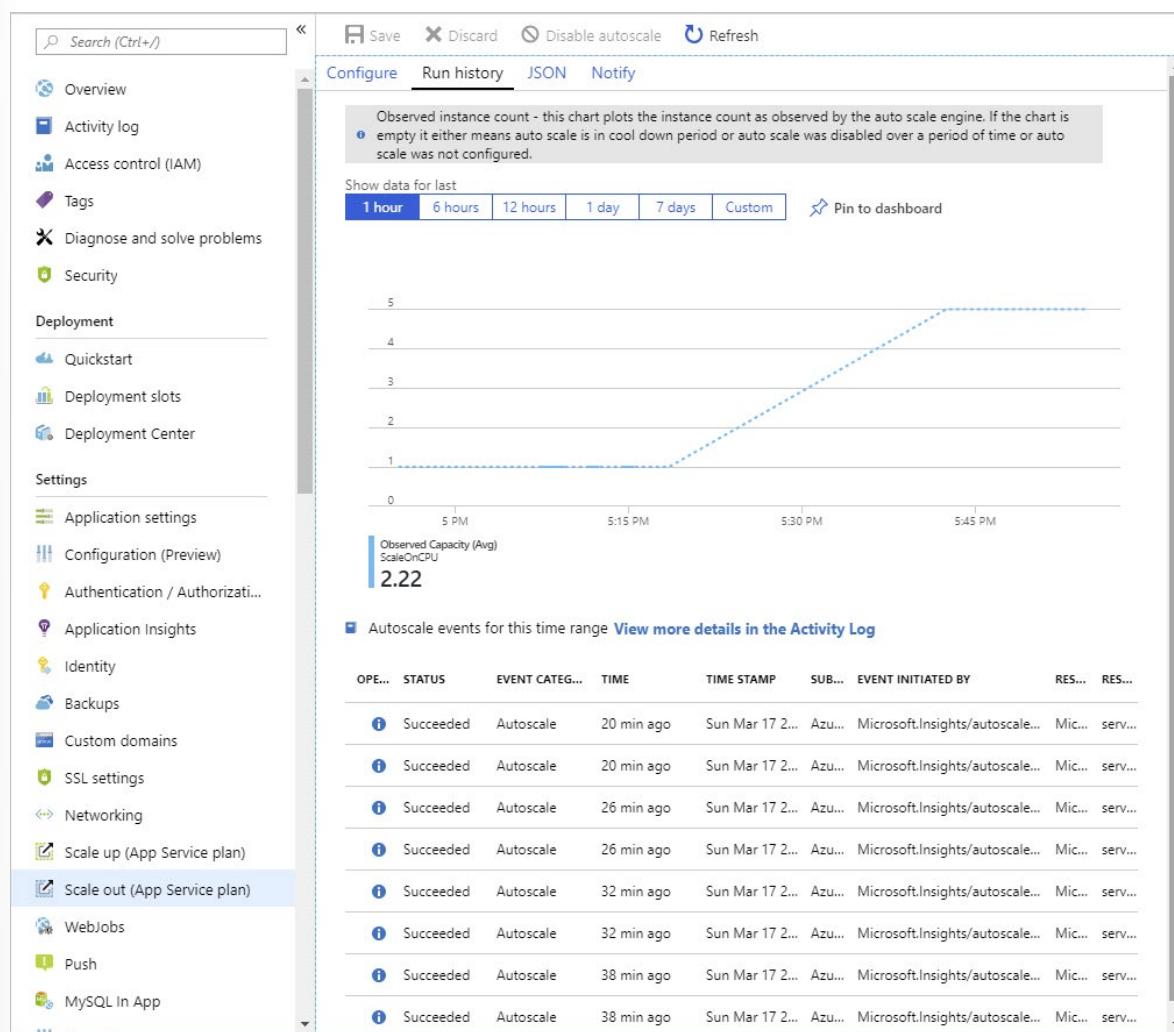
Create scale rules

A metric-based scale condition contains one or more scale rules. You use the **Add a rule** link to add your own custom rules. You define the criteria that indicate when a rule should trigger an autoscale action, and the autoscale action to be performed (scale out or scale in) using the metrics, aggregations, operators, and thresholds described earlier.

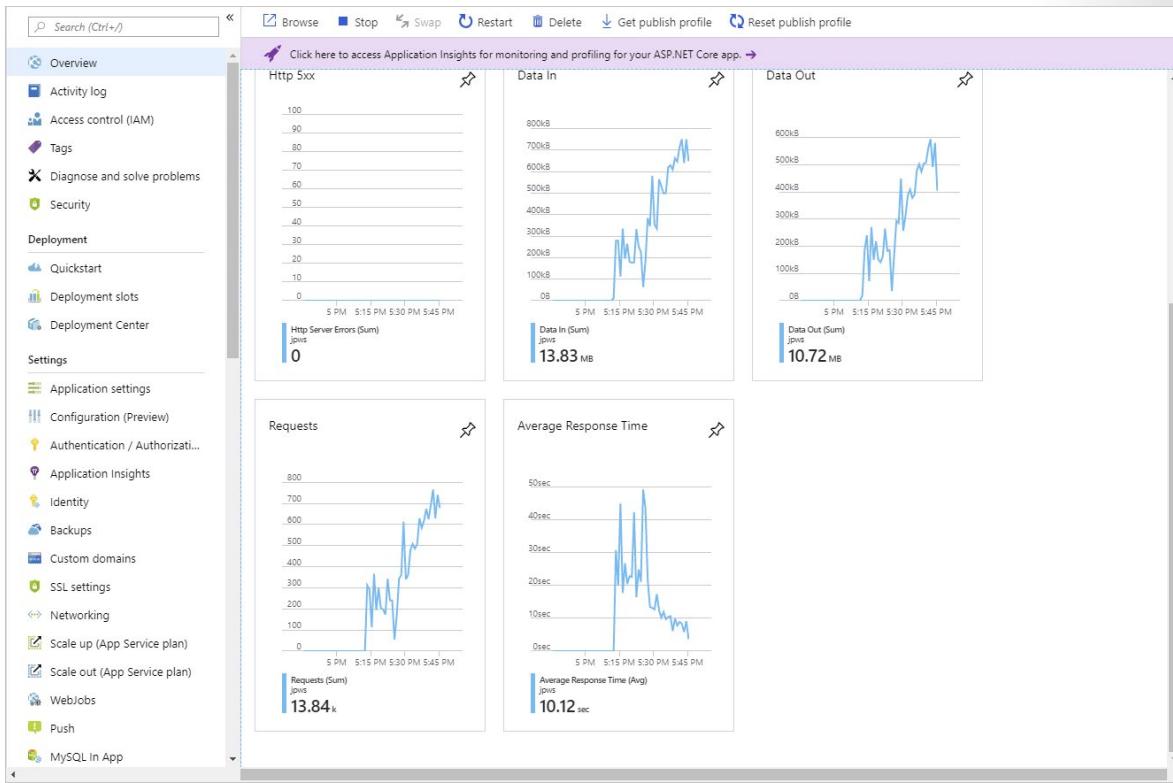
The screenshot shows the Azure portal interface for managing an App Service plan. On the left, there's a navigation bar with Save, Discard, Refresh, Logs, and Feedback buttons. Below that, resource details are shown: Resource group (game), Instance count (1). Under the 'Default' section, it says 'Auto created default scale condition'. It shows 'Scale mode' as 'Scale based on a metric' (selected) and 'Scale to a specific instance count' (unchecked). The 'Instance count*' field is set to 1. A note below states: 'This scale condition is executed when none of the other scale condition(s) match'. In the 'Auto created scale condition 1' section, the 'Scale mode' is also set to 'Scale based on a metric'. A red box highlights the 'Add a rule' button, which is described as 'No metric rules defined; click Add a rule to scale out and scale in your instances based on example: Add a rule that increases instance count by 1 when CPU percentage is above 70%'. The 'Add a rule' button is also highlighted with a red box. The 'Scale rule' modal window is open on the right, showing the configuration for the scale condition. It includes fields for Metric source (Current resource), Resource type (App Service plans), Metric namespace (App Service plans standard metrics), Metric name (CPU Percentage), and various operators and thresholds for scaling.

Monitor autoscaling activity

The Azure portal enables you to track when autoscaling has occurred through the **Run history** chart. This chart shows how the number of instances varies over time, and which autoscale conditions caused each change.



You can use the **Run history** chart in conjunction with the metrics shown on the **Overview** page to correlate the autoscaling events with resource utilization.



Explore autoscale best practices

If you're not following good practices when creating autoscale settings you can create conditions that lead to undesirable results. In this unit you will learn how to avoid creating rules that conflict with each other.

Autoscale concepts

- An autoscale setting scales instances horizontally, which is *out* by increasing the instances and *in* by decreasing the number of instances. An autoscale setting has a maximum, minimum, and default value of instances.
- An autoscale job always reads the associated metric to scale by, checking if it has crossed the configured threshold for scale-out or scale-in.
- All thresholds are calculated at an instance level. For example, "scale out by one instance when average CPU > 80% when instance count is 2", means scale-out when the average CPU across all instances is greater than 80%.
- All autoscale successes and failures are logged to the Activity Log. You can then configure an activity log alert so that you can be notified via email, SMS, or webhooks whenever there is activity.

Autoscale best practices

Use the following best practices as you create your autoscale rules.

Ensure the maximum and minimum values are different and have an adequate margin between them

If you have a setting that has `minimum=2, maximum=2` and the current instance count is 2, no scale action can occur. Keep an adequate margin between the maximum and minimum instance counts, which are inclusive. Autoscale always scales between these limits.

Choose the appropriate statistic for your diagnostics metric

For diagnostics metrics, you can choose among *Average*, *Minimum*, *Maximum* and *Total* as a metric to scale by. The most common statistic is *Average*.

Choose the thresholds carefully for all metric types

We recommend carefully choosing different thresholds for scale-out and scale-in based on practical situations.

We *do not recommend* autoscale settings like the examples below with the same or very similar threshold values for out and in conditions:

- Increase instances by 1 count when Thread Count ≥ 600
- Decrease instances by 1 count when Thread Count ≤ 600

Let's look at an example of what can lead to a behavior that may seem confusing. Consider the following sequence.

1. Assume there are two instances to begin with and then the average number of threads per instance grows to 625.
2. Autoscale scales out adding a third instance.
3. Next, assume that the average thread count across instance falls to 575.
4. Before scaling in, autoscale tries to estimate what the final state will be if it scaled in. For example, 575×3 (current instance count) = $1,725 / 2$ (final number of instances when scaled in) = 862.5 threads. This means autoscale would have to immediately scale-out again even after it scaled in, if the average thread count remains the same or even falls only a small amount. However, if it scaled out again, the whole process would repeat, leading to an infinite loop.
5. To avoid this situation (termed "flapping"), autoscale does not scale in at all. Instead, it skips and reevaluates the condition again the next time the service's job executes. This can confuse many people because autoscale wouldn't appear to work when the average thread count was 575.

Estimation during a scale-in is intended to avoid "flapping" situations, where scale-in and scale-out actions continually go back and forth. Keep this behavior in mind when you choose the same thresholds for scale-out and in.

We recommend choosing an adequate margin between the scale-out and in thresholds. As an example, consider the following better rule combination.

- Increase instances by 1 count when CPU% ≥ 80
- Decrease instances by 1 count when CPU% ≤ 60

In this case

1. Assume there are 2 instances to start with.
2. If the average CPU% across instances goes to 80, autoscale scales out adding a third instance.

3. Now assume that over time the CPU% falls to 60.
4. Autoscale's scale-in rule estimates the final state if it were to scale-in. For example, 60×3 (current instance count) = $180 / 2$ (final number of instances when scaled in) = 90. So autoscale does not scale-in because it would have to scale-out again immediately. Instead, it skips scaling in.
5. The next time autoscale checks, the CPU continues to fall to 50. It estimates again - 50×3 instance = $150 / 2$ instances = 75, which is below the scale-out threshold of 80, so it scales in successfully to 2 instances.

Considerations for scaling when multiple rules are configured in a profile

There are cases where you may have to set multiple rules in a profile. The following set of autoscale rules are used by services when multiple rules are set.

On *scale-out*, autoscale runs if any rule is met. On *scale-in*, autoscale require all rules to be met.

To illustrate, assume that you have the following four autoscale rules:

- If CPU < 30 %, scale-in by 1
- If Memory < 50%, scale-in by 1
- If CPU > 75%, scale-out by 1
- If Memory > 75%, scale-out by 1

Then the follow occurs:

- If CPU is 76% and Memory is 50%, we scale-out.
- If CPU is 50% and Memory is 76% we scale-out.

On the other hand, if CPU is 25% and memory is 51% autoscale does not scale-in. In order to scale-in, CPU must be 29% and Memory 49%.

Always select a safe default instance count

The default instance count is important autoscale scales your service to that count when metrics are not available. Therefore, select a default instance count that's safe for your workloads.

Configure autoscale notifications

Autoscale will post to the Activity Log if any of the following conditions occur:

- Autoscale issues a scale operation
- Autoscale service successfully completes a scale action
- Autoscale service fails to take a scale action.
- Metrics are not available for autoscale service to make a scale decision.
- Metrics are available (recovery) again to make a scale decision.

You can also use an Activity Log alert to monitor the health of the autoscale engine. In addition to using activity log alerts, you can also configure email or webhook notifications to get notified for successful scale actions via the notifications tab on the autoscale setting.

Knowledge check

Multiple choice

Which of these statements best describes autoscaling?

- Autoscaling requires an administrator to actively monitor the workload on a system.
- Autoscaling is a scale out/scale in solution.
- Scaling up/scale down provides better availability than autoscaling.

Multiple choice

Which of these scenarios is a suitable candidate for autoscaling?

- The number of users requiring access to an application varies according to a regular schedule. For example, more users use the system on a Friday than other days of the week.
- The system is subject to a sudden influx of requests that grinds your system to a halt.
- Your organization is running a promotion and expects to see increased traffic to their web site for the next couple of weeks.

Multiple choice

There are multiple rules in an autoscale profile. Which of the following scale operations will run if any of the rule conditions are met?

- scale-out
- scale-in
- scale-out/in

Summary

In this module, you learned how to:

- Identify scenarios for which autoscaling is an appropriate solution
- Create autoscaling rules for a web app
- Monitor the effects of autoscaling

Explore Azure App Service deployment slots

Introduction

The deployment slot functionality in App Service is a powerful tool that enables you to preview, manage, test, and deploy your different development environments.

Learning objectives

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

- Describe the benefits of using deployment slots
- Understand how slot swapping operates in App Service
- Perform manual swaps and enable auto swap
- Route traffic manually and automatically

Prerequisites

- Experience using the Azure portal to create and manage App Service web apps

Explore staging environments

When you deploy your web app, web app on Linux, mobile back end, or API app to Azure App Service, you can use a separate deployment slot instead of the default production slot when you're running in the **Standard**, **Premium**, or **Isolated** App Service plan tier. Deployment slots are live apps with their own host names. App content and configurations elements can be swapped between two deployment slots, including the production slot.

Deploying your application to a non-production slot has the following benefits:

- You can validate app changes in a staging deployment slot before swapping it with the production slot.
- Deploying an app to a slot first and swapping it into production makes sure that all instances of the slot are warmed up before being swapped into production. This eliminates downtime when you deploy your app. The traffic redirection is seamless, and no requests are dropped because of swap operations. You can automate this entire workflow by configuring auto swap when pre-swap validation isn't needed.
- After a swap, the slot with previously staged app now has the previous production app. If the changes swapped into the production slot aren't as you expect, you can perform the same swap immediately to get your "last known good site" back.

Each App Service plan tier supports a different number of deployment slots. There's no additional charge for using deployment slots. To find out the number of slots your app's tier supports, visit [App Service limits²](#).

To scale your app to a different tier, make sure that the target tier supports the number of slots your app already uses. For example, if your app has more than five slots, you can't scale it down to the **Standard** tier, because the **Standard** tier supports only five deployment slots.

² <https://docs.microsoft.com/azure/azure-resource-manager/management/azure-subscription-service-limits#app-service-limits>

When you create a new slot the new deployment slot has no content, even if you clone the settings from a different slot. You can deploy to the slot from a different repository branch or a different repository.

Examine slot swapping

When you swap slots (for example, from a staging slot to the production slot), App Service does the following to ensure that the target slot doesn't experience downtime:

1. Apply the following settings from the target slot (for example, the production slot) to all instances of the source slot:
 - Slot-specific app settings and connection strings, if applicable.
 - Continuous deployment settings, if enabled.
 - App Service authentication settings, if enabled.Any of these cases trigger all instances in the source slot to restart. During **swap with preview**, this marks the end of the first phase. The swap operation is paused, and you can validate that the source slot works correctly with the target slot's settings.
2. Wait for every instance in the source slot to complete its restart. If any instance fails to restart, the swap operation reverts all changes to the source slot and stops the operation.
3. If local cache is enabled, trigger local cache initialization by making an HTTP request to the application root ("/") on each instance of the source slot. Wait until each instance returns any HTTP response. Local cache initialization causes another restart on each instance.
4. If auto swap is enabled with custom warm-up, trigger Application Initiation by making an HTTP request to the application root ("/") on each instance of the source slot.
 - If `applicationInitialization` isn't specified, trigger an HTTP request to the application root of the source slot on each instance.
 - If an instance returns any HTTP response, it's considered to be warmed up.
5. If all instances on the source slot are warmed up successfully, swap the two slots by switching the routing rules for the two slots. After this step, the target slot (for example, the production slot) has the app that's previously warmed up in the source slot.
6. Now that the source slot has the pre-swap app previously in the target slot, perform the same operation by applying all settings and restarting the instances.

At any point of the swap operation, all work of initializing the swapped apps happens on the source slot. The target slot remains online while the source slot is being prepared and warmed up, regardless of where the swap succeeds or fails. To swap a staging slot with the production slot, make sure that the production slot is always the target slot. This way, the swap operation doesn't affect your production app.

When you clone configuration from another deployment slot, the cloned configuration is editable. Some configuration elements follow the content across a swap (not slot specific), whereas other configuration elements stay in the same slot after a swap (slot specific). The following table shows the settings that change when you swap slots.

Settings that are swapped	Settings that aren't swapped
General settings, such as framework version, 32/64-bit, web sockets	Publishing endpoints
App settings (can be configured to stick to a slot)	Custom domain names

Settings that are swapped	Settings that aren't swapped
Connection strings (can be configured to stick to a slot)	Non-public certificates and TLS/SSL settings
Handler mappings	Scale settings
Public certificates	WebJobs schedulers
WebJobs content	IP restrictions
Hybrid connections *	Always On
Virtual network integration *	Diagnostic log settings
Service endpoints *	Cross-origin resource sharing (CORS)
Azure Content Delivery Network *	

Features marked with an asterisk (*) are planned to be unswapped.

Note: To make settings swappable, add the app setting `WEBSITE_OVERRIDE_DEFAULT_STICKY_SLOT_SETTINGS` in every slot of the app and set its value to `0` or `false`. These settings are either all swappable or not at all. You can't make just some settings swappable and not the others. Managed identities are never swapped and are not affected by this override app setting.

To configure an app setting or connection string to stick to a specific slot (not swapped), go to the Configuration page for that slot. Add or edit a setting, and then select **Deployment slot setting**. Selecting this check box tells App Service that the setting is not swappable.

Swap deployment slots

You can swap deployment slots on your app's Deployment slots page and the Overview page. Before you swap an app from a deployment slot into production, make sure that production is your target slot and that all settings in the source slot are configured exactly as you want to have them in production.

Manually swapping deployment slots

To swap deployment slots:

1. Go to your app's **Deployment slots** page and select **Swap**. The **Swap** dialog box shows settings in the selected source and target slots that will be changed.
 2. Select the desired **Source** and **Target** slots. Usually, the target is the production slot. Also, select the **Source Changes** and **Target Changes** tabs and verify that the configuration changes are expected. When you're finished, you can swap the slots immediately by selecting **Swap**.
- To see how your target slot would run with the new settings before the swap actually happens, don't select Swap, but follow the instructions in *Swap with preview* below.
3. When you're finished, close the dialog box by selecting Close.

Swap with preview (multi-phase swap)

Before you swap into production as the target slot, validate that the app runs with the swapped settings. The source slot is also warmed up before the swap completion, which is desirable for mission-critical applications.

When you perform a swap with preview, App Service performs the same swap operation but pauses after the first step. You can then verify the result on the staging slot before completing the swap.

If you cancel the swap, App Service reapplies configuration elements to the source slot.

To swap with preview:

1. Follow the steps above in Swap deployment slots but select **Perform swap with preview**. The dialog box shows you how the configuration in the source slot changes in phase 1, and how the source and target slot change in phase 2.

2. When you're ready to start the swap, select **Start Swap**.

When phase 1 finishes, you're notified in the dialog box. Preview the swap in the source slot by going to https://<app_name>-<source-slot-name>.azurewebsites.net.

3. When you're ready to complete the pending swap, select **Complete Swap** in **Swap action** and select **Complete Swap**.

To cancel a pending swap, select **Cancel Swap** instead.

4. When you're finished, close the dialog box by selecting **Close**.

Configure auto swap

Auto swap streamlines Azure DevOps scenarios where you want to deploy your app continuously with zero cold starts and zero downtime for customers of the app. When auto swap is enabled from a slot into production, every time you push your code changes to that slot, App Service automatically swaps the app into production after it's warmed up in the source slot.

Note: Auto swap isn't currently supported in web apps on Linux.

To configure auto swap:

1. Go to your app's resource page and select the deployment slot you want to configure to auto swap. The setting is on the **Configuration > General settings** page.
2. Set **Auto swap enabled** to **On**. Then select the desired target slot for Auto swap deployment slot, and select **Save** on the command bar.
3. Execute a code push to the source slot. Auto swap happens after a short time, and the update is reflected at your target slot's URL.

Specify custom warm-up

Some apps might require custom warm-up actions before the swap. The `applicationInitialization` configuration element in `web.config` lets you specify custom initialization actions. The swap operation waits for this custom warm-up to finish before swapping with the target slot. Here's a sample `web.config` fragment.

```
<system.webServer>
    <applicationInitialization>
        <add initializationPage="/" hostName="[app hostname]" />
        <add initializationPage="/Home/About" hostName="[app hostname]" />
    </applicationInitialization>
</system.webServer>
```

For more information on customizing the `applicationInitialization` element, see [Most common deployment slot swap failures and how to fix them³](#).

³ <https://ruslany.net/2017/11/most-common-deployment-slot-swap-failures-and-how-to-fix-them/>

You can also customize the warm-up behavior with one or both of the following app settings:

- WEBSITE_SWAP_WARMUP_PING_PATH: The path to ping to warm up your site. Add this app setting by specifying a custom path that begins with a slash as the value. An example is /statuscheck. The default value is /.
- WEBSITE_SWAP_WARMUP_PING_STATUSES: Valid HTTP response codes for the warm-up operation. Add this app setting with a comma-separated list of HTTP codes. An example is 200, 202. If the returned status code isn't in the list, the warmup and swap operations are stopped. By default, all response codes are valid.

Roll back and monitor a swap

If any errors occur in the target slot (for example, the production slot) after a slot swap, restore the slots to their pre-swap states by swapping the same two slots immediately.

If the swap operation takes a long time to complete, you can get information on the swap operation in the activity log.

On your app's resource page in the portal, in the left pane, select **Activity log**.

A swap operation appears in the log query as `Swap Web App Slots`. You can expand it and select one of the suboperations or errors to see the details.

Route traffic in App Service

By default, all client requests to the app's production URL (`http://<app_name>.azurewebsites.net`) are routed to the production slot. You can route a portion of the traffic to another slot. This feature is useful if you need user feedback for a new update, but you're not ready to release it to production.

Route production traffic automatically

To route production traffic automatically:

1. Go to your app's resource page and select **Deployment slots**.
2. In the **Traffic %** column of the slot you want to route to, specify a percentage (between 0 and 100) to represent the amount of total traffic you want to route. Select **Save**.

After the setting is saved, the specified percentage of clients is randomly routed to the non-production slot.

After a client is automatically routed to a specific slot, it's "pinned" to that slot for the life of that client session. On the client browser, you can see which slot your session is pinned to by looking at the `x-ms-routing-name` cookie in your HTTP headers. A request that's routed to the "staging" slot has the cookie `x-ms-routing-name=staging`. A request that's routed to the production slot has the cookie `x-ms-routing-name=self`.

Route production traffic manually

In addition to automatic traffic routing, App Service can route requests to a specific slot. This is useful when you want your users to be able to opt in to or opt out of your beta app. To route production traffic manually, you use the `x-ms-routing-name` query parameter.

To let users opt out of your beta app, for example, you can put this link on your webpage:

```
<a href="Go back to production app</a>
```

The string `x-ms-routing-name=self` specifies the production slot. After the client browser accesses the link, it's redirected to the production slot. Every subsequent request has the `x-ms-routing-name=self` cookie that pins the session to the production slot.

To let users opt in to your beta app, set the same query parameter to the name of the non-production slot. Here's an example:

```
<webappname>.azurewebsites.net/?x-ms-routing-name=staging
```

By default, new slots are given a routing rule of 0%, a default value is displayed in grey. When you explicitly set this value to 0% it is displayed in black, your users can access the staging slot manually by using the `x-ms-routing-name` query parameter. But they won't be routed to the slot automatically because the routing percentage is set to 0. This is an advanced scenario where you can "hide" your staging slot from the public while allowing internal teams to test changes on the slot.

Knowledge check

Multiple choice

By default, all client requests to the app's production URL (`http://<app_name>.azurewebsites.net`) are routed to the production slot. One can automatically route a portion of the traffic to another slot. What is the default routing rule applied to new deployment slots?

- 0%
- 10%
- 20%

Multiple choice

Some configuration elements follow the content across a swap (not slot specific), whereas other configuration elements stay in the same slot after a swap (slot specific). Which of the settings below are swapped?

- Publishing endpoints
- WebJobs content
- WebJobs schedulers

Summary

In this module, you learned how to:

- Describe the benefits of using deployment slots
- Understand how slot swapping operates in App Service
- Perform manual swaps and enable auto swap
- Route traffic manually and automatically

Answers

Multiple choice

Which of the following App Service plans supports only function apps?

- Dedicated
- Isolated
- Consumption

Explanation

That's correct. The consumption tier is only available to function apps. It scales the functions dynamically depending on workload.

Multiple choice

Which of the following networking features of App Service can be used to control outbound network traffic?

- App-assigned address
- Hybrid Connections
- Service endpoints

Explanation

That's correct. Hybrid Connections are an outbound network feature.

Multiple choice

In which of the app configuration settings categories below would you set the language and SDK version?

- Application settings
- Path mappings
- General settings

Explanation

That's correct. This category is used to configure stack, platform, debugging, and incoming client certificate settings.

Multiple choice

Which of the following types of application logging is supported on the Linux platform?

- Web server logging
- Failed request tracing
- Deployment logging

Explanation

That's correct. Deployment logging is supported on the Linux platform.

Multiple choice

Which of the following choices correctly lists the two parts of a feature flag?

- Name, App Settings
- Name, one or more filters
- Feature manager, one or more filters

Explanation

That's correct. 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.

Multiple choice

Which of these statements best describes autoscaling?

- Autoscaling requires an administrator to actively monitor the workload on a system.
- Autoscaling is a scale out/scale in solution.
- Scaling up/scale down provides better availability than autoscaling.

Explanation

That's correct. The system can scale out when specified resource metrics indicate increasing usage, and scale in when these metrics drop.

Multiple choice

Which of these scenarios is a suitable candidate for autoscaling?

- The number of users requiring access to an application varies according to a regular schedule. For example, more users use the system on a Friday than other days of the week.
- The system is subject to a sudden influx of requests that grinds your system to a halt.
- Your organization is running a promotion and expects to see increased traffic to their web site for the next couple of weeks.

Explanation

That's correct. Changes in application load that are predictable are good candidates for autoscaling.

Multiple choice

There are multiple rules in an autoscale profile. Which of the following scale operations will run if any of the rule conditions are met?

- scale-out
- scale-in
- scale-out/in

Explanation

That's correct. Scale-out operations will trigger if any of the rule conditions are met.

Multiple choice

By default, all client requests to the app's production URL (`http://<app_name>.azurewebsites.net`) are routed to the production slot. One can automatically route a portion of the traffic to another slot. What is the default routing rule applied to new deployment slots?

- 0%
- 10%
- 20%

Explanation

That's correct. By default, new slots are given a routing rule of 0%.

Multiple choice

Some configuration elements follow the content across a swap (not slot specific), whereas other configuration elements stay in the same slot after a swap (slot specific). Which of the settings below are swapped?

- Publishing endpoints
- WebJobs content
- WebJobs schedulers

Explanation

That's correct. WebJobs content are swapped.

Module 2 Implement Azure Functions

Explore Azure Functions

Introduction

Azure Functions lets you develop serverless applications on Microsoft Azure. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it.

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

- Explain functional differences between Azure Functions, Azure Logic Apps, and WebJobs
- Describe Azure Functions hosting plan options
- Describe how Azure Functions scale to meet business needs

Discover Azure Functions

Azure Functions are a great solution for processing data, integrating systems, working with the internet-of-things (IoT), and building simple APIs and microservices. Consider Functions for tasks like image or order processing, file maintenance, or for any tasks that you want to run on a schedule. Functions provides templates to get you started with key scenarios.

Azure Functions supports *triggers*, which are ways to start execution of your code, and *bindings*, which are ways to simplify coding for input and output data. There are other integration and automation services in Azure and they all can solve integration problems and automate business processes. They can all define input, actions, conditions, and output.

Compare Azure Functions and Azure Logic Apps

Both Functions and Logic Apps enable serverless workloads. Azure Functions is a serverless compute service, whereas Azure Logic Apps provides serverless workflows. Both can create complex orchestrations. An orchestration is a collection of functions or steps, called actions in Logic Apps, that are executed to accomplish a complex task.

For Azure Functions, you develop orchestrations by writing code and using the Durable Functions extension. For Logic Apps, you create orchestrations by using a GUI or editing configuration files.

You can mix and match services when you build an orchestration, calling functions from logic apps and calling logic apps from functions. The following table lists some of the key differences between these:

	Azure Functions	Logic Apps
Development	Code-first (imperative)	Designer-first (declarative)
Connectivity	About a dozen built-in binding types, write code for custom bindings	Large collection of connectors, Enterprise Integration Pack for B2B scenarios, build custom connectors
Actions	Each activity is an Azure function; write code for activity functions	Large collection of ready-made actions
Monitoring	Azure Application Insights	Azure portal, Azure Monitor logs
Management	REST API, Visual Studio	Azure portal, REST API, PowerShell, Visual Studio
Execution context	Can run locally or in the cloud	Supports run-anywhere scenarios

Compare Functions and WebJobs

Like Azure Functions, Azure App Service WebJobs with the WebJobs SDK is a code-first integration service that is designed for developers. Both are built on Azure App Service and support features such as source control integration, authentication, and monitoring with Application Insights integration.

Azure Functions is built on the WebJobs SDK, so it shares many of the same event triggers and connections to other Azure services. Here are some factors to consider when you're choosing between Azure Functions and WebJobs with the WebJobs SDK:

	Functions	WebJobs with WebJobs SDK
Serverless app model with automatic scaling	Yes	No
Develop and test in browser	Yes	No
Pay-per-use pricing	Yes	No
Integration with Logic Apps	Yes	No
Trigger events	Timer Azure Storage queues and blobs Azure Service Bus queues and topics Azure Cosmos DB Azure Event Hubs HTTP/WebHook (GitHub Slack) Azure Event Grid	Timer Azure Storage queues and blobs Azure Service Bus queues and topics Azure Cosmos DB Azure Event Hubs File system

Azure Functions offers more developer productivity than Azure App Service WebJobs does. It also offers more options for programming languages, development environments, Azure service integration, and pricing. For most scenarios, it's the best choice.

Compare Azure Functions hosting options

When you create a function app in Azure, you must choose a hosting plan for your app. There are three basic hosting plans available for Azure Functions: Consumption plan, Functions Premium plan, and App service (Dedicated) plan. All hosting plans are generally available (GA) on both Linux and Windows virtual machines.

The hosting plan you choose dictates the following behaviors:

- How your function app is scaled.
- The resources available to each function app instance.
- Support for advanced functionality, such as Azure Virtual Network connectivity.

The following is a summary of the benefits of the three main hosting plans for Functions:

Plan	Benefits
Consumption plan	This is the default hosting plan. It scales automatically and you only pay for compute resources when your functions are running. Instances of the Functions host are dynamically added and removed based on the number of incoming events.
Premium plan	Automatically scales based on demand using pre-warmed workers which run applications with no delay after being idle, runs on more powerful instances, and connects to virtual networks.
Dedicated plan	Run your functions within an App Service plan at regular App Service plan rates. Best for long-running scenarios where Durable Functions can't be used.

There are two other hosting options which provide the highest amount of control and isolation in which to run your function apps.

Hosting option	Details
ASE	App Service Environment (ASE) (https://docs.microsoft.com/azure/app-service/environment/intro) is an App Service feature that provides a fully isolated and dedicated environment for securely running App Service apps at high scale.
Kubernetes	Kubernetes provides a fully isolated and dedicated environment running on top of the Kubernetes platform. For more information visit Azure Functions on Kubernetes with KEDA (https://docs.microsoft.com/azure/azure-functions/functions-kubernetes-keda).

Always on

If you run on a Dedicated plan, you should enable the **Always on** setting so that your function app runs correctly. On an App Service plan, the functions runtime goes idle after a few minutes of inactivity, so only HTTP triggers will "wake up" your functions. Always on is available only on an App Service plan. On a Consumption plan, the platform activates function apps automatically.

Storage account requirements

On any plan, a function app requires a general Azure Storage account, which supports Azure Blob, Queue, Files, and Table storage. This is because Functions relies on Azure Storage for operations such as managing triggers and logging function executions, but some storage accounts do not support queues and tables. These accounts, which include blob-only storage accounts (including premium storage) and general-purpose storage accounts with zone-redundant storage replication, are filtered-out from your existing **Storage Account** selections when you create a function app.

The same storage account used by your function app can also be used by your triggers and bindings to store your application data. However, for storage-intensive operations, you should use a separate storage account.

Scale Azure Functions

In the Consumption and Premium plans, Azure Functions scales CPU and memory resources by adding additional instances of the Functions host. The number of instances is determined on the number of events that trigger a function.

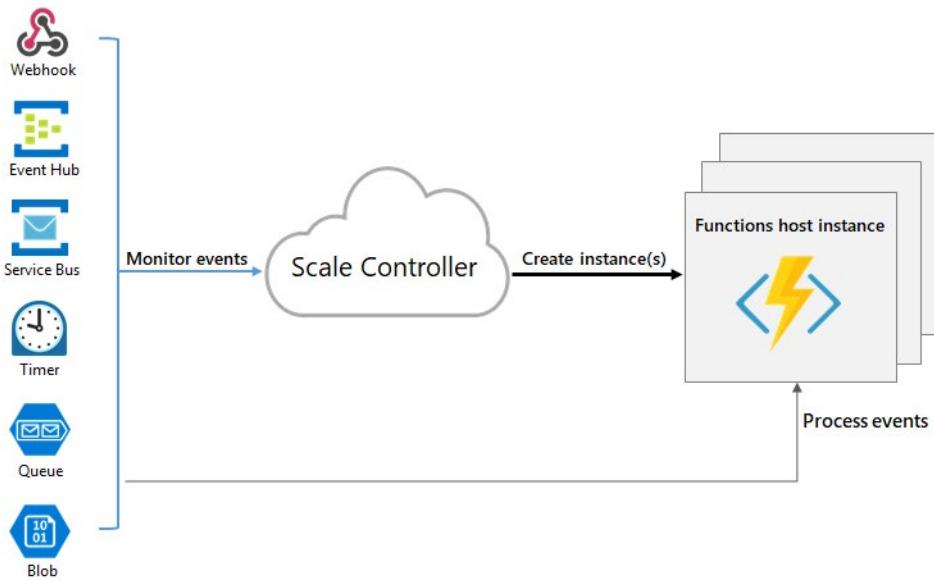
Each instance of the Functions host in the Consumption plan is limited to 1.5 GB of memory and one CPU. An instance of the host is the entire function app, meaning all functions within a function app share resource within an instance and scale at the same time. Function apps that share the same Consumption plan scale independently. In the Premium plan, the plan size determines the available memory and CPU for all apps in that plan on that instance.

Function code files are stored on Azure Files shares on the function's main storage account. When you delete the main storage account of the function app, the function code files are deleted and cannot be recovered.

Runtime scaling

Azure Functions uses a component called the *scale controller* to monitor the rate of events and determine whether to scale out or scale in. The scale controller uses heuristics for each trigger type. For example, when you're using an Azure Queue storage trigger, it scales based on the queue length and the age of the oldest queue message.

The unit of scale for Azure Functions is the function app. When the function app is scaled out, additional resources are allocated to run multiple instances of the Azure Functions host. Conversely, as compute demand is reduced, the scale controller removes function host instances. The number of instances is eventually "scaled in" to zero when no functions are running within a function app.



Note: After your function app has been idle for a number of minutes, the platform may scale the number of instances on which your app runs down to zero. The next request has the added latency of scaling from zero to one. This latency is referred to as a *cold start*.

Scaling behaviors

Scaling can vary on a number of factors, and scale differently based on the trigger and language selected. There are a few intricacies of scaling behaviors to be aware of:

- **Maximum instances:** A single function app only scales out to a maximum of 200 instances. A single instance may process more than one message or request at a time though, so there isn't a set limit on number of concurrent executions.
- **New instance rate:** For HTTP triggers, new instances are allocated, at most, once per second. For non-HTTP triggers, new instances are allocated, at most, once every 30 seconds.

Limit scale out

You may wish to restrict the maximum number of instances an app used to scale out. This is most common for cases where a downstream component like a database has limited throughput. By default, Consumption plan functions scale out to as many as 200 instances, and Premium plan functions will scale out to as many as 100 instances. You can specify a lower maximum for a specific app by modifying the `functionAppScaleLimit` value. The `functionAppScaleLimit` can be set to 0 or null for unrestricted, or a valid value between 1 and the app maximum.

Azure Functions scaling in an App service plan

Using an App Service plan, you can manually scale out by adding more VM instances. You can also enable autoscale, though autoscale will be slower than the elastic scale of the Premium plan.

Knowledge check

Multiple choice

Which of the following Azure Functions hosting plans is best when predictive scaling and costs are required?

- Functions Premium Plan
- App service plan
- Consumption plan

Multiple choice

An organization wants to implement a serverless workflow to solve a business problem. One of the requirements is the solution needs to use a designer-first (declarative) development model. Which of the choices below meets the requirements?

- Azure Functions
- Azure Logic Apps
- WebJobs

Summary

In this module, you learned how to:

- Explain functional differences between Azure Functions, Azure Logic Apps, and WebJobs
- Describe Azure Functions hosting plan options
- Describe how Azure Functions scale to meet business needs

Develop Azure Functions

Introduction

Functions share a few core technical concepts and components, regardless of the language or binding you use.

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

- Explain the key components of a function and how they are structured
- Create triggers and bindings to control when a function runs and where the output is directed
- Connect a function to services in Azure
- Create a function by using Visual Studio Code and the Azure Functions Core Tools

Explore Azure Functions development

A function contains two important pieces - your code, which can be written in a variety of languages, and some config, the *function.json* file. For compiled languages, this config file is generated automatically from annotations in your code. For scripting languages, you must provide the config file yourself.

The *function.json* file defines the function's trigger, bindings, and other configuration settings. Every function has one and only one trigger. The runtime uses this config file to determine the events to monitor and how to pass data into and return data from a function execution. The following is an example *function.json* file.

```
{
    "disabled": false,
    "bindings": [
        // ... bindings here
        {
            "type": "bindingType",
            "direction": "in",
            "name": "myParamName",
            // ... more depending on binding
        }
    ]
}
```

The *bindings* property is where you configure both triggers and bindings. Each binding shares a few common settings and some settings which are specific to a particular type of binding. Every binding requires the following settings:

Property	Types	Comments
type	string	Name of binding. For example, queueTrigger.
direction	string	Indicates whether the binding is for receiving data into the function or sending data from the function. For example, in or out.

Property	Types	Comments
name	string	The name that is used for the bound data in the function. For example, myQueue.

Function app

A function app provides an execution context in Azure in which your functions run. As such, it is the unit of deployment and management for your functions. A function app is comprised of one or more individual functions that are managed, deployed, and scaled together. All of the functions in a function app share the same pricing plan, deployment method, and runtime version. Think of a function app as a way to organize and collectively manage your functions.

Note: In Functions 2.x all functions in a function app must be authored in the same language. In previous versions of the Azure Functions runtime, this wasn't required.

Folder structure

The code for all the functions in a specific function app is located in a root project folder that contains a host configuration file. The **host.json**¹ file contains runtime-specific configurations and is in the root folder of the function app. A *bin* folder contains packages and other library files that the function app requires. Specific folder structures required by the function app depend on language:

- **C# compiled (.csproj)**²
- **C# script (.csx)**³
- **F# script**⁴
- **Java**⁵
- **JavaScript**⁶
- **Python**⁷

Local development environments

Functions makes it easy to use your favorite code editor and development tools to create and test functions on your local computer. Your local functions can connect to live Azure services, and you can debug them on your local computer using the full Functions runtime.

The way in which you develop functions on your local computer depends on your language and tooling preferences. See **Code and test Azure Functions locally**⁸ for more information.

Warning: Do not mix local development with portal development in the same function app. When you create and publish functions from a local project, you should not try to maintain or modify project code in the portal.

¹ <https://docs.microsoft.com/azure/azure-functions/functions-host-json>

² <https://docs.microsoft.com/azure/azure-functions/functions-dotnet-class-library#functions-class-library-project>

³ <https://docs.microsoft.com/azure/azure-functions/functions-reference-csharp#folder-structure>

⁴ <https://docs.microsoft.com/azure/azure-functions/functions-reference-fsharp#folder-structure>

⁵ <https://docs.microsoft.com/azure/azure-functions/functions-reference-java#folder-structure>

⁶ <https://docs.microsoft.com/azure/azure-functions/functions-reference-node#folder-structure>

⁷ <https://docs.microsoft.com/azure/azure-functions/functions-reference-python#folder-structure>

⁸ <https://docs.microsoft.com/azure/azure-functions/functions-develop-local>

Create triggers and bindings

Triggers are what cause a function to run. A trigger defines how a function is invoked and a function must have exactly one trigger. Triggers have associated data, which is often provided as the payload of the function.

Binding to a function is a way of declaratively connecting another resource to the function; bindings may be connected as *input bindings*, *output bindings*, or both. Data from bindings is provided to the function as parameters.

You can mix and match different bindings to suit your needs. Bindings are optional and a function might have one or multiple input and/or output bindings.

Triggers and bindings let you avoid hardcoding access to other services. Your function receives data (for example, the content of a queue message) in function parameters. You send data (for example, to create a queue message) by using the return value of the function.

Trigger and binding definitions

Triggers and bindings are defined differently depending on the development language.

Language	Triggers and bindings are configured by...
C# class library	decorating methods and parameters with C# attributes
Java	decorating methods and parameters with Java annotations
JavaScript/PowerShell/Python/TypeScript	updating <i>function.json</i> schema

For languages that rely on *function.json*, the portal provides a UI for adding bindings in the **Integration** tab. You can also edit the file directly in the portal in the **Code + test** tab of your function.

In .NET and Java, the parameter type defines the data type for input data. For instance, use `string` to bind to the text of a queue trigger, a byte array to read as binary, and a custom type to de-serialize to an object. Since .NET class library functions and Java functions don't rely on *function.json* for binding definitions, they can't be created and edited in the portal. C# portal editing is based on C# script, which uses *function.json* instead of attributes.

For languages that are dynamically typed such as JavaScript, use the `dataType` property in the *function.json* file. For example, to read the content of an HTTP request in binary format, set `dataType` to `binary`:

```
{  
    "dataType": "binary",  
    "type": "httpTrigger",  
    "name": "req",  
    "direction": "in"  
}
```

Other options for `dataType` are `stream` and `string`.

Binding direction

All triggers and bindings have a `direction` property in the *function.json* file:

- For triggers, the direction is always `in`

- Input and output bindings use `in` and `out`
- Some bindings support a special direction `inout`. If you use `inout`, only the **Advanced editor** is available via the **Integrate** tab in the portal.

When you use attributes in a class library to configure triggers and bindings, the direction is provided in an attribute constructor or inferred from the parameter type.

Azure Functions trigger and binding example

Suppose you want to write a new row to Azure Table storage whenever a new message appears in Azure Queue storage. This scenario can be implemented using an Azure Queue storage trigger and an Azure Table storage output binding.

Here's a `function.json` file for this scenario.

```
{
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "order",
      "queueName": "myqueue-items",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    },
    {
      "type": "table",
      "direction": "out",
      "name": "$return",
      "tableName": "outTable",
      "connection": "MY_TABLE_STORAGE_ACCT_APP_SETTING"
    }
  ]
}
```

The first element in the `bindings` array is the Queue storage trigger. The `type` and `direction` properties identify the trigger. The `name` property identifies the function parameter that receives the queue message content. The name of the queue to monitor is in `queueName`, and the connection string is in the app setting identified by `connection`.

The second element in the `bindings` array is the Azure Table Storage output binding. The `type` and `direction` properties identify the binding. The `name` property specifies how the function provides the new table row, in this case by using the function return value. The name of the table is in `tableName`, and the connection string is in the app setting identified by `connection`.

C# script example

Here's C# script code that works with this trigger and binding. Notice that the name of the parameter that provides the queue message content is `order`; this name is required because the `name` property value in `function.json` is `order`.

```
#r "Newtonsoft.Json"

using Microsoft.Extensions.Logging;
```

```
using Newtonsoft.Json.Linq;

// From an incoming queue message that is a JSON object, add fields and
// write to Table storage
// The method return value creates a new row in Table Storage
public static Person Run(JObject order, ILogger log)
{
    return new Person() {
        PartitionKey = "Orders",
        RowKey = Guid.NewGuid().ToString(),
        Name = order["Name"].ToString(),
        MobileNumber = order["MobileNumber"].ToString() };
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}
```

JavaScript example

The same *function.json* file can be used with a JavaScript function:

```
// From an incoming queue message that is a JSON object, add fields and
// write to Table Storage
// The second parameter to context.done is used as the value for the new
// row
module.exports = function (context, order) {
    order.PartitionKey = "Orders";
    order.RowKey = generateRandomId();

    context.done(null, order);
};

function generateRandomId() {
    return Math.random().toString(36).substring(2, 15) +
        Math.random().toString(36).substring(2, 15);
}
```

Class library example

In a class library, the same trigger and binding information — queue and table names, storage accounts, function parameters for input and output — is provided by attributes instead of a *function.json* file. Here's an example:

```
public static class QueueTriggerTableOutput
{
```

```
[FunctionName("QueueTriggerTableOutput")]
[return: Table("outTable", Connection = "MY_TABLE_STORAGE_ACCT_APP_SETTING")]
public static Person Run(
    [QueueTrigger("myqueue-items", Connection = "MY_STORAGE_ACCT_APP_SETTING")] JObject order,
    ILogger log)
{
    return new Person() {
        PartitionKey = "Orders",
        RowKey = Guid.NewGuid().ToString(),
        Name = order["Name"].ToString(),
        MobileNumber = order["MobileNumber"].ToString() };
}
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}
```

Additional resource

For more detailed examples of triggers and bindings please visit:

- [Azure Blob storage bindings for Azure Functions⁹](#)
- [Azure Cosmos DB bindings for Azure Functions 2.x¹⁰](#)
- [Timer trigger for Azure Functions¹¹](#)
- [Azure Functions HTTP triggers and bindings¹²](#)

Connect functions to Azure services

Your function project references connection information by name from its configuration provider. It does not directly accept the connection details, allowing them to be changed across environments. For example, a trigger definition might include a `connection` property. This might refer to a connection string, but you cannot set the connection string directly in a `function.json`. Instead, you would set `connection` to the name of an environment variable that contains the connection string.

The default configuration provider uses environment variables. These might be set by [Application Settings¹³](#) when running in the Azure Functions service, or from the [local settings file¹⁴](#) when developing locally.

⁹ <https://docs.microsoft.com/azure/azure-functions/functions-bindings-storage-blob>

¹⁰ <https://docs.microsoft.com/azure/azure-functions/functions-bindings-cosmosdb-v2>

¹¹ <https://docs.microsoft.com/azure/azure-functions/functions-bindings-timer>

¹² <https://docs.microsoft.com/azure/azure-functions/functions-bindings-http-webhook>

¹³ <https://docs.microsoft.com/azure/azure-functions/functions-how-to-use-azure-function-app-settings?tabs=portal#settings>

¹⁴ <https://docs.microsoft.com/azure/azure-functions/functions-develop-local#local-settings-file>

Connection values

When the connection name resolves to a single exact value, the runtime identifies the value as a *connection string*, which typically includes a secret. The details of a connection string are defined by the service to which you wish to connect.

However, a connection name can also refer to a collection of multiple configuration items. Environment variables can be treated as a collection by using a shared prefix that ends in double underscores __. The group can then be referenced by setting the connection name to this prefix.

For example, the `connection` property for a Azure Blob trigger definition might be `Storage1`. As long as there is no single string value configured with `Storage1` as its name, `Storage1__serviceUri` would be used for the `serviceUri` property of the connection. The connection properties are different for each service.

Configure an identity-based connection

Some connections in Azure Functions are configured to use an identity instead of a secret. Support depends on the extension using the connection. In some cases, a connection string may still be required in Functions even though the service to which you are connecting supports identity-based connections.

Note: Identity-based connections are not supported with Durable Functions.

When hosted in the Azure Functions service, identity-based connections use a **managed identity**¹⁵. The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized using alternative connection parameters.

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. This is typically done by assigning a role in Azure RBAC or specifying the identity in an access policy, depending on the service to which you are connecting.

Important: Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges.

Exercise: Create an Azure Function by using Visual Studio Code

In this exercise you'll learn how to create a simple C# function that responds to HTTP requests. After creating and testing the code locally in Visual Studio Code you will deploy to Azure.

Prerequisites

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

- 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>.

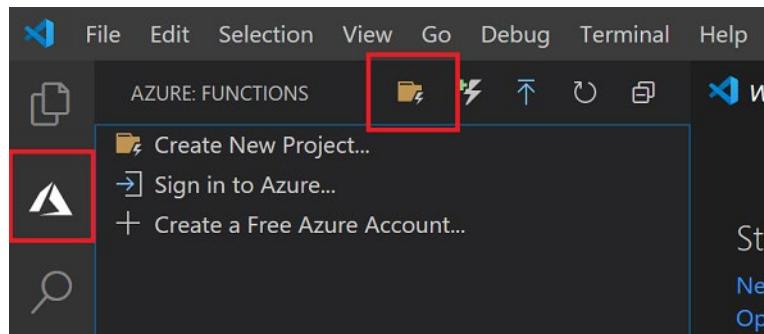
¹⁵ <https://docs.microsoft.com/azure/app-service/overview-managed-identity?toc=/azure/azure-functions/toc.json>

- The **Azure Functions Core Tools**¹⁶ version 3.x.
- **Visual Studio Code**¹⁷ on one of the **supported platforms**¹⁸.
- **.NET Core 3.1**¹⁹ is the target framework for the steps below.
- The **C# extension**²⁰ for Visual Studio Code.
- The **Azure Functions extension**²¹ for Visual Studio Code.

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project in C#. Later in this exercise, you'll publish your function code to Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, select **Create new project....**



2. Choose a directory location for your project workspace and choose **Select**.

Note: Be sure to select a project folder that is outside of a workspace.

3. Provide the following information at the prompts:

- **Select a language:** Choose C#.
- **Select a .NET runtime:** Choose .NET Core 3.1
- **Select a template for your project's first function:** Choose HTTP trigger.
- **Provide a function name:** Type HttpExample.
- **Provide a namespace:** Type My.Functions.
- **Authorization level:** Choose Anonymous, which enables anyone to call your function endpoint.
- **Select how you would like to open your project:** Choose Add to workspace.

4. Using this information, Visual Studio Code generates an Azure Functions project with an HTTP trigger.

¹⁶ <https://docs.microsoft.com/azure/azure-functions/functions-run-local#install-the-azure-functions-core-tools>

¹⁷ <https://code.visualstudio.com/>

¹⁸ https://code.visualstudio.com/docs/supporting/requirements#_platforms

¹⁹ <https://dotnet.microsoft.com/download/dotnet/3.1>

²⁰ <https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp>

²¹ <https://marketplace.visualstudio.com/items?itemName=ms-azurertools.vscode-azurefunctions>

Run the function locally

Visual Studio Code integrates with Azure Functions Core tools to let you run this project on your local development computer before you publish to Azure.

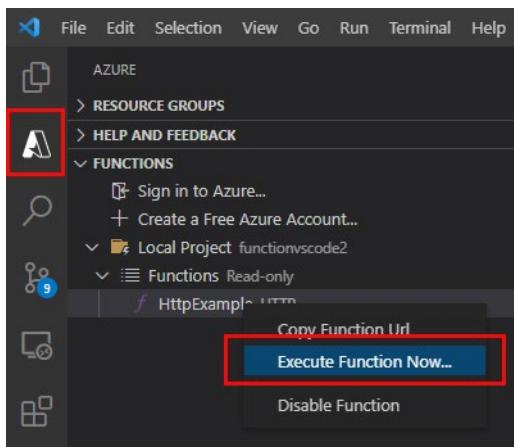
- To call your function, press **F5** to start the function app project. Output from Core Tools is displayed in the **Terminal** panel. Your app starts in the **Terminal** panel. You can see the URL endpoint of your HTTP-triggered function running locally.

```
PROBLEMS TERMINAL OUTPUT DEBUG CONSOLE
2: host start (Task) + - ×
Core Tools Version: 3.0.3477 Commit hash: 5fbb9a76fc00e4168f2cc90d6ff0afe5373afc6d (64-bit)
Function Runtime Version: 3.0.15584.0

Functions:
HttpExample: [GET,POST] http://localhost:7071/api/HttpExample

For detailed output, run func with --verbose flag.
[2021-06-07T15:43:22.457Z] Worker process started and initialized.
[2021-06-07T15:43:26.923Z] Host lock lease acquired by instance ID '00000000000000000000000000000008E0725BD'.
```

- With Core Tools running, go to the **Azure: Functions** area. Under **FUNCTIONS**, expand **Local Project > FUNCTIONS**. Right-click the `HttpExample` function and choose **Execute Function Now...**



- In **Enter request body** type the request message body value of `{ "name": "Azure" }`. Press **Enter** to send this request message to your function. When the function executes locally and returns a response, a notification is raised in Visual Studio Code. Information about the function execution is shown in **Terminal** panel.

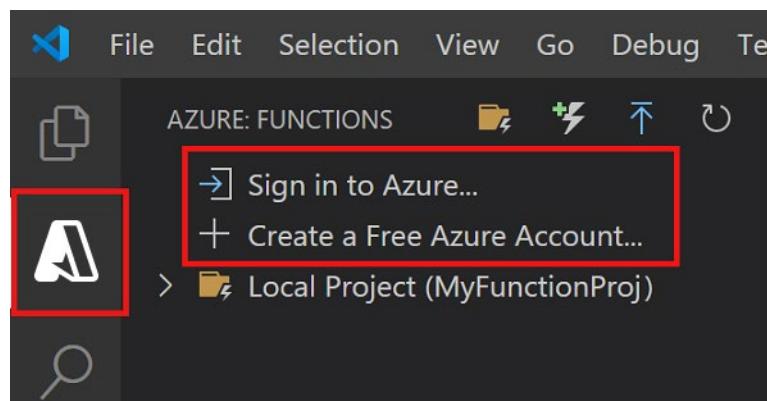
- Press Ctrl + C** to stop Core Tools and disconnect the debugger.

After you've verified that the function runs correctly on your local computer, it's time to use Visual Studio Code to publish the project directly to Azure.

Sign in to Azure

Before you can publish your app, you must sign in to Azure. If you're already signed in, go to the next section.

- If you aren't already signed in, choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose **Sign in to Azure....**



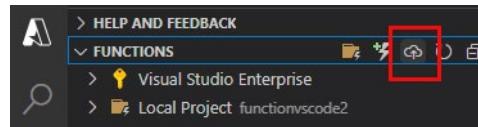
2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the Side bar.

Publish the project to Azure

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

Important: Publishing to an existing function app overwrites the content of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Azure: Functions** area, choose the **Deploy to Function app...** button.



2. Provide the following information at the prompts:
 - **Select Function App in Azure:** Choose **+ Create new Function App**. (Don't choose the Advanced option, which isn't covered in this article.)
 - **Enter a globally unique name for the function app:** Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.
 - **Select a runtime stack:** Use the same choice you made in the *Create your local project* section above.
 - **Select a location for new resources:** For better performance, choose a region near you.
 - **Select subscription:** Choose the subscription to use. *You won't see this if you only have one subscription.*

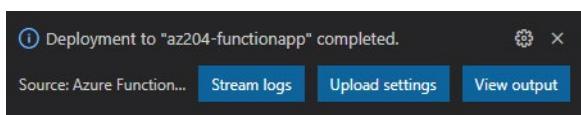
The extension shows the status of individual resources as they are being created in Azure in the notification area.

Creating new function app "az204-functionapp" ... (6/6)

3. When completed, the following Azure resources are created in your subscription, using names based on your function app name:

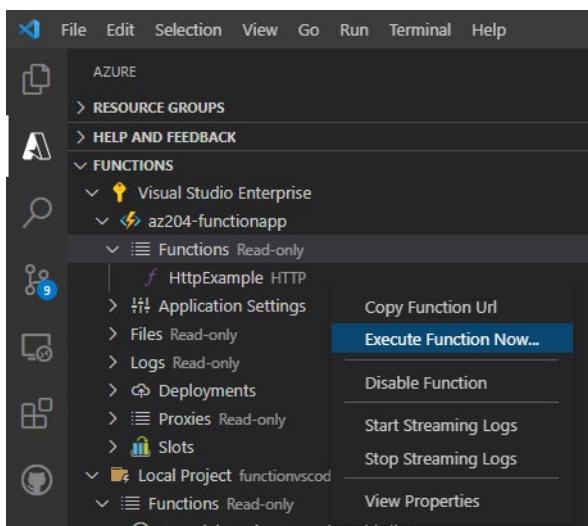
- A resource group, which is a logical container for related resources.
- A standard Azure Storage account, which maintains state and other information about your projects.
- A consumption plan, which defines the underlying host for your serverless function app.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An Application Insights instance connected to the function app, which tracks usage of your serverless function.

A notification is displayed after your function app is created and the deployment package is applied.

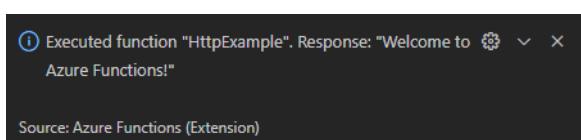


Run the function in Azure

1. Back in the **Azure: Functions** area in the side bar, expand your subscription, your new function app, and **Functions**. Right-click the **HttpExample** function and choose **Execute Function Now....**



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



Clean up resources

Use the following steps to delete the function app and its related resources to avoid incurring any further costs.

1. In Visual Studio Code, press **F1** to open the command palette. In the command palette, search for and select Azure Functions: Open in portal.
2. Choose your function app from the list, and press **Enter**. The function app page opens in the Azure portal.
3. In the **Overview** tab, select the named link next to **Resource group**.
4. In the **Resource group** page, review the list of included resources, and verify that they are the ones you want to delete.
5. Select **Delete resource group**, and follow the instructions.

Deletion may take a couple of minutes. When it's done, a notification appears for a few seconds. You can also select the bell icon at the top of the page to view the notification.

Knowledge check

Multiple choice

Which of the following is required for a function to run?

- Binding
- Trigger
- Both triggers and bindings

Multiple choice

Which of the following supports both the in and out direction settings?

- Bindings
- Trigger
- Connection value

Summary

In this module, you learned how to:

- Explain the key components of a function and how they are structured
- Create triggers and bindings to control when a function runs and where the output is directed
- Connect a function to services in Azure
- Create a function by using Visual Studio Code and the Azure Functions Core Tools

Implement Durable Functions

Introduction

Durable Functions is an extension of Azure Functions that lets you write stateful functions in a serverless compute environment.

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

- Describe the app patterns typically used in durable functions
- Describe the four durable function types
- Explain the function Task Hubs perform in durable functions
- Describe the use of durable orchestrations, timers, and events

Explore Durable Functions app patterns

The *durable functions* extension lets you define stateful workflows by writing *orchestrator functions* and stateful entities by writing *entity functions* using the Azure Functions programming model. Behind the scenes, the extension manages state, checkpoints, and restarts for you, allowing you to focus on your business logic.

Supported languages

Durable Functions currently supports the following languages:

- **C#**: both precompiled class libraries and C# script.
- **JavaScript**: supported only for version 2.x of the Azure Functions runtime. Requires version 1.7.0 of the Durable Functions extension, or a later version.
- **Python**: requires version 2.3.1 of the Durable Functions extension, or a later version.
- **F#**: precompiled class libraries and F# script. F# script is only supported for version 1.x of the Azure Functions runtime.
- **PowerShell**: Supported only for version 3.x of the Azure Functions runtime and PowerShell7. Requires version 2.x of the bundle extensions.

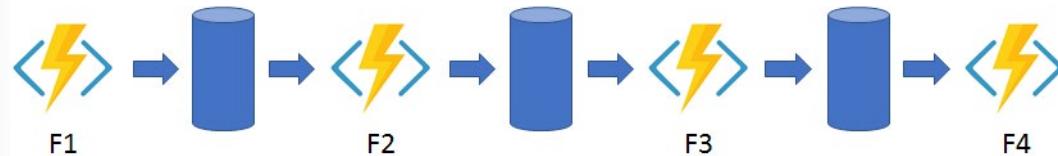
Application patterns

The primary use case for Durable Functions is simplifying complex, stateful coordination requirements in serverless applications. The following sections describe typical application patterns that can benefit from Durable Functions:

- Function chaining
- Fan-out/fan-in
- Async HTTP APIs
- Monitor
- Human interaction

Function chaining

In the function chaining pattern, a sequence of functions executes in a specific order. In this pattern, the output of one function is applied to the input of another function.

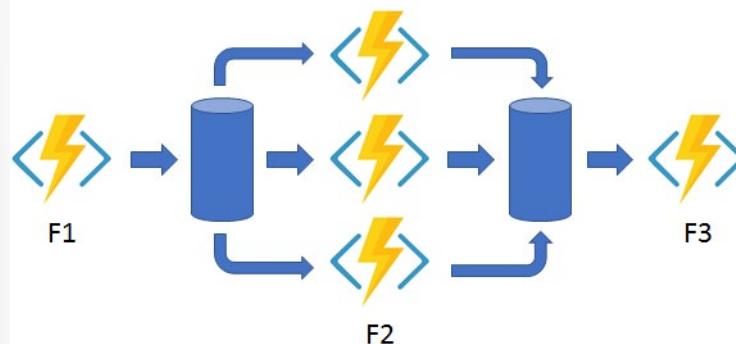


In the code example below, the values F1, F2, F3, and F4 are the names of other functions in the function app. You can implement control flow by using normal imperative coding constructs. Code executes from the top down. The code can involve existing language control flow semantics, like conditionals and loops. You can include error handling logic in try/catch/finally blocks.

```
[FunctionName("Chaining")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1", null);
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return await context.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // Error handling or compensation goes here.
    }
}
```

Fan out/fan in

In the fan out/fan in pattern, you execute multiple functions in parallel and then wait for all functions to finish. Often, some aggregation work is done on the results that are returned from the functions.



With normal functions, you can fan out by having the function send multiple messages to a queue. To fan in you write code to track when the queue-triggered functions end, and then store function outputs.

In the code example below, the fan-out work is distributed to multiple instances of the F2 function. The work is tracked by using a dynamic list of tasks. The .NET Task.WhenAll API or JavaScript context.df.Task.all API is called, to wait for all the called functions to finish. Then, the F2 function outputs are aggregated from the dynamic task list and passed to the F3 function.

```
[FunctionName ("FanOutFanIn")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var parallelTasks = new List<Task<int>>();

    // Get a list of N work items to process in parallel.
    object[] workBatch = await context.CallActivityAsync<object[]>("F1",
        null);
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<int> task = context.CallActivityAsync<int>("F2", work-
        Batch[i]);
        parallelTasks.Add(task);
    }

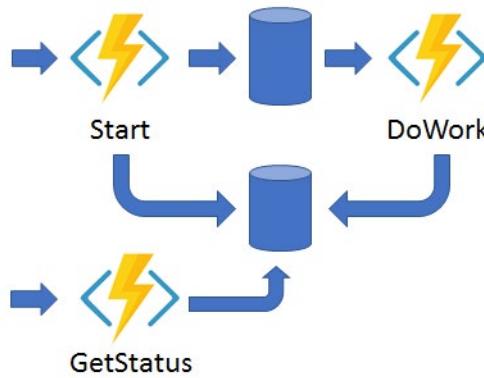
    await Task.WhenAll(parallelTasks);

    // Aggregate all N outputs and send the result to F3.
    int sum = parallelTasks.Sum(t => t.Result);
    await context.CallActivityAsync("F3", sum);
}
```

The automatic checkpointing that happens at the `await` or `yield` call on `Task.WhenAll` or `context.df.Task.all` ensures that a potential midway crash or reboot doesn't require restarting an already completed task.

Async HTTP APIs

The async HTTP API pattern addresses the problem of coordinating the state of long-running operations with external clients. A common way to implement this pattern is by having an HTTP endpoint trigger the long-running action. Then, redirect the client to a status endpoint that the client polls to learn when the operation is finished.



Durable Functions provides **built-in support** for this pattern, simplifying or even removing the code you need to write to interact with long-running function executions. After an instance starts, the extension exposes webhook HTTP APIs that query the orchestrator function status.

The following example shows REST commands that start an orchestrator and query its status. For clarity, some protocol details are omitted from the example.

```
> curl -X POST https://myfunc.azurewebsites.net/orchestrators/DoWork -H  
"Content-Length: 0" -i  
HTTP/1.1 202 Accepted  
Content-Type: application/json  
Location: https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/  
b79baf67f717453ca9e86c5da21e03ec  
  
{"id":"b79baf67f717453ca9e86c5da21e03ec", ...}  
  
> curl https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/  
b79baf67f717453ca9e86c5da21e03ec -i  
HTTP/1.1 202 Accepted  
Content-Type: application/json  
Location: https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/  
b79baf67f717453ca9e86c5da21e03ec  
  
{"runtimeStatus":"Running","lastUpdatedTime":"2019-03-16T21:20:47Z", ...}  
  
> curl https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/  
b79baf67f717453ca9e86c5da21e03ec -i  
HTTP/1.1 200 OK  
Content-Length: 175  
Content-Type: application/json  
  
{"runtimeStatus":"Completed","lastUpdatedTime":"2019-03-16T21:20:57Z", ...}
```

The Durable Functions extension exposes built-in HTTP APIs that manage long-running orchestrations. You can alternatively implement this pattern yourself by using your own function triggers (such as HTTP, a queue, or Azure Event Hubs) and the orchestration client binding.

You can use the `HttpStart` triggered function to start instances of an orchestrator function using a client function.

```
public static class HttpStart
{
    [FunctionName("HttpStart")]
    public static async Task<HttpResponseMessage> Run(
        [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route =
"orchestrators/{functionName}")] HttpRequestMessage req,
        [DurableClient] IDurableClient starter,
        string functionName,
        ILogger log)
    {
        // Function input comes from the request content.
        object eventData = await req.Content.ReadAsAsync<object>();
        string instanceId = await starter.StartNewAsync(functionName,
eventData);

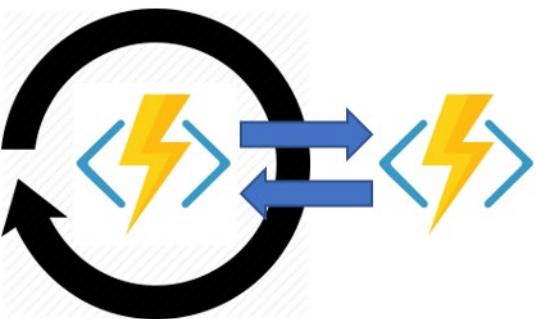
        log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

        return starter.CreateCheckStatusResponse(req, instanceId);
    }
}
```

To interact with orchestrators, the function must include a `DurableClient` input binding. You use the client to start an orchestration. It can also help you return an HTTP response containing URLs for checking the status of the new orchestration.

Monitor

The monitor pattern refers to a flexible, recurring process in a workflow. An example is polling until specific conditions are met. You can use a regular timer trigger to address a basic scenario, such as a periodic cleanup job, but its interval is static and managing instance lifetimes becomes complex. You can use Durable Functions to create flexible recurrence intervals, manage task lifetimes, and create multiple monitor processes from a single orchestration.



In a few lines of code, you can use Durable Functions to create multiple monitors that observe arbitrary endpoints. The monitors can end execution when a condition is met, or another function can use the durable orchestration client to terminate the monitors. You can change a monitor's wait interval based on a specific condition (for example, exponential backoff).

The following code implements a basic monitor:

```
[FunctionName("MonitorJobStatus")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    int jobId = context.GetInput<int>();
    int pollingInterval = GetPollingInterval();
    DateTime expiryTime = GetExpiryTime();

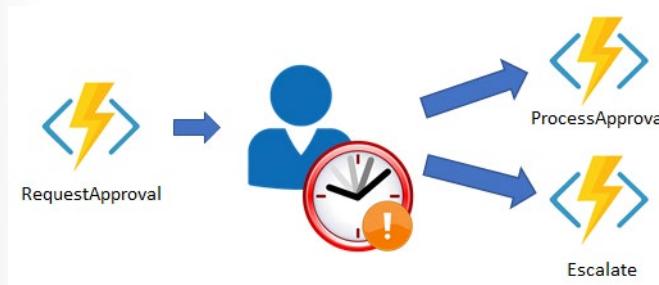
    while (context.CurrentUtcDateTime < expiryTime)
    {
        var jobStatus = await context.CallActivityAsync<string>("GetJobStatus", jobId);
        if (jobStatus == "Completed")
        {
            // Perform an action when a condition is met.
            await context.CallActivityAsync("SendAlert", machineId);
            break;
        }

        // Orchestration sleeps until this time.
        var nextCheck = context.CurrentUtcDateTime.AddSeconds(pollingInterval);
        await context.CreateTimer(nextCheck, CancellationToken.None);
    }

    // Perform more work here, or let the orchestration end.
}
```

Human interaction

Many automated processes involve some kind of human interaction. Involving humans in an automated process is tricky because people aren't as highly available and as responsive as cloud services. An automated process might allow for this interaction by using timeouts and compensation logic.



You can implement the pattern in this example by using an orchestrator function. The orchestrator uses a durable timer to request approval. The orchestrator escalates if timeout occurs. The orchestrator waits for an external event, such as a notification that's generated by a human interaction.

```
[FunctionName("ApprovalWorkflow")]
public static async Task Run(
```

```
[OrchestrationTrigger] IDurableOrchestrationContext context)
{
    await context.CallActivityAsync("RequestApproval", null);
    using (var timeoutCts = new CancellationTokenSource())
    {
        DateTime dueTime = context.CurrentUtcDateTime.AddHours(72);
        Task durableTimeout = context.CreateTimer(dueTime, timeoutCts.
Token);

        Task<bool> approvalEvent = context.WaitForExternalEvent<bool>("Ap-
provalEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent, durableTime-
out))
        {
            timeoutCts.Cancel();
            await context.CallActivityAsync("ProcessApproval", approvalEv-
ent.Result);
        }
        else
        {
            await context.CallActivityAsync("Escalate", null);
        }
    }
}
```

Additional resources

- To learn about the differences between Durable Functions 1.x and 2.x visit:
 - **Durable Functions versions overview²²**

Discover the four function types

There are currently four durable function types in Azure Functions: orchestrator, activity, entity, and client. The rest of this section goes into more details about the types of functions involved in an orchestration.

Orchestrator functions

Orchestrator functions describe how actions are executed and the order in which actions are executed. Orchestrator functions describe the orchestration in code (C# or JavaScript) as shown in the previous unit. An orchestration can have many different types of actions, including activity functions, sub-orchestra-tions, waiting for external events, HTTP, and timers. Orchestrator functions can also interact with entity functions.

Orchestrator functions are written using ordinary code, but there are strict requirements on how to write the code. Specifically, orchestrator function code must be deterministic. Failing to follow these determinism requirements can cause orchestrator functions to fail to run correctly.

²² <https://docs.microsoft.com/azure/azure-functions/durable/durable-functions-versions>

Note: The [Orchestrator function code constraints](#)²³ article has detailed information on this requirement.

Activity functions

Activity functions are the basic unit of work in a durable function orchestration. For example, you might create an orchestrator function to process an order. The tasks involve checking the inventory, charging the customer, and creating a shipment. Each task would be a separate activity function. These activity functions may be executed serially, in parallel, or some combination of both.

Unlike orchestrator functions, activity functions aren't restricted in the type of work you can do in them. Activity functions are frequently used to make network calls or run CPU intensive operations. An activity function can also return data back to the orchestrator function.

An activity trigger is used to define an activity function. .NET functions receive a `DurableActivityContext` as a parameter. You can also bind the trigger to any other JSON-serializable object to pass in inputs to the function. In JavaScript, you can access an input via the `<activity trigger binding name>` property on the `context.bindings` object. Activity functions can only have a single value passed to them. To pass multiple values, you must use tuples, arrays, or complex types.

Entity functions

Entity functions define operations for reading and updating small pieces of state. We often refer to these stateful entities as durable entities. Like orchestrator functions, entity functions are functions with a special trigger type, *entity trigger*. They can also be invoked from client functions or from orchestrator functions. Unlike orchestrator functions, entity functions do not have any specific code constraints. Entity functions also manage state explicitly rather than implicitly representing state via control flow. Some things to note:

- Entities are accessed via a unique identifier, the *entity ID*. An entity ID is simply a pair of strings that uniquely identifies an entity instance.
- Operations on entities require that you specify the **Entity ID** of the target entity, and the **Operation name**, which is a string that specifies the operation to perform.

Client functions

Orchestrator and entity functions are triggered by their bindings and both of these triggers work by reacting to messages that are enqueued in a task hub. The primary way to deliver these messages is by using an orchestrator client binding, or an entity client binding, from within a *client function*. Any non-orchestrator function can be a client function. For example, You can trigger the orchestrator from an HTTP-triggered function, an Azure Event Hub triggered function, etc. What makes a function a client function is its use of the *durable client output binding*.

Unlike other function types, orchestrator and entity functions cannot be triggered directly using the buttons in the Azure portal. If you want to test an orchestrator or entity function in the Azure portal, you must instead run a client function that starts an orchestrator or entity function as part of its implementation. For the simplest testing experience, a *manual trigger* function is recommended.

²³ <https://docs.microsoft.com/azure/azure-functions/durable/durable-functions-code-constraints>

Explore task hubs

A task hub in Durable Functions is a logical container for durable storage resources that are used for orchestrations and entities. Orchestrator, activity, and entity functions can only directly interact with each other when they belong to the same task hub.

If multiple function apps share a storage account, each function app must be configured with a separate task hub name. A storage account can contain multiple task hubs. This restriction generally applies to other storage providers as well.

Azure Storage resources

A task hub in Azure Storage consists of the following resources:

- One or more control queues.
- One work-item queue.
- One history table.
- One instances table.
- One storage container containing one or more lease blobs.
- A storage container containing large message payloads, if applicable.

All of these resources are created automatically in the configured Azure Storage account when orchestrator, entity, or activity functions run or are scheduled to run.

Task hub names

Task hubs in Azure Storage are identified by a name that conforms to these rules:

- Contains only alphanumeric characters
- Starts with a letter
- Has a minimum length of 3 characters, maximum length of 45 characters

The task hub name is declared in the *host.json* file, as shown in the following example:

```
{  
    "version": "2.0",  
    "extensions": {  
        "durableTask": {  
            "hubName": "MyTaskHub"  
        }  
    }  
}
```

The name is what differentiates one task hub from another when there are multiple task hubs in a shared storage account. If you have multiple function apps sharing a shared storage account, you must explicitly configure different names for each task hub in the *host.json* files. Otherwise the multiple function apps will compete with each other for messages, which could result in undefined behavior, including orchestrations getting unexpectedly “stuck” in the Pending or Running state.

Explore durable orchestrations

You can use an *orchestrator function* to orchestrate the execution of other Durable functions within a function app. Orchestrator functions have the following characteristics:

- Orchestrator functions define function workflows using procedural code. No declarative schemas or designers are needed.
- Orchestrator functions can call other durable functions synchronously and asynchronously. Output from called functions can be reliably saved to local variables.
- Orchestrator functions are durable and reliable. Execution progress is automatically checkpointed when the function "awaits" or "yields". Local state is never lost when the process recycles or the VM reboots.
- Orchestrator functions can be long-running. The total lifespan of an *orchestration instance* can be seconds, days, months, or never-ending.

Orchestration identity

Each *instance* of an orchestration has an instance identifier (also known as an *instance ID*). By default, each instance ID is an autogenerated GUID. However, instance IDs can also be any user-generated string value. Each orchestration instance ID must be unique within a task hub.

Note: It is generally recommended to use autogenerated instance IDs whenever possible. User-generated instance IDs are intended for scenarios where there is a one-to-one mapping between an orchestration instance and some external application-specific entity.

An orchestration's instance ID is a required parameter for most instance management operations. They are also important for diagnostics, such as searching through orchestration tracking data in Application Insights for troubleshooting or analytics purposes. For this reason, it is recommended to save generated instance IDs to some external location (for example, a database or in application logs) where they can be easily referenced later.

Reliability

Orchestrator functions reliably maintain their execution state by using the event sourcing design pattern. Instead of directly storing the current state of an orchestration, the Durable Task Framework uses an append-only store to record the full series of actions the function orchestration takes.

Durable Functions uses event sourcing transparently. Behind the scenes, the `await` (C#) or `yield` (JavaScript) operator in an orchestrator function yields control of the orchestrator thread back to the Durable Task Framework dispatcher. The dispatcher then commits any new actions that the orchestrator function scheduled (such as calling one or more child functions or scheduling a durable timer) to storage. The transparent commit action appends to the execution history of the orchestration instance. The history is stored in a storage table. The commit action then adds messages to a queue to schedule the actual work. At this point, the orchestrator function can be unloaded from memory.

When an orchestration function is given more work to do, the orchestrator wakes up and re-executes the entire function from the start to rebuild the local state. During the replay, if the code tries to call a function (or do any other async work), the Durable Task Framework consults the execution history of the current orchestration. If it finds that the activity function has already executed and yielded a result, it replays that function's result and the orchestrator code continues to run. Replay continues until the function code is finished or until it has scheduled new async work.

Features and patterns

The table below describes the features and patterns of orchestrator functions.

Pattern/Feature	Description
Sub-orchestrations	Orchestrator functions can call activity functions, but also other orchestrator functions. For example, you can build a larger orchestration out of a library of orchestrator functions. Or, you can run multiple instances of an orchestrator function in parallel.
Durable timers	Orchestrations can schedule durable timers to implement delays or to set up timeout handling on async actions. Use durable timers in orchestrator functions instead of <code>Thread.Sleep</code> and <code>Task.Delay</code> (C#) or <code>setTimeout()</code> and <code>setInterval()</code> (JavaScript).
External events	Orchestrator functions can wait for external events to update an orchestration instance. This Durable Functions feature often is useful for handling a human interaction or other external callbacks.
Error handling	Orchestrator functions can use the error-handling features of the programming language. Existing patterns like <code>try/catch</code> are supported in orchestration code.
Critical sections	Orchestration instances are single-threaded so it isn't necessary to worry about race conditions <i>within</i> an orchestration. However, race conditions are possible when orchestrations interact with external systems. To mitigate race conditions when interacting with external systems, orchestrator functions can define <i>critical sections</i> using a <code>LockAsync</code> method in .NET.
Calling HTTP endpoints	Orchestrator functions aren't permitted to do I/O. The typical workaround for this limitation is to wrap any code that needs to do I/O in an activity function. Orchestrations that interact with external systems frequently use activity functions to make HTTP calls and return the result to the orchestration.
Passing multiple parameters	It isn't possible to pass multiple parameters to an activity function directly. The recommendation is to pass in an array of objects or to use <code>ValueTuples</code> objects in .NET.

Control timing in durable functions

Durable Functions provides *durable timers* for use in orchestrator functions to implement delays or to set up timeouts on async actions. Durable timers should be used in orchestrator functions instead of `Thread.Sleep` and `Task.Delay` (C#), or `setTimeout()` and `setInterval()` (JavaScript), or `time.sleep()` (Python).

You create a durable timer by calling the `CreateTimer` (.NET) method or the `createTimer` (JavaScript) method of the orchestration trigger binding. The method returns a task that completes on a specified date and time.

Timer limitations

When you create a timer that expires at 4:30 pm, the underlying Durable Task Framework enqueues a message that becomes visible only at 4:30 pm. When running in the Azure Functions Consumption plan, the newly visible timer message will ensure that the function app gets activated on an appropriate VM.

Usage for delay

The following example illustrates how to use durable timers for delaying execution. The example is issuing a billing notification every day for 10 days.

```
[FunctionName("BillingIssuer")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    for (int i = 0; i < 10; i++)
    {
        DateTime deadline = context.CurrentUtcDateTime.Add(TimeSpan.From-
Days(1));
        await context.CreateTimer(deadline, CancellationToken.None);
        await context.CallActivityAsync("SendBillingEvent");
    }
}
```

Usage for timeout

This example illustrates how to use durable timers to implement timeouts.

```
[FunctionName("TryGetQuote")]
public static async Task<bool> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    TimeSpan timeout = TimeSpan.FromSeconds(30);
    DateTime deadline = context.CurrentUtcDateTime.Add(timeout);

    using (var cts = new CancellationTokenSource())
    {
        Task activityTask = context.CallActivityAsync("GetQuote");
        Task timeoutTask = context.CreateTimer(deadline, cts.Token);

        Task winner = await Task.WhenAny(activityTask, timeoutTask);
        if (winner == activityTask)
        {
            // success case
            cts.Cancel();
            return true;
        }
    }
}
```

```
        else
        {
            // timeout case
            return false;
        }
    }
}
```

Warning: Use a `CancellationTokenSource` to cancel a durable timer (.NET) or call `cancel()` on the returned `TimerTask` (JavaScript) if your code will not wait for it to complete. The Durable Task Framework will not change an orchestration's status to "completed" until all outstanding tasks are completed or canceled.

This cancellation mechanism doesn't terminate in-progress activity function or sub-orchestration executions. It simply allows the orchestrator function to ignore the result and move on.

Send and wait for events

Orchestrator functions have the ability to wait and listen for external events. This feature of Durable Functions is often useful for handling human interaction or other external triggers.

Wait for events

The `WaitForExternalEvent` (.NET), `waitForExternalEvent` (JavaScript), and `wait_for_external_event` (Python) methods of the orchestration trigger binding allows an orchestrator function to asynchronously wait and listen for an external event. The listening orchestrator function declares the *name* of the event and the *shape of the data* it expects to receive.

The following example listens for a specific single event and takes action when it's received.

```
[FunctionName ("BudgetApproval")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    bool approved = await context.WaitForExternalEvent<bool>("Approval");
    if (approved)
    {
        // approval granted - do the approved action
    }
    else
    {
        // approval denied - send a notification
    }
}
```

Send events

The `RaiseEventAsync` (.NET) or `raiseEvent` (JavaScript) method of the orchestration client binding sends the events that `WaitForExternalEvent` (.NET) or `waitForExternalEvent` (JavaScript) waits for. The `RaiseEventAsync` method takes `eventName` and `eventData` as parameters. The event data must be JSON-serializable.

Below is an example queue-triggered function that sends an "Approval" event to an orchestrator function instance. The orchestration instance ID comes from the body of the queue message.

```
[FunctionName("ApprovalQueueProcessor")]
public static async Task Run(
    [QueueTrigger("approval-queue")] string instanceId,
    [DurableClient] IDurableOrchestrationClient client)
{
    await client.RaiseEventAsync(instanceId, "Approval", true);
}
```

Internally, `RaiseEventAsync` (.NET) or `raiseEvent` (JavaScript) enqueues a message that gets picked up by the waiting orchestrator function. If the instance is not waiting on the specified *event name*, the event message is added to an in-memory queue. If the orchestration instance later begins listening for that *event name*, it will check the queue for event messages.

Knowledge check

Multiple choice

Which of the following durable function types is used to read and update small pieces of state?

- Orchestrator
- Activity
- Entity

Multiple choice

Which application pattern would you use for a durable function that is polling a resource until a specific condition is met?

- Function chaining
- Fan out/fan in
- Monitor

Summary

In this module, you learned how to:

- Describe the app patterns typically used in Durable Functions
- Describe the four durable function types
- Explain the function Task Hubs perform in Durable Functions
- Describe the use of durable orchestrations, timers, and events

Answers

Multiple choice

Which of the following Azure Functions hosting plans is best when predictive scaling and costs are required?

- Functions Premium Plan
- App service plan
- Consumption plan

Explanation

That's correct. App service plans support setting autoscaling rules based on predictive usage.

Multiple choice

An organization wants to implement a serverless workflow to solve a business problem. One of the requirements is the solution needs to use a designer-first (declarative) development model. Which of the choices below meets the requirements?

- Azure Functions
- Azure Logic Apps
- WebJobs

Explanation

That's correct. Azure Logic Apps enables serverless workloads and uses a designer-first (declarative) development model.

Multiple choice

Which of the following is required for a function to run?

- Binding
- Trigger
- Both triggers and bindings

Explanation

That's correct. A trigger defines how a function is invoked and a function must have exactly one trigger.

Multiple choice

Which of the following supports both the `in` and `out` direction settings?

- Bindings
- Trigger
- Connection value

Explanation

That's correct. Input and output bindings use `in` and `out`.

Multiple choice

Which of the following durable function types is used to read and update small pieces of state?

- Orchestrator
- Activity
- Entity

Explanation

That's correct. Entity functions define operations for reading and updating small pieces of state.

Multiple choice

Which application pattern would you use for a durable function that is polling a resource until a specific condition is met?

- Function chaining
- Fan out/fan in
- Monitor

Explanation

That's correct. The monitor pattern refers to a flexible, recurring process in a workflow. An example is polling until specific conditions are met.

Module 3 Develop solutions that use Blob storage

Explore Azure Blob storage

Introduction

Azure Blob storage is Microsoft's object storage solution for the cloud. Blob storage is optimized for storing massive amounts of unstructured data.

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

- Identify the different types of storage accounts and the resource hierarchy for blob storage.
- Explain how data is securely stored and protected through redundancy.
- Create a block blob storage account by using the Azure Cloud Shell.

Explore Azure Blob storage

Azure Blob storage is Microsoft's object storage solution for the cloud. Blob storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that does not adhere to a particular data model or definition, such as text or binary data.

Blob storage is designed for:

- Serving images or documents directly to a browser.
- Storing files for distributed access.
- Streaming video and audio.
- Writing to log files.
- Storing data for backup and restore, disaster recovery, and archiving.
- Storing data for analysis by an on-premises or Azure-hosted service.

Users or client applications can access objects in Blob storage via HTTP/HTTPS, from anywhere in the world. Objects in Blob storage are accessible via the Azure Storage REST API, Azure PowerShell, Azure CLI, or an Azure Storage client library.

An Azure Storage account is the top-level container for all of your Azure Blob storage. The storage account provides a unique namespace for your Azure Storage data that is accessible from anywhere in the world over HTTP or HTTPS.

Types of storage accounts

Azure Storage offers two performance levels of storage accounts, standard and premium. Each performance level supports different features and has its own pricing model.

- **Standard:** This is the standard general-purpose v2 account and is recommended for most scenarios using Azure Storage.
- **Premium:** Premium accounts offer higher performance by using solid-state drives. If you create a premium account you can choose between three account types, block blobs, page blobs, or file shares.

The following table describes the types of storage accounts recommended by Microsoft for most scenarios using Blob storage.

Storage account type	Supported storage services	Usage
Standard general-purpose v2	Blob, Queue, and Table storage, Azure Files	Standard storage account type for blobs, file shares, queues, and tables. Recommended for most scenarios using Azure Storage. If you want support for NFS file shares in Azure Files, use the premium file shares account type.
Premium block blobs	Blob storage	Premium storage account type for block blobs and append blobs. Recommended for scenarios with high transaction rates, or scenarios that use smaller objects or require consistently low storage latency.
Premium page blobs	Page blobs only	Premium storage account type for page blobs only.
Premium file shares	Azure Files	Premium storage account type for file shares only.

Access tiers for block blob data

Azure Storage provides different options for accessing block blob data based on usage patterns. Each access tier in Azure Storage is optimized for a particular pattern of data usage. By selecting the right access tier for your needs, you can store your block blob data in the most cost-effective manner.

The available access tiers are:

- The **Hot** access tier, which is optimized for frequent access of objects in the storage account. The Hot tier has the highest storage costs, but the lowest access costs. New storage accounts are created in the hot tier by default.
- The **Cool** access tier, which is optimized for storing large amounts of data that is infrequently accessed and stored for at least 30 days. The Cool tier has lower storage costs and higher access costs compared to the Hot tier.
- The **Archive** tier, which is available only for individual block blobs. The archive tier is optimized for data that can tolerate several hours of retrieval latency and will remain in the Archive tier for at least 180 days. The archive tier is the most cost-effective option for storing data, but accessing that data is more expensive than accessing data in the hot or cool tiers.

If there is a change in the usage pattern of your data, you can switch between these access tiers at any time.

Discover Azure Blob storage resource types

Blob storage offers three types of resources:

- The **storage account**.
- A **container** in the storage account
- A **blob** in a container

Storage accounts

A storage account provides a unique namespace in Azure for your data. Every object that you store in Azure Storage has an address that includes your unique account name. The combination of the account name and the Azure Storage blob endpoint forms the base address for the objects in your storage account.

For example, if your storage account is named *mystorageaccount*, then the default endpoint for Blob storage is:

`http://mystorageaccount.blob.core.windows.net`

Containers

A container organizes a set of blobs, similar to a directory in a file system. A storage account can include an unlimited number of containers, and a container can store an unlimited number of blobs. The container name must be lowercase.

Blobs

Azure Storage supports three types of blobs:

- **Block blobs** store text and binary data, up to about 4.7 TB. Block blobs are made up of blocks of data that can be managed individually.
- **Append blobs** are made up of blocks like block blobs, but are optimized for append operations. Append blobs are ideal for scenarios such as logging data from virtual machines.

- **Page blobs** store random access files up to 8 TB in size. Page blobs store virtual hard drive (VHD) files and serve as disks for Azure virtual machines.

Explore Azure Storage security features

Azure Storage provides a comprehensive set of security capabilities that together enable developers to build secure applications:

- All data (including metadata) written to Azure Storage is automatically encrypted using Storage Service Encryption (SSE).
- Azure Active Directory (Azure AD) and Role-Based Access Control (RBAC) are supported for Azure Storage for both resource management operations and data operations, as follows:
 - You can assign RBAC roles scoped to the storage account to security principals and use Azure AD to authorize resource management operations such as key management.
 - Azure AD integration is supported for blob and queue data operations. You can assign RBAC roles scoped to a subscription, resource group, storage account, or an individual container or queue to a security principal or a managed identity for Azure resources.
- Data can be secured in transit between an application and Azure by using Client-Side Encryption, HTTPS, or SMB 3.0.
- OS and data disks used by Azure virtual machines can be encrypted using Azure Disk Encryption.
- Delegated access to the data objects in Azure Storage can be granted using a shared access signature.

Azure Storage encryption for data at rest

Azure Storage automatically encrypts your data when persisting it to the cloud. Encryption protects your data and help you meet your organizational security and compliance commitments. Data in Azure Storage is encrypted and decrypted transparently using 256-bit AES encryption, one of the strongest block ciphers available, and is FIPS 140-2 compliant. Azure Storage encryption is similar to BitLocker encryption on Windows.

Azure Storage encryption is enabled for all new and existing storage accounts and cannot be disabled. Because your data is secured by default, you don't need to modify your code or applications to take advantage of Azure Storage encryption.

Storage accounts are encrypted regardless of their performance tier (standard or premium) or deployment model (Azure Resource Manager or classic). All Azure Storage redundancy options support encryption, and all copies of a storage account are encrypted. All Azure Storage resources are encrypted, including blobs, disks, files, queues, and tables. All object metadata is also encrypted.

Encryption does not affect Azure Storage performance. There is no additional cost for Azure Storage encryption.

Encryption key management

You can rely on Microsoft-managed keys for the encryption of your storage account, or you can manage encryption with your own keys. If you choose to manage encryption with your own keys, you have two options:

- You can specify a *customer-managed* key to use for encrypting and decrypting all data in the storage account. A customer-managed key is used to encrypt all data in all services in your storage account.

- You can specify a *customer-provided* key on Blob storage operations. A client making a read or write request against Blob storage can include an encryption key on the request for granular control over how blob data is encrypted and decrypted.

The following table compares key management options for Azure Storage encryption.

	Microsoft-managed keys	Customer-managed keys	Customer-provided keys
Encryption/decryption operations	Azure	Azure	Azure
Azure Storage services supported	All	Blob storage, Azure Files	Blob storage
Key storage	Microsoft key store	Azure Key Vault	Azure Key Vault or any other key store
Key rotation responsibility	Microsoft	Customer	Customer
Key usage	Microsoft	Azure portal, Storage Resource Provider REST API, Azure Storage management libraries, PowerShell, CLI	Azure Storage REST API (Blob storage), Azure Storage client libraries
Key access	Microsoft only	Microsoft, Customer	Customer only

Evaluate Azure Storage redundancy options

Azure Storage always stores multiple copies of your data so that it is protected from planned and unplanned events, including transient hardware failures, network or power outages, and massive natural disasters. Redundancy ensures that your storage account meets its availability and durability targets even in the face of failures.

When deciding which redundancy option is best for your scenario, consider the tradeoffs between lower costs and higher availability. The factors that help determine which redundancy option you should choose include:

- How your data is replicated in the primary region
- Whether your data is replicated to a second region that is geographically distant to the primary region, to protect against regional disasters
- Whether your application requires read access to the replicated data in the secondary region if the primary region becomes unavailable for any reason

Redundancy in the primary region

Data in an Azure Storage account is always replicated three times in the primary region. Azure Storage offers two options for how your data is replicated in the primary region.

- **Locally redundant storage (LRS):** Copies your data synchronously three times within a single physical location in the primary region. LRS is the least expensive replication option, but is not recommended for applications requiring high availability or durability.
- **Zone-redundant storage (ZRS):** Copies your data synchronously across three Azure availability zones in the primary region. For applications requiring high availability, Microsoft recommends using ZRS in the primary region, and also replicating to a secondary region.

Redundancy in a secondary region

For applications requiring high durability, you can choose to additionally copy the data in your storage account to a secondary region that is hundreds of miles away from the primary region. If your storage account is copied to a secondary region, then your data is durable even in the case of a complete regional outage or a disaster in which the primary region isn't recoverable.

When you create a storage account, you select the primary region for the account. The paired secondary region is determined based on the primary region, and can't be changed.

Azure Storage offers two options for copying your data to a secondary region:

- **Geo-redundant storage (GRS)** copies your data synchronously three times within a single physical location in the primary region using LRS. It then copies your data asynchronously to a single physical location in the secondary region. Within the secondary region, your data is copied synchronously three times using LRS.
- **Geo-zone-redundant storage (GZRS)** copies your data synchronously across three Azure availability zones in the primary region using ZRS. It then copies your data asynchronously to a single physical location in the secondary region. Within the secondary region, your data is copied synchronously three times using LRS.

Exercise: Create a block blob storage account

The block blob storage account type lets you create block blobs with premium performance characteristics. This type of storage account is optimized for workloads with high transaction rates or that require very fast access times.

In this exercise you will create a block blob storage account by using the Azure portal, and in the Cloud Shell using the Azure CLI.

Prerequisites

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

- 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 account in the Azure portal

To create a block blob storage account in the Azure portal, follow these steps:

1. In the Azure portal, select **All services** > the **Storage** category > **Storage accounts**.
2. Under **Storage accounts**, select **+ Create**.
3. In the **Subscription** field, select the subscription in which to create the storage account.
4. In the **Resource group** field, select **Create new** and enter *az204-blob-rg* as the name for the new resource group.
5. In the **Storage account name** field, enter a name for the account. Note the following guidelines:
 - The name must be unique across Azure.
 - The name must be between three and 24 characters long.
 - The name can include only numbers and lowercase letters.

6. In the **Location** field, select a location for the storage account, or use the default location.
7. For the rest of the settings, configure the following:

Field	Value
Performance	Select Premium .
Premium account type	Select Block blobs .
Replication	Leave the default setting of Locally-redundant storage (LRS) .

8. Select **Review + create** to review the storage account settings.
9. Select **Create**.

Create account by using Azure Cloud Shell

1. Login to the **Azure portal**¹ and open the Cloud Shell.
 - You can also login to the **Azure Cloud Shell**² directly.
2. Create a new resource group. Replace <myLocation> with a region near you.

Note: Skip this step if you created a resource group in the *Create account in the Azure portal* section above.

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

3. Create the block blob storage account. See Step 5 in the *Create account in the Azure portal* instructions above for the storage account name requirements. Replace <myLocation> with a region near you.

```
az storage account create --resource-group az204-blob-rg --name \
<myStorageAcct> --location <myLocation> \
--kind BlockBlobStorage --sku Premium_LRS
```

Clean up resources

When you no longer need the resources in this walkthrough use the following command to delete the resource group and associated resources.

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

¹ <https://portal.azure.com>

² <https://shell.azure.com>

Knowledge check

Multiple choice

Which of the following types of blobs are used to store virtual hard drive files?

- Block blobs
- Append blobs
- Page blobs

Multiple choice

Which of the following types of storage accounts is recommended for most scenarios using Azure Storage?

- General-purpose v2
- General-purpose v1
- FileStorage

Summary

In this module you learned how to:

- Identify the different types of storage accounts and the resource hierarchy for blob storage.
- Explain how data is securely stored and protected through redundancy.
- Create a block blob storage account by using the Azure Cloud Shell.

Manage the Azure Blob storage lifecycle

Introduction

Data sets have unique lifecycles. Early in the lifecycle, people access some data often. But the need for access drops drastically as the data ages. Some data stays idle in the cloud and is rarely accessed once stored.

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

- Describe how each of the access tiers are optimized.
- Create and implement a lifecycle policy.
- Rehydrate blob data stored in an archive tier.

Explore the Azure Blob storage lifecycle

Data sets have unique lifecycles. Early in the lifecycle, people access some data often. But the need for access drops drastically as the data ages. Some data stays idle in the cloud and is rarely accessed once stored. Some data expires days or months after creation, while other data sets are actively read and modified throughout their lifetimes.

Access tiers

Azure storage offers different access tiers, allowing you to store blob object data in the most cost-effective manner. Available access tiers include:

- **Hot** - Optimized for storing data that is accessed frequently.
- **Cool** - Optimized for storing data that is infrequently accessed and stored for at least 30 days.
- **Archive** - Optimized for storing data that is rarely accessed and stored for at least 180 days with flexible latency requirements, on the order of hours.

The following considerations apply to the different access tiers:

- The access tier can be set on a blob during or after upload.
- Only the hot and cool access tiers can be set at the account level. The archive access tier can only be set at the blob level.
- Data in the cool access tier has slightly lower availability, but still has high durability, retrieval latency, and throughput characteristics similar to hot data.
- Data in the archive access tier is stored offline. The archive tier offers the lowest storage costs but also the highest access costs and latency.
- The hot and cool tiers support all redundancy options. The archive tier supports only LRS, GRS, and RA-GRS.
- Data storage limits are set at the account level and not per access tier. You can choose to use all of your limit in one tier or across all three tiers.

Manage the data lifecycle

Azure Blob storage lifecycle management offers a rich, rule-based policy for General Purpose v2 and Blob storage accounts. Use the policy to transition your data to the appropriate access tiers or expire at the end of the data's lifecycle. The lifecycle management policy lets you:

- Transition blobs to a cooler storage tier (hot to cool, hot to archive, or cool to archive) to optimize for performance and cost
- Delete blobs at the end of their lifecycles
- Define rules to be run once per day at the storage account level
- Apply rules to containers or a subset of blobs (using prefixes as filters)

Consider a scenario where data gets frequent access during the early stages of the lifecycle, but only occasionally after two weeks. Beyond the first month, the data set is rarely accessed. In this scenario, hot storage is best during the early stages. Cool storage is most appropriate for occasional access. Archive storage is the best tier option after the data ages over a month. By adjusting storage tiers in respect to the age of data, you can design the least expensive storage options for your needs. To achieve this transition, lifecycle management policy rules are available to move aging data to cooler tiers.

Discover Blob storage lifecycle policies

A lifecycle management policy is a collection of rules in a JSON document. Each rule definition within a policy includes a filter set and an action set. The filter set limits rule actions to a certain set of objects within a container or objects names. The action set applies the tier or delete actions to the filtered set of objects.:

```
{
  "rules": [
    {
      "name": "rule1",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {...}
    },
    {
      "name": "rule2",
      "type": "Lifecycle",
      "definition": {...}
    }
  ]
}
```

A policy is a collection of rules:

Parameter name	Parameter type	Notes
rules	An array of rule objects	At least one rule is required in a policy. You can define up to 100 rules in a policy.

Each rule within the policy has several parameters:

Parameter name	Parameter type	Notes	Required
name	String	A rule name can include up to 256 alphanumeric characters. Rule name is case-sensitive. It must be unique within a policy.	True
enabled	Boolean	An optional boolean to allow a rule to be temporary disabled. Default value is true if it's not set.	False
type	An enum value	The current valid type is Lifecycle.	True
definition	An object that defines the lifecycle rule	Each definition is made up of a filter set and an action set.	True

Rules

Each rule definition includes a filter set and an action set. The filter set limits rule actions to a certain set of objects within a container or objects names. The action set applies the tier or delete actions to the filtered set of objects.

The following sample rule filters the account to run the actions on objects that exist inside `container1` and start with `foo`.

- Tier blob to cool tier 30 days after last modification
- Tier blob to archive tier 90 days after last modification
- Delete blob 2,555 days (seven years) after last modification
- Delete blob snapshots 90 days after snapshot creation

```
{
  "rules": [
    {
      "name": "ruleFoo",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {
        "filters": {
          "blobTypes": [ "blockBlob" ],
          "prefixMatch": [ "container1/foo" ]
        },
        "actions": {
          "baseBlob": {
            "tierToCool": { "daysAfterModificationGreaterThan": 30 },
            "tierToArchive": { "daysAfterModificationGreaterThan": 90 },
            "delete": { "daysAfterModificationGreaterThan": 2555 }
          },
          "snapshot": {
            ...
          }
        }
      }
    }
  ]
}
```

```
        "delete": { "daysAfterCreationGreaterThanOrEqual": 90 }
    }
}
]
}
```

Rule filters

Filters limit rule actions to a subset of blobs within the storage account. If more than one filter is defined, a logical AND runs on all filters. Filters include:

Filter name	Filter type	Is Required
blobTypes	An array of predefined enum values.	Yes
prefixMatch	An array of strings for prefixes to be match. Each rule can define up to 10 prefixes. A prefix string must start with a container name.	No
blobIndexMatch	An array of dictionary values consisting of blob index tag key and value conditions to be matched. Each rule can define up to 10 blob index tag condition.	No

Rule actions

Actions are applied to the filtered blobs when the run condition is met.

Lifecycle management supports tiering and deletion of blobs and deletion of blob snapshots. Define at least one action for each rule on blobs or blob snapshots.

Action	Base Blob	Snapshot	Version
tierToCool	Supported for block-Blob	Supported	Supported
enableAutoTierToHot-FromCool	Supported for block-Blob	Not supported	Not supported
tierToArchive	Supported for block-Blob	Supported	Supported
delete	Supported for block-Blob and appendBlob	Supported	Supported

Note: If you define more than one action on the same blob, lifecycle management applies the least expensive action to the blob. For example, action `delete` is cheaper than action `tierToArchive`. Action `tierToArchive` is cheaper than action `tierToCool`.

The run conditions are based on age. Base blobs use the last modified time to track age, and blob snapshots use the snapshot creation time to track age.

Action run condition	Condition value	Description
daysAfterModificationGreaterThan	Integer value indicating the age in days	The condition for base blob actions
daysAfterCreationGreater Than	Integer value indicating the age in days	The condition for blob snapshot actions

Implement Blob storage lifecycle policies

You can add, edit, or remove a policy by using any of the following methods:

- Azure portal
- Azure PowerShell
- Azure CLI
- REST APIs

Below are the steps and some examples for the Portal and Azure CLI.

Azure portal

There are two ways to add a policy through the Azure portal: Azure portal List view, and Azure portal Code view.

Azure portal List view

1. Sign in to the [Azure portal](#)³.
2. Select **All resources** and then select your storage account.
3. Under **Data management**, select **Lifecycle management** to view or change your rules.
4. Select the **List view** tab.
5. Select **Add rule** and then fill out the **Action set** form fields. In the following example, blobs are moved to cool storage if they haven't been modified for 30 days.
6. Select **Filter set** to add an optional filter. Then, select Browse to specify a container and folder by which to filter.
7. Select **Review + add** to review the policy settings.
8. Select **Add** to add the new policy.

Azure portal Code view

1. Follow the first three steps above in the **List view** section.
2. Select the **Code view** tab. The following JSON is an example of a policy that moves a block blob whose name begins with *log* to the cool tier if it has been more than 30 days since the blob was modified.

```
{
  "rules": [
    {
      "blobType": "BlockBlob",
      "name_starts_with": "log",
      "coolTier": true,
      "daysAfterModificationGreaterThan": 30
    }
  ]
}
```

³ <https://portal.azure.com>

```
"enabled": true,
"name": "move-to-cool",
"type": "Lifecycle",
"definition": {
  "actions": {
    "baseBlob": {
      "tierToCool": {
        "daysAfterModificationGreaterThan": 30
      }
    }
  },
  "filters": {
    "blobTypes": [
      "blockBlob"
    ],
    "prefixMatch": [
      "sample-container/log"
    ]
  }
}
]
}
```

3. Select **Save**.

Azure CLI

To add a lifecycle management policy with Azure CLI, write the policy to a JSON file, then call the `az storage account management-policy create` command to create the policy.

```
az storage account management-policy create \
  --account-name <storage-account> \
  --policy @policy.json \
  --resource-group <resource-group>
```

A lifecycle management policy must be read or written in full. Partial updates are not supported.

Rehydrate blob data from the archive tier

While a blob is in the archive access tier, it's considered to be offline and can't be read or modified. In order to read or modify data in an archived blob, you must first rehydrate the blob to an online tier, either the hot or cool tier. There are two options for rehydrating a blob that is stored in the archive tier:

- **Copy an archived blob to an online tier:** You can rehydrate an archived blob by copying it to a new blob in the hot or cool tier with the **Copy Blob⁴** or **Copy Blob from URL⁵** operation. Microsoft recommends this option for most scenarios.

⁴ <https://docs.microsoft.com/rest/api/storageservices/copy-blob>

⁵ <https://docs.microsoft.com/rest/api/storageservices/copy-blob-from-url>

- **Change a blob's access tier to an online tier:** You can rehydrate an archived blob to hot or cool by changing its tier using the **Set Blob Tier**⁶ operation.

Rehydrating a blob from the archive tier can take several hours to complete. Microsoft recommends rehydrating larger blobs for optimal performance. Rehydrating several small blobs concurrently may require additional time.

Rehydration priority

When you rehydrate a blob, you can set the priority for the rehydration operation via the optional `x-ms-rehydrate-priority` header on a **Set Blob Tier**⁷ or **Copy Blob/Copy Blob From URL** operation. Rehydration priority options include:

- **Standard priority:** The rehydration request will be processed in the order it was received and may take up to 15 hours.
- **High priority:** The rehydration request will be prioritized over standard priority requests and may complete in under one hour for objects under 10 GB in size.

To check the rehydration priority while the rehydration operation is underway, call **Get Blob Properties**⁸ to return the value of the `x-ms-rehydrate-priority` header. The rehydration priority property returns either *Standard* or *High*.

Copy an archived blob to an online tier

You can use either the **Copy Blob** or **Copy Blob from URL** operation to copy the blob. When you copy an archived blob to a new blob an online tier, the source blob remains unmodified in the archive tier. You must copy the archived blob to a new blob with a different name or to a different container. You cannot overwrite the source blob by copying to the same blob.

Copying an archived blob to an online destination tier is supported within the same storage account only. You cannot copy an archived blob to a destination blob that is also in the archive tier.

The following table shows the behavior of a blob copy operation, depending on the tiers of the source and destination blob.

	Hot tier source	Cool tier source	Archive tier source
Hot tier destination	Supported	Supported	Supported within the same account. Requires blob rehydration.
Cool tier destination	Supported	Supported	Supported within the same account. Requires blob rehydration.
Archive tier destination	Supported	Supported	Unsupported

Change a blob's access tier to an online tier

The second option for rehydrating a blob from the archive tier to an online tier is to change the blob's tier by calling **Set Blob Tier**. With this operation, you can change the tier of the archived blob to either hot or cool.

⁶ <https://docs.microsoft.com/rest/api/storageservices/set-blob-tier>

⁷ <https://docs.microsoft.com/rest/api/storageservices/set-blob-tier>

⁸ <https://docs.microsoft.com/rest/api/storageservices/get-blob-properties>

Once a **Set Blob Tier** request is initiated, it cannot be canceled. During the rehydration operation, the blob's access tier setting continues to show as archived until the rehydration process is complete.

To learn how to rehydrate a blob by changing its tier to an online tier, see **Rehydrate a blob by changing its tier⁹**.

Caution: Changing a blob's tier doesn't affect its last modified time. If there is a lifecycle management policy in effect for the storage account, then rehydrating a blob with **Set Blob Tier** can result in a scenario where the lifecycle policy moves the blob back to the archive tier after rehydration because the last modified time is beyond the threshold set for the policy.

Knowledge check

Multiple choice

Which access tier is considered to be offline and can't be read or modified?

- Cool
- Archive
- Hot

Multiple choice

Which of the following storage account types supports lifecycle policies?

- General Purpose v1
- General Purpose v2
- FileStorage

Summary

In this module you learned how to:

- Describe how each of the access tiers are optimized.
- Create and implement a lifecycle policy.
- Rehydrate blob data stored in an archive tier.

⁹ <https://docs.microsoft.com/azure/storage/blobs/archive-rehydrate-to-online-tier#rehydrate-a-blob-by-changing-its-tier>

Work with Azure Blob storage

Introduction

The Azure Storage client libraries for .NET offer a convenient interface for making calls to Azure Storage.

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

- Create an application to create and manipulate data by using the Azure Storage client library for Blob storage.
- Manage container properties and metadata by using .NET and REST.

Explore Azure Blob storage client library

The Azure Storage client libraries for .NET offer a convenient interface for making calls to Azure Storage. The latest version of the Azure Storage client library is version 12.x. Microsoft recommends using version 12.x for new applications.

Below are the classes in the `Azure.Storage.Blobs` namespace and their purpose:

Class	Description
<code>BlobClient</code>	The <code>BlobClient</code> allows you to manipulate Azure Storage blobs.
<code>BlobClientOptions</code>	Provides the client configuration options for connecting to Azure Blob Storage.
<code>BlobContainerClient</code>	The <code>BlobContainerClient</code> allows you to manipulate Azure Storage containers and their blobs.
<code>BlobServiceClient</code>	The <code>BlobServiceClient</code> allows you to manipulate Azure Storage service resources and blob containers. The storage account provides the top-level namespace for the Blob service.
<code>BlobUriBuilder</code>	The <code>BlobUriBuilder</code> class provides a convenient way to modify the contents of a Uri instance to point to different Azure Storage resources like an account, container, or blob.

Exercise: Create blob storage resources by using the .NET client library

This exercise uses the Azure Blob storage client library to show you how to perform the following actions on Azure Blob storage in a console app:

- Create a container
- Upload blobs to a container
- List the blobs in a container
- Download blobs
- Delete 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:** You can install Visual Studio Code from <https://code.visualstudio.com>¹⁰.
- **Azure CLI:** You can install the Azure CLI from <https://docs.microsoft.com/cli/azure/install-azure-cli>.
- The **.NET Core 3.1 SDK**, or **.NET 5.0 SDK**. You can install from <https://dotnet.microsoft.com/download>.

Setting up

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

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

```
az login
```

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

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

4. Create a storage account. We need a storage account created to use in the application. Replace <myLocation> with the same region you used for the resource group. Replace <myStorageAcct> with a unique name.

```
az storage account create --resource-group az204-blob-rg --name <myStorageAcct> --location <myLocation> --sku Standard_LRS
```

Note: Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only. Your storage account name must be unique within Azure. No two storage accounts can have the same name.

5. Get credentials for the storage account.

- Navigate to the [Azure portal](#)¹¹.
- Locate the storage account created.
- Select **Access keys** in the **Security + networking** section of the navigation pane. Here, you can view your account access keys and the complete connection string for each key.
- Find the **Connection string** value under **key1**, and select the **Copy** button to copy the connection string. You will add the connection string value to the code in the next section.
- In the **Blob** section of the storage account overview, select **Containers**. Leave the windows open so you can view changes made to the storage as you progress through the exercise.

¹⁰ <https://code.visualstudio.com/>

¹¹ <https://portal.azure.com/>

Prepare the .NET project

In this section we'll create project named *az204-blob* and install the Azure Blob Storage client library.

1. In the VS Code terminal navigate to a directory where you want to store your project.
2. In the terminal, use the `dotnet new` command to create a new console app. This command creates a simple "Hello World" C# project with a single source file: *Program.cs*.

```
dotnet new console -n az204-blob
```

3. Use the following commands to switch to the newly created *az204-blob* folder and build the app to verify that all is well.

```
cd az204-blob  
dotnet build
```

4. Inside the *az204-blob* folder, create another folder named *data*. This is where the blob data files will be created and stored.

```
mkdir data
```

5. While still in the application directory, install the Azure Blob Storage client library for .NET package by using the `dotnet add package` command.

```
dotnet add package Azure.Storage.Blobs
```

Tip: Leave the console window open so you can use it to build and run the app later in the exercise.

6. Open the *Program.cs* file in your editor, and replace the contents with the following code.

```
using Azure.Storage.Blobs;  
using Azure.Storage.Blobs.Models;  
using System;  
using System.IO;  
using System.Threading.Tasks;  
  
namespace az204_blob  
{  
    class Program  
    {  
        public static void Main()  
        {  
            Console.WriteLine("Azure Blob Storage exercise\n");  
  
            // Run the examples asynchronously, wait for the results before proceeding  
            ProcessAsync().GetAwaiter().GetResult();  
  
            Console.WriteLine("Press enter to exit the sample application.");  
            Console.ReadLine();  
        }  
  
        private static async Task ProcessAsync()
```

```
{  
    // Copy the connection string from the portal in the variable below.  
    string storageConnectionString = "CONNECTION STRING";  
  
    // Create a client that can authenticate with a connection string  
    BlobServiceClient blobServiceClient = new BlobServiceClient(storageConnectionString);  
  
    // EXAMPLE CODE STARTS BELOW HERE  
  
}  
}  
}
```

7. Set the `storageConnectionString` variable to the value you copied from the portal.

Build the full app

For each of the following sections below you'll find a brief description of the action being taken as well as the code snippet you'll add to the project. Each new snippet is appended to the one before it, and we'll build and run the console app at the end.

For each example below copy the code and append it to the previous snippet in the example code section of the `Program.cs` file.

Create a container

To create the container first create an instance of the `BlobServiceClient` class, then call the `CreateBlobContainerAsync` method to create the container in your storage account. A GUID value is appended to the container name to ensure that it is unique. The `CreateBlobContainerAsync` method will fail if the container already exists.

```
//Create a unique name for the container  
string containerName = "wtblob" + Guid.NewGuid().ToString();  
  
// Create the container and return a container client object  
BlobContainerClient containerClient = await blobServiceClient.CreateBlob-  
ContainerAsync(containerName);  
Console.WriteLine("A container named '" + containerName + "' has been  
created. " +  
    "\nTake a minute and verify in the portal." +  
    "\nNext a file will be created and uploaded to the container.");  
Console.WriteLine("Press 'Enter' to continue.");  
Console.ReadLine();
```

Upload blobs to a container

The following code snippet gets a reference to a `BlobClient` object by calling the `GetBlobClient` method on the container created in the previous section. It then uploads the selected local file to the

blob by calling the `UploadAsync` method. This method creates the blob if it doesn't already exist, and overwrites it if it does.

```
// Create a local file in the ./data/ directory for uploading and download-
ing
string localPath = "./data/";
string fileName = "wtfile" + Guid.NewGuid().ToString() + ".txt";
string localFilePath = Path.Combine(localPath, fileName);

// Write text to the file
await File.WriteAllTextAsync(localFilePath, "Hello, World!");

// Get a reference to the blob
BlobClient blobClient = containerClient.GetBlobClient(fileName);

Console.WriteLine("Uploading to Blob storage as blob:\n\t {0}\n", blobCli-
ent.Uri);

// Open the file and upload its data
using FileStream uploadFileStream = File.OpenRead(localFilePath);
await blobClient.UploadAsync(uploadFileStream, true);
uploadFileStream.Close();

Console.WriteLine("\nThe file was uploaded. We'll verify by listing" +
    " the blobs next.");
Console.WriteLine("Press 'Enter' to continue.");
Console.ReadLine();
```

List the blobs in a container

List the blobs in the container by using the `GetBlobsAsync` method. In this case, only one blob has been added to the container, so the listing operation returns just that one blob.

```
// List blobs in the container
Console.WriteLine("Listing blobs...");
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())
{
    Console.WriteLine("\t" + blobItem.Name);
}

Console.WriteLine("\nYou can also verify by looking inside the " +
    "container in the portal." +
    "\nNext the blob will be downloaded with an altered file name.");
Console.WriteLine("Press 'Enter' to continue.");
Console.ReadLine();
```

Download blobs

Download the blob created previously to your local file system by using the `DownloadAsync` method. The example code adds a suffix of "DOWNLOADED" to the blob name so that you can see both files in local file system.

```
// Download the blob to a local file
// Append the string "DOWNLOADED" before the .txt extension
string downloadFilePath = localFilePath.Replace(".txt", "DOWNLOADED.txt");

Console.WriteLine("\nDownloading blob to\n\t{0}\n", downloadFilePath);

// Download the blob's contents and save it to a file
BlobDownloadInfo download = await blobClient.DownloadAsync();

using (FileStream downloadFileStream = File.OpenWrite(downloadFilePath))
{
    await download.Content.CopyToAsync(downloadFileStream);
    downloadFileStream.Close();
}
Console.WriteLine("\nLocate the local file to verify it was downloaded.");
Console.WriteLine("The next step is to delete the container and local
files.");
Console.WriteLine("Press 'Enter' to continue.");
Console.ReadLine();
```

Delete a container

The following code cleans up the resources the app created by deleting the entire container using `DeleteAsync`. It also deletes the local files created by the app.

```
// Delete the container and clean up local files created
Console.WriteLine("\nDeleting blob container...");
await containerClient.DeleteAsync();

Console.WriteLine("Deleting the local source and downloaded files...");
File.Delete(localFilePath);
File.Delete(downloadFilePath);

Console.WriteLine("Finished cleaning up.");
```

Run the code

Now that the app is complete it's time to build and run it. Ensure you are in your application directory and run the following commands:

```
dotnet build
dotnet run
```

There are many prompts in the app to allow you to take the time to see what's happening in the portal after each step.

Clean up other resources

The app deleted the resources it created. You can delete all of the resources created for this exercise by using the command below. You will need to confirm that you want to delete the resources.

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

Manage container properties and metadata by using .NET

Blob containers support system properties and user-defined metadata, in addition to the data they contain.

- **System properties:** System properties exist on each Blob storage resource. Some of them can be read or set, while others are read-only. Under the covers, some system properties correspond to certain standard HTTP headers. The Azure Storage client library for .NET maintains these properties for you.
- **User-defined metadata:** User-defined metadata consists of one or more name-value pairs that you specify for a Blob storage resource. You can use metadata to store additional values with the resource. Metadata values are for your own purposes only, and do not affect how the resource behaves.

Metadata name/value pairs are valid HTTP headers, and so should adhere to all restrictions governing HTTP headers. Metadata names must be valid HTTP header names and valid C# identifiers, may contain only ASCII characters, and should be treated as case-insensitive. Metadata values containing non-ASCII characters should be Base64-encoded or URL-encoded.

Retrieve container properties

To retrieve container properties, call one of the following methods of the `BlobContainerClient` class:

- `GetProperties`
- `GetPropertiesAsync`

The following code example fetches a container's system properties and writes some property values to a console window:

```
private static async Task ReadContainerPropertiesAsync(BlobContainerClient container)
{
    try
    {
        // Fetch some container properties and write out their values.
        var properties = await container.GetPropertiesAsync();
        Console.WriteLine($"Properties for container {container.Uri}");
        Console.WriteLine($"Public access level: {properties.Value.PublicAccess}");
        Console.WriteLine($"Last modified time in UTC: {properties.Value.LastModified}");
    }
    catch (RequestFailedException e)
```

```
        {
            Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
            Console.WriteLine(e.Message);
            Console.ReadLine();
        }
    }
```

Set and retrieve metadata

You can specify metadata as one or more name-value pairs on a blob or container resource. To set metadata, add name-value pairs to an `IDictionary` object, and then call one of the following methods of the `BlobContainerClient` class to write the values:

- `SetMetadata`
- `SetMetadataAsync`

The name of your metadata must conform to the naming conventions for C# identifiers. Metadata names preserve the case with which they were created, but are case-insensitive when set or read. If two or more metadata headers with the same name are submitted for a resource, Blob storage comma-separates and concatenates the two values and return HTTP response code 200 (OK).

The following code example sets metadata on a container.

```
public static async Task AddContainerMetadataAsync(BlobContainerClient
    container)
{
    try
    {
        IDictionary<string, string> metadata =
            new Dictionary<string, string>();

        // Add some metadata to the container.
        metadata.Add("docType", "textDocuments");
        metadata.Add("category", "guidance");

        // Set the container's metadata.
        await container.SetMetadataAsync(metadata);
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
```

The `GetProperties` and `GetPropertiesAsync` methods are used to retrieve metadata in addition to properties as shown earlier.

The following code example retrieves metadata from a container.

```
public static async Task ReadContainerMetadataAsync(BlobContainerClient
    container)
```

```
{  
    try  
    {  
        var properties = await container.GetPropertiesAsync();  
  
        // Enumerate the container's metadata.  
        Console.WriteLine("Container metadata:");  
        foreach (var metadataItem in properties.Value.Metadata)  
        {  
            Console.WriteLine($"\\tKey: {metadataItem.Key}");  
            Console.WriteLine($"\\tValue: {metadataItem.Value}");  
        }  
    }  
    catch (RequestFailedException e)  
    {  
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");  
        Console.WriteLine(e.Message);  
        Console.ReadLine();  
    }  
}
```

Set and Retrieve properties and metadata for blob resources by using REST

Containers and blobs support custom metadata, represented as HTTP headers. Metadata headers can be set on a request that creates a new container or blob resource, or on a request that explicitly creates a property on an existing resource.

Metadata header format

Metadata headers are name/value pairs. The format for the header is:

```
x-ms-meta-name:string-value
```

Beginning with version 2009-09-19, metadata names must adhere to the naming rules for C# identifiers.

Names are case-insensitive. Note that metadata names preserve the case with which they were created, but are case-insensitive when set or read. If two or more metadata headers with the same name are submitted for a resource, the Blob service returns status code 400 (Bad Request).

The metadata consists of name/value pairs. The total size of all metadata pairs can be up to 8KB in size.

Metadata name/value pairs are valid HTTP headers, and so they adhere to all restrictions governing HTTP headers.

Operations on metadata

Metadata on a blob or container resource can be retrieved or set directly, without returning or altering the content of the resource.

Note that metadata values can only be read or written in full; partial updates are not supported. Setting metadata on a resource overwrites any existing metadata values for that resource.

Retrieving properties and metadata

The GET and HEAD operations both retrieve metadata headers for the specified container or blob. These operations return headers only; they do not return a response body. The URI syntax for retrieving metadata headers on a container is as follows:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer?restype=container
```

The URI syntax for retrieving metadata headers on a blob is as follows:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=metadata
```

Setting Metadata Headers

The PUT operation sets metadata headers on the specified container or blob, overwriting any existing metadata on the resource. Calling PUT without any headers on the request clears all existing metadata on the resource.

The URI syntax for setting metadata headers on a container is as follows:

```
PUT https://myaccount.blob.core.windows.net/mycontainer?comp=metadata&restype=container
```

The URI syntax for setting metadata headers on a blob is as follows:

```
PUT https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=metadata
```

Standard HTTP properties for containers and blobs

Containers and blobs also support certain standard HTTP properties. Properties and metadata are both represented as standard HTTP headers; the difference between them is in the naming of the headers. Metadata headers are named with the header prefix `x-ms-meta-` and a custom name. Property headers use standard HTTP header names, as specified in the Header Field Definitions section 14 of the HTTP/1.1 protocol specification.

The standard HTTP headers supported on containers include:

- ETag
- Last-Modified

The standard HTTP headers supported on blobs include:

- ETag
- Last-Modified
- Content-Length
- Content-Type
- Content-MD5
- Content-Encoding

- Content-Language
- Cache-Control
- Origin
- Range

Knowledge check

Multiple choice

Which of the following standard HTTP headers are supported for both containers and blobs when setting properties by using REST?

- Last-Modified
- Content-Length
- Origin

Multiple choice

Which of the following classes of the Azure Storage client library for .NET allows you to manipulate both Azure Storage containers and their blobs?

- BlobClient
- BlobContainerClient
- BlobUriBuilder

Summary

In this module you learned how to:

- Create an application to create and manipulate data by using the Azure Storage client library for Blob storage.
- Manage container properties and metadata by using .NET and REST.

Answers

Multiple choice

Which of the following types of blobs are used to store virtual hard drive files?

- Block blobs
- Append blobs
- Page blobs

Explanation

That's correct. Page blobs store random access files up to 8 TB in size, and are used to store virtual hard drive (VHD) files and serve as disks for Azure virtual machines.

Multiple choice

Which of the following types of storage accounts is recommended for most scenarios using Azure Storage?

- General-purpose v2
- General-purpose v1
- FileStorage

Explanation

That's correct. This supports blobs, files, queues, and tables. It is recommended for most scenarios using Azure Storage.

Multiple choice

Which access tier is considered to be offline and can't be read or modified?

- Cool
- Archive
- Hot

Explanation

That's correct. Blobs in the archive tier must be rehydrated to either the hot or cool tears before it can be read or modified.

Multiple choice

Which of the following storage account types supports lifecycle policies?

- General Purpose v1
- General Purpose v2
- FileStorage

Explanation

That's correct. Azure Blob storage lifecycle management offers a rich, rule-based policy for General Purpose v2 and Blob storage accounts.

Multiple choice

Which of the following standard HTTP headers are supported for both containers and blobs when setting properties by using REST?

- Last-Modified
- Content-Length
- Origin

Explanation

That's correct. Last-Modified is supported on both containers and blobs.

Multiple choice

Which of the following classes of the Azure Storage client library for .NET allows you to manipulate both Azure Storage containers and their blobs?

- BlobClient
- BlobContainerClient
- BlobUriBuilder

Explanation

That's correct. The BlobContainerClient can be used to manipulate both containers and blobs.

Module 4 Develop solutions that use Azure Cosmos DB

Explore Azure Cosmos DB

Introduction

Azure Cosmos DB is a globally distributed database system that allows you to read and write data from the local replicas of your database and it transparently replicates the data to all the regions associated with your Cosmos account.

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

- Identify the key benefits provided by Azure Cosmos DB
- Describe the elements in an Azure Cosmos DB account and how they are organized
- Explain the different consistency levels and choose the correct one for your project
- Explore the APIs supported in Azure Cosmos DB and choose the appropriate API for your solution
- Describe how request units impact costs
- Create Azure Cosmos DB resources by using the Azure portal.

Identify key benefits of Azure Cosmos DB

Azure Cosmos DB is designed to provide low latency, elastic scalability of throughput, well-defined semantics for data consistency, and high availability.

You can configure your databases to be globally distributed and available in any of the Azure regions. To lower the latency, place the data close to where your users are. Choosing the required regions depends on the global reach of your application and where your users are located.

With Azure Cosmos DB, you can add or remove the regions associated with your account at any time. Your application doesn't need to be paused or redeployed to add or remove a region.

Key benefits of global distribution

With its novel multi-master replication protocol, every region supports both writes and reads. The multi-master capability also enables:

- Unlimited elastic write and read scalability.
- 99.999% read and write availability all around the world.
- Guaranteed reads and writes served in less than 10 milliseconds at the 99th percentile.

Your application can perform near real-time reads and writes against all the regions you chose for your database. Azure Cosmos DB internally handles the data replication between regions with consistency level guarantees of the level you've selected.

Running a database in multiple regions worldwide increases the availability of a database. If one region is unavailable, other regions automatically handle application requests. Azure Cosmos DB offers 99.999% read and write availability for multi-region databases.

Explore the resource hierarchy

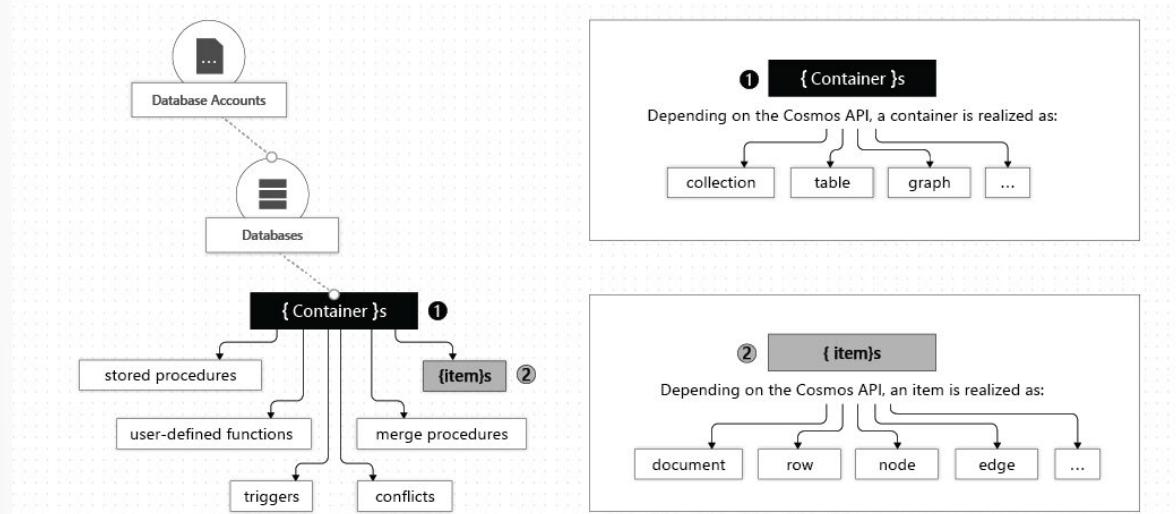
The Azure Cosmos account is the fundamental unit of global distribution and high availability. Your Azure Cosmos account contains a unique DNS name and you can manage an account by using the Azure portal or the Azure CLI, or by using different language-specific SDKs. For globally distributing your data and throughput across multiple Azure regions, you can add and remove Azure regions to your account at any time.

Elements in an Azure Cosmos account

An Azure Cosmos container is the fundamental unit of scalability. You can virtually have an unlimited provisioned throughput (RU/s) and storage on a container. Azure Cosmos DB transparently partitions your container using the logical partition key that you specify in order to elastically scale your provisioned throughput and storage.

Currently, you can create a maximum of 50 Azure Cosmos accounts under an Azure subscription (this is a soft limit that can be increased via support request). After you create an account under your Azure subscription, you can manage the data in your account by creating databases, containers, and items.

The following image shows the hierarchy of different entities in an Azure Cosmos DB account:



Azure Cosmos databases

You can create one or multiple Azure Cosmos databases under your account. A database is analogous to a namespace. A database is the unit of management for a set of Azure Cosmos containers. The following table shows how an Azure Cosmos database is mapped to various API-specific entities:

Azure Cosmos entity	SQL API	Cassandra API	Azure Cosmos DB API for MongoDB	Gremlin API	Table API
Azure Cosmos database	Database	Keyspace	Database	Database	NA

Note: With Table API accounts, when you create your first table, a default database is automatically created in your Azure Cosmos account.

Azure Cosmos containers

An Azure Cosmos container is the unit of scalability both for provisioned throughput and storage. A container is horizontally partitioned and then replicated across multiple regions. The items that you add to the container are automatically grouped into logical partitions, which are distributed across physical partitions, based on the partition key. The throughput on a container is evenly distributed across the physical partitions.

When you create a container, you configure throughput in one of the following modes:

- **Dedicated provisioned throughput mode:** The throughput provisioned on a container is exclusively reserved for that container and it is backed by the SLAs.
- **Shared provisioned throughput mode:** These containers share the provisioned throughput with the other containers in the same database (excluding containers that have been configured with dedicated provisioned throughput). In other words, the provisioned throughput on the database is shared among all the “shared throughput” containers.

A container is a schema-agnostic container of items. Items in a container can have arbitrary schemas. For example, an item that represents a person and an item that represents an automobile can be placed in the same container. By default, all items that you add to a container are automatically indexed without requiring explicit index or schema management.

Azure Cosmos items

Depending on which API you use, an Azure Cosmos item can represent either a document in a collection, a row in a table, or a node or edge in a graph. The following table shows the mapping of API-specific entities to an Azure Cosmos item:

Cosmos entity	SQL API	Cassandra API	Azure Cosmos DB API for MongoDB	Gremlin API	Table API
Azure Cosmos item	Item	Row	Document	Node or edge	Item

Explore consistency levels

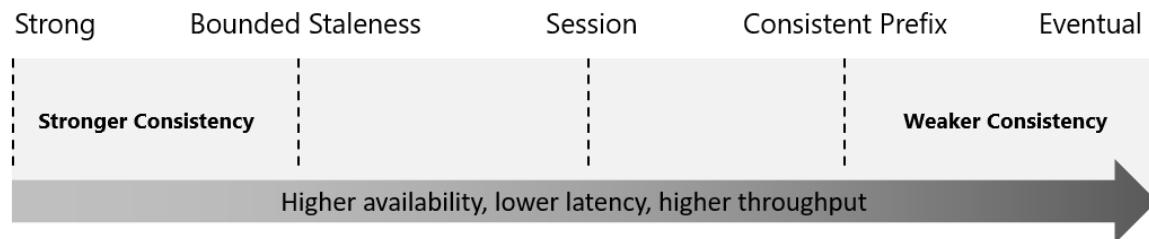
Azure Cosmos DB approaches data consistency as a spectrum of choices instead of two extremes. Strong consistency and eventual consistency are at the ends of the spectrum, but there are many consistency

choices along the spectrum. Developers can use these options to make precise choices and granular tradeoffs with respect to high availability and performance.

With Azure Cosmos DB, developers can choose from five well-defined consistency models on the consistency spectrum. From strongest to more relaxed, the models include:

- *strong*
- *bounded staleness*
- *session*
- *consistent prefix*
- *eventual*

Each level provides availability and performance tradeoffs. The following image shows the different consistency levels as a spectrum.



The consistency levels are region-agnostic and are guaranteed for all operations regardless of the region from which the reads and writes are served, the number of regions associated with your Azure Cosmos account, or whether your account is configured with a single or multiple write regions.

Read consistency applies to a single read operation scoped within a partition-key range or a logical partition. The read operation can be issued by a remote client or a stored procedure.

Choose the right consistency level

Azure Cosmos DB allows developers to choose among the five consistency models: strong, bounded staleness, session, consistent prefix and eventual. Each of these consistency models can be used for specific real-world scenarios. Each provides precise availability and performance tradeoffs and are backed by comprehensive SLAs. The following simple considerations will help you make the right choice in many common scenarios.

SQL API and Table API

Consider the following points if your application is built using SQL API or Table API:

- For many real-world scenarios, session consistency is optimal and it's the recommended option.
- If your application requires strong consistency, it is recommended that you use bounded staleness consistency level.
- If you need stricter consistency guarantees than the ones provided by session consistency and single-digit-millisecond latency for writes, it is recommended that you use bounded staleness consistency level.
- If your application requires eventual consistency, it is recommended that you use consistent prefix consistency level.

- If you need less strict consistency guarantees than the ones provided by session consistency, it is recommended that you use consistent prefix consistency level.
- If you need the highest availability and the lowest latency, then use eventual consistency level.
- If you need even higher data durability without sacrificing performance, you can create a custom consistency level at the application layer.

Cassandra, MongoDB, and Gremlin APIs

Azure Cosmos DB provides native support for wire protocol-compatible APIs for popular databases. These include MongoDB, Apache Cassandra, and Gremlin. When using Gremlin API the default consistency level configured on the Azure Cosmos account is used. For details on consistency level mapping between Cassandra API or the API for MongoDB and Azure Cosmos DB's consistency levels see, [Cassandra API consistency mapping¹](#) and [API for MongoDB consistency mapping²](#).

Consistency guarantees in practice

In practice, you may often get stronger consistency guarantees. Consistency guarantees for a read operation correspond to the freshness and ordering of the database state that you request. Read-consistency is tied to the ordering and propagation of the write/update operations.

- When the consistency level is set to **bounded staleness**, Cosmos DB guarantees that the clients always read the value of a previous write, with a lag bounded by the staleness window.
- When the consistency level is set to **strong**, the staleness window is equivalent to zero, and the clients are guaranteed to read the latest committed value of the write operation.
- For the remaining three consistency levels, the staleness window is largely dependent on your workload. For example, if there are no write operations on the database, a read operation with **eventual**, **session**, or **consistent prefix** consistency levels is likely to yield the same results as a read operation with strong consistency level.

If your Azure Cosmos account is configured with a consistency level other than the strong consistency, you can find out the probability that your clients may get strong and consistent reads for your workloads by looking at the *Probabilistically Bounded Staleness* (PBS) metric.

Probabilistic bounded staleness shows how eventual your eventual consistency is. This metric provides an insight into how often you can get a stronger consistency than the consistency level that you have currently configured on your Azure Cosmos account. In other words, you can see the probability (measured in milliseconds) of getting strongly consistent reads for a combination of write and read regions.

Explore supported APIs

Azure Cosmos DB offers multiple database APIs, which include the Core (SQL) API, API for MongoDB, Cassandra API, Gremlin API, and Table API. By using these APIs, you can model real world data using documents, key-value, graph, and column-family data models. These APIs allow your applications to treat Azure Cosmos DB as if it were various other databases technologies, without the overhead of management, and scaling approaches.

¹ <https://docs.microsoft.com/azure/cosmos-db/cassandra/apache-cassandra-consistency-mapping>

² <https://docs.microsoft.com/azure/cosmos-db/mongodb/consistency-mapping>

Core(SQL) API

This API stores data in document format. It offers the best end-to-end experience as we have full control over the interface, service, and the SDK client libraries. Any new feature that is rolled out to Azure Cosmos DB is first available on SQL API accounts. Azure Cosmos DB SQL API accounts provide support for querying items using the Structured Query Language (SQL) syntax, one of the most familiar and popular query languages to query JSON objects.

If you are migrating from other databases such as Oracle, DynamoDB, HBase etc. SQL API is the recommended option. SQL API supports analytics and offers performance isolation between operational and analytical workloads.

API for MongoDB

This API stores data in a document structure, via BSON format. It is compatible with MongoDB wire protocol; however, it does not use any native MongoDB related code. This API is a great choice if you want to use the broader MongoDB ecosystem and skills, without compromising on using Azure Cosmos DB's features such as scaling, high availability, geo-replication, multiple write locations, automatic and transparent shard management, transparent replication between operational and analytical stores, and more. You can use your existing MongoDB apps with API for MongoDB by just changing the connection string.

API for MongoDB is compatible with the 4.0, 3.6, and 3.2 MongoDB server versions. Server version 4.0 is recommended as it offers the best performance and full feature support

Cassandra API

This API stores data in column-oriented schema. Cassandra API is wire protocol compatible with the Apache Cassandra. You should consider Cassandra API if you want to benefit the elasticity and fully managed nature of Azure Cosmos DB and still use most of the native Apache Cassandra features, tools, and ecosystem. This means on Cassandra API you don't need to manage the OS, Java VM, garbage collector, read/write performance, nodes, clusters, etc.

You can use Apache Cassandra client drivers to connect to the Cassandra API. The Cassandra API enables you to interact with data using the Cassandra Query Language (CQL), and tools like CQL shell, Cassandra client drivers that you're already familiar with. Cassandra API currently only supports OLTP scenarios. Using Cassandra API, you can also use the unique features of Azure Cosmos DB such as change feed.

Table API

This API stores data in key/value format. If you are currently using Azure Table storage, you may see some limitations in latency, scaling, throughput, global distribution, index management, low query performance. Table API overcomes these limitations and it's recommended to migrate your app if you want to use the benefits of Azure Cosmos DB. Table API only supports OLTP scenarios.

Applications written for Azure Table storage can migrate to the Table API with little code changes and take advantage of premium capabilities.

Gremlin API

This API allows users to make graph queries and stores data as edges and vertices. Use this API for scenarios involving dynamic data, data with complex relations, data that is too complex to be modeled

with relational databases, and if you want to use the existing Gremlin ecosystem and skills. Gremlin API currently only supports OLTP scenarios.

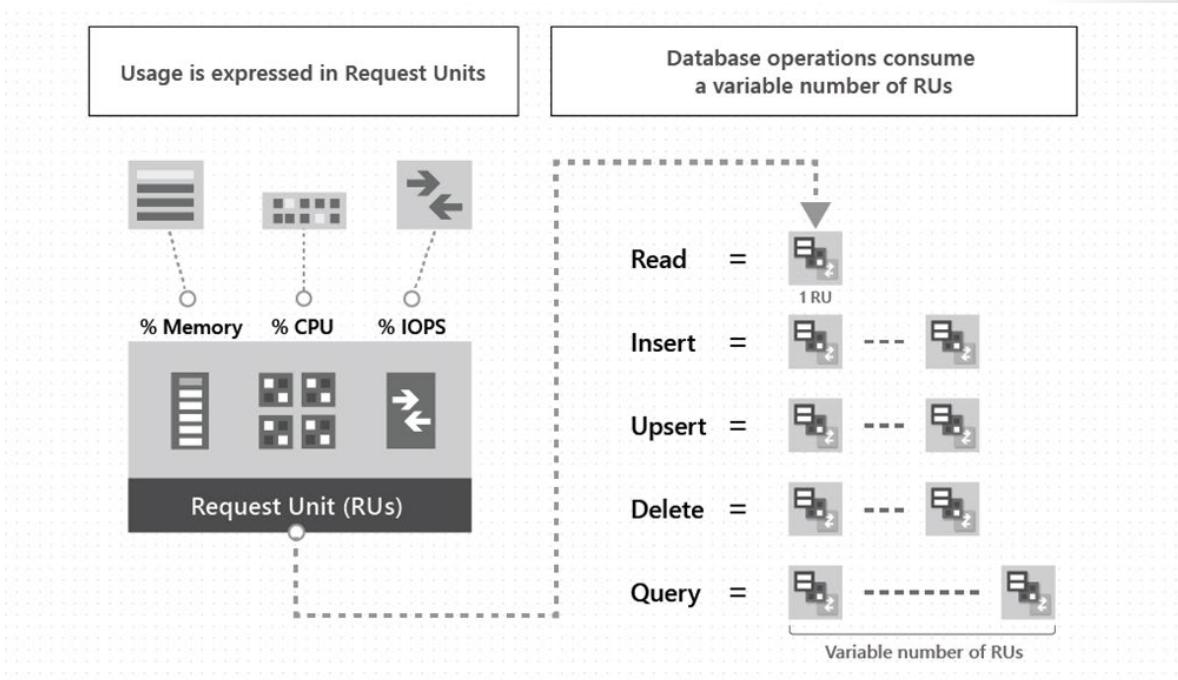
Discover Request Units

With Azure Cosmos DB, you pay for the throughput you provision and the storage you consume on an hourly basis. Throughput must be provisioned to ensure that sufficient system resources are available for your Azure Cosmos database at all times.

The cost of all database operations is normalized by Azure Cosmos DB and is expressed by *request units* (or RUs, for short). A request unit represents the system resources such as CPU, IOPS, and memory that are required to perform the database operations supported by Azure Cosmos DB.

The cost to do a point read, which is fetching a single item by its ID and partition key value, for a 1KB item is 1RU. All other database operations are similarly assigned a cost using RUs. No matter which API you use to interact with your Azure Cosmos container, costs are always measured by RUs. Whether the database operation is a write, point read, or query, costs are always measured in RUs.

The following image shows the high-level idea of RUs:



The type of Azure Cosmos account you're using determines the way consumed RUs get charged. There are three modes in which you can create an account:

- **Provisioned throughput mode:** In this mode, you provision the number of RUs for your application on a per-second basis in increments of 100 RUs per second. To scale the provisioned throughput for your application, you can increase or decrease the number of RUs at any time in increments or decrements of 100 RUs. You can make your changes either programmatically or by using the Azure portal. You can provision throughput at container and database granularity level.
- **Serverless mode:** In this mode, you don't have to provision any throughput when creating resources in your Azure Cosmos account. At the end of your billing period, you get billed for the amount of request units that has been consumed by your database operations.

- **Autoscale mode:** In this mode, you can automatically and instantly scale the throughput (RU/s) of your database or container based on its usage. This mode is well suited for mission-critical workloads that have variable or unpredictable traffic patterns, and require SLAs on high performance and scale.

Exercise: Create Azure Cosmos DB resources by using the Azure portal

In this exercise you'll learn how to perform the following actions in the Azure portal:

- Create an Azure Cosmos DB account
- Add a database and a container
- Add data to your database
- Clean up resources

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 an Azure Cosmos DB account

1. Log in to the [Azure portal](#)³.
2. From the Azure portal navigation pane, select **+ Create a resource**.
3. Search for **Azure Cosmos DB**, then select **Create/Azure Cosmos DB** to get started.
4. On the Select API option page, select **Create** in the **Core (SQL) - Recommended** box.
5. In the **Create Azure Cosmos DB Account - Core (SQL)** page, enter the basic settings for the new Azure Cosmos account.
 - **Subscription:** Select the subscription you want to use.
 - **Resource Group:** Select **Create new**, then enter *az204-cosmos-rg*.
 - **Account Name:** Enter a *unique* name to identify your Azure Cosmos account. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.
 - **Location:** Use the location that is closest to your users to give them the fastest access to the data.
 - **Capacity mode:** Select **Serverless**.
6. Select **Review + create**.
7. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.
8. Select **Go to resource** to go to the Azure Cosmos DB account page.

³ <https://portal.azure.com>

Add a database and a container

You can use the Data Explorer in the Azure portal to create a database and container.

1. Select **Data Explorer** from the left navigation on your Azure Cosmos DB account page, and then select **New Container**.

The screenshot shows the Azure Data Explorer interface for the 'az204dev-cosmos' account. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The 'Data Explorer' link is highlighted with a red box. At the top right, there is a 'New Container' button, which is also highlighted with a red box. Below the top navigation bar, it says 'SQL API'.

2. In the **Add container** pane, enter the settings for the new container.
 - **Database ID:** Select **Create new**, and enter *ToDoList*.
 - **Container ID:** Enter *Items*
 - **Partition key:** Enter */category*. The samples in this demo use */category* as the partition key.
3. Select **OK**. The Data Explorer displays the new database and the container that you created.

Add data to your database

Add data to your new database using Data Explorer.

1. In **Data Explorer**, expand the **ToDoList** database, and expand the **Items** container. Next, select **Items**, and then select **New Item**.

The screenshot shows the Azure Data Explorer interface for the 'ToDoList' database. The left sidebar shows 'Items' selected under 'ToDoList'. The main area has a toolbar with a 'New Item' button, which is highlighted with a red box. Below the toolbar, it says 'Items' and 'SELECT * FROM c'. On the right, there is a table with columns 'id' and '/cat...'. The 'id' column has a value of '/cat...', and the '/cat...' column has a dropdown menu open with options like 'Load more'.

2. Add the following structure to the item on the right side of the **Items** pane:

```
{  
    "id": "1",  
    "category": "personal",  
    "name": "groceries",  
    "description": "Pick up apples and strawberries.",  
    "isComplete": false  
}
```

3. Select **Save**.
4. Select **New Item** again, and create and save another item with a unique `id`, and any other properties and values you want. Your items can have any structure, because Azure Cosmos DB doesn't impose any schema on your data.

Clean up resources

1. Select **Overview** from the left navigation on your Azure Cosmos DB account page.
2. Select the **az204-cosmos-rg** resource group link in the Essentials group.
3. Select **Delete** resource group and follow the directions to delete the resource group and all of the resources it contains.

Knowledge check

Multiple choice

When setting up Azure Cosmos DB there are three account type options. Which of the account type options below is used to specify the number of RUs for an application on a per-second basis?

- Provisioned throughput
- Serverless
- Autoscale

Multiple choice

Which of the following consistency levels below offers the greatest throughput?

- Strong
- Session
- Eventual

Summary

In this module you learned how to:

- Identify the key benefits provided by Azure Cosmos DB
- Describe the elements in an Azure Cosmos DB account and how they are organized
- Explain the different consistency levels and choose the correct one for your project
- Explore the APIs supported in Azure Cosmos DB and choose the appropriate API for your solution

- Describe how request units impact costs
- Create Azure Cosmos DB resources by using the Azure portal.

Implement partitioning in Azure Cosmos DB

Introduction

Azure Cosmos DB uses partitioning to scale individual containers in a database to meet the performance needs of your application.

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

- Describe the differences between logical and physical partitions
- Choose the appropriate partition key for your solution
- Create a synthetic partition key

Explore partitions

In partitioning, the items in a container are divided into distinct subsets called *logical partitions*. Logical partitions are formed based on the value of a *partition key* that is associated with each item in a container. All the items in a logical partition have the same partition key value.

For example, a container holds items. Each item has a unique value for the `UserID` property. If `UserID` serves as the partition key for the items in the container and there are 1,000 unique `UserID` values, 1,000 logical partitions are created for the container.

In addition to a partition key that determines the item's logical partition, each item in a container has an *item ID* which is unique within a logical partition. Combining the partition key and the *item ID* creates the item's *index*, which uniquely identifies the item. Choosing a partition key is an important decision that will affect your application's performance.

Logical partitions

A logical partition consists of a set of items that have the same partition key. For example, in a container that contains data about food nutrition, all items contain a `foodGroup` property. You can use `foodGroup` as the partition key for the container. Groups of items that have specific values for `foodGroup`, such as Beef Products, Baked Products, and Sausages and Luncheon Meats, form distinct logical partitions.

A logical partition also defines the scope of database transactions. You can update items within a logical partition by using a transaction with snapshot isolation. When new items are added to a container, new logical partitions are transparently created by the system. You don't have to worry about deleting a logical partition when the underlying data is deleted.

Physical partitions

A container is scaled by distributing data and throughput across physical partitions. Internally, one or more logical partitions are mapped to a single physical partition. Typically smaller containers have many logical partitions but they only require a single physical partition. Unlike logical partitions, physical partitions are an internal implementation of the system and they are entirely managed by Azure Cosmos DB.

The number of physical partitions in your container depends on the following:

- The number of throughput provisioned (each individual physical partition can provide a throughput of up to 10,000 request units per second). The 10,000 RU/s limit for physical partitions implies that

logical partitions also have a 10,000 RU/s limit, as each logical partition is only mapped to one physical partition.

- The total data storage (each individual physical partition can store up to 50GB data).

Note: Physical partitions are an internal implementation of the system and they are entirely managed by Azure Cosmos DB. When developing your solutions, don't focus on physical partitions because you can't control them. Instead, focus on your partition keys. If you choose a partition key that evenly distributes throughput consumption across logical partitions, you will ensure that throughput consumption across physical partitions is balanced.

Throughput provisioned for a container is divided evenly among physical partitions. A partition key design that doesn't distribute requests evenly might result in too many requests directed to a small subset of partitions that become "hot." Hot partitions lead to inefficient use of provisioned throughput, which might result in rate-limiting and higher costs.

Choose a partition key

A partition key has two components: **partition key path** and the **partition key value**. For example, consider an item { "userId" : "Andrew", "worksFor": "Microsoft" } if you choose "userId" as the partition key, the following are the two partition key components:

- The partition key path (for example: "/userId"). The partition key path accepts alphanumeric and underscore(_) characters. You can also use nested objects by using the standard path notation(/).
- The partition key value (for example: "Andrew"). The partition key value can be of string or numeric types.

Selecting your partition key is a simple but important design choice in Azure Cosmos DB. Once you select your partition key, it is not possible to change it in-place. If you need to change your partition key, you should move your data to a new container with your new desired partition key.

For **all** containers, your partition key should:

- Be a property that has a value which does not change. If a property is your partition key, you can't update that property's value.
- Have a high cardinality. In other words, the property should have a wide range of possible values.
- Spread request unit (RU) consumption and data storage evenly across all logical partitions. This ensures even RU consumption and storage distribution across your physical partitions.

Partition keys for read-heavy containers

For large read-heavy containers you might want to choose a partition key that appears frequently as a filter in your queries. Queries can be **efficiently routed to only the relevant physical partitions**⁴ by including the partition key in the filter predicate.

If most of your workload's requests are queries and most of your queries have an equality filter on the same property, this property can be a good partition key choice.

However, if your container is small, you probably don't have enough physical partitions to need to worry about the performance impact of cross-partition queries. If your container could grow to more than a few physical partitions, then you should make sure you pick a partition key that minimizes cross-partition queries.

⁴ <https://docs.microsoft.com/azure/cosmos-db/how-to-query-container#in-partition-query>

Your container will require more than a few physical partitions when either of the following are true:

- Your container will have over 30,000 RU's provisioned
- Your container will store over 100 GB of data

Using item ID as the partition key

If your container has a property that has a wide range of possible values, it is likely a great partition key choice. One possible example of such a property is the *item ID*. For small read-heavy containers or write-heavy containers of any size, the *item ID* is naturally a great choice for the partition key.

The system property *item ID* exists in every item in your container. You may have other properties that represent a logical ID of your item. In many cases, these are also great partition key choices for the same reasons as the *item ID*.

The *item ID* is a great partition key choice for the following reasons:

- There are a wide range of possible values (one unique *item ID* per item).
- Because there is a unique *item ID* per item, the *item ID* does a great job at evenly balancing RU consumption and data storage.
- You can easily do efficient point reads since you'll always know an item's partition key if you know its *item ID*.

Some things to consider when selecting the *item ID* as the partition key include:

- If the *item ID* is the partition key, it will become a unique identifier throughout your entire container. You won't be able to have items that have a duplicate *item ID*.
- If you have a read-heavy container that has a lot of **physical partitions**⁵, queries will be more efficient if they have an equality filter with the *item ID*.
- You can't run stored procedures or triggers across multiple logical partitions.

Create a synthetic partition key

It's the best practice to have a partition key with many distinct values, such as hundreds or thousands. The goal is to distribute your data and workload evenly across the items associated with these partition key values. If such a property doesn't exist in your data, you can construct a *synthetic partition key*. This unit describes several basic techniques for generating a synthetic partition key for your Cosmos container.

Concatenate multiple properties of an item

You can form a partition key by concatenating multiple property values into a single artificial `partitionKey` property. These keys are referred to as synthetic keys. For example, consider the following example document:

```
{  
  "deviceId": "abc-123",  
  "date": 2018  
}
```

⁵ <https://docs.microsoft.com/azure/cosmos-db/partitioning-overview#physical-partitions>

For the previous document, one option is to set /deviceId or /date as the partition key. Use this option, if you want to partition your container based on either device ID or date. Another option is to concatenate these two values into a synthetic `partitionKey` property that's used as the partition key.

```
{  
  "deviceId": "abc-123",  
  "date": 2018,  
  "partitionKey": "abc-123-2018"  
}
```

In real-time scenarios, you can have thousands of items in a database. Instead of adding the synthetic key manually, define client-side logic to concatenate values and insert the synthetic key into the items in your Cosmos containers.

Use a partition key with a random suffix

Another possible strategy to distribute the workload more evenly is to append a random number at the end of the partition key value. When you distribute items in this way, you can perform parallel write operations across partitions.

An example is if a partition key represents a date. You might choose a random number between 1 and 400 and concatenate it as a suffix to the date. This method results in partition key values like 2018-08-09.1, 2018-08-09.2, and so on, through 2018-08-09.400. Because you randomize the partition key, the write operations on the container on each day are spread evenly across multiple partitions. This method results in better parallelism and overall higher throughput.

Use a partition key with pre-calculated suffixes

The random suffix strategy can greatly improve write throughput, but it's difficult to read a specific item. You don't know the suffix value that was used when you wrote the item. To make it easier to read individual items, use the pre-calculated suffixes strategy. Instead of using a random number to distribute the items among the partitions, use a number that is calculated based on something that you want to query.

Consider the previous example, where a container uses a date as the partition key. Now suppose that each item has a Vehicle-Identification-Number (VIN) attribute that we want to access. Further, suppose that you often run queries to find items by the VIN, in addition to date. Before your application writes the item to the container, it can calculate a hash suffix based on the VIN and append it to the partition key date. The calculation might generate a number between 1 and 400 that is evenly distributed. This result is similar to the results produced by the random suffix strategy method. The partition key value is then the date concatenated with the calculated result.

With this strategy, the writes are evenly spread across the partition key values, and across the partitions. You can easily read a particular item and date, because you can calculate the partition key value for a specific Vehicle-Identification-Number. The benefit of this method is that you can avoid creating a single hot partition key, i.e., a partition key that takes all the workload.

Knowledge check

Multiple choice

Which of the options below best describes the relationship between logical and physical partitions?

- Logical partitions are collections of physical partitions.
- Physical partitions are collections of logical partitions
- There is no relationship between physical and logical partitions.

Multiple choice

Which of the below correctly lists the two components of a partition key?

- Key path, synthetic key
- Key path, key value
- Key value, item ID

Summary

In this module you learned how to:

- Describe the differences between logical and physical partitions
- Choose the appropriate partition key for your solution
- Create a synthetic partition key

Work with Azure Cosmos DB

Introduction

This module is an introduction to both client and server-side programming on Azure Cosmos DB.

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

- Identify classes and methods used to create resources
- Create resources by using the Azure Cosmos DB .NET v3 SDK
- Write stored procedures, triggers, and user-defined functions by using JavaScript

Explore Microsoft .NET SDK v3 for Azure Cosmos DB

This unit focuses on version 3 of the .NET SDK. (**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.

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 above. 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 which enables efficient connection management and performance.

```
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 is an existing database.

```
// 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 service as an asynchronous operation.

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

Delete a database

Delete a Database as an asynchronous operation.

```
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 is an existing container.

```
// 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

```
Container container = database.GetContainer(containerId);
ContainerProperties containerProperties = await container.ReadContainerA-
sync();
```

Delete a container

Delete a Container as an asynchronous operation.

```
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`.

```
ItemResponse<SalesOrder> response = await container.CreateItemAsync(sale-
sOrder, 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`.

```
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`.

```
QueryDefinition query = new QueryDefinition(  
    "select * from sales s where s.AccountNumber = @AccountInput")  
.WithParameter("@AccountInput", "Account1");  
  
FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<Sale-  
sOrder>(  
    query,  
    requestOptions: new QueryRequestOptions()  
    {  
        PartitionKey = new PartitionKey("Account1"),  
        MaxItemCount = 1  
    });
```

Additional resources

- The [azure-cosmos-dotnet-v3⁶](https://github.com/Azure/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 Cosmos DB.NET V3 SDK \(Microsoft.Azure.Cosmos\) examples for the SQL API⁷](https://docs.microsoft.com/azure/cosmos-db/sql-api-dotnet-v3-sdk-samples) for direct links to specific examples in the GitHub repository.

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

In this exercise you'll 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

⁶ <https://github.com/Azure/azure-cosmos-dotnet-v3/tree/master/Microsoft.Azure.Cosmos.Samples/Usage>

⁷ <https://docs.microsoft.com/azure/cosmos-db/sql-api-dotnet-v3-sdk-samples>

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>⁸.
- **Azure CLI:** You can install the Azure CLI from <https://docs.microsoft.com/cli/azure/install-azure-cli>.
- The **.NET Core 3.1 SDK**, or **.NET 5.0 SDK**. You can install from <https://dotnet.microsoft.com/download>.

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. Log in to Azure by using the command below. A browser window should open letting you choose which account to log in with.

```
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.

```
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 account. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length. *This command will take a few minutes to complete.*

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

Record the documentEndpoint shown in the JSON response, it will be used below.

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

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

⁸ <https://code.visualstudio.com/>

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.

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

2. Create the .NET console app.

```
dotnet new console
```

3. Open the current folder in VS Code using the command below. The `-r` option will open the folder without launching a new VS Code window.

```
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 VS Code and use the command below to add the `Microsoft.Azure.Cosmos` package to the project.

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

2. Add using statements to include `Microsoft.Azure.Cosmos` and to enable async operations.

```
using System.Threading.Tasks;  
using Microsoft.Azure.Cosmos;
```

Add code to connect to an Azure Cosmos DB account

1. Replace the entire `class Program` with the code snippet below. 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.

```
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();
    }
}
```

2. Below the `Main` method, add a new asynchronous task called `CosmosAsync`, which instantiates our new `CosmosClient`.

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

Create a database

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

```
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);
}
```

2. Add the code below at the end of the `CosmosAsync` method, it calls the `CreateDatabaseAsync` method you just added.

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

Create a container

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

```
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);
}
```

2. Copy and paste the code below where you instantiated the `CosmosClient` to call the `CreateContainer` method you just added.

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

Run the application

1. Save your work and, in a terminal in VS Code, run the `dotnet run` command. The console will display the following messages.

```
Beginning operations...
```

```
Created Database: az204Database
```

```
Created Container: az204Container
```

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

2. You can 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 command below.

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

Create stored procedures

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 DB⁹](#).

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 is a simple stored procedure that returns a "Hello World" response.

```
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, as well as 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 is 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 is not provided and there is an error, the Azure Cosmos DB runtime will throw an error.

⁹ <https://docs.microsoft.com/azure/cosmos-db/sql/how-to-use-stored-procedures-triggers-udfs>

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 will throw an exception. Otherwise, the rest of the stored procedure continues to run.

The following example stored procedure takes an input parameter named `documentToCreate` and the parameter's value is the body of a document to be created in the current collection. The callback throws an error if the operation fails. Otherwise, it sets the `id` of the created document as the body of the response to the client.

```
function createSampleDocument(documentToCreate) {
    var context = getContext();
    var collection = context.getCollection();
    var accepted = collection.createDocument(
        collection.getSelfLink(),
        documentToCreate,
        function (error, documentCreated) {
            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:

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

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

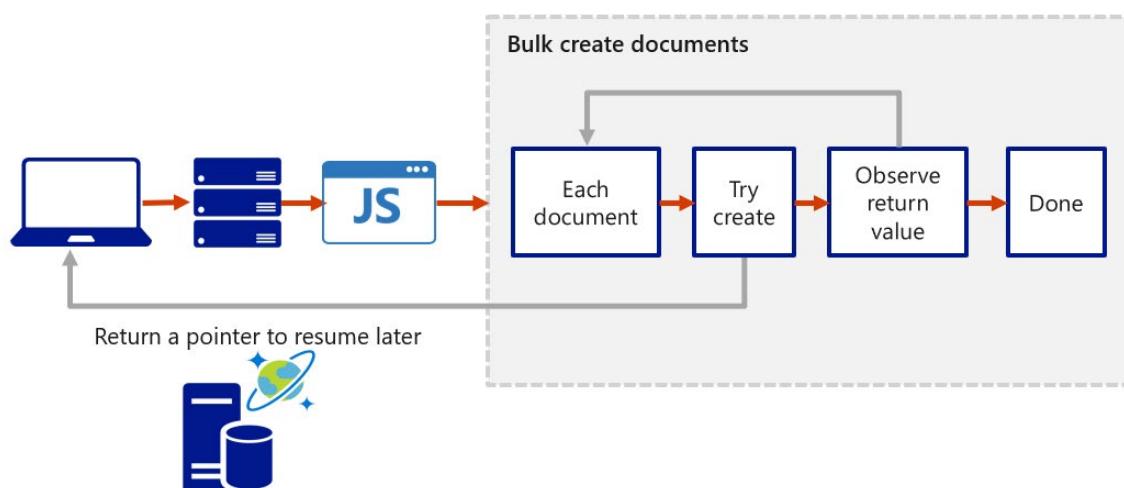
Bounded execution

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 will complete 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 transac-

tion 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

Azure Cosmos DB supports pre-triggers and post-triggers. Pre-triggers are executed before modifying a database item and post-triggers are executed after modifying a database item. Triggers are not 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 [pre-triggers¹⁰](#) and [post-triggers¹¹](#).

Pre-triggers

The following example shows how a pre-trigger 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.

```
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);
```

¹⁰ <https://docs.microsoft.com/azure/cosmos-db/sql/how-to-use-stored-procedures-triggers-udfs#pre-triggers>

¹¹ <https://docs.microsoft.com/azure/cosmos-db/sql/how-to-use-stored-procedures-triggers-udfs#post-triggers>

```
}
```

Pre-triggers cannot 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 pre-trigger 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 is not permitted.

For examples of how to register and call a pre-trigger, visit the [pre-triggers¹²](#) 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.

```
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, requestOptions) {
        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;
    }
}
```

¹² <https://docs.microsoft.com/azure/cosmos-db/sql/how-to-use-stored-procedures-triggers-udfs#pre-triggers>

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 will fail the whole transaction. Anything committed will be 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 is a container called "Incomes" with properties as follows:

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

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

```
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;  
}
```

Knowledge check

Multiple choice

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

- String
- Integer
- Boolean

Multiple choice

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

- Pre-trigger
- Post-trigger
- User-defined function

Summary

In this module you learned how to:

- Identify classes and methods used to create resources
- Create resources by using the Azure Cosmos DB .NET v3 SDK
- Write stored procedures, triggers, and user-defined functions by using JavaScript

Answers

Multiple choice

When setting up Azure Cosmos DB there are three account type options. Which of the account type options below is used to specify the number of RUs for an application on a per-second basis?

- Provisioned throughput
- Serverless
- Autoscale

Explanation

That's correct. In this mode, you provision the number of RUs for your application on a per-second basis in increments of 100 RUs per second.

Multiple choice

Which of the following consistency levels below offers the greatest throughput?

- Strong
- Session
- Eventual

Explanation

That's correct. The eventual consistency level offers the greatest throughput at the cost of weaker consistency.

Multiple choice

Which of the options below best describes the relationship between logical and physical partitions?

- Logical partitions are collections of physical partitions.
- Physical partitions are collections of logical partitions
- There is no relationship between physical and logical partitions.

Explanation

That's correct. One or more logical partitions are mapped to a single physical partition..

Multiple choice

Which of the below correctly lists the two components of a partition key?

- Key path, synthetic key
- Key path, key value
- Key value, item ID

Explanation

That's correct. A partition key has two components: partition key path and the partition key value.

Multiple choice

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

- String
- Integer
- Boolean

Explanation

That's correct. When defining a stored procedure in Azure portal, input parameters are always sent as a string to the stored procedure.

Multiple choice

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

- Pre-trigger
- Post-trigger
- User-defined function

Explanation

That's correct. Pre-triggers can be used to conform data before it is added to the container.

Module 5 Implement infrastructure as a service solutions

Provision virtual machines in Azure

Introduction

Azure virtual machines (VM) is one of several types of on-demand, scalable computing resources that Azure offers. Typically, you choose a VM when you need more control over the computing environment than the other choices offer.

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

- Describe the design considerations for creating a virtual machine to support your apps needs
- Explain the different availability options for Azure VMs
- Describe the VM sizing options
- Create an Azure VM by using the Azure CLI

Explore Azure Virtual Machines

An Azure virtual machine gives you the flexibility of virtualization without having to buy and maintain the physical hardware that runs it. However, you still need to maintain the VM by performing tasks, such as configuring, patching, and installing the software that runs on it. This unit gives you information about what you should consider before you create a VM, how you create it, and how you manage it.

Azure virtual machines can be used in various ways. Some examples are:

- **Development and test** – Azure VMs offer a quick and easy way to create a computer with specific configurations required to code and test an application.
- **Applications in the cloud** – Because demand for your application can fluctuate, it might make economic sense to run it on a VM in Azure. You pay for extra VMs when you need them and shut them down when you don't.
- **Extended datacenter** – Virtual machines in an Azure virtual network can easily be connected to your organization's network.

Design considerations for virtual machine creation

There are always a multitude of design considerations when you build out an application infrastructure in Azure. These aspects of a VM are important to think about before you start:

- **Availability:** Azure supports a single instance virtual machine Service Level Agreement of 99.9% provided you deploy the VM with premium storage for all disks.
- **VM size:** The size of the VM that you use is determined by the workload that you want to run. The size that you choose then determines factors such as processing power, memory, and storage capacity.
- **VM limits:** Your subscription has default quota limits in place that could impact the deployment of many VMs for your project. The current limit on a per subscription basis is 20 VMs per region. Limits can be raised by filing a [support ticket requesting an increase](#)¹.
- **VM image:** You can either use your own image, or you can use one of the images in the Azure Marketplace. You can get a list of images in the marketplace by using the `az vm image list` command. See [list popular images](#)² for more information on using the command.
- **VM disks:** There are two components that make up this area. The type of disks which determines the performance level and the storage account type that contains the disks. Azure provides two types of disks:
 - **Standard disks:** Backed by HDDs, and delivers cost-effective storage while still being performant. Standard disks are ideal for a cost effective dev and test workload.
 - **Premium disks:** Backed by SSD-based, high-performance, low-latency disk. Perfect for VMs running production workload.

And, there are two options for the disk storage:

- **Managed disks:** Managed disks are the newer and recommended disk storage model and they are managed by Azure. You specify the size of the disk, which can be up to 4 terabytes (TB), and Azure creates and manages both the disk and the storage. You don't have to worry about storage account limits, which makes managed disks easier to scale out than unmanaged discs.
- **Unmanaged disks:** With unmanaged disks, you're responsible for the storage accounts that hold the virtual hard disks (VHDs) that correspond to your VM disks. You pay the storage account rates for the amount of space you use. A single storage account has a fixed-rate limit of 20,000 input/output (I/O) operations per second. This means that a storage account is capable of supporting 40 standard VHDs at full utilization. If you need to scale out with more disks, then you'll need more storage accounts, which can get complicated.

Virtual machine extensions

Windows VMs have extensions which give your VM additional capabilities through post deployment configuration and automated tasks. These common tasks can be accomplished using extensions:

- **Run custom scripts:** The Custom Script Extension helps you configure workloads on the VM by running your script when the VM is provisioned.
- **Deploy and manage configurations:** The PowerShell Desired State Configuration (DSC) Extension helps you set up DSC on a VM to manage configurations and environments.

¹ <https://docs.microsoft.com/azure/azure-portal/supportability/regional-quota-requests>

² <https://docs.microsoft.com/azure/virtual-machines/linux/cli-ps-findimage#list-popular-images>

- **Collect diagnostics data:** The Azure Diagnostics Extension helps you configure the VM to collect diagnostics data that can be used to monitor the health of your application.

For Linux VMs, Azure supports **cloud-init**³ across most Linux distributions that support it and works with all the major automation tooling like Ansible, Chef, SaltStack, and Puppet.

Compare virtual machine availability options

Azure offers several options for ensuring the availability of the virtual machines, and applications, you have deployed.

Availability zones

Availability zones⁴ expands the level of control you have to maintain the availability of the applications and data on your VMs. An Availability Zone is a physically separate zone, within an Azure region. There are three Availability Zones per supported Azure region.

An Availability Zone in an Azure region is a combination of a fault domain and an update domain. For example, if you create three or more VMs across three zones in an Azure region, your VMs are effectively distributed across three fault domains and three update domains. The Azure platform recognizes this distribution across update domains to make sure that VMs in different zones are not scheduled to be updated at the same time.

Build high-availability into your application architecture by co-locating your compute, storage, networking, and data resources within a zone and replicating in other zones. Azure services that support Availability Zones fall into two categories:

- **Zonal services:** Where a resource is pinned to a specific zone (for example, virtual machines, managed disks, Standard IP addresses), or
- **Zone-redundant services:** When the Azure platform replicates automatically across zones (for example, zone-redundant storage, SQL Database).

Availability sets

An **availability set**⁵ is a logical grouping of VMs that allows Azure to understand how your application is built to provide for redundancy and availability. An availability set is composed of two additional groupings that protect against hardware failures and allow updates to safely be applied - fault domains (FDs) and update domains (UDs).

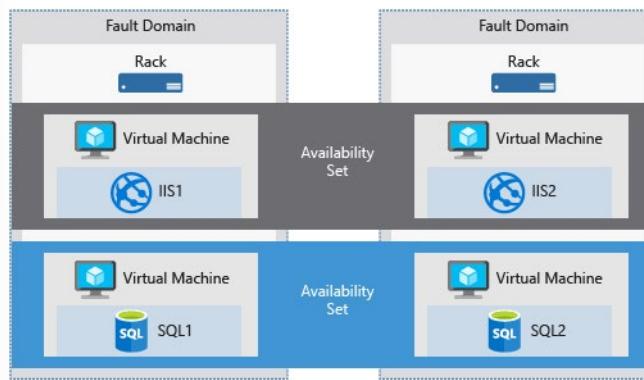
Fault domains

A fault domain is a logical group of underlying hardware that share a common power source and network switch, similar to a rack within an on-premises datacenter. As you create VMs within an availability set, the Azure platform automatically distributes your VMs across these fault domains. This approach limits the impact of potential physical hardware failures, network outages, or power interruptions.

³ <https://cloud-init.io/>

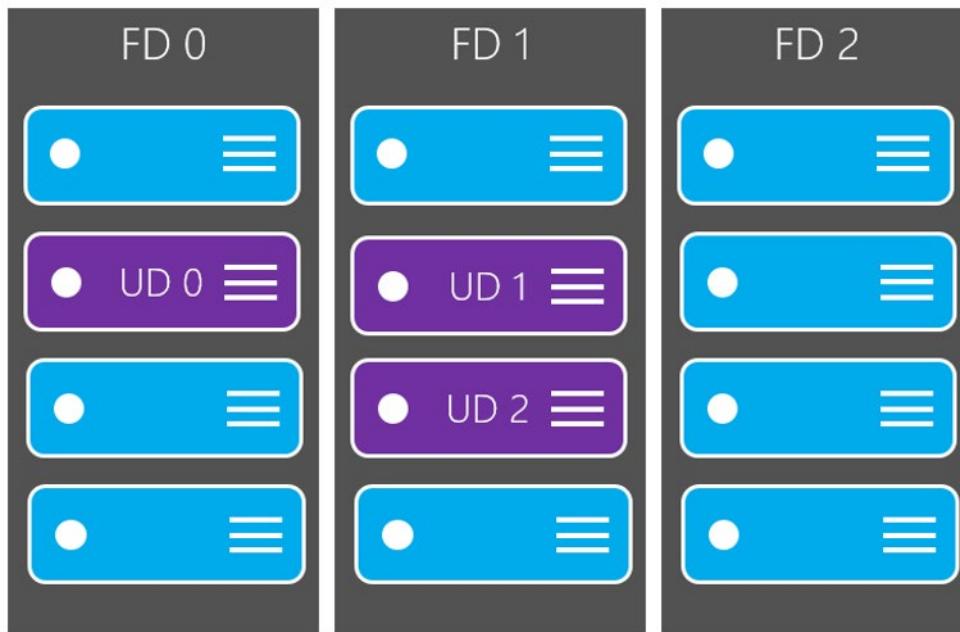
⁴ <https://docs.microsoft.com/azure/availability-zones/az-overview?context=/azure/virtual-machines/context/context>

⁵ <https://docs.microsoft.com/azure/virtual-machines/availability-set-overview>



Update domains

An update domain is a logical group of underlying hardware that can undergo maintenance or be rebooted at the same time. As you create VMs within an availability set, the Azure platform automatically distributes your VMs across these update domains. This approach ensures that at least one instance of your application always remains running as the Azure platform undergoes periodic maintenance. The order of update domains being rebooted may not proceed sequentially during planned maintenance, but only one update domain is rebooted at a time.



Virtual machine scale sets

Azure virtual machine scale sets⁶ let you create and manage a group of load balanced VMs. The number of VM instances can automatically increase or decrease in response to demand or a defined schedule.

⁶ <https://docs.microsoft.com/azure/virtual-machine-scale-sets/overview?context=/azure/virtual-machines/context/context>

Load balancer

Combine the **Azure Load Balancer**⁷ with an availability zone or availability set to get the most application resiliency. An Azure load balancer is a Layer-4 (TCP, UDP) load balancer that provides high availability by distributing incoming traffic among healthy VMs. A load balancer health probe monitors a given port on each VM and only distributes traffic to an operational VM.

You define a front-end IP configuration that contains one or more public IP addresses. This front-end IP configuration allows your load balancer and applications to be accessible over the Internet.

Virtual machines connect to a load balancer using their virtual network interface card (NIC). To distribute traffic to the VMs, a back-end address pool contains the IP addresses of the virtual (NICs) connected to the load balancer.

To control the flow of traffic, you define load balancer rules for specific ports and protocols that map to your VMs.

Determine appropriate virtual machine size

The best way to determine the appropriate VM size is to consider the type of workload your VM needs to run. Based on the workload, you're able to choose from a subset of available VM sizes. Workload options are classified as follows on Azure:

VM Type	Description
General Purpose	Balanced CPU-to-memory ratio. Ideal for testing and development, small to medium databases, and low to medium traffic web servers.
Compute Optimized	High CPU-to-memory ratio. Good for medium traffic web servers, network appliances, batch processes, and application servers.
Memory Optimized	High memory-to-CPU ratio. Great for relational database servers, medium to large caches, and in-memory analytics.
Storage Optimized	High disk throughput and IO ideal for Big Data, SQL, NoSQL databases, data warehousing and large transactional databases.
GPU	Specialized virtual machines targeted for heavy graphic rendering and video editing, as well as model training and inferencing (ND) with deep learning. Available with single or multiple GPUs.
High Performance Compute	Our fastest and most powerful CPU virtual machines with optional high-throughput network interfaces (RDMA).

What if my size needs change?

Azure allows you to change the VM size when the existing size no longer meets your needs. You can resize the VM - as long as your current hardware configuration is allowed in the new size. This provides a fully agile and elastic approach to VM management.

If you stop and deallocate the VM, you can then select any size available in your region since this removes your VM from the cluster it was running on.

⁷ <https://docs.microsoft.com/azure/load-balancer/load-balancer-overview>

Caution: Be cautious when resizing production VMs - they will be rebooted automatically which can cause a temporary outage and change some configuration settings such as the IP address.

Additional resources

- For information about pricing of the various sizes, see the pricing pages for [Linux⁸](#) or [Windows⁹](#).
- For availability of VM sizes in Azure regions, see [Products available by region¹⁰](#).

Exercise: Create a virtual machine by using the Azure CLI

In this exercise you'll create a Linux virtual machine by performing the following operations using Azure CLI commands:

- Create a resource group and a virtual machine
- Install a web server
- View the web server in action
- Clean up resources

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>.

Login to Azure and start the Cloud Shell

1. Login to the [Azure portal¹¹](#) and open the Cloud Shell.



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



Create a resource group and virtual machine

1. Create a resource group with the `az group create` command. The command below creates a resource group named `az204-vm-rg`. Replace `<myLocation>` with a region near you.

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

⁸ <https://azure.microsoft.com/pricing/details/virtual-machines/#Linux>

⁹ <https://azure.microsoft.com/pricing/details/virtual-machines/Windows/#Windows>

¹⁰ <https://azure.microsoft.com/regions/services/>

¹¹ <https://portal.azure.com>

2. Create a VM with the `az vm create` command. The command below creates a Linux VM named `az204vm` with an admin user named `azureuser`. After executing the command you will need to supply a password that meets the password requirements.

```
az vm create \
--resource-group az204-vm-rg \
--name az204vm \
--image UbuntuLTS \
--generate-ssh-keys \
--admin-username azureuser
```

It will take a few minutes for the operation to complete. When it is finished note the `publicIpAddress` in the output, you'll use it in the next step.

Note: When creating VMs with the Azure CLI passwords need to be between 12-123 characters, have both uppercase and lowercase characters, a digit, and have a special character (@, !, etc.). Be sure to remember the password.

Install a web server

1. By default, only SSH connections are opened when you create a Linux VM in Azure. Use `az vm open-port` to open TCP port 80 for use with the NGINX web server:

```
az vm open-port --port 80 \
--resource-group az204-vm-rg \
--name az204vm
```

2. Connect to your VM by using SSH. Replace `<publicIPAddress>` in the example with the public IP address of your VM as noted in the previous output:

```
ssh azureuser@<publicIPAddress>
```

3. To see your VM in action, install the NGINX web server. Update your package sources and then install the latest NGINX package.

```
sudo apt-get -y update
sudo apt-get -y install nginx
```

4. When done type `exit` to leave the SSH session.

View the web server in action

Use a web browser of your choice to view the default NGINX welcome page. Use the public IP address of your VM as the web address. The following example shows the default NGINX web site:



Clean up Azure resources

You can now safely delete the `az204-vm-rg` resource group from your account by running the command below.

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

Knowledge check

Multiple choice

Which of the following Azure virtual machine types is most appropriate for testing and development?

- Compute optimized
- General Purpose
- Storage optimized

Multiple choice

Which of the below represents a logical grouping of VMs that allows Azure to understand how your application is built to provide for redundancy and availability?

- Load balancer
- Availability zone
- Availability set

Summary

In this module, you learned how to:

- Describe the design considerations for creating a virtual machine to support your apps needs
- Explain the different availability options for Azure VMs
- Describe the VM sizing options
- Create an Azure VM by using the Azure CLI

Create and deploy Azure Resource Manager templates

Introduction

Azure Resource Manager is the deployment and management service for Azure. It provides a management layer that enables you to create, update, and delete resources in your Azure subscription.

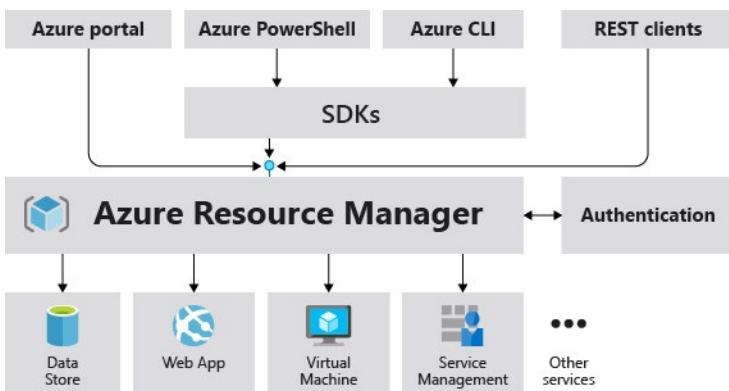
After completing this module, you'll be able to:

- Describe what role Azure Resource Manager has in Azure and the benefits of using Azure Resource Manager templates
- Explain what happens when Azure Resource Manager templates are deployed and how to structure them to support your solution
- Create a template with conditional resource deployments
- Choose the correct deployment mode for your solution
- Create and deploy an Azure Resource Manager template by using Visual Studio Code

Explore Azure Resource Manager

When a user sends a request from any of the Azure tools, APIs, or SDKs, Resource Manager receives the request. It authenticates and authorizes the request. Resource Manager sends the request to the Azure service, which takes the requested action. Because all requests are handled through the same API, you see consistent results and capabilities in all the different tools.

The following image shows the role Azure Resource Manager plays in handling Azure requests.



Why choose Azure Resource Manager templates?

If you're trying to decide between using Azure Resource Manager templates and one of the other infrastructure as code services, consider the following advantages of using templates:

- **Declarative syntax:** Azure Resource Manager templates allow you to create and deploy an entire Azure infrastructure declaratively. For example, you can deploy not only virtual machines, but also the network infrastructure, storage systems, and any other resources you may need.
- **Repeatable results:** Repeatedly deploy your infrastructure throughout the development lifecycle and have confidence your resources are deployed in a consistent manner. Templates are idempotent,

which means you can deploy the same template many times and get the same resource types in the same state. You can develop one template that represents the desired state, rather than developing lots of separate templates to represent updates.

- **Orchestration:** You don't have to worry about the complexities of ordering operations. Resource Manager orchestrates the deployment of interdependent resources so they're created in the correct order. When possible, Resource Manager deploys resources in parallel so your deployments finish faster than serial deployments. You deploy the template through one command, rather than through multiple imperative commands.

Template file

Within your template, you can write template expressions that extend the capabilities of JSON. These expressions make use of the **functions¹²** provided by Resource Manager.

The template has the following sections:

- **Parameters¹³** - Provide values during deployment that allow the same template to be used with different environments.
- **Variables¹⁴** - Define values that are reused in your templates. They can be constructed from parameter values.
- **User-defined functions¹⁵** - Create customized functions that simplify your template.
- **Resources¹⁶** - Specify the resources to deploy.
- **Outputs¹⁷** - Return values from the deployed resources.

Deploy multi-tiered solutions

With Resource Manager, you can create a template (in JSON format) that defines the infrastructure and configuration of your Azure solution. By using a template, you can repeatedly deploy your solution throughout its lifecycle and have confidence your resources are deployed in a consistent state.

When you deploy a template, Resource Manager converts the template into REST API operations. For example, when Resource Manager receives a template with the following resource definition:

```
"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-04-01",
    "name": "mystorageaccount",
    "location": "westus",
    "sku": {
      "name": "Standard_LRS"
    },
    "kind": "StorageV2",
    "properties": {}
  }
]
```

¹² <https://docs.microsoft.com/azure/azure-resource-manager/templates/template-functions>

¹³ <https://docs.microsoft.com/azure/azure-resource-manager/templates/parameters>

¹⁴ <https://docs.microsoft.com/azure/azure-resource-manager/templates/variables>

¹⁵ <https://docs.microsoft.com/azure/azure-resource-manager/templates/user-defined-functions>

¹⁶ <https://docs.microsoft.com/azure/azure-resource-manager/templates/resource-declaration>

¹⁷ <https://docs.microsoft.com/azure/azure-resource-manager/templates/outputs>

]

It converts the definition to the following REST API operation, which is sent to the Microsoft.Storage resource provider:

```
PUT  
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/  
{resourceGroupName}/providers/Microsoft.Storage/storageAccounts/mystorage-  
account?api-version=2019-04-01  
REQUEST BODY  
{  
    "location": "westus",  
    "sku": {  
        "name": "Standard_LRS"  
    },  
    "kind": "StorageV2",  
    "properties": {}  
}
```

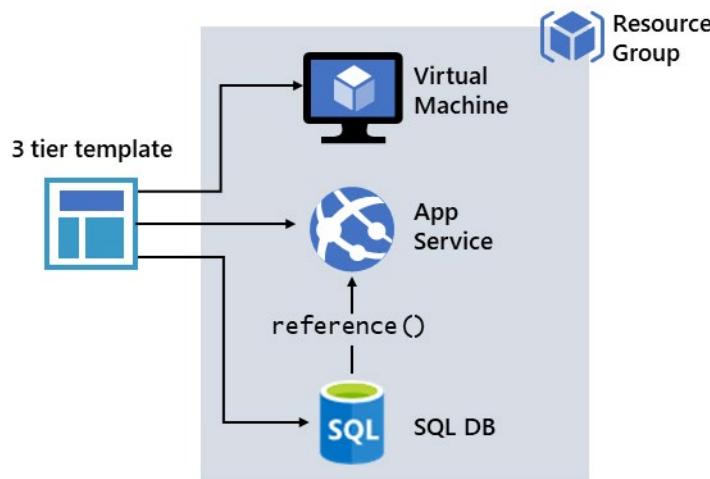
Notice that the **apiVersion** you set in the template for the resource is used as the API version for the REST operation. You can repeatedly deploy the template and have confidence it will continue to work. By using the same API version, you don't have to worry about breaking changes that might be introduced in later versions.

You can deploy a template using any of the following options:

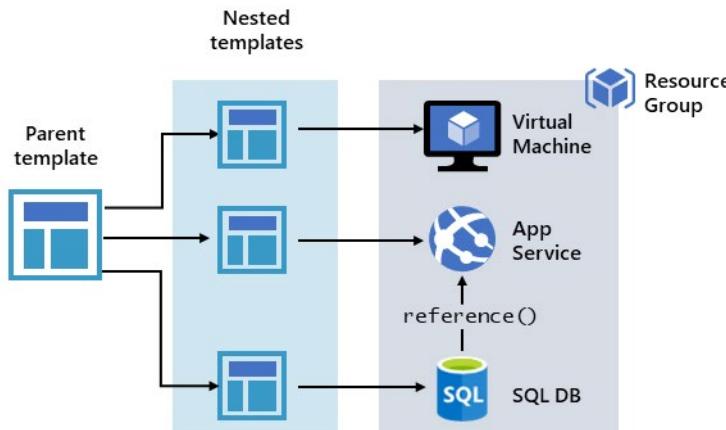
- Azure portal
- Azure CLI
- PowerShell
- REST API
- Button in GitHub repository
- Azure Cloud Shell

Defining multi-tiered templates

How you define templates and resource groups is entirely up to you and how you want to manage your solution. For example, you can deploy a three tier application through a single template to a single resource group.



But, you don't have to define your entire infrastructure in a single template. Often, it makes sense to divide your deployment requirements into a set of targeted, purpose-specific templates. You can easily reuse these templates for different solutions. To deploy a particular solution, you create a master template that links all the required templates. The following image shows how to deploy a three tier solution through a parent template that includes three nested templates.



If you envision your tiers having separate lifecycles, you can deploy your three tiers to separate resource groups. The resources can still be linked to resources in other resource groups.

Azure Resource Manager analyzes dependencies to ensure resources are created in the correct order. If one resource relies on a value from another resource (such as a virtual machine needing a storage account for disks), you set a dependency. For more information, see [Defining dependencies in Azure Resource Manager templates](#).

You can also use the template for updates to the infrastructure. For example, you can add a resource to your solution and add configuration rules for the resources that are already deployed. If the template specifies creating a resource but that resource already exists, Azure Resource Manager performs an update instead of creating a new asset. Azure Resource Manager updates the existing asset to the same state as it would be as new.

Resource Manager provides extensions for scenarios when you need additional operations such as installing particular software that isn't included in the setup. If you're already using a configuration management service, like DSC, Chef or Puppet, you can continue working with that service by using extensions.

Finally, the template becomes part of the source code for your app. You can check it in to your source code repository and update it as your app evolves. You can edit the template through Visual Studio.

Share templates

After creating your template, you may wish to share it with other users in your organization. **Template specs**¹⁸ enable you to store a template as a resource type. You use role-based access control to manage access to the template spec. Users with read access to the template spec can deploy it, but not change the template.

This approach means you can safely share templates that meet your organization's standards.

Explore conditional deployment

Sometimes you need to optionally deploy a resource in an Azure Resource Manager template (Azure Resource Manager template). Use the `condition` element to specify whether the resource is deployed. The value for the condition resolves to true or false. When the value is true, the resource is created. When the value is false, the resource isn't created. The value can only be applied to the whole resource.

Note: Conditional deployment doesn't cascade to **child resources**¹⁹. If you want to conditionally deploy a resource and its child resources, you must apply the same condition to each resource type.

New or existing resource

You can use conditional deployment to create a new resource or use an existing one. The following example shows how to use condition to deploy a new storage account or use an existing storage account. It contains a parameter named `newOrExisting` which is used as a condition in the `resources` section.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "storageAccountName": {
            "type": "string"
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]"
        },
        "newOrExisting": {
            "type": "string",
            "defaultValue": "new",
            "allowedValues": [
                "new",
                "existing"
            ]
        }
    }
}
```

¹⁸ <https://docs.microsoft.com/azure/azure-resource-manager/templates/template-specs>

¹⁹ <https://docs.microsoft.com/azure/azure-resource-manager/templates/child-resource-name-type>

```
        ]
    },
},
"functions": [],
"resources": [
{
    "condition": "[equals(parameters('newOrExisting'), 'new')]",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-06-01",
    "name": "[parameters('storageAccountName')]",
    "location": "[parameters('location')]",
    "sku": {
        "name": "Standard_LRS",
        "tier": "Standard"
    },
    "kind": "StorageV2",
    "properties": {
        "accessTier": "Hot"
    }
}
]
}
```

When the parameter **newOrExisting** is set to **new**, the condition evaluates to true. The storage account is deployed. However, when **newOrExisting** is set to **existing**, the condition evaluates to false and the storage account isn't deployed.

Runtime functions

If you use a `reference` or `list` function with a resource that is conditionally deployed, the function is evaluated even if the resource isn't deployed. You get an error if the function refers to a resource that doesn't exist.

Use the `if` function to make sure the function is only evaluated for conditions when the resource is deployed.

You set a resource as dependent on a conditional resource exactly as you would any other resource. When a conditional resource isn't deployed, Azure Resource Manager automatically removes it from the required dependencies.

Additional resources

- [Azure Resource Manager template functions²⁰](#)

Set the correct deployment mode

When deploying your resources, you specify that the deployment is either an incremental update or a complete update. The difference between these two modes is how Resource Manager handles existing resources in the resource group that aren't in the template. The default mode is incremental.

²⁰ <https://docs.microsoft.com/azure/azure-resource-manager/templates/template-functions>

For both modes, Resource Manager tries to create all resources specified in the template. If the resource already exists in the resource group and its settings are unchanged, no operation is taken for that resource. If you change the property values for a resource, the resource is updated with those new values. If you try to update the location or type of an existing resource, the deployment fails with an error. Instead, deploy a new resource with the location or type that you need.

Complete mode

In complete mode, Resource Manager **deletes** resources that exist in the resource group but aren't specified in the template.

If your template includes a resource that isn't deployed because condition evaluates to false, the result depends on which REST API version you use to deploy the template. If you use a version earlier than 2019-05-10, the resource **isn't deleted**. With 2019-05-10 or later, the resource **is deleted**. The latest versions of Azure PowerShell and Azure CLI delete the resource.

Be careful using complete mode with `copy` loops. Any resources that aren't specified in the template after resolving the copy loop are deleted.

Incremental mode

In incremental mode, Resource Manager **leaves unchanged** resources that exist in the resource group but aren't specified in the template.

However, when redeploying an existing resource in incremental mode, the outcome is different. Specify all properties for the resource, not just the ones you're updating. A common misunderstanding is to think properties that aren't specified are left unchanged. If you don't specify certain properties, Resource Manager interprets the update as overwriting those values.

Example result

To illustrate the difference between incremental and complete modes, consider the following table.

Resource Group contains	Template contains	Incremental result	Complete result
Resource A	Resource A	Resource A	Resource A
Resource B	Resource B	Resource B	Resource B
Resource C	Resource D	Resource C Resource D	Resource D

When deployed in **incremental** mode, Resource D is added to the existing resource group. When deployed in **complete** mode, Resource D is added and Resource C is deleted.

Set deployment mode

To set the deployment mode when deploying with PowerShell, use the `Mode` parameter.

```
New-AzResourceGroupDeployment ` 
    -Mode Complete ` 
    -Name ExampleDeployment ` 
    -ResourceGroupName ExampleResourceGroup ` 
    -TemplateFile c:\MyTemplates\storage.json
```

To set the deployment mode when deploying with Azure CLI, use the `mode` parameter.

```
az deployment group create \
--mode Complete \
--name ExampleDeployment \
--resource-group ExampleResourceGroup \
--template-file storage.json
```

Exercise: Create and deploy Azure Resource Manager templates by using Visual Studio Code

In this exercise you will learn how to use Visual Studio Code, and the Azure Resource Manager Tools extension, to create and edit Azure Resource Manager templates.

- Create an Azure Resource Manager template
- Add an Azure resource to the template
- Add parameters to the template
- Create a parameter file
- Deploy the template
- Clean up resources

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 Code²¹** with the **Azure Resource Manager Tools²²** installed.
- **Azure CLI²³** installed locally

Create an Azure Resource Manager template

1. Create and open a new file named `azuredeploy.json` with Visual Studio Code.
2. Enter `arm` in the `azuredeploy.json` file and select `arm!` from the autocomplete options. This will insert a snippet with the basic building blocks for an Azure resource group deployment.



²¹ <https://code.visualstudio.com/>

²² <https://marketplace.visualstudio.com/items?itemName=msazurermttools.azure-vmss-tools>

²³ <https://docs.microsoft.com/cli/azure/>

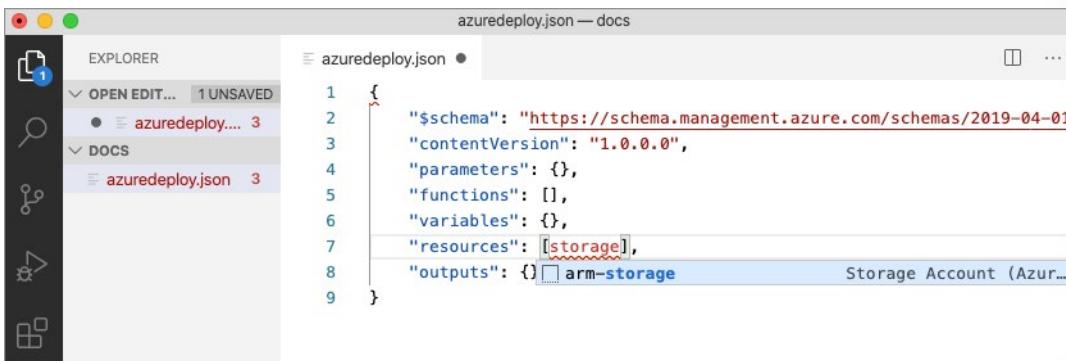
Your file should contain something similar to the example below.

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {},  
    "functions": [],  
    "variables": {},  
    "resources": [],  
    "outputs": {}  
}
```

Add an Azure resource to the template

In this section you will add a snippet to support the creation of an Azure storage account to the template.

Place the cursor in the template resources block, type in `storage`, and select the **arm-storage** snippet.



The resources block should look similar to the example below.

```
"resources": [  
    {  
        "name": "storageaccount1",  
        "type": "Microsoft.Storage/storageAccounts",  
        "apiVersion": "2019-06-01",  
        "tags": {  
            "displayName": "storageaccount1"  
        },  
        "location": "[resourceGroup().location]",  
        "kind": "StorageV2",  
        "sku": {  
            "name": "Premium_LRS",  
            "tier": "Premium"  
        }  
    }]  
,
```

Add parameters to the template

Now you will create and use a parameter to specify the storage account name.

Place your cursor in the parameters block, add a carriage return, type ", and then select the new-parameter snippet. This action adds a generic parameter to the template.

```
azuredeploy.json
1  {
2    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/dep...
3    "contentVersion": "1.0.0.0",
4    "parameters": [
5      ""
6      , "new-paramet... ARM Template Parameter Definition...
7      "functions": [],
8      "variables": {},
9      "resources": [
10        {
11          "name": "storageaccount1",
12          "type": "Microsoft.Storage/storageAccounts",
13          "apiVersion": "2019-06-01"
14        }
15      ]
16    ]
17  }
```

Make the following changes to the new parameter you just added:

1. Update the name of the parameter to `storageAccountName` and the description to Storage Account Name.
2. Azure storage account names have a minimum length of 3 characters and a maximum of 24. Add both `minLength` and `maxLength` to the parameter and provide appropriate values.

The `parameters` block should look similar to the example below.

```
"parameters": {
  "storageAccountName": {
    "type": "string",
    "metadata": {
      "description": "Storage Account Name"
    },
    "minLength": 3,
    "maxLength": 24
  }
},
```

Follow the steps below to update the name property of the storage resource to use the parameter.

1. In the `resources` block, delete the current default name which is `storageaccount1` in the examples above. Leave the quotes ("") around the name in place.
2. Enter a square bracket [, which produces a list of Azure Resource Manager template functions. Select **parameters** from the list.
3. Add () at the end of **parameters** and select **storageAccountName** from the pop-up. If the list of parameters does not show up automatically you can enter a single quote ' inside of the round brackets to display the list.

The resources block of the template should now be similar to the example below.

```
"resources": [
  {
    "name": "[parameters('storageAccountName')]",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-06-01",
    "dependsOn": []
  }
],
```

```

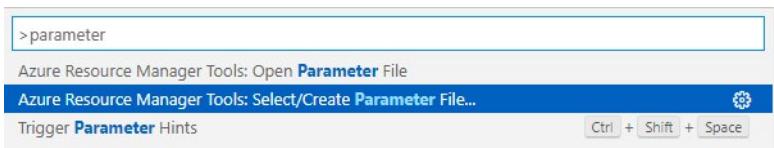
    "tags": {
        "displayName": "storageaccount1"
    },
    "location": "[resourceGroup().location]",
    "kind": "StorageV2",
    "sku": {
        "name": "Premium_LRS",
        "tier": "Premium"
    }
},

```

Create a parameter file

An Azure Resource Manager template parameter file allows you to store environment-specific parameter values and pass these values in as a group at deployment time. This useful if you want to have values specific to a test or production environment, for example. The extension makes it easy to create a parameter file that is mapped to your existing template. Follow the steps below to create a parameter file.

1. With the `azuredeploy.json` file in focus open the **Command Palette** by selecting **View > Command Palette** from the menu bar.
2. In the **Command Palette** enter “parameter” in the search bar and select **Azure Resource Manager Tools:Select/Create Parameter File**.



3. A new dialog box will open at the top of the editor. From those options select **New**, then select **All Parameters**. Accept the default name for the new file.
4. Edit the `value` parameter and type in a name that meets the naming requirements. The `azuredeploy.parameters.json` file should be similar to the example below.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "storageAccountName": {
            "value": "az204storageacctarm"
        }
    }
}
```

Deploy the template

It's time to deploy the template. Follow the steps below, in the VS Code terminal, to connect to Azure and deploy the new storage account resource.

1. Connect to Azure by using the `az login` command.

```
az login
```

2. Create a resource group to contain the new resource. Replace <myLocation> with a region near you.

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

3. Use the az deployment group create command to deploy your template. The deployment will take a few minutes to complete, progress will be shown in the terminal.

```
az deployment group create --resource-group az204-arm-rg --template-file azuredeploy.json  
--parameters azuredeploy.parameters.json
```

4. You can verify the deployment by running the command below. Replace <myStorageAccount> with the name you used earlier.

```
az storage account show --resource-group az204-arm-rg --name <myStorageAccount>
```

Clean up resources

When the Azure resources are no longer needed use the Azure CLI command below to delete the resource group.

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

Knowledge check

Multiple choice

*What purpose does the **outputs** section of an Azure Resource Manager template serve?*

- Specify the resources to deploy.
- Return values from the deployed resources
- Define values that are reused in your templates.

Multiple choice

Which Azure Resource Manager template deployment mode deletes resources in a resource group that aren't specified in the template?

- Incremental
- Complete
- Both incremental and complete delete resources

Summary

In this module, you learned how to:

- Describe what role Azure Resource Manager has in Azure and the benefits of using Azure Resource Manager templates

- Explain what happens when Azure Resource Manager templates are deployed and how to structure them to support your solution
- Create a template with conditional resource deployments
- Choose the correct deployment mode for your solution
- Create and deploy an Azure Resource Manager template by using Visual Studio Code

Manage container images in Azure Container Registry

Introduction

Azure Container Registry (ACR) is a managed, private Docker registry service based on the open-source Docker Registry 2.0. Create and maintain Azure container registries to store and manage your private Docker container images.

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

- Explain the features and benefits Azure Container Registry offers
- Describe how to use ACR Tasks to automate builds and deployments
- Explain the elements in a Dockerfile
- Build and run an image in the ACR by using Azure CLI

Discover the Azure Container Registry

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, 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**²⁴ and images built to the **Open Container Initiative (OCI) Image Format Specification**²⁵.

Automated image builds

Use **Azure Container Registry Tasks**²⁶ (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

Every Basic, Standard, and Premium Azure container registry benefits 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

²⁴ <https://docs.microsoft.com/azure/container-registry/container-registry-helm-repos>

²⁵ <https://github.com/opencontainers/image-spec/blob/master/spec.md>

²⁶ <https://docs.microsoft.com/azure/container-registry/container-registry-tasks-overview>

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 is not 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**²⁷.

Very 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 *cannot* be recovered after deletion.

Build and manage containers with tasks

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.

²⁷ <https://docs.microsoft.com/azure/container-registry/container-registry-skus#service-tier-features-and-limits>

Using the familiar `docker build` format, the **az acr build²⁸** 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 public or private Git repository in GitHub or Azure DevOps. 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 file²⁹** 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

Explore elements of a Dockerfile

If you want to create a custom container you will need to understand the elements of a Dockerfile. A Dockerfile is a text file that contains the instructions we use to build and run a Docker image. The following aspects of the image are defined:

- The base or parent image we use to create the new image
- Commands to update the base OS and install additional software

²⁸ https://docs.microsoft.com/cli/azure/acr#az_acr_build

²⁹ <https://docs.microsoft.com/azure/container-registry/container-registry-tasks-reference-yaml>

- 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

Let's map these aspects to an example Dockerfile. Suppose we're creating a Docker image for an ASP.NET Core website. The Dockerfile may look like the following example.

```
# Step 1: Specify the parent image for the new image
FROM ubuntu:18.04

# Step 2: Update OS packages and install additional software
RUN apt -y update && apt install -y wget nginx software-properties-common
apt-transport-https \
    && wget -q https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb \
    && dpkg -i packages-microsoft-prod.deb \
    && add-apt-repository universe \
    && apt -y update \
    && apt install -y dotnet-sdk-3.0

# Step 3: Configure Nginx environment
CMD service nginx start

# Step 4: Configure Nginx environment
COPY ./default /etc/nginx/sites-available/default

# STEP 5: Configure work directory
WORKDIR /app

# STEP 6: Copy website code to container
COPY ./website/ .

# STEP 7: Configure network requirements
EXPOSE 80:8080

# STEP 8: Define the entry point of the process that runs in the container
ENTRYPOINT ["dotnet", "website.dll"]
```

We're not going to cover the Dockerfile file specification here or the detail of each command in our above example. However, notice that there are several commands in this file that allow us to manipulate the structure of the image.

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.

Finally, notice the last step, step 8. The `ENTRYPOINT` in the file indicates which process will execute once we run a container from an image.

Additional resources

- Dockerfile reference
 - <https://docs.docker.com/engine/reference/builder/>

Exercise: Build and run a container image by using Azure Container Registry Tasks

In this exercise you will use ACR Tasks to perform the following actions:

- Create an Azure Container Registry
- Build and push image from a Dockerfile
- Verify the results
- Run the image in the ACR

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>

Login to Azure and start the Cloud Shell

1. Login to the **Azure portal**³⁰ and open the Cloud Shell.



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



Create an Azure Container Registry

1. Create a resource group for the registry, replace <myLocation> in the command below with a location near you.

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

2. Create a basic container registry. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. Replace <myContainerRegistry> in the command below with a unique value.

```
az acr create --resource-group az204-acr-rg \
--name <myContainerRegistry> --sku Basic
```

³⁰ <https://portal.azure.com>

Note: The command above creates a *Basic* registry, a cost-optimized option for developers learning about Azure Container Registry.

Build and push image from a Dockerfile

Now use Azure Container Registry to build and push an image based on a local Dockerfile.

1. Create, or navigate, to a local directory and then use the command below to create the Dockerfile.

The Dockerfile will contain a single line that references the `hello-world` image hosted at the Microsoft Container Registry.

```
echo FROM mcr.microsoft.com/hello-world > Dockerfile
```

2. Run the `az acr build` command, which builds the image and, after the image is successfully built, pushes it to your registry. Replace `<myContainerRegistry>` with the name you used earlier.

```
az acr build --image sample/hello-world:v1 \
--registry <myContainerRegistry> \
--file Dockerfile .
```

The command above will generate a lot of output, below is shortened sample of that output showing the last few lines with the final results. You can see in the `repository` field the `sample/hello-world` image is listed.

```
- image:
  registry: <myContainerRegistry>.azurecr.io
  repository: sample/hello-world
  tag: v1
  digest: sha256:92c7f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3b-
c396879d95e899a
  runtime-dependency:
    registry: mcr.microsoft.com
    repository: hello-world
    tag: latest
    digest: sha256:92c7f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3b-
c396879d95e899a
  git: {}
```

```
Run ID: cf1 was successful after 11s
```

Verify the results

1. Use the `az acr repository list` command to list the repositories in your registry. Replace `<myContainerRegistry>` with the name you used earlier.

```
az acr repository list --name <myContainerRegistry> --output table
```

Output:

Result

```
sample/hello-world
```

2. Use the `az acr repository show-tags` command to list the tags on the **sample/hello-world** repository. Replace `<myContainerRegistry>` with the name you used earlier.

```
az acr repository show-tags --name <myContainerRegistry> \  
--repository sample/hello-world --output table
```

Output:

```
Result  
-----  
v1
```

Run the image in the ACR

1. Run the `sample/hello-world:v1` container image from your container registry by using the `az acr run` command. The following example uses `$Registry` to specify the registry where you run the command. Replace `<myContainerRegistry>` with the name you used earlier.

```
az acr run --registry <myContainerRegistry> \  
--cmd '$Registry/sample/hello-world:v1' /dev/null
```

The `cmd` parameter in this example runs the container in its default configuration, but `cmd` supports additional `docker run` parameters or even other `docker` commands.

Below is shortened sample of the output:

```
Packing source code into tar to upload...  
Uploading archived source code from '/tmp/run_archive_ebf74da7fcb-  
04683867b129e2ccad5e1.tar.gz'...  
Sending context (1.855 KiB) to registry: mycontainerre...  
Queued a run with ID: cab  
Waiting for an agent...  
2019/03/19 19:01:53 Using acb_vol_60e9a538-b466-475f-9565-80c5b93ea15 as  
the home volume  
2019/03/19 19:01:53 Creating Docker network: acb_default_network, driver:  
'bridge'  
2019/03/19 19:01:53 Successfully set up Docker network: acb_default_network  
2019/03/19 19:01:53 Setting up Docker configuration...  
2019/03/19 19:01:54 Successfully set up Docker configuration  
2019/03/19 19:01:54 Logging in to registry: mycontainerregistry008.azurecr.  
io  
2019/03/19 19:01:55 Successfully logged into mycontainerregistry008.azurecr.  
io  
2019/03/19 19:01:55 Executing step ID: acb_step_0. Working directory: '',  
Network: 'acb_default_network'  
2019/03/19 19:01:55 Launching container with name: acb_step_0  
  
Hello from Docker!  
This message shows that your installation appears to be working correctly.
```

```
2019/03/19 19:01:56 Successfully executed container: acb_step_0
2019/03/19 19:01:56 Step ID: acb_step_0 marked as successful (elapsed time
in seconds: 0.843801)
```

Run ID: cab was successful after 6s

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.

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

Knowledge check

Multiple choice

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

- Basic
- Standard
- Premium

Summary

In this module, you learned how to:

- Explain the features and benefits Azure Container Registry offers
- Describe how to use ACR Tasks to automate builds and deployments
- Explain the elements in a Dockerfile
- Build and run an image in the ACR by using Azure CLI

Run container images in Azure Container Instances

Introduction

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.

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

- Describe the benefits of Azure Container Instances and how resources are grouped
- Deploy a container instance in Azure by using the Azure CLI
- Start and stop containers using policies
- Set environment variables in your container instances
- Mount file shares in your container instances

Explore Azure Container Instances

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.

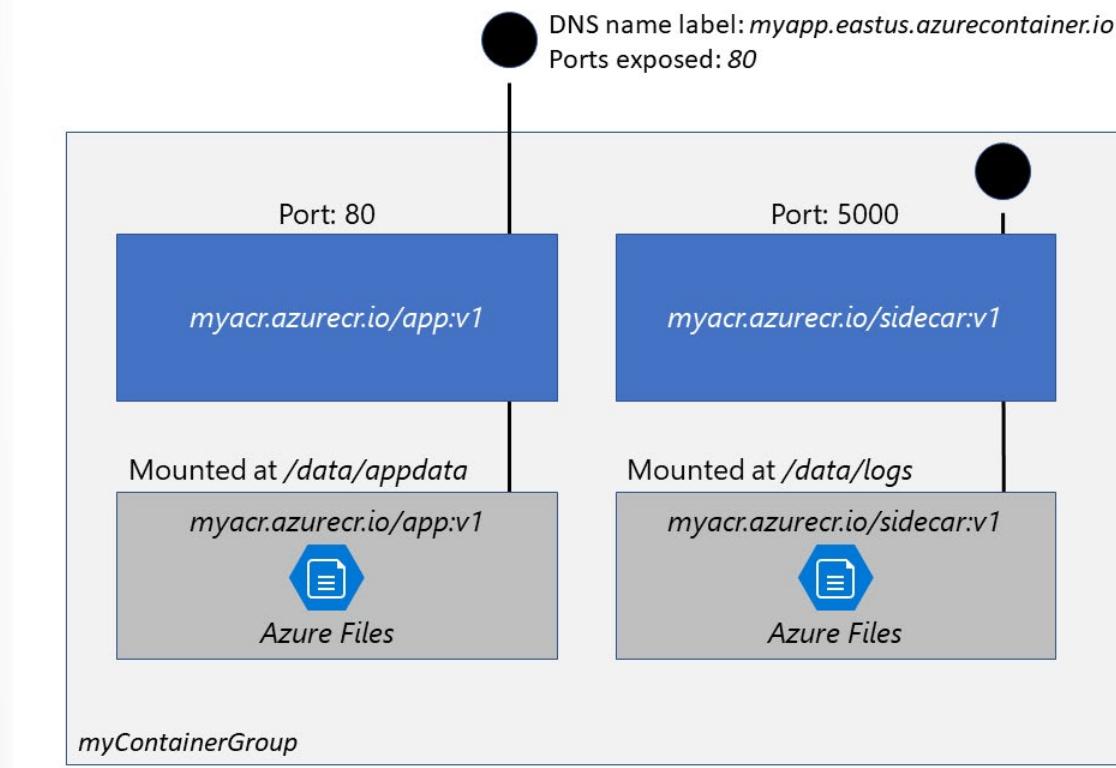
For scenarios where you need full container orchestration, including service discovery across multiple containers, automatic scaling, and coordinated application upgrades, we recommend **Azure Kubernetes Service (AKS)**³¹.

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:

³¹ <https://docs.microsoft.com/azure/aks/>



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. Taking CPU resources as

an example, if you create a container group with two instances, each requesting 1 CPU, then the container group is allocated 2 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 have 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 small number of 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

In this exercise you'll 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>

Login to Azure and create the resource group

1. Login to the **Azure portal**³² and open the Cloud Shell.



2. After 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 will be easier to clean up these resources when you are finished with the module. Replace <myLocation> with a region near you.

```
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 will 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.

```
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 will take a few minutes for the operation to complete.

```
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 commands above, `$DNS_NAME_LABEL` specifies your DNS name. The image name, `mcr.microsoft.com/azuredocs/aci-helloworld`, refers to a Docker image hosted on Docker Hub that runs a basic Node.js web application.

³² <https://portal.azure.com>

Verify the container is running

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

```
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.

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.

- 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.

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

Run containerized tasks with restart policies

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.

Restart policy	Description
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`.

```
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

Setting environment variables in your container instances allows you to provide dynamic configuration of the application or script run by the container. This is 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 example below two variables are passed to the container when it is created. The example below is assuming you are 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"`.

```
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.

```
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:

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

Mount an Azure file share in Azure Container Instances

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`. Below is an example of the command:

```
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. An example YAML file is show below.

```
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:

```
"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:

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

```
    "mountPath": "/mnt/share2/"  
  } ]
```

Knowledge check

Multiple choice

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

- Azure Resource Management template
- YAML file
- az container create command

Summary

In this module, you learned how to:

- Describe the benefits of Azure Container Instances and how resources are grouped
- Deploy a container instance in Azure by using the Azure CLI
- Start and stop containers using policies
- Set environment variables in your container instances
- Mount file shares in your container instances

Answers

Multiple choice

Which of the following Azure virtual machine types is most appropriate for testing and development?

- Compute optimized
- General Purpose
- Storage optimized

Explanation

That's correct. This type has a balanced CPU-to-memory ratio, and is ideal for testing and development.

Multiple choice

Which of the below represents a logical grouping of VMs that allows Azure to understand how your application is built to provide for redundancy and availability?

- Load balancer
- Availability zone
- Availability set

Explanation

That's correct. An availability set is a logical grouping of VMs Reason.

Multiple choice

What purpose does the **outputs** section of an Azure Resource Manager template serve?

- Specify the resources to deploy.
- Return values from the deployed resources
- Define values that are reused in your templates.

Explanation

That's correct. The "outputs" section returns values from the resource(s) that were deployed.

Multiple choice

Which Azure Resource Manager template deployment mode deletes resources in a resource group that aren't specified in the template?

- Incremental
- Complete
- Both incremental and complete delete resources

Explanation

That's correct. Complete mode will delete resources not specified in an Azure Resource Manager template deployment.

Multiple choice

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

- Basic
- Standard
- Premium

Explanation

That's correct. The premium tier adds geo-replication as a feature.

Multiple choice

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

- Azure Resource Management template
- YAML file
- az container create command

Explanation

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

Module 6 Implement user authentication and authorization

Explore the Microsoft identity platform

Introduction

The Microsoft identity platform for developers is a set of tools which 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

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

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) as well as 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, which 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 entity¹** defines the schema for an application object's properties.

Service principal object

To access resources that are 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).

¹ <https://docs.microsoft.com/graph/api/resources/application>

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** - The 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**². 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 cannot 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 does not 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 1:1 relationship with the software application, and
- A 1:many relationship with its corresponding service principal object(s).

A service principal must be created in each tenant where the application is used, enabling 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

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**³ 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*.

² <https://docs.microsoft.com/azure/active-directory/managed-identities-azure-resources/overview>

³ <https://docs.microsoft.com/azure/active-directory/develop/active-directory-v2-protocols>

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 scopes⁴** as well as 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 endpoint⁵**.

Note: 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 *application permissions*.

- **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.
- **Application 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 application permissions.

Consent types

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

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

⁴ <https://docs.microsoft.com/azure/active-directory/develop/v2-permissions-and-consent#openid-connect-scopes>

⁵ <https://docs.microsoft.com/azure/active-directory/develop/v2-permissions-and-consent#admin-restricted-permissions>

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) has not granted consent for this app, then Microsoft identity platform will prompt the user to provide consent at this time. Static permissions also enables 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 is 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 additional 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 pre-define them in the application registration information. If the user hasn't yet consented to new scopes added to the request, they'll be prompted to consent only to the new permissions. Incremental, or dynamic consent, only applies to delegated permissions and not to application 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 additional 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.

```
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

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 authentication⁶**
- 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 does not 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.

⁶ <https://docs.microsoft.com/azure/active-directory/authentication/concept-mfa-howitworks>

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 are 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 are building a native 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 native 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 native app, which needs to comply with the Conditional Access policy.

Knowledge check

Multiple choice

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
- Application permissions
- Both delegated and application permissions

Multiple choice

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
- Apps performing the Integrated Windows authentication flow

Summary

In this module, you learned how 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

Implement authentication by using the Microsoft Authentication Library

Introduction

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

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 a number of 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 are 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 a number of 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 table below.

Library	Supported platforms and frameworks
MSAL for Android (https://github.com/AzureAD/microsoft-authentication-library-for-android)	Android
MSAL Angular (https://github.com/AzureAD/microsoft-authentication-library-for-js/tree/dev/lib/msal-angular)	Single-page apps with Angular and Angular.js frameworks
MSAL for iOS and macOS (https://github.com/AzureAD/microsoft-authentication-library-for-objc)	iOS and macOS
MSAL Go (Preview) (https://github.com/AzureAD/microsoft-authentication-library-for-go)	Windows, macOS, Linux

Library	Supported platforms and frameworks
MSAL Java (https://github.com/AzureAD/microsoft-authentication-library-for-java)	Windows, macOS, Linux
MSAL.js (https://github.com/AzureAD/microsoft-authentication-library-for-js/tree/dev/lib/msal-browser)	JavaScript/TypeScript frameworks such as Vue.js, Ember.js, or Durandal.js
MSAL.NET (https://github.com/AzureAD/microsoft-authentication-library-for-dotnet)	.NET Framework, .NET Core, Xamarin Android, Xamarin iOS, Universal Windows Platform
MSAL Node (https://github.com/AzureAD/microsoft-authentication-library-for-js/tree/dev/lib/msal-node)	Web apps with Express, desktop apps with Electron, Cross-platform console apps
MSAL Python (https://github.com/AzureAD/microsoft-authentication-library-for-python)	Windows, macOS, Linux
MSAL React (https://github.com/AzureAD/microsoft-authentication-library-for-js/tree/dev/lib/msal-react)	Single-page apps with React and React-based libraries (Next.js, Gatsby.js)

Authentication flows

Below are some of the different authentication flows provided by Microsoft Authentication Library (MSAL). These flows can be used in a variety of different 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 are 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

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 are 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'll have also set the redirectUri where the identity provider will contact back 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.

```
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:

```
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, a number of `.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.

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

- `.WithRedirectUri` modifier: The `.WithRedirectUri` modifier overrides the default redirect URI. In the case of public client applications, this will be useful for scenarios which require a broker.

```
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 client confidential client.

Modifier	Description
<code>.WithAuthority()</code>	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.
<code>.WithTenantId(string tenantId)</code>	Overrides the tenant ID, or the tenant description.
<code>.WithClientId(string)</code>	Overrides the client ID.
<code>.WithRedirectUri(string redirectUri)</code>	Overrides the default redirect URI. In the case of public client applications, this will be useful for scenarios requiring a broker.
<code>.WithComponent(string)</code>	Sets the name of the library using MSAL.NET (for telemetry reasons).
<code>.WithDebugLoggingCallback()</code>	If called, the application will call <code>Debug.WriteLine</code> simply enabling debugging traces.
<code>.WithLogging()</code>	If called, the application will call a callback with debugging traces.
<code>.WithTelemetry(TelemetryCallback telemetryCallback)</code>	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 Directory.
.WithClientSecret(string clientSecret)	Sets the client secret (app password) identifying the application with Azure Active Directory.

Exercise: Implement interactive authentication by using MSAL.NET

In this exercise you'll 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>⁷.

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 <code>http://localhost</code> in the box to the right.

Below are more details on the **Supported account types**.

5. Select **Register**.

⁷ <https://code.visualstudio.com/>

The screenshot shows the 'Register an application' page in the Microsoft Azure portal. At the top, there's a navigation bar with links for Home, Contoso AD (dev), and a search bar. The main title is 'Register an application'. A required field 'Name' is highlighted with a red asterisk, followed by a placeholder 'The user-facing display name for this application (this can be changed later)'. Below this is a large empty input field. The next section is titled 'Supported account types' with the question 'Who can use this application or access this API?'. There are four radio button options: 'Accounts in this organizational directory only (Contoso AD (dev) - Single tenant)' (selected), 'Accounts in any organizational directory (Any Azure AD directory - Multitenant)', 'Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)', and 'Personal Microsoft accounts only'. A 'Help me choose...' link is provided. The next section is 'Redirect URI (optional)' with the note 'We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.' It includes a dropdown menu set to 'Web' and an input field containing 'e.g. https://myapp.com/auth'. At the bottom, there's a link 'By proceeding, you agree to the Microsoft Platform Policies' with a 'View' link, and a blue 'Register' button.

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.

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

3. Create the .NET console app.

```
dotnet new console
```

4. Open the *az204-auth* folder in VS Code.

```
code . -r
```

Build the console app

In this section you will 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 VS Code.

```
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.

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

3. Change the `Main` method to enable `async`.

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

Add code for the interactive authentication

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

```
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.

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

Code	Description
<code>.Create</code>	Creates a <code>PublicClientApplicationBuilder</code> from a clientID.
<code>.WithAuthority</code>	Adds a known Authority corresponding to an ADFS server. In the code we're specifying the Public cloud, and using 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. We'll use that permission to acquire a token.

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

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

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

```
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 example below.

```
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";

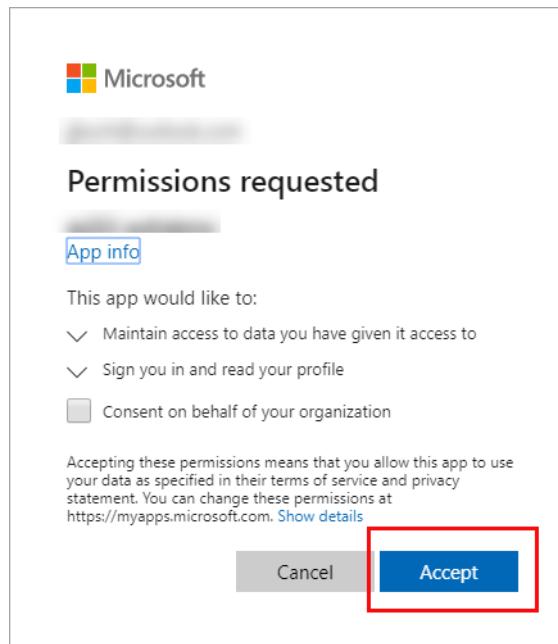
        public static async Task Main(string[] args)
        {
            var app = PublicClientApplicationBuilder
                .Create(_clientId)
                .WithAuthority(AzureCloudInstance.AzurePublic, _tenantId)
                .WithRedirectUri("http://localhost")
                .Build();
    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

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

Run the application

1. In the VS Code terminal run `dotnet build` to check for errors, then `dotnet run` to run the app.

2. The app will open 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 will receive a **Permissions requested** notification asking you to approve the app to read data associated with your account. Select **Accept**.



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

Token: eyJ0eXAiOiJKV1QiLCJub25jZSI6I1VhU.....

Knowledge check

Multiple choice

Which of the following MSAL libraries supports single-page web apps?

- MSAL Node
- MSAL.js
- MSAL.NET

Summary

In this module, you learned how 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

Implement shared access signatures

Introduction

A shared access signature (SAS) is a URI that grants restricted access rights to Azure Storage resources. You can provide a shared access signature to clients that you want grant delegate access to certain storage account resources.

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

- Identify the three types of shared access signatures
- Explain when to implement shared access signatures
- Create a stored access policy

Discover shared access signatures

A shared access signature (SAS) is a signed URI that points to one or more storage resources and includes a token that contains a special set of query parameters. The token indicates how the resources may be accessed by the client. One of the query parameters, the signature, is constructed from the SAS parameters and signed with the key that was used to create the SAS. This signature is used by Azure Storage to authorize access to the storage resource.

Types of shared access signatures

Azure Storage supports three types of shared access signatures:

- **User delegation SAS:** A user delegation SAS is secured with Azure Active Directory credentials and also by the permissions specified for the SAS. A user delegation SAS applies to Blob storage only.
- **Service SAS:** A service SAS is secured with the storage account key. A service SAS delegates access to a resource in the following Azure Storage services: Blob storage, Queue storage, Table storage, or Azure Files.
- **Account SAS:** An account SAS is secured with the storage account key. An account SAS delegates access to resources in one or more of the storage services. All of the operations available via a service or user delegation SAS are also available via an account SAS.

Note: Microsoft recommends that you use Azure Active Directory credentials when possible as a security best practice, rather than using the account key, which can be more easily compromised. When your application design requires shared access signatures for access to Blob storage, use Azure Active Directory credentials to create a user delegation SAS when possible for superior security.

How shared access signatures work

When you use a SAS to access data stored in Azure Storage, you need two components. The first is a URI to the resource you want to access. The second part is a SAS token that you've created to authorize access to that resource.

In a single URI, such as `https://medicalrecords.blob.core.windows.net/patient-images/patient-116139-nq8z7f.jpg?sp=r&st=2020-01-20T11:42:32Z&se=2020-01-20T19:42:32`

Z&spr=https&sv=2019-02-02&sr=b&sig=SrW1HZ5Nb6MbRzTbXCaPm%2BJiSEn15tC91Y4umMP-wVZs%3D, you can separate the URI from the SAS token as follows:

- **URI:**https://medicalrecords.blob.core.windows.net/patient-images/patient-116139-nq8z7f.jpg?
- **SAS token:**sp=r&st=2020-01-20T11:42:32Z&se=2020-01-20T19:42:32Z&spr=https&sv=2019-02-02&sr=b&sig=SrW1HZ5Nb6MbRzTbXCaPm%2BJiSEn15tC91Y4umMPwVZs%3D

The SAS token itself is made up of several components.

Component	Description
sp=r	Controls the access rights. The values can be a for add, c for create, d for delete, l for list, r for read, or w for write. This example is read only. The example sp=acdlrw grants all the available rights.
st=2020-01-20T11:42:32Z	The date and time when access starts.
se=2020-01-20T19:42:32Z	The date and time when access ends. This example grants eight hours of access.
sv=2019-02-02	The version of the storage API to use.
sr=b	The kind of storage being accessed. In this example, b is for blob.
sig=SrW1HZ5Nb6MbRzTbXCaPm%2BJiSEn15t-C91Y4umMPwVZs%3D	The cryptographic signature.

Best practices

To reduce the potential risks of using a SAS, Microsoft provides some guidance:

- To securely distribute a SAS and prevent man-in-the-middle attacks, always use HTTPS.
- The most secure SAS is a user delegation SAS. Use it wherever possible because it removes the need to store your storage account key in code. You must use Azure Active Directory to manage credentials. This option might not be possible for your solution.
- Try to set your expiration time to the smallest useful value. If a SAS key becomes compromised, it can be exploited for only a short time.
- Apply the rule of minimum-required privileges. Only grant the access that's required. For example, in your app, read-only access is sufficient.
- There are some situations where a SAS isn't the correct solution. When there's an unacceptable risk of using a SAS, create a middle-tier service to manage users and their access to storage.

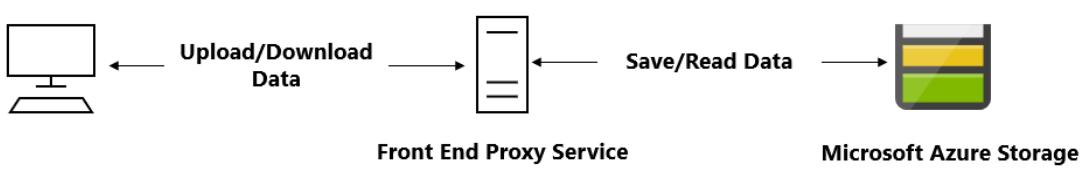
The most flexible and secure way to use a service or account SAS is to associate the SAS tokens with a stored access policy.

Choose when to use shared access signatures

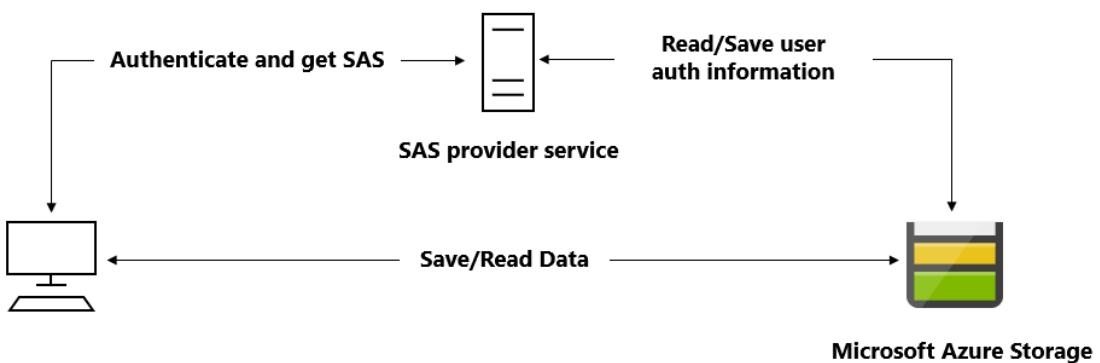
Use a SAS when you want to provide secure access to resources in your storage account to any client who does not otherwise have permissions to those resources.

A common scenario where a SAS is useful is a service where users read and write their own data to your storage account. In a scenario where a storage account stores user data, there are two typical design patterns:

- Clients upload and download data via a front-end proxy service, which performs authentication. This front-end proxy service has the advantage of allowing validation of business rules, but for large amounts of data or high-volume transactions, creating a service that can scale to match demand may be expensive or difficult.



- A lightweight service authenticates the client as needed and then generates a SAS. Once the client application receives the SAS, they can access storage account resources directly with the permissions defined by the SAS and for the interval allowed by the SAS. The SAS mitigates the need for routing all data through the front-end proxy service.



Many real-world services may use a hybrid of these two approaches. For example, some data might be processed and validated via the front-end proxy, while other data is saved and/or read directly using SAS.

Additionally, a SAS is required to authorize access to the source object in a copy operation in certain scenarios:

- When you copy a blob to another blob that resides in a different storage account, you must use a SAS to authorize access to the source blob. You can optionally use a SAS to authorize access to the destination blob as well.
- When you copy a file to another file that resides in a different storage account, you must use a SAS to authorize access to the source file. You can optionally use a SAS to authorize access to the destination file as well.
- When you copy a blob to a file, or a file to a blob, you must use a SAS to authorize access to the source object, even if the source and destination objects reside within the same storage account.

Explore stored access policies

A stored access policy provides an additional level of control over service-level shared access signatures (SAS) on the server side. Establishing a stored access policy groups shared access signatures and provides additional restrictions for signatures that are bound by the policy. You can use a stored access policy to change the start time, expiry time, or permissions for a signature, or to revoke it after it has been issued.

The following storage resources support stored access policies:

- Blob containers
- File shares
- Queues
- Tables

Creating a stored access policy

The access policy for a SAS consists of the start time, expiry time, and permissions for the signature. You can specify all of these parameters on the signature URI and none within the stored access policy; all on the stored access policy and none on the URI; or some combination of the two. However, you cannot specify a given parameter on both the SAS token and the stored access policy.

To create or modify a stored access policy, call the `Set ACL` operation for the resource (see [Set Container ACL⁸](#), [Set Queue ACL⁹](#), [Set Table ACL¹⁰](#), or [Set Share ACL¹¹](#)) with a request body that specifies the terms of the access policy. The body of the request includes a unique signed identifier of your choosing, up to 64 characters in length, and the optional parameters of the access policy, as follows:

Note: When you establish a stored access policy on a container, table, queue, or share, it may take up to 30 seconds to take effect. During this time requests against a SAS associated with the stored access policy may fail with status code 403 (Forbidden), until the access policy becomes active. Table entity range restrictions (`startpk`, `startrk`, `endpk`, and `endrk`) cannot be specified in a stored access policy.

Below are examples of creating a stored access policy by using C# .NET and the Azure CLI.

```
BlobSignedIdentifier identifier = new BlobSignedIdentifier
{
    Id = "stored access policy identifier",
    AccessPolicy = new BlobAccessPolicy
    {
        ExpiresOn = DateTimeOffset.UtcNow.AddHours(1),
        Permissions = "rw"
    }
};

blobContainer.SetAccessPolicy(permissions: new BlobSignedIdentifier[] {
    identifier });

az storage container policy create \
    --name <stored access policy identifier> \
    --container-name <container name> \
```

⁸ <https://docs.microsoft.com/rest/api/storageservices/set-container-acl>

⁹ <https://docs.microsoft.com/rest/api/storageservices/set-queue-acl>

¹⁰ <https://docs.microsoft.com/rest/api/storageservices/set-table-acl>

¹¹ <https://docs.microsoft.com/rest/api/storageservices/set-share-acl>

```
--start <start time UTC datetime> \
--expiry <expiry time UTC datetime> \
--permissions <(a)dd, (c)reate, (d)elete, (l)ist, (r)ead, or (w)rite> \
--account-key <storage account key> \
--account-name <storage account name> \
```

Modifying or revoking a stored access policy

To modify the parameters of the stored access policy you can call the access control list operation for the resource type to replace the existing policy. For example, if your existing policy grants read and write permissions to a resource, you can modify it to grant only read permissions for all future requests.

To revoke a stored access policy you can delete it, rename it by changing the signed identifier, or change the expiry time to a value in the past. Changing the signed identifier breaks the associations between any existing signatures and the stored access policy. Changing the expiry time to a value in the past causes any associated signatures to expire. Deleting or modifying the stored access policy immediately affects all of the SAS associated with it.

To remove a single access policy, call the resource's `Set ACL` operation, passing in the set of signed identifiers that you wish to maintain on the container. To remove all access policies from the resource, call the `Set ACL` operation with an empty request body.

Knowledge check

Multiple choice

Which of the following types of shared access signatures (SAS) applies to Blob storage only?

- Account SAS
- Service SAS
- User delegation SAS

Multiple choice

Which of the following best practices provides the most flexible and secure way to use a service or account shared access signature (SAS)?

- Associate SAS tokens with a stored access policy.
- Always use HTTPS
- Implement a user delegation SAS

Summary

In this module, you learned how to:

- Identify the three types of shared access signatures
- Explain when to implement shared access signatures
- Create a stored access policy

Explore Microsoft Graph

Introduction

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

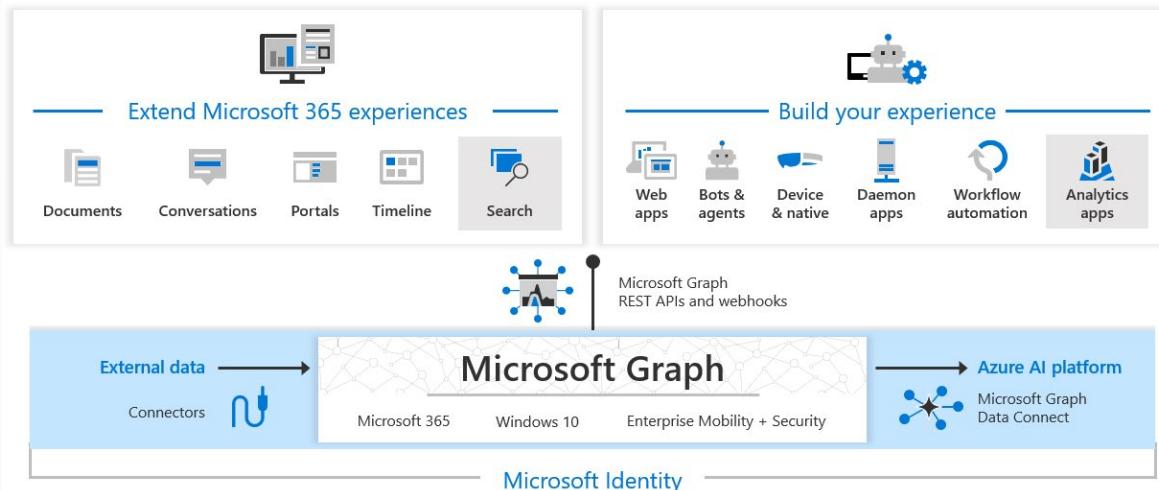
After completing this module, you'll be able to:

- 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

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.

Microsoft 365 Platform



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 connectors¹²** 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.

¹² <https://docs.microsoft.com/microsoftsearch/connectors-overview>

- **Microsoft Graph Data Connect**¹³ 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

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**¹⁴. A small number of API sets are defined in their sub-namespaces, such as the **call records API**¹⁵ which defines resources like `callRecord`¹⁶ 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, you construct a request that looks like the following:

```
{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.

¹³ <https://docs.microsoft.com/graph/overview#access-microsoft-graph-data-at-scale-using-microsoft-graph-data-connect>

¹⁴ <https://docs.microsoft.com/graph/traverse-the-graph#microsoft-graph-api-metadata>

¹⁵ <https://docs.microsoft.com/graph/api/resources/callrecords-api-overview>

¹⁶ <https://docs.microsoft.com/graph/api/resources/callrecords-callrecord>

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, usually 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; do not 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 will include the resource you are 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 additional 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 will 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 additional 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`.

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

Additional resources

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

- **Graph Explorer**¹⁷
- **Postman**¹⁸

Query Microsoft Graph by using SDKs

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 will 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**¹⁹ - 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**²⁰ - 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**²¹ - The core library for making calls to Microsoft Graph.
- **Microsoft.Graph.Auth**²² - Provides an authentication scenario-based wrapper of the Microsoft Authentication Library (MSAL) for use with the Microsoft Graph SDK. Microsoft.Graph.Auth has a dependency on Microsoft.Graph.Core.

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 will handle 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**²³.

¹⁷ <https://developer.microsoft.com/graph/graph-explorer>

¹⁸ <https://www.getpostman.com/>

¹⁹ <https://github.com/microsoftgraph/msgraph-sdk-dotnet>

²⁰ <https://github.com/microsoftgraph/msgraph-beta-sdk-dotnet>

²¹ <https://github.com/microsoftgraph/msgraph-sdk-dotnet>

²² <https://github.com/microsoftgraph/msgraph-sdk-dotnet-auth>

²³ <https://docs.microsoft.com/graph/sdk/choose-authentication-providers>

```
// Build a client application.  
IPublicClientApplication publicClientApplication = PublicClientApplication-  
Builder  
    .Create("INSERT-CLIENT-APP-ID")  
    .Build();  
// Create an authentication provider by passing in a client application and  
graph scopes.  
DeviceCodeProvider authProvider = new DeviceCodeProvider(publicClientAppli-  
cation, graphScopes);  
// Create a new instance of GraphServiceClient with the authentication  
provider.  
GraphServiceClient graphClient = new GraphServiceClient(authProvider);
```

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.

```
// 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 a number of 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 will request that the server provide the list of entities sorted by the specified properties.

```
// GET https://graph.microsoft.com/v1.0/me/messages?$select=subject,send-  
er&$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.

```
// 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.

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

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

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

Additional resources

- Microsoft Graph REST API v1.0 reference²⁴

Apply best practices to Microsoft Graph

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 will need to acquire an OAuth 2.0 access token, and present it to Microsoft Graph in either of the following:

- 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**²⁵ to acquire the access token to Microsoft Graph.

²⁴ <https://docs.microsoft.com/graph/api/overview>

²⁵ <https://docs.microsoft.com/azure/active-directory/develop/active-directory-v2-libraries>

Consent and authorization

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

- **Use least privilege.** Only request permissions that are absolutely 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 user²⁶](#) 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 appropriately²⁷](#).
 - Ensure that you understand the difference between [static, dynamic and incremental consent²⁸](#).
 - **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 a resource collection, you should expect that Microsoft Graph will return 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 will not contain an `@odata.nextLink` property. For more details, see [paging²⁹](#).
- **Evolvable enumerations:** Adding members to existing enumerations can break applications already using these enums. Evolvable enums is 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.

²⁶ <https://docs.microsoft.com/graph/api/user-post-users>

²⁷ <https://docs.microsoft.com/azure/active-directory/develop/v2-permissions-and-consent>

²⁸ <https://docs.microsoft.com/azure/active-directory/develop/v2-permissions-and-consent#consent-types>

²⁹ <https://docs.microsoft.com/graph/paging>

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 if necessary for a specific scenario, and if that use case is covered by your terms of use and privacy policy, and does not violate the **Microsoft APIs Terms of Use³⁰**. Your application should also implement proper retention and deletion policies.

Knowledge check

Multiple choice

Which HTTP method below is used to update a resource with new values?

- POST
- PATCH
- PUT

Multiple choice

Which of the components of the Microsoft 365 platform is used to deliver data external to Azure into Microsoft Graph services and applications?

- Microsoft Graph API
- Microsoft Graph connectors
- Microsoft Graph Data Connect

Multiple choice

Which of the following Microsoft Graph .NET SDK packages provides an authentication scenario-based wrapper of the Microsoft Authentication Library?

- Microsoft.Graph
- Microsoft.Graph.Core
- Microsoft.Graph.Auth

Summary

In this module, you learned how 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

³⁰ <https://docs.microsoft.com/legal/microsoft-apis/terms-of-use?context=/graph/context>

Answers

Multiple choice

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
- Application permissions
- Both delegated and application permissions

Explanation

That's 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.

Multiple choice

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
- Apps performing the Integrated Windows authentication flow

Explanation

That's correct. Apps performing the on-behalf-of flow require code to handle Conditional Access challenges.

Multiple choice

Which of the following MSAL libraries supports single-page web apps?

- MSAL Node
- MSAL.js
- MSAL.NET

Explanation

That's correct. MSAL.js supports single-page applications.

Multiple choice

Which of the following types of shared access signatures (SAS) applies to Blob storage only?

- Account SAS
- Service SAS
- User delegation SAS

Explanation

That's correct. A user delegation SAS is secured with Azure Active Directory credentials and also by the permissions specified for the SAS. A user delegation SAS applies to Blob storage only.

Multiple choice

Which of the following best practices provides the most flexible and secure way to use a service or account shared access signature (SAS)?

- Associate SAS tokens with a stored access policy.
- Always use HTTPS
- Implement a user delegation SAS

Explanation

That's correct. The most flexible and secure way to use a service or account SAS is to associate the SAS tokens with a stored access policy.

Multiple choice

Which HTTP method below is used to update a resource with new values?

- POST
- PATCH
- PUT

Explanation

That's correct. The PATCH method does update a resource with a new value.

Multiple choice

Which of the components of the Microsoft 365 platform is used to deliver data external to Azure into Microsoft Graph services and applications?

- Microsoft Graph API
- Microsoft Graph connectors
- Microsoft Graph Data Connect

Explanation

That's correct. Microsoft Graph connectors work in the incoming direction. Connectors exist for many commonly used data sources such as Box, Google Drive, Jira, and Salesforce.

Multiple choice

Which of the following Microsoft Graph .NET SDK packages provides an authentication scenario-based wrapper of the Microsoft Authentication Library?

- Microsoft.Graph
- Microsoft.Graph.Core
- Microsoft.Graph.Auth

Explanation

That's correct. The Microsoft.Graph.Auth package provides an authentication scenario-based wrapper of the Microsoft Authentication Library for use with the Microsoft Graph SDK.

Module 7 Implement secure cloud solutions

Implement Azure Key Vault

Introduction

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

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 page¹](https://azure.microsoft.com/pricing/details/key-vault/).

¹ <https://azure.microsoft.com/pricing/details/key-vault/>

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

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 to use a vault per application per environment (Development, Pre-Production and Production). This helps you not share secrets across environments and also reduces the threat in case of 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**² and purge protection if you want to guard against force deletion of the secret.

Authenticate to Azure Key Vault

Authentication with Key Vault works in conjunction 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 a variety of services.
- If you cannot 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.

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

² <https://docs.microsoft.com/azure/key-vault/general/soft-delete-overview>

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 (https://docs.microsoft.com/dotnet/api/overview/azure/identity-readme)	Azure Identity SDK Python (https://docs.microsoft.com/python/api/overview/azure/)	Azure Identity SDK Java (https://docs.microsoft.com/java/api/overview/azure/identity-readme)	Azure Identity SDK JavaScript (https://docs.microsoft.com/javascript/api/overview/azure/identity-readme)

Authentication to Key Vault with REST

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

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

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

```
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.

Additional resources

- [Azure Key Vault developer's guide³](#)
- [Azure Key Vault availability and redundancy⁴](#)

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

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

- Create a Key Vault
- Add and retrieve a secret

³ <https://docs.microsoft.com/azure/key-vault/general/developers-guide>

⁴ <https://docs.microsoft.com/azure/key-vault/general/disaster-recovery-guidance>

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>

Login to Azure and start the Cloud Shell

1. Login to the **Azure portal**⁵ and open the Cloud Shell.



2. After 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 below with a region that makes sense for you. The Key Vault name needs to be a globally unique name, and the script below generates a random string.

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

2. Create a resource group.

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

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

```
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 additional steps.

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

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

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

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

⁵ <https://portal.azure.com>

This command will return some JSON. The last line will contain the password in plain text.

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

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

Clean up resources

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

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

Knowledge check

Multiple choice

Which of the below methods of authenticating to Azure Key Vault is recommended for most scenarios?

- Service principal and certificate
- Service principal and secret
- Managed identities

Multiple choice

Azure Key Vault protects data when it is traveling between Azure Key Vault and clients. What protocol does it use for encryption?

- Secure Sockets Layer
- Transport Layer Security
- Presentation Layer

Summary

In this module, you learned how 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

Implement managed identities

Introduction

A common challenge for developers is the management of secrets and credentials used to secure communication between different components making up a solution. Managed identities eliminate the need for developers to manage credentials.

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

- Explain the differences between the two types of managed identities
- Describe the flows for user- and system-assigned managed identities
- Configure managed identities
- Acquire access tokens by using REST and code

Explore managed identities

Managed identities provide an identity for applications to use when connecting to resources that support Azure Active Directory (Azure AD) authentication. Applications may use the managed identity to obtain Azure AD tokens. For example, an application may use a managed identity to access resources like Azure Key Vault where developers can store credentials in a secure manner or to access storage accounts.

Types of managed identities

There are two types of managed identities:

- A **system-assigned managed identity** is enabled directly on an Azure service instance. When the identity is enabled, Azure creates an identity for the instance in the Azure AD tenant that's trusted by the subscription of the instance. After the identity is created, the credentials are provisioned onto the instance. The lifecycle of a system-assigned identity is directly tied to the Azure service instance that it's enabled on. If the instance is deleted, Azure automatically cleans up the credentials and the identity in Azure AD.
- A **user-assigned managed identity** is created as a standalone Azure resource. Through a create process, Azure creates an identity in the Azure AD tenant that's trusted by the subscription in use. After the identity is created, the identity can be assigned to one or more Azure service instances. The lifecycle of a user-assigned identity is managed separately from the lifecycle of the Azure service instances to which it's assigned.

Internally, managed identities are service principals of a special type, which are locked to only be used with Azure resources. When the managed identity is deleted, the corresponding service principal is automatically removed.

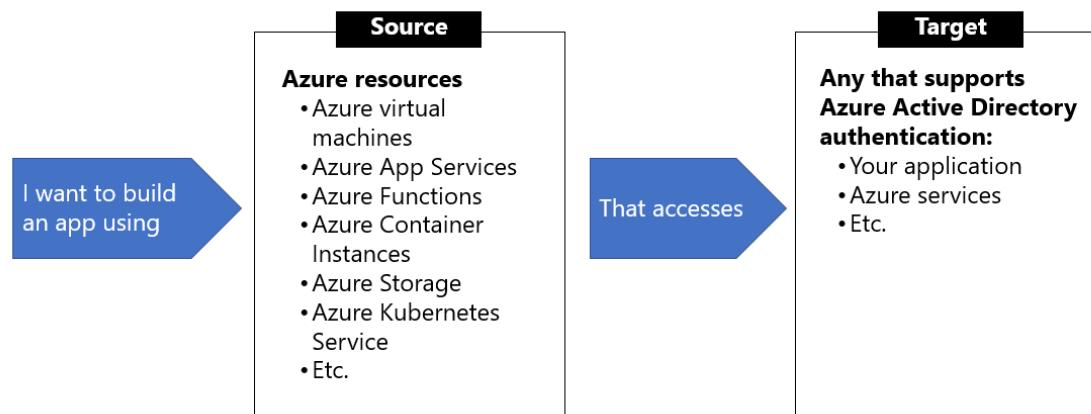
Characteristics of managed identities

The table below highlights some of the key differences between the two types of managed identities.

Characteristic	System-assigned managed identity	User-assigned managed identity
Creation	Created as part of an Azure resource (for example, an Azure virtual machine or Azure App Service)	Created as a stand-alone Azure resource
Lifecycle	Shared lifecycle with the Azure resource that the managed identity is created with. When the parent resource is deleted, the managed identity is deleted as well.	Independent life-cycle. Must be explicitly deleted.
Sharing across Azure resources	Cannot be shared, it can only be associated with a single Azure resource.	Can be shared, the same user-assigned managed identity can be associated with more than one Azure resource.

When to use managed identities

The image below gives an overview the scenarios that support using managed identities. For example, you can use managed identities if you want to build an app using Azure App Services that accesses Azure Storage without having to manage any credentials.



What Azure services support managed identities?

Managed identities for Azure resources can be used to authenticate to services that support Azure Active Directory authentication. For a list of Azure services that support the managed identities for Azure resources feature, visit [Services that support managed identities for Azure resources](#)⁶.

The rest of this module will use Azure virtual machines in the examples, but the same concepts and similar actions can be applied to any resource in Azure that supports Azure Active Directory authentication.

⁶ <https://docs.microsoft.com/azure/active-directory/managed-identities-azure-resources/services-support-msi>

Discover the managed identities authentication flow

In this unit, you learn how managed identities work with Azure virtual machines. Below are the flows detailing how the two types of managed identities work with an Azure virtual machine.

How a system-assigned managed identity works with an Azure virtual machine

1. Azure Resource Manager receives a request to enable the system-assigned managed identity on a virtual machine.
2. Azure Resource Manager creates a service principal in Azure Active Directory for the identity of the virtual machine. The service principal is created in the Azure Active Directory tenant that's trusted by the subscription.
3. Azure Resource Manager configures the identity on the virtual machine by updating the Azure Instance Metadata Service identity endpoint with the service principal client ID and certificate.
4. After the virtual machine has an identity, use the service principal information to grant the virtual machine access to Azure resources. To call Azure Resource Manager, use role-based access control in Azure Active Directory to assign the appropriate role to the virtual machine service principal. To call Key Vault, grant your code access to the specific secret or key in Key Vault.
5. Your code that's running on the virtual machine can request a token from the Azure Instance Metadata service endpoint, accessible only from within the virtual machine: <http://169.254.169.254/metadata/identity/oauth2/token>
6. A call is made to Azure Active Directory to request an access token (as specified in step 5) by using the client ID and certificate configured in step 3. Azure Active Directory returns a JSON Web Token (JWT) access token.
7. Your code sends the access token on a call to a service that supports Azure Active Directory authentication.

How a user-assigned managed identity works with an Azure virtual machine

1. Azure Resource Manager receives a request to create a user-assigned managed identity.
2. Azure Resource Manager creates a service principal in Azure Active Directory for the user-assigned managed identity. The service principal is created in the Azure Active Directory tenant that's trusted by the subscription.
3. Azure Resource Manager receives a request to configure the user-assigned managed identity on a virtual machine and updates the Azure Instance Metadata Service identity endpoint with the user-assigned managed identity service principal client ID and certificate.
4. After the user-assigned managed identity is created, use the service principal information to grant the identity access to Azure resources. To call Azure Resource Manager, use role-based access control in Azure Active Directory to assign the appropriate role to the service principal of the user-assigned identity. To call Key Vault, grant your code access to the specific secret or key in Key Vault.

Note: You can also do this step before step 3.

5. Your code that's running on the virtual machine can request a token from the Azure Instance Metadata Service identity endpoint, accessible only from within the virtual machine:
<http://169.254.169.254/metadata/identity/oauth2/token>
6. A call is made to Azure Active Directory to request an access token (as specified in step 5) by using the client ID and certificate configured in step 3. Azure Active Directory returns a JSON Web Token (JWT) access token.
7. Your code sends the access token on a call to a service that supports Azure Active Directory authentication.

Configure managed identities

You can configure an Azure virtual machine with a managed identity during, or after, the creation of the virtual machine. Below are some CLI examples showing the commands for both system- and user-assigned identities.

System-assigned managed identity

To create, or enable, an Azure virtual machine with the system-assigned managed identity your account needs the **Virtual Machine Contributor** role assignment. No additional Azure AD directory role assignments are required.

Enable system-assigned managed identity during creation of an Azure virtual machine

The following example creates a virtual machine named *myVM* with a system-assigned managed identity, as requested by the `--assign-identity` parameter. The `--admin-username` and `--admin-password` parameters specify the administrative user name and password account for virtual machine sign-in. Update these values as appropriate for your environment:

```
az vm create --resource-group myResourceGroup \
    --name myVM --image win2016datacenter \
    --generate-ssh-keys \
    --assign-identity \
    --admin-username azureuser \
    --admin-password myPassword12
```

Enable system-assigned managed identity on an existing Azure virtual machine

Use `az vm identity assign` command enable the system-assigned identity to an existing virtual machine:

```
az vm identity assign -g myResourceGroup -n myVm
```

User-assigned managed identity

To assign a user-assigned identity to a virtual machine during its creation, your account needs the **Virtual Machine Contributor** and **Managed Identity Operator** role assignments. No additional Azure AD directory role assignments are required.

Enabling user-assigned managed identities is a two-step process:

1. Create the user-assigned identity
2. Assign the identity to a virtual machine

Create a user-assigned identity

Create a user-assigned managed identity using `az identity create`. The `-g` parameter specifies the resource group where the user-assigned managed identity is created, and the `-n` parameter specifies its name.

```
az identity create -g myResourceGroup -n myUserAssignedIdentity
```

Assign a user-assigned managed identity during the creation of an Azure virtual machine

The following example creates a virtual machine associated with the new user-assigned identity, as specified by the `--assign-identity` parameter.

```
az vm create \
--resource-group <RESOURCE GROUP> \
--name <VM NAME> \
--image UbuntuLTS \
--admin-username <USER NAME> \
--admin-password <PASSWORD> \
--assign-identity <USER ASSIGNED IDENTITY NAME>
```

Assign a user-assigned managed identity to an existing Azure virtual machine

Assign the user-assigned identity to your virtual machine using `az vm identity assign`.

```
az vm identity assign \
-g <RESOURCE GROUP> \
-n <VM NAME> \
--identities <USER ASSIGNED IDENTITY>
```

Azure SDKs with managed identities for Azure resources support

Azure supports multiple programming platforms through a series of **Azure SDKs⁷**. Several of them have been updated to support managed identities for Azure resources, and provide corresponding samples to demonstrate usage.

SDK	Sample
.NET	Manage resource from a virtual machine enabled with managed identities for Azure resources enabled (https://github.com/Azure-Samples/aad-dotnet-manage-resources-from-vm-with-msi)
Java	Manage storage from a virtual machine enabled with managed identities for Azure resources (https://github.com/Azure-Samples/compute-java-manage-resources-from-vm-with-msi-in-aad-group)
Node.js	Create a virtual machine with system-assigned managed identity enabled (https://azure.microsoft.com/resources/samples/compute-node-msi-vm/)
Python	Create a virtual machine with system-assigned managed identity enabled (https://azure.microsoft.com/resources/samples/compute-python-msi-vm/)
Ruby	Create Azure virtual machine with an system-assigned identity enabled (https://github.com/Azure-Samples/compute-ruby-msi-vm/)

Acquire an access token

A client application can request managed identities for Azure resources app-only access token for accessing a given resource. The token is based on the managed identities for Azure resources service principal. As such, there is no need for the client to register itself to obtain an access token under its own service principal. The token is suitable for use as a bearer token in service-to-service calls requiring client credentials.

This unit provides various code and script examples for token acquisition, as well as guidance on important topics such as handling token expiration and HTTP errors.

Important: All sample code/script in this unit assumes the client is running on a virtual machine with managed identities for Azure resources.

Acquire a token

The fundamental interface for acquiring an access token is based on REST, making it accessible to any client application running on the VM that can make HTTP REST calls. This is similar to the Azure AD

⁷ <https://azure.microsoft.com/downloads>

programming model, except the client uses an endpoint on the virtual machine versus an Azure AD endpoint.

Sample request using the Azure Instance Metadata Service (IMDS) endpoint:

```
GET 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https://management.azure.com/' HTTP/1.1 Metadata: true
```

Element	Description
GET	The HTTP verb, indicating you want to retrieve data from the endpoint. In this case, an OAuth access token.
http://169.254.169.254/metadata/identity/oauth2/token	The managed identities for Azure resources endpoint for the Instance Metadata Service.
api-version	A query string parameter, indicating the API version for the IMDS endpoint. Please use API version 2018-02-01 or greater.
resource	A query string parameter, indicating the App ID URI of the target resource. It also appears in the aud (audience) claim of the issued token. This example requests a token to access Azure Resource Manager, which has an App ID URI of https://management.azure.com/.
Metadata	An HTTP request header field, required by managed identities for Azure resources as a mitigation against Server Side Request Forgery (SSRF) attack. This value must be set to "true", in all lower case.

Sample response:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "access_token": "eyJ0eXAi...",
    "refresh_token": "",
    "expires_in": "3599",
    "expires_on": "1506484173",
    "not_before": "1506480273",
    "resource": "https://management.azure.com/",
    "token_type": "Bearer"
}
```

Get a token by using C#

The code sample below builds the request to acquire a token, calls the endpoint, and then extracts the token from the response.

```
using System;
using System.Collections.Generic;
using System.IO;
```

```
using System.Net;
using System.Web.Script.Serialization;

// Build request to acquire managed identities for Azure resources token
HttpWebRequest request = (HttpWebRequest)WebRequest.Create("http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https://management.azure.com/");
request.Headers["Metadata"] = "true";
request.Method = "GET";

try
{
    // Call /token endpoint
    HttpWebResponse response = (HttpWebResponse)request.GetResponse();

    // Pipe response Stream to a StreamReader, and extract access token
    StreamReader streamResponse = new StreamReader(response.GetResponseStream());
    string stringResponse = streamResponse.ReadToEnd();
    JavaScriptSerializer j = new JavaScriptSerializer();
    Dictionary<string, string> list = (Dictionary<string, string>) j.Deserialize(stringResponse, typeof(Dictionary<string, string>));
    string accessToken = list["access_token"];
}
catch (Exception e)
{
    string errorText = String.Format("{0} \n\n{1}", e.Message, e.InnerException != null ? e.InnerException.Message : "Acquire token failed");
}
```

Token caching

While the managed identities for Azure resources subsystem does cache tokens, we also recommend to implement token caching in your code. As a result, you should prepare for scenarios where the resource indicates that the token is expired.

On-the-wire calls to Azure Active Directory result only when:

- Cache miss occurs due to no token in the managed identities for Azure resources subsystem cache.
- The cached token is expired.

Retry guidance

It is recommended to retry if you receive a 404, 429, or 5xx error code.

Throttling limits apply to the number of calls made to the IMDS endpoint. When the throttling threshold is exceeded, IMDS endpoint limits any further requests while the throttle is in effect. During this period, the IMDS endpoint will return the HTTP status code 429 ("Too many requests"), and the requests fail.

Knowledge check

Multiple choice

Which of the following characteristics is indicative of user-assigned identities?

- Shared lifecycle with an Azure resource
- Independent life-cycle
- Can only be associated with a single Azure resource

Multiple choice

A client app requests managed identities for an access token for a given resource. Which of the below is the basis for the token?

- Oauth 2.0
- Service principal
- Virtual machine

Summary

In this module, you learned how to:

- Explain the differences between the two types of managed identities
- Describe the flows for user- and system-assigned managed identities
- Configure managed identities
- Acquire access tokens by using REST and code

Implement Azure App Configuration

Introduction

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

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 access 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
- Complete data encryptions, at rest or 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

Programming language and framework	How to connect
.NET Framework and ASP.NET	App Configuration builder for .NET
Java Spring	App Configuration client for Spring Cloud
Others	App Configuration REST API

Create paired keys and values

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 a number of 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.

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

- Based on component services

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

- Based on deployment regions

```
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:

```
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

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 additional 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:

```
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:

```
bool featureFlag = true;
```

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

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

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

```
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:

```
"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

In this unit you will 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.

Private endpoints for App Configuration

When creating a private endpoint, you must specify the App Configuration store to which it connects. If you have multiple App Configuration stores, you need a separate private endpoint for each store. Azure relies upon DNS resolution to route connections from the VNet to the configuration store over a private link. You can quickly find connection strings in the Azure portal by selecting your App Configuration store, then selecting **Settings > Access Keys**.

DNS changes for private endpoints

When you create a private endpoint, the DNS CNAME resource record for the configuration store is updated to an alias in a subdomain with the prefix `privatelink`. Azure also creates a **private DNS zone**⁸ corresponding to the `privatelink` subdomain, with the DNS A resource records for the private endpoints.

When you resolve the endpoint URL from within the VNet hosting the private endpoint, it resolves to the private endpoint of the store. When resolved from outside the VNet, the endpoint URL resolves to the public endpoint. When you create a private endpoint, the public endpoint is disabled.

Managed identities

A managed identity from Azure Active Directory (AAD) 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 does not require you to provision or rotate any secrets.

⁸ <https://docs.microsoft.com/azure/dns/private-dns-overview>

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`.

```
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 assigns it to an Azure App Configuration store named `myTestAppConfigStore`.

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

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

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

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

Knowledge check

Multiple choice

Which type of encryption does Azure App Configuration use to encrypt data at rest?

- 64-bit AES
- 128-bit AES
- 256-bit AES

Multiple choice

Which of the below evaluates the state of a feature flag?

- Feature flag
- Feature manager
- Filter

Summary

In this module, you learned how 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

Answers

Multiple choice

Which of the below methods of authenticating to Azure Key Vault is recommended for most scenarios?

- Service principal and certificate
- Service principal and secret
- Managed identities

Explanation

That's correct. The benefit of this approach is that Azure automatically rotates the identity.

Multiple choice

Azure Key Vault protects data when it is traveling between Azure Key Vault and clients. What protocol does it use for encryption?

- Secure Sockets Layer
- Transport Layer Security
- Presentation Layer

Explanation

That's correct. Azure Key Vault enforces Transport Layer Security protocol to protect data when it's traveling between Azure Key Vault and clients.

Multiple choice

Which of the following characteristics is indicative of user-assigned identities?

- Shared lifecycle with an Azure resource
- Independent life-cycle
- Can only be associated with a single Azure resource

Explanation

That's correct. User-assigned identities exist independently from the resources they are associated with and must be explicitly deleted.

Multiple choice

A client app requests managed identities for an access token for a given resource. Which of the below is the basis for the token?

- OAuth 2.0
- Service principal
- Virtual machine

Explanation

That's correct. The token is based on the managed identities for Azure resources service principal.

Multiple choice

Which type of encryption does Azure App Configuration use to encrypt data at rest?

- 64-bit AES
- 128-bit AES
- 256-bit AES

Explanation

That's correct. Azure App Configuration encrypts sensitive information at rest using a 256-bit AES encryption key provided by Microsoft.

Multiple choice

Which of the below evaluates the state of a feature flag?

- Feature flag
- Feature manager
- Filter

Explanation

That's correct. 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.

Module 8 Implement API Management

Explore API Management

Introduction

API Management helps organizations publish APIs to external, partner, and internal developers to unlock the potential of their data and services.

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

- Describe the components, and their function, of the API Management service.
- Explain how API gateways can help manage calls to your APIs.
- Secure access to APIs by using subscriptions and certificates.
- Create a backend API.

Discover the API Management service

API Management provides the core functionality to ensure a successful API program through developer engagement, business insights, analytics, security, and protection. Each API consists of one or more operations, and each API can be added to one or more products. To use an API, developers subscribe to a product that contains that API, and then they can call the API's operation, subject to any usage policies that may be in effect.

The system is made up of the following components:

- The **API gateway** is the endpoint that:
 - Accepts API calls and routes them to your backend(s).
 - Verifies API keys, JWT tokens, certificates, and other credentials.
 - Enforces usage quotas and rate limits.
 - Transforms your API on the fly without code modifications.
 - Caches backend responses where set up.
 - Logs call metadata for analytics purposes.

- The **Azure portal** is the administrative interface where you set up your API program. Use it to:
 - Define or import API schema.
 - Package APIs into products.
 - Set up policies like quotas or transformations on the APIs.
 - Get insights from analytics.
 - Manage users.
- The **Developer portal** serves as the main web presence for developers, where they can:
 - Read API documentation.
 - Try out an API via the interactive console.
 - Create an account and subscribe to get API keys.
 - Access analytics on their own usage.

Products

Products are how APIs are surfaced to developers. Products in API Management have one or more APIs, and are configured with a title, description, and terms of use. Products can be **Open** or **Protected**. Protected products must be subscribed to before they can be used, while open products can be used without a subscription. Subscription approval is configured at the product level and can either require administrator approval, or be auto-approved.

Groups

Groups are used to manage the visibility of products to developers. API Management has the following immutable system groups:

- **Administrators** - Azure subscription administrators are members of this group. Administrators manage API Management service instances, creating the APIs, operations, and products that are used by developers.
- **Developers** - Authenticated developer portal users fall into this group. Developers are the customers that build applications using your APIs. Developers are granted access to the developer portal and build applications that call the operations of an API.
- **Guests** - Unauthenticated developer portal users, such as prospective customers visiting the developer portal of an API Management instance fall into this group. They can be granted certain read-only access, such as the ability to view APIs but not call them.

In addition to these system groups, administrators can create custom groups or leverage external groups in associated Azure Active Directory tenants.

Developers

Developers represent the user accounts in an API Management service instance. Developers can be created or invited to join by administrators, or they can sign up from the Developer portal. Each developer is a member of one or more groups, and can subscribe to the products that grant visibility to those groups.

Policies

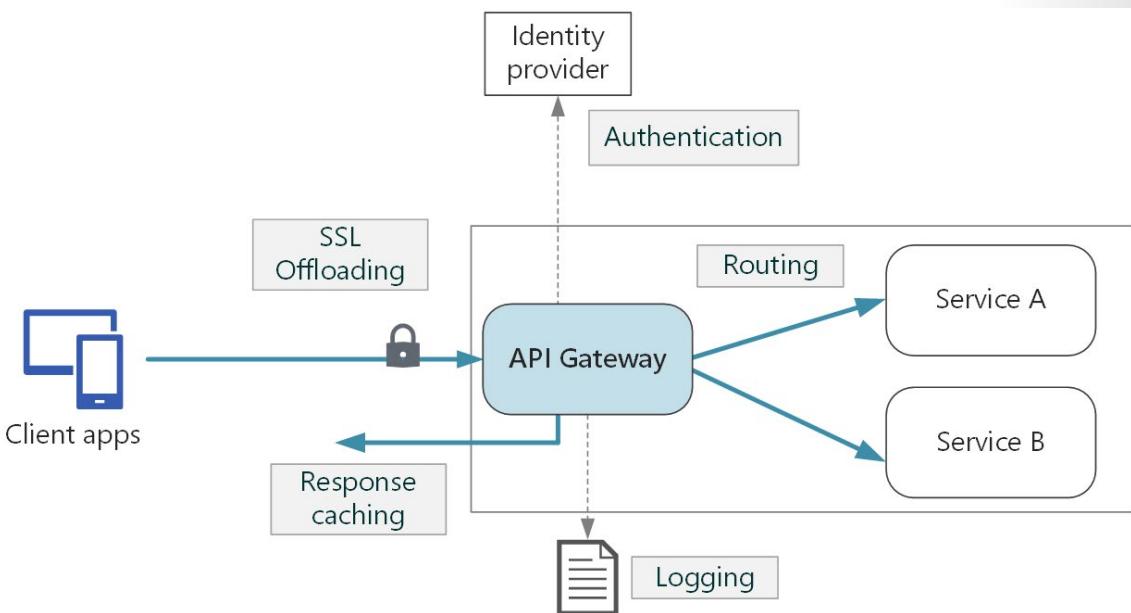
Policies are a powerful capability of API Management that allow the Azure portal to change the behavior of the API through configuration. Policies are a collection of statements that are executed sequentially on the request or response of an API. Popular statements include format conversion from XML to JSON and call rate limiting to restrict the number of incoming calls from a developer, and many other policies are available.

Developer portal

The developer portal is where developers can learn about your APIs, view and call operations, and subscribe to products. Prospective customers can visit the developer portal, view APIs and operations, and sign up. The URL for your developer portal is located on the dashboard in the Azure portal for your API Management service instance.

Explore API gateways

Your solution may contain several front- and back-end services. In this scenario, how does a client know what endpoints to call? What happens when new services are introduced, or existing services are refactored? How do services handle SSL termination, authentication, and other concerns? An *API gateway* can help to address these challenges.



An API gateway sits between clients and services. It acts as a reverse proxy, routing requests from clients to services. It may also perform various cross-cutting tasks such as authentication, SSL termination, and rate limiting. If you don't deploy a gateway, clients must send requests directly to front-end services. However, there are some potential problems with exposing services directly to clients:

- It can result in complex client code. The client must keep track of multiple endpoints, and handle failures in a resilient way.
- It creates coupling between the client and the backend. The client needs to know how the individual services are decomposed. That makes it harder to maintain the client and also harder to refactor services.

- A single operation might require calls to multiple services. That can result in multiple network round trips between the client and the server, adding significant latency.
- Each public-facing service must handle concerns such as authentication, SSL, and client rate limiting.
- Services must expose a client-friendly protocol such as HTTP or WebSocket. This limits the choice of communication protocols.
- Services with public endpoints are a potential attack surface, and must be hardened.

A gateway helps to address these issues by decoupling clients from services. Gateways can perform a number of different functions, and you may not need all of them. The functions can be grouped into the following design patterns:

- **Gateway routing:** Use the gateway as a reverse proxy to route requests to one or more backend services, using layer 7 routing. The gateway provides a single endpoint for clients, and helps to decouple clients from services.
- **Gateway aggregation:** Use the gateway to aggregate multiple individual requests into a single request. This pattern applies when a single operation requires calls to multiple backend services. The client sends one request to the gateway. The gateway dispatches requests to the various backend services, and then aggregates the results and sends them back to the client. This helps to reduce chattiness between the client and the backend.
- **Gateway Offloading:** Use the gateway to offload functionality from individual services to the gateway, particularly cross-cutting concerns. It can be useful to consolidate these functions into one place, rather than making every service responsible for implementing them.

Here are some examples of functionality that could be offloaded to a gateway:

- SSL termination
- Authentication
- IP allow/block list
- Client rate limiting (throttling)
- Logging and monitoring
- Response caching
- GZIP compression
- Servicing static content

Explore API Management policies

In Azure API Management, policies are a powerful capability of the system that allow the publisher to change the behavior of the API through configuration. Policies are a collection of Statements that are executed sequentially on the request or response of an API.

Policies are applied inside the gateway which sits between the API consumer and the managed API. The gateway receives all requests and usually forwards them unaltered to the underlying API. However a policy can apply changes to both the inbound request and outbound response. Policy expressions can be used as attribute values or text values in any of the API Management policies, unless the policy specifies otherwise.

Understanding policy configuration

The policy definition is a simple XML document that describes a sequence of inbound and outbound statements. The XML can be edited directly in the definition window.

The configuration is divided into `inbound`, `backend`, `outbound`, and `on-error`. The series of specified policy statements is executes in order for a request and a response.

```
<policies>
  <inbound>
    <!-- statements to be applied to the request go here -->
  </inbound>
  <backend>
    <!-- statements to be applied before the request is forwarded to
        the backend service go here -->
  </backend>
  <outbound>
    <!-- statements to be applied to the response go here -->
  </outbound>
  <on-error>
    <!-- statements to be applied if there is an error condition go here
-->
  </on-error>
</policies>
```

If there is an error during the processing of a request, any remaining steps in the `inbound`, `backend`, or `outbound` sections are skipped and execution jumps to the statements in the `on-error` section. By placing policy statements in the `on-error` section you can review the error by using the `contextLastError` property, inspect and customize the error response using the `set-body` policy, and configure what happens if an error occurs.

Examples

Apply policies specified at different scopes

If you have a policy at the global level and a policy configured for an API, then whenever that particular API is used both policies will be applied. API Management allows for deterministic ordering of combined policy statements via the `base` element.

```
<policies>
  <inbound>
    <cross-domain />
    <base />
    <find-and-replace from="xyz" to="abc" />
  </inbound>
</policies>
```

In the example policy definition above, the `cross-domain` statement would execute before any higher policies which would in turn, be followed by the `find-and-replace` policy.

Filter response content

The policy defined in example below demonstrates how to filter data elements from the response payload based on the product associated with the request.

The snippet assumes that response content is formatted as JSON and contains root-level properties named "minutely", "hourly", "daily", "flags".

```
<policies>
    <inbound>
        <base />
    </inbound>
    <backend>
        <base />
    </backend>
    <outbound>
        <base />
        <choose>
            <when condition="@((context.Response.StatusCode == 200 && context.Product.Name.Equals("Starter")))">
                <!-- NOTE that we are not using preserveContent=true when deserializing response body stream into a JSON object since we don't intend to access it again. See details on https://docs.microsoft.com/azure/api-management/api-management-transformation-policies#SetBody --&gt;
                &lt;set-body&gt;
                    @{
                        var response = context.Response.Body.As&lt;JObject&gt;();
                        foreach (var key in new [] {"minutely", "hourly", "daily",
"flags"}) {
                            response.Property (key).Remove ();
                        }
                        return response.ToString ();
                    }
                &lt;/set-body&gt;
            &lt;/when&gt;
        &lt;/choose&gt;
    &lt;/outbound&gt;
    &lt;on-error&gt;
        &lt;base /&gt;
    &lt;/on-error&gt;
&lt;/policies&gt;</pre>
```

Create advanced policies

This unit provides a reference for the following API Management policies:

- Control flow - Conditionally applies policy statements based on the results of the evaluation of Boolean expressions.
- Forward request - Forwards the request to the backend service.
- Limit concurrency - Prevents enclosed policies from executing by more than the specified number of requests at a time.

- Log to Event Hub - Sends messages in the specified format to an Event Hub defined by a Logger entity.
- Mock response - Aborts pipeline execution and returns a mocked response directly to the caller.
- Retry - Retries execution of the enclosed policy statements, if and until the condition is met. Execution will repeat at the specified time intervals and up to the specified retry count.

Control flow

The choose policy applies enclosed policy statements based on the outcome of evaluation of boolean expressions, similar to an if-then-else or a switch construct in a programming language.

```
<choose>
    <when condition="Boolean expression | Boolean constant">
        <!-- one or more policy statements to be applied if the above condition is true -->
    </when>
    <when condition="Boolean expression | Boolean constant">
        <!-- one or more policy statements to be applied if the above condition is true -->
    </when>
    <otherwise>
        <!-- one or more policy statements to be applied if none of the above conditions are true -->
    </otherwise>
</choose>
```

The control flow policy must contain at least one `<when/>` element. The `<otherwise/>` element is optional. Conditions in `<when/>` elements are evaluated in order of their appearance within the policy. Policy statement(s) enclosed within the first `<when/>` element with condition attribute equals true will be applied. Policies enclosed within the `<otherwise/>` element, if present, will be applied if all of the `<when/>` element condition attributes are false.

Forward request

The forward-request policy forwards the incoming request to the backend service specified in the request context. The backend service URL is specified in the API settings and can be changed using the set backend service policy.

Removing this policy results in the request not being forwarded to the backend service and the policies in the outbound section are evaluated immediately upon the successful completion of the policies in the inbound section.

```
<forward-request timeout="time in seconds" follow-redirects="true | false"/>
```

Limit concurrency

The limit-concurrency policy prevents enclosed policies from executing by more than the specified number of requests at any time. Upon exceeding that number, new requests will fail immediately with a 429 Too Many Requests status code.

```
<limit-concurrency key="expression" max-count="number">
    <!-- nested policy statements -->
</limit-concurrency>
```

Log to Event Hub

The log-to-eventhub policy sends messages in the specified format to an Event Hub defined by a Logger entity. As its name implies, the policy is used for saving selected request or response context information for online or offline analysis.

```
<log-to-eventhub logger-id="id of the logger entity" partition-id="index of
the partition where messages are sent" partition-key="value used for parti-
tion assignment">
    Expression returning a string to be logged
</log-to-eventhub>
```

Mock response

The mock-response, as the name implies, is used to mock APIs and operations. It aborts normal pipeline execution and returns a mocked response to the caller. The policy always tries to return responses of highest fidelity. It prefers response content examples, whenever available. It generates sample responses from schemas, when schemas are provided and examples are not. If neither examples or schemas are found, responses with no content are returned.

```
<mock-response status-code="code" content-type="media type"/>
```

Retry

The retry policy executes its child policies once and then retries their execution until the retry condition becomes false or retry count is exhausted.

```
<retry>
    condition="boolean expression or literal"
    count="number of retry attempts"
    interval="retry interval in seconds"
    max-interval="maximum retry interval in seconds"
    delta="retry interval delta in seconds"
    first-fast-retry="boolean expression or literal">
        <!-- One or more child policies. No restrictions -->
</retry>
```

Return response

The return-response policy aborts pipeline execution and returns either a default or custom response to the caller. Default response is 200 OK with no body. Custom response can be specified via a context variable or policy statements. When both are provided, the response contained within the context variable is modified by the policy statements before being returned to the caller.

```
<return-response response-variable-name="existing context variable">
  <set-header/>
  <set-body/>
  <set-status/>
</return-response>
```

Additional resources

- Visit **API Management policies**¹ for more policy examples.
- **Error handling in API Management policies**²

Secure APIs by using subscriptions

When you publish APIs through API Management, it's easy and common to secure access to those APIs by using subscription keys. Developers who need to consume the published APIs must include a valid subscription key in HTTP requests when they make calls to those APIs. Otherwise, the calls are rejected immediately by the API Management gateway. They aren't forwarded to the back-end services.

To get a subscription key for accessing APIs, a subscription is required. A subscription is essentially a named container for a pair of subscription keys. Developers who need to consume the published APIs can get subscriptions. And they don't need approval from API publishers. API publishers can also create subscriptions directly for API consumers.

Note: API Management also supports other mechanisms for securing access to APIs, including: OAuth2.0, Client certificates, and IP allow listing.

Subscriptions and Keys

A subscription key is a unique auto-generated key that can be passed through in the headers of the client request or as a query string parameter. The key is directly related to a subscription, which can be scoped to different areas. Subscriptions give you granular control over permissions and policies.

The three main subscription scopes are:

Scope	Details
All APIs	Applies to every API accessible from the gateway
Single API	This scope applies to a single imported API and all of its endpoints
Product	A product is a collection of one or more APIs that you configure in API Management. You can assign APIs to more than one product. Products can have different access rules, usage quotas, and terms of use.

Applications that call a protected API must include the key in every request.

You can regenerate these subscription keys at any time, for example, if you suspect that a key has been shared with unauthorized users.

¹ <https://docs.microsoft.com/azure/api-management/api-management-policies>

² <https://docs.microsoft.com/azure/api-management/api-management-error-handling-policies>

DISPLAY NAME	PRIMARY KEY	SECONDARY KEY	SCOPE	STATE	OWNER	ALLOW TRACING
Product: Starter	*****	*****	Product: Starter	Active	Administrator	✓
Product: Unlimited	*****	*****	Product: Unlimited	Active	Administrator	✓
Built-in all-access su...	*****	*****	Service	Active	Administrator	✓
Unlimited	*****	*****	Product: Unlimited	Active	Administrator	✓
	*****	*****	Product: NorthWind...	Active	Administrator	✓

Every subscription has two keys, a primary and a secondary. Having two keys makes it easier when you do need to regenerate a key. For example, if you want to change the primary key and avoid downtime, use the secondary key in your apps.

For products where subscriptions are enabled, clients must supply a key when making calls to APIs in that product. Developers can obtain a key by submitting a subscription request. If you approve the request, you must send them the subscription key securely, for example, in an encrypted message. This step is a core part of the API Management workflow.

Call an API with the subscription key

Applications must include a valid key in all HTTP requests when they make calls to API endpoints that are protected by a subscription. Keys can be passed in the request header, or as a query string in the URL.

The default header name is **Ocp-Apim-Subscription-Key**, and the default query string is **subscription-key**.

To test out your API calls, you can use the developer portal, or command-line tools, such as **curl**. Here's an example of a **GET** request using the developer portal, which shows the subscription key header:

The screenshot shows the NorthWind Shoes API documentation. At the top, there's a navigation bar with links for HOME, APIS, PRODUCTS, APPLICATIONS, and ISSUES. Below the navigation, there's a search bar and a sidebar with several API endpoints listed under the 'Products' category. One endpoint is highlighted with a blue background: 'GET Retrieve the details of every product sold'. To the right of this endpoint, there's a summary: 'Retrieve the details of every product sold' with a 'Try it' button. Further down, there's a 'Request' section with a 'Request URL' field containing 'https://apim-northwindshoes.az.../api/Products'. A 'Request headers' section is shown with a red box around the 'Ocp-Apim-Subscription-Key' header, which is described as a string type that provides access to the API. Below the headers, there's a 'Request body' section and a 'Responses' section indicating a '200 OK' response with 'Success' status.

Here's how you can pass a key in the request header using **curl**:

```
curl --header "Ocp-Apim-Subscription-Key: <key string>" https://<apim gateway>.azure-api.net/api/path
```

Here's an example **curl** command that passes a key in the URL as a query string:

```
curl https://<apim gateway>.azure-api.net/api/path?subscription-key=<key string>
```

If the key is not passed in the header, or as a query string in the URL, you'll get a **401 Access Denied** response from the API gateway.

Secure APIs by using certificates

Certificates can be used to provide Transport Layer Security (TLS) mutual authentication between the client and the API gateway. You can configure the API Management gateway to allow only requests with certificates containing a specific thumbprint. The authorization at the gateway level is handled through inbound policies.

Transport Layer Security client authentication

With TLS client authentication, the API Management gateway can inspect the certificate contained within the client request and check for properties like:

Property	Description
Certificate Authority (CA)	Only allow certificates signed by a particular CA
Thumbprint	Allow certificates containing a specified thumbprint
Subject	Only allow certificates with a specified subject
Expiration Date	Only allow certificates that have not expired

These properties are not mutually exclusive and they can be mixed together to form your own policy requirements. For instance, you can specify that the certificate passed in the request is signed by a certain certificate authority and hasn't expired.

Client certificates are signed to ensure that they are not tampered with. When a partner sends you a certificate, verify that it comes from them and not an imposter. There are two common ways to verify a certificate:

- Check who issued the certificate. If the issuer was a certificate authority that you trust, you can use the certificate. You can configure the trusted certificate authorities in the Azure portal to automate this process.
- If the certificate is issued by the partner, verify that it came from them. For example, if they deliver the certificate in person, you can be sure of its authenticity. These are known as self-signed certificates.

Accepting client certificates in the Consumption tier

The Consumption tier in API Management is designed to conform with serverless design principals. If you build your APIs from serverless technologies, such as Azure Functions, this tier is a good fit. In the Consumption tier, you must explicitly enable the use of client certificates, which you can do on the **Custom domains** page. This step is not necessary in other tiers.

The screenshot shows the 'Custom domains' blade for an API Management service named 'apim-WeatherData'. On the left, there's a navigation sidebar with 'Subscriptions', 'Security' (OAuth 2.0, OpenID Connect, Client certificates), and 'Settings' (Properties). The main area has a search bar and buttons for '+ Add', 'Save', 'Discard', and 'Columns'. Under 'Client certificates', a dropdown menu shows 'Request client certificate' set to 'Yes'. Below this is a table for 'Custom domains' with columns: ENDPOINT, HOSTNAME, CERTIFICATE, and CERTIFICATE KEY VAULT ID. A single row is listed: 'Gateway' with 'apim-weatherdata.azure-api.net' in the HOSTNAME column and an empty field in the others.

Certificate Authorization Policies

Create these policies in the inbound processing policy file within the API Management gateway:

The screenshot shows the Azure API Management - APIs interface. On the left, there's a sidebar with various options like Quickstart, APIs, Products, Named values, Tags, Analytics, Users, Subscriptions, Groups, Notifications, Notification templates, and Issues. The 'APIs' option is highlighted with a red box. In the main area, there's a search bar and a 'Search APIs' section with filters for tags and group by tag. Below that is a 'Design' tab with sub-sections for 'Frontend' and 'Inbound processing'. The 'Frontend' section shows an 'Echo API' and a 'Weather Data' operation. The 'Inbound processing' section shows a 'Policies' section with a 'base' policy and a '+ Add policy' button. A red box highlights the 'Policies' section.

Check the thumbprint of a client certificate

Every client certificate includes a thumbprint, which is a hash, calculated from other certificate properties. The thumbprint ensures that the values in the certificate have not been altered since the certificate was issued by the certificate authority. You can check the thumbprint in your policy. The following example checks the thumbprint of the certificate passed in the request:

```
<choose>
    <when condition="@ (context.Request.Certificate == null || context.
Request.Certificate.Thumbprint != "desired-thumbprint")" >
        <return-response>
            <set-status code="403" reason="Invalid client certificate" />
        </return-response>
    </when>
</choose>
```

Check the thumbprint against certificates uploaded to API Management

In the previous example, only one thumbprint would work so only one certificate would be validated. Usually, each customer or partner company would pass a different certificate with a different thumbprint. To support this scenario, obtain the certificates from your partners and use the **Client certificates** page in the Azure portal to upload them to the API Management resource. Then add this code to your policy:

```
<choose>
    <when condition="@ (context.Request.Certificate == null || !context.
Request.Certificate.Verify() || !context.Deployment.Certificates.Any(c =>
c.Value.Thumbprint == context.Request.Certificate.Thumbprint))" >
        <return-response>
            <set-status code="403" reason="Invalid client certificate" />
        </return-response>
    </when>
</choose>
```

Check the issuer and subject of a client certificate

This example checks the issuer and subject of the certificate passed in the request:

```
<choose>
    <when condition="@{context.Request.Certificate == null || context.
Request.Certificate.Issuer != "trusted-issuer" || context.Request.Certifi-
cate.SubjectName.Name != "expected-subject-name")" >
        <return-response>
            <set-status code="403" reason="Invalid client certificate" />
        </return-response>
    </when>
</choose>
```

Exercise: Create a backend API

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

- Create an API Management (APIM) instance
- Import an API
- Configure the backend settings
- Test the API

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>.

Login to Azure

1. Login to the **Azure portal**³ and open the Cloud Shell.



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



Create an API Management instance

1. Let's set some variables for the CLI commands to use to reduce the amount of retyping. Replace <myLocation> with a region that makes sense for you. The APIM name needs to be a globally unique name, and the script below generates a random string. Replace <myEmail> with an email address you can access.

```
myApiName=az204-apim-$RANDOM
myLocation=<myLocation>
```

³ <https://portal.azure.com>

```
myEmail=<myEmail>
```

2. Create a resource group. The commands below will create a resource group named *az204-apim-rg*.

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

3. Create an APIM instance. The `az apim create` command is used to create the instance. The `--sku-name Consumption` option is used to speed up the process for the walkthrough.

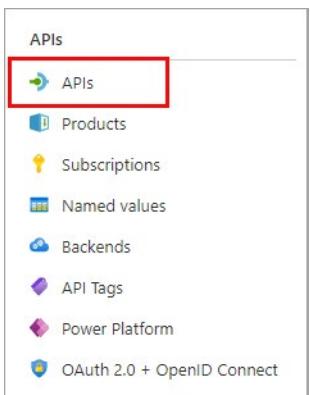
```
az apim create -n $myApiName \
--location $myLocation \
--publisher-email $myEmail \
--resource-group az204-apim-rg \
--publisher-name AZ204-APIM-Exercise \
--sku-name Consumption
```

Note: The operation should complete in about five minutes.

Import a backend API

This section shows how to import and publish an OpenAPI specification backend API.

1. In the Azure portal, search for and select **API Management services**.
2. On the **API Management** screen, select the API Management instance you created.
3. Select **APIs** in the **API management service** navigation pane.



4. Select **OpenAPI** from the list and select **Full** in the pop-up.

Create from OpenAPI specification

Basic Full

* OpenAPI specification or Select a file (maximum size 4 MiB)

* Display name

* Name

Description

URL scheme HTTP HTTPS Both

API URL suffix

Base URL

Tags

Products

! To publish the API, you must associate it with a product. [Learn more.](#)

Version this API?

Create **Cancel**

Use the values from the table below to fill out the form. You can leave any fields not mentioned their default value.

Setting	Value	Description
OpenAPI Specification	<code>https://conferenceapi.azurewebsites.net?format=json</code>	References the service implementing the API, requests are forwarded to this address. Most of the necessary information in the form is automatically populated after you enter this.
Display name	<i>Demo Conference API</i>	This name is displayed in the Developer portal.
Name	<i>demo-conference-api</i>	Provides a unique name for the API.
Description	Automatically populated	Provide an optional description of the API.
API URL suffix	<i>conference</i>	The suffix is appended to the base URL for the API management service. API Management distinguishes APIs by their suffix and therefore the suffix must be unique for every API for a given publisher.

5. Select **Create**.

Configure the backend settings

The *Demo Conference API* is created and a backend needs to be specified.

1. Select **Settings** in the blade to the right and enter `https://conferenceapi.azurewebsites.net/` in the **Web service URL** field.
2. Deselect the **Subscription required** checkbox.

REVISION 1 CREATED Oct 21, 2021, 11:27:06 AM

Design Settings Test Revisions Change log

General

* Display name Demo Conference API
* Name demo-conference-api
Description A sample API with information related to a technical conference. The available resources include *Speakers*, *Sessions* and *Topics*. A single write operation is available to provide feedback on a session.
Web service URL `https://conferenceapi.azurewebsites.net/`
URL scheme HTTP HTTPS Both
API URL suffix conference
Base URL `http(s)://az204-apim-19520.azure-api.net/conference`
Tags e.g. Booking
Products No products selected

Subscription

Subscription required
Header name Ocp-Apim-Subscription-Key
Query parameter name subscription-key

Save Discard

3. Select **Save**.

Test the API

Now that the API has been imported and the backend configured it is time to test the API.

1. Select **Test**.

The screenshot shows a top navigation bar with 'REVISION 1' and 'CREATED Oct 19, 2021, 3:23:42 PM'. Below it are tabs: 'Design', 'Settings', 'Test' (which is highlighted with a red box), 'Revisions', and 'Change log'. A search bar says 'Search operations' and a dropdown says 'Please select an operation'. The main area is empty.

2. Select **GetSpeakers**. The page shows **Query parameters** and **Headers**, if any. The Ocp-Apim-Subscription-Key is filled in automatically for the subscription key associated with this API.
3. Select **Send**.

Backend responds with **200 OK** and some data.

Clean up Azure resources

When you are finished with the resources you created in this exercise you can use the command below to delete the resource group and all related resources.

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

Knowledge check

Multiple choice

Which of the following components of the API Management service would a developer use if they need to create an account and subscribe to get API keys?

- API gateway
- Azure portal
- Developer portal

Multiple choice

Which of the following API Management policies would one use if one wants to apply a policy based on a condition?

- forward-request
- choose
- return-response

Summary

In this module, you learned how to:

- Describe the components, and their function, of the API Management service.
- Explain how API gateways can help manage calls to your APIs.
- Secure access to APIs by using subscriptions and certificates.
- Create a backend API.

Answers

Multiple choice

Which of the following components of the API Management service would a developer use if they need to create an account and subscribe to get API keys?

- API gateway
- Azure portal
- Developer portal

Explanation

That's correct. The Developer portal serves as the main web presence for developers, and is where they can subscribe to get API keys.

Multiple choice

Which of the following API Management policies would one use if one wants to apply a policy based on a condition?

- forward-request
- choose
- return-response

Explanation

That's correct. The choose policy applies enclosed policy statements based on the outcome of evaluation of boolean expressions.

Module 9 Develop event-based solutions

Explore Azure Event Grid

Introduction

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.

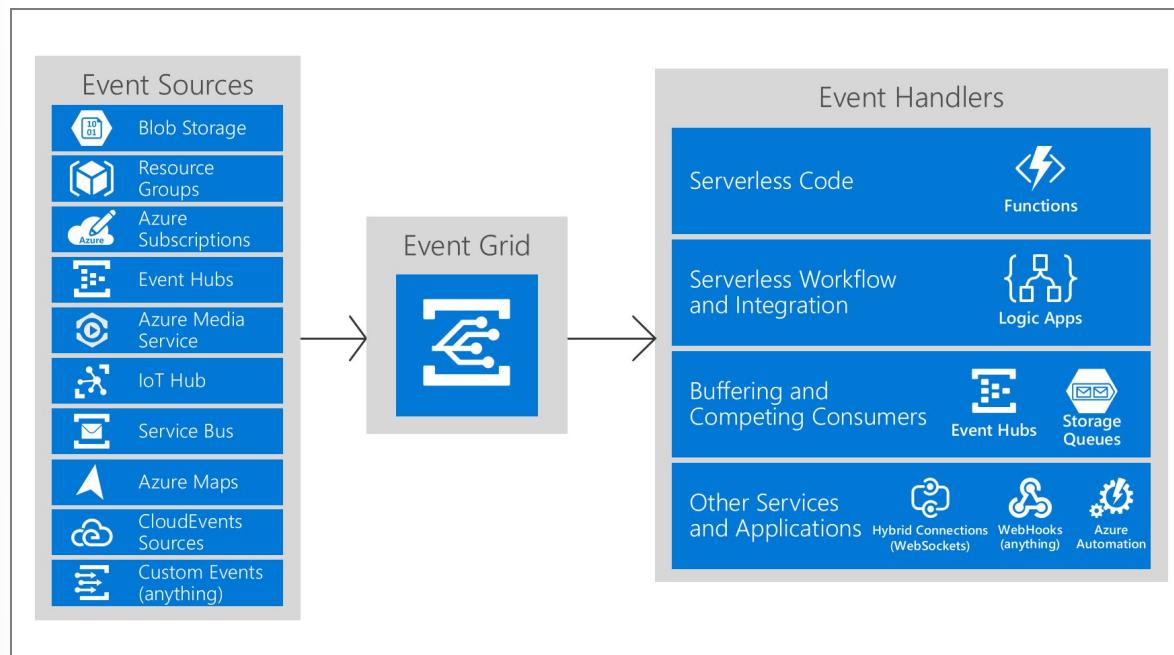
Explore Azure Event Grid

Azure Event Grid is an eventing backplane that enables event-driven, reactive programming. It uses the publish-subscribe model. 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. First, select the Azure resource you would like to subscribe to, and then give the event handler or WebHook endpoint to send the event to. 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.

This image below 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, and are described in more detail below:

- **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

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 will incur operations charges as though they were multiple events. For example, an event that is 130 KB would incur operations as though it were 3 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 store¹](#).

¹ <https://github.com/Azure/azure-rest-api-specs/tree/master/specification/eventgrid/data-plane>

Event schema

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

```
[  
  {  
    "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 will stamp onto the event. If included it must match the event grid topic Azure Resource Manager ID exactly.	Full resource path to the event source. This field isn't writeable. Event Grid provides this value.
subject	string	Yes	Publisher-defined path to the event subject.
eventType	string	Yes	One of the registered event types for this event source.
eventTime	string	Yes	The time the event is generated based on the provider's UTC time.
id	string	Yes	Unique identifier for the event.
data	object	No	Event data specific to the resource provider.
dataVersion	string	No. If not included it will be stamped with an empty value.	The schema version of the data object. The publisher defines the schema version.

Property	Type	Required	Description
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 event metadata. Event Grid defines the schema of the top-level properties. Event Grid provides this value.

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.

CloudEvents v1.0 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:

```
{
  "specversion": "1.0",
  "type": "Microsoft.Storage.BlobCreated",
  "source": "/subscriptions/{subscription-id}/resourceGroups/{re-
  source-group}/providers/Microsoft.Storage/storageAccounts/{storage-ac-
  count}",
  "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",
    "blobType": "Block"
  }
}
```

```
        "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": "0000000000000000000000000000000099240000000000c41c18",
        "storageDiagnostics": {
            "batchId": "681fe319-3006-00a8-0022-9e7cde000000"
        }
    }
}
```

A detailed description of the available fields, their types, and definitions in CloudEvents v1.0 is **available here²**.

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

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 is 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, EventGrid 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), EventGrid will either perform dead-lettering on 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

² <https://github.com/cloudevents/spec/blob/v1.0/spec.md#required-attributes>

Endpoint Type	Error codes
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 will attempt 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 will be 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 example below shows setting the maximum number of attempts to 18 by using the Azure CLI.

```
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 will deliver per batch. This number will never 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 will begin 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 will assume 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 certain 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 is a five-minute delay between the last attempt to deliver an event and when it is 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 (4K) 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

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 format of the resource is: /subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/{resource-provider}/{resource-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 resource is: /subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/Microsoft.EventGrid/topics/{topic-name}

Receive events by using webhooks

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 three Azure services listed below, 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**³ or **IFTTT**⁴), 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'll 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

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

- Event types
- Subject begins with or ends with
- Advanced fields and operators

³ <https://zapier.com/>

⁴ <https://ifttt.com/>

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:

```
"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:

```
"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:

```
"filter": {  
    "advancedFilters": [  
        {  
            "operatorType": "NumberGreaterThanOrEquals",  
            "key": "Data.Key1",  
            "value": 5  
        },  
        {  
            "operatorType": "StringContains",  
            "key": "Subject",  
            "value": "testcontainer"  
        }  
    ]  
}
```

```
        "values": ["container1", "container2"]
    }
]
}
```

Exercise: Route custom events to web endpoint by using Azure CLI

In this exercise you will 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 will open your terminal and create some variables that will be 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 commands below to create the variables. Replace <myLocation> with a region near you.

```
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 will be creating.

```
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.

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

It can take a few minutes for the registration to complete. To check the status run the command below.

```
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 is part of the DNS entry.

```
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 script below uses a pre-built 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.

```
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 script above 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 script below will grab the needed subscription ID from your account and use in the creation of the event subscription.

```
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.

```
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.

```
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.

```
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. 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.

```
{
  "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.

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

Knowledge check

Multiple choice

Which of the following event schema properties requires a value?

- Topic
- Data
- Subject

Multiple choice

Which of the following Event Grid built-in roles is appropriate for managing Event Grid resources?

- Event Grid Contributor
- Event Grid Subscription Contributor
- Event Grid Data Sender

Summary

In this module, you learned how 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 Hubs

Introduction

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

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 table below 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 overhead, so you focus on your business solutions. Event Hubs for Apache Kafka ecosystems gives you the PaaS Kafka experience without having to 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 concurrently and letting you control the speed of 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 to talk to Event Hubs. You do not need to set up, configure, and manage your own Kafka clusters.

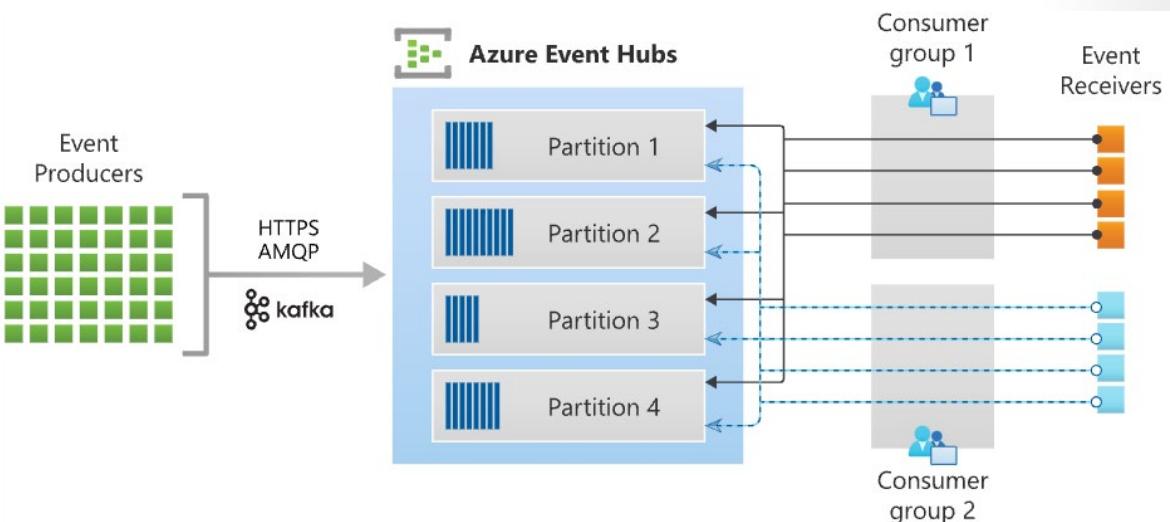
Key concepts

Event Hubs contains the following key components:

- An **Event Hub client** is the primary interface for developers interacting with the Event Hubs client library. There are several different Event Hub clients, each dedicated to a specific use of Event Hubs, such as publishing or consuming events.

- An **Event Hub 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 Hub consumer** is a type of client which reads information from the Event Hub 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 Hub consumers are often robust and high-scale platform infrastructure parts with built-in analytics capabilities, like Azure Stream Analytics, Apache Spark, or Apache Storm.
- A **partition** is an ordered sequence of events that is held in an Event Hub. 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 are added to the end of this sequence. The number of partitions is specified at the time an Event Hub is created and cannot be changed.
- A **consumer group** is a view of an entire Event Hub. 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 5 concurrent readers on a partition per consumer group; however it is recommended that there is 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 will receive duplicate events.
- **Event receivers:** Any entity that reads event data from an event hub. 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:** Pre-purchased units of capacity that control the throughput capacity of Event Hubs.

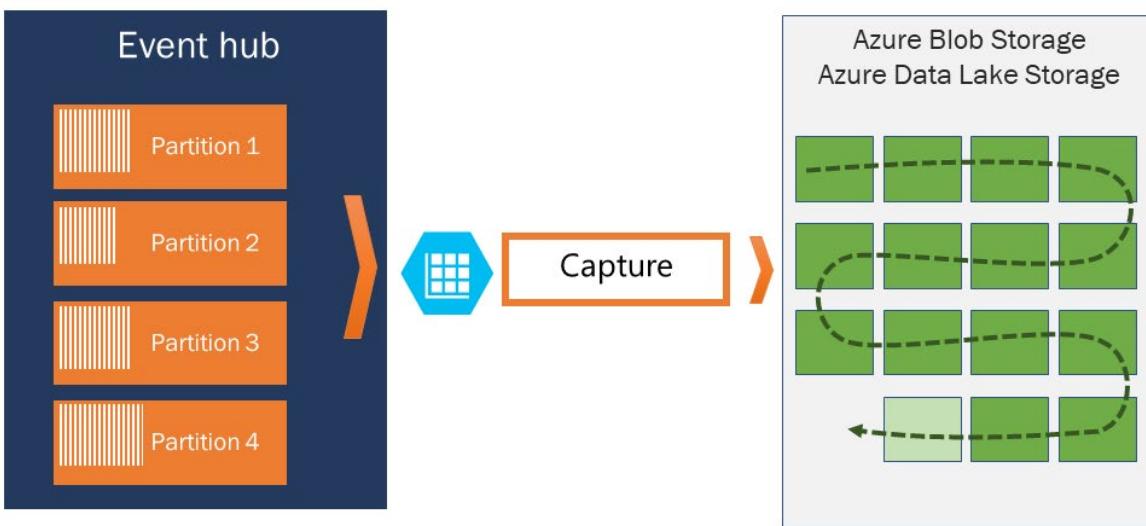
The following figure shows the Event Hubs stream processing architecture:



Explore Event Hubs Capture

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:

```
{Namespace} / {EventHub} / {PartitionId} / {Year} / {Month} / {Day} / {Hour} / {Minute} /  
{Second}
```

Note that the date values are padded with zeroes; an example filename might be:

[https://mystorageaccount.blob.core.windows.net/mycontainer/mynamespace/
myeventhub/0/2017/12/08/03/03/17.avro](https://mystorageaccount.blob.core.windows.net/mycontainer/mynamespace/myeventhub/0/2017/12/08/03/03/17.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 is no data. This process provides a predictable cadence and marker that can feed your batch processors.

Scale your processing application

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 as well as 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 will 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

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 Owner⁵:** Use this role to give *complete access* to Event Hubs resources.
- **Azure Event Hubs Data Sender⁶:** Use this role to give *send access* to Event Hubs resources.
- **Azure Event Hubs Data Receiver⁷:** 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 Hub. 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 which 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.

⁵ <https://docs.microsoft.com/azure/role-based-access-control/built-in-roles#azure-event-hubs-data-owner>

⁶ <https://docs.microsoft.com/azure/role-based-access-control/built-in-roles#azure-event-hubs-data-sender>

⁷ <https://docs.microsoft.com/azure/role-based-access-control/built-in-roles#azure-event-hubs-data-receiver>

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 will be applied to the consumer groups of that entity.

Perform common operations with the Event Hubs client library

This unit contains examples of common operations you can perform with the Event Hubs client library (`Azure.Messaging.EventHubs`) to interact with an Event Hub.

Inspect an Event Hub

Many Event Hub operations take place within the scope of a specific partition. Because partitions are owned by the Event Hub, their names are assigned at the time of creation. To understand what partitions are available, you query the Event Hub using one of the Event Hub clients. For illustration, the `EventHubProducerClient` is demonstrated in these examples, but the concept and form are common across clients.

```
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 an Event Hub

In order to publish events, you'll 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 will take advantage of automatic routing.

```
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 Hub

In order to read events from an Event Hub, you'll need to create an `EventHubConsumerClient` for a given consumer group. When an Event Hub is created, it provides a default consumer group that can be used to get started with exploring Event Hubs. In our example, we will focus on reading all events that have been published to the Event Hub 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**⁸, as it provides a more robust and performant experience.

```
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.ReadEventsAsyn-
c(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 Hub partition

To read from a specific partition, the consumer will need to specify where in the event stream to begin receiving events; in our example, we will focus on reading all published events for the first partition of the Event Hub.

⁸ <https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/eventhub/Azure.Messaging.EventHubs.Processor>

```
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.ReadEvents-
FromPartitionAsync(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 is 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'll need to provide a `BlobContainerClient` for the processor, which has been configured for the storage account and container that should be used.

```
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(ProcessEventArgs eventArgs) => Task.Complet-
edTask;
```

```
var storageClient = new BlobContainerClient(storageConnectionString, blob-
ContainerName);
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;
}
```

Knowledge check

Multiple choice

Which of the following Event Hubs concepts represents an ordered sequence of events that is held in an Event Hub?

- Consumer group
- Partition
- Event Hub producer

Multiple choice

Which of the following represents when an event processor marks or commits the position of the last successfully processed event within a partition?

- Checkpointing
- Scale
- Load balance

Summary

In this module, you learned how 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.

Answers

Multiple choice

Which of the following event schema properties requires a value?

- Topic
- Data
- Subject

Explanation

That's correct. The subject property specifies the publisher-defined path to the event subject and is required.

Multiple choice

Which of the following Event Grid built-in roles is appropriate for managing Event Grid resources?

- Event Grid Contributor
- Event Grid Subscription Contributor
- Event Grid Data Sender

Explanation

That's correct. The Event Grid Contributor role has permissions to manage resources.

Multiple choice

Which of the following Event Hubs concepts represents an ordered sequence of events that is held in an Event Hub?

- Consumer group
- Partition
- Event Hub producer

Explanation

That's correct. A partition is an ordered sequence of events that is held in an Event Hub.

Multiple choice

Which of the following represents when an event processor marks or commits the position of the last successfully processed event within a partition?

- Checkpointing
- Scale
- Load balance

Explanation

That's correct. Checkpointing is a process by which an event processor marks or commits the position of the last successfully processed event within a partition.

Module 10 Develop message-based solutions

Discover Azure message queues

Introduction

Azure supports two types of queue mechanisms: **Service Bus queues** and **Storage queues**.

Service Bus queues are part of a broader Azure messaging infrastructure that supports queuing, publish/subscribe, and more advanced integration patterns. They're designed to integrate applications or application components that may span multiple communication protocols, data contracts, trust domains, or network environments.

Storage queues are part of the Azure Storage infrastructure. They allow you to store large numbers of messages. You access messages from anywhere in the world via authenticated calls using HTTP or HTTPS. A queue message can be up to 64 KB in size. A queue may contain millions of messages, up to the total capacity limit of a storage account. Queues are commonly used to create a backlog of work to process asynchronously.

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

- Choose the appropriate queue mechanism for your solution.
- Explain how the messaging entities that form the core capabilities of Service Bus operate.
- Send and receive message from a Service Bus queue by using .NET.
- Identify the key components of Azure Queue Storage
- Create queues and manage messages in Azure Queue Storage by using .NET.

Choose a message queue solution

Storage queues and Service Bus queues have a slightly different feature set. You can choose either one or both, depending on the needs of your particular solution.

When determining which queuing technology fits the purpose of a given solution, solution architects and developers should consider these recommendations.

Consider using Service Bus queues

As a solution architect/developer, **you should consider using Service Bus queues** when:

- Your solution needs to receive messages without having to poll the queue. With Service Bus, you can achieve it by using a long-polling receive operation using the TCP-based protocols that Service Bus supports.
- Your solution requires the queue to provide a guaranteed first-in-first-out (FIFO) ordered delivery.
- Your solution needs to support automatic duplicate detection.
- You want your application to process messages as parallel long-running streams (messages are associated with a stream using the **session ID** property on the message). In this model, each node in the consuming application competes for streams, as opposed to messages. When a stream is given to a consuming node, the node can examine the state of the application stream state using transactions.
- Your solution requires transactional behavior and atomicity when sending or receiving multiple messages from a queue.
- Your application handles messages that can exceed 64 KB but won't likely approach the 256-KB limit.

Consider using Storage queues

As a solution architect/developer, **you should consider using Storage queues** when:

- Your application must store over 80 gigabytes of messages in a queue.
- Your application wants to track progress for processing a message in the queue. It's useful if the worker processing a message crashes. Another worker can then use that information to continue from where the prior worker left off.
- You require server side logs of all of the transactions executed against your queues.

Explore Azure Service Bus

Microsoft Azure Service Bus is a fully managed enterprise integration message broker. Service Bus can decouple applications and services. Data is transferred between different applications and services using **messages**. A message is a container decorated with metadata, and contains data. The data can be any kind of information, including structured data encoded with the common formats such as the following ones: JSON, XML, Apache Avro, Plain Text.

Some common messaging scenarios are:

- *Messaging*. Transfer business data, such as sales or purchase orders, journals, or inventory movements.
- *Decouple applications*. Improve reliability and scalability of applications and services. Client and service don't have to be online at the same time.
- *Topics and subscriptions*. Enable 1:n relationships between publishers and subscribers.
- *Message sessions*. Implement workflows that require message ordering or message deferral.

Service Bus tiers

Service Bus offers a standard and premium tier. The *premium* tier of Service Bus Messaging addresses common customer requests around scale, performance, and availability for mission-critical applications. The premium tier is recommended for production scenarios. Although the feature sets are nearly identical, these two tiers of Service Bus Messaging are designed to serve different use cases.

Some high-level differences are highlighted in the following table.

Premium	Standard
High throughput	Variable throughput
Predictable performance	Variable latency
Fixed pricing	Pay as you go variable pricing
Ability to scale workload up and down	N/A
Message size up to 1 MB. Support for message payloads up to 100 MB currently exists in preview.	Message size up to 256 KB

Advanced features

Service Bus includes advanced features that enable you to solve more complex messaging problems. The following table describes several of these features.

Feature	Description
Message sessions	To create a first-in, first-out (FIFO) guarantee in Service Bus, use sessions. Message sessions enable exclusive, ordered handling of unbounded sequences of related messages.
Autoforwarding	The autoforwarding feature chains a queue or subscription to another queue or topic that is in the same namespace.
Dead-letter queue	Service Bus supports a dead-letter queue (DLQ). A DLQ holds messages that can't be delivered to any receiver. Service Bus lets you remove messages from the DLQ and inspect them.
Scheduled delivery	You can submit messages to a queue or topic for delayed processing. You can schedule a job to become available for processing by a system at a certain time.
Message deferral	A queue or subscription client can defer retrieval of a message until a later time. The message remains in the queue or subscription, but it's set aside.
Batching	Client-side batching enables a queue or topic client to delay sending a message for a certain period of time.
Transactions	A transaction groups two or more operations together into an <i>execution scope</i> . Service Bus supports grouping operations against a single messaging entity within the scope of a single transaction. A message entity can be a queue, topic, or subscription.
Filtering and actions	Subscribers can define which messages they want to receive from a topic. These messages are specified in the form of one or more named subscription rules.

Feature	Description
Autodelete on idle	Autodelete on idle enables you to specify an idle interval after which a queue is automatically deleted. The minimum duration is 5 minutes.
Duplicate detection	An error could cause the client to have a doubt about the outcome of a send operation. Duplicate detection enables the sender to resend the same message, or for the queue or topic to discard any duplicate copies.
Security protocols	Service Bus supports security protocols such as Shared Access Signatures (SAS), Role Based Access Control (RBAC) and Managed identities for Azure resources.
Geo-disaster recovery	When Azure regions or datacenters experience downtime, Geo-disaster recovery enables data processing to continue operating in a different region or datacenter.
Security	Service Bus supports standard AMQP 1.0 and HTTP/REST protocols.

Compliance with standards and protocols

The primary wire protocol for Service Bus is **Advanced Messaging Queueing Protocol (AMQP) 1.0**¹, an open ISO/IEC standard. It allows customers to write applications that work against Service Bus and on-premises brokers such as ActiveMQ or RabbitMQ. The **AMQP protocol guide**² provides detailed information in case you want to build such an abstraction.

Service Bus Premium is fully compliant with the Java/Jakarta EE **Java Message Service (JMS) 2.0**³ API.

Client libraries

Fully supported Service Bus client libraries are available via the Azure SDK.

- **Azure Service Bus for .NET**⁴
- **Azure Service Bus libraries for Java**⁵
- **Azure Service Bus provider for Java JMS 2.0**⁶
- **Azure Service Bus Modules for JavaScript and TypeScript**⁷
- **Azure Service Bus libraries for Python**⁸

¹ <https://docs.microsoft.com/azure/service-bus-messaging/service-bus-amqp-overview>

² <https://docs.microsoft.com/azure/service-bus-messaging/service-bus-amqp-protocol-guide>

³ <https://docs.microsoft.com/azure/service-bus-messaging/how-to-use-java-message-service-20>

⁴ <https://docs.microsoft.com/dotnet/api/overview/azure/service-bus>

⁵ <https://docs.microsoft.com/java/api/overview/azure/servicebus>

⁶ <https://docs.microsoft.com/azure/service-bus-messaging/how-to-use-java-message-service-20>

⁷ <https://docs.microsoft.com/javascript/api/overview/azure/service-bus>

⁸ <https://docs.microsoft.com/python/api/overview/azure/servicebus>

Discover Service Bus queues, topics, and subscriptions

The messaging entities that form the core of the messaging capabilities in Service Bus are **queues**, **topics** and **subscriptions**, and rules/actions.

Queues

Queues offer **First In, First Out** (FIFO) message delivery to one or more competing consumers. That is, receivers typically receive and process messages in the order in which they were added to the queue. And, only one message consumer receives and processes each message. Because messages are stored durably in the queue producers (senders) and consumers (receivers) don't have to process messages concurrently.

A related benefit is **load-leveling**, which enables producers and consumers to send and receive messages at different rates. In many applications, the system load varies over time. However, the processing time required for each unit of work is typically constant. Intermediating message producers and consumers with a queue means that the consuming application only has to be able to handle average load instead of peak load.

Using queues to intermediate between message producers and consumers provides an inherent loose coupling between the components. Because producers and consumers are not aware of each other, a consumer can be upgraded without having any effect on the producer.

You can create queues using the Azure portal, PowerShell, CLI, or Resource Manager templates. Then, send and receive messages using clients written in C#, Java, Python, and JavaScript.

Receive modes

You can specify two different modes in which Service Bus receives messages: **Receive and delete** or **Peek lock**.

Receive and delete

In this mode, when Service Bus receives the request from the consumer, it marks the message as being consumed and returns it to the consumer application. This mode is the simplest model. It works best for scenarios in which the application can tolerate not processing a message if a failure occurs. For example, consider a scenario in which the consumer issues the receive request and then crashes before processing it. As Service Bus marks the message as being consumed, the application begins consuming messages upon restart. It will miss the message that it consumed before the crash.

Peek lock

In this mode, the receive operation becomes two-stage, which makes it possible to support applications that can't tolerate missing messages.

1. Finds the next message to be consumed, **locks** it to prevent other consumers from receiving it, and then, return the message to the application.
2. After the application finishes processing the message, it requests the Service Bus service to complete the second stage of the receive process. Then, the service **marks the message as being consumed**.

If the application is unable to process the message for some reason, it can request the Service Bus service to **abandon** the message. Service Bus **unlocks** the message and makes it available to be received again, either by the same consumer or by another competing consumer. Secondly, there's a **timeout** associated with the lock. If the application fails to process the message before the lock timeout expires, Service Bus unlocks the message and makes it available to be received again.

Topics and subscriptions

A queue allows processing of a message by a single consumer. In contrast to queues, topics and subscriptions provide a one-to-many form of communication in a publish and subscribe pattern. It's useful for scaling to large numbers of recipients. Each published message is made available to each subscription registered with the topic. Publisher sends a message to a topic and one or more subscribers receive a copy of the message, depending on filter rules set on these subscriptions. The subscriptions can use additional filters to restrict the messages that they want to receive.

Publishers send messages to a topic in the same way that they send messages to a queue. But, consumers don't receive messages directly from the topic. Instead, consumers receive messages from subscriptions of the topic. A topic subscription resembles a virtual queue that receives copies of the messages that are sent to the topic. Consumers receive messages from a subscription identically to the way they receive messages from a queue.

Creating a topic is similar to creating a queue, as described in the previous section. You can create topics and subscriptions using the Azure portal, PowerShell, CLI, or Resource Manager templates. Then, send messages to a topic and receive messages from subscriptions using clients written in C#, Java, Python, and JavaScript.

Rules and actions

In many scenarios, messages that have specific characteristics must be processed in different ways. To enable this processing, you can configure subscriptions to find messages that have desired properties and then perform certain modifications to those properties. While Service Bus subscriptions see all messages sent to the topic, you can only copy a subset of those messages to the virtual subscription queue. This filtering is accomplished using subscription filters. Such modifications are called **filter actions**. When a subscription is created, you can supply a filter expression that operates on the properties of the message. The properties can be both the system properties (for example, **Label**) and custom application properties (for example, **StoreName**). The SQL filter expression is optional in this case. Without a SQL filter expression, any filter action defined on a subscription will be done on all the messages for that subscription.

Explore Service Bus message payloads and serialization

Messages carry a payload and metadata. The metadata is in the form of key-value pair properties, and describes the payload, and gives handling instructions to Service Bus and applications. Occasionally, that metadata alone is sufficient to carry the information that the sender wants to communicate to receivers, and the payload remains empty.

The object model of the official Service Bus clients for .NET and Java reflect the abstract Service Bus message structure, which is mapped to and from the wire protocols Service Bus supports.

A Service Bus message consists of a binary payload section that Service Bus never handles in any form on the service-side, and two sets of properties. The *broker properties* are predefined by the system. These

predefined properties either control message-level functionality inside the broker, or they map to common and standardized metadata items. The *user properties* are a collection of key-value pairs that can be defined and set by the application.

Message routing and correlation

A subset of the broker properties described previously, specifically `To`, `ReplyTo`, `ReplyToSessionId`, `MessageId`, `CorrelationId`, and `SessionId`, are used to help applications route messages to particular destinations. To illustrate this, consider a few patterns:

- **Simple request/reply:** A publisher sends a message into a queue and expects a reply from the message consumer. To receive the reply, the publisher owns a queue into which it expects replies to be delivered. The address of that queue is expressed in the `ReplyTo` property of the outbound message. When the consumer responds, it copies the `MessageId` of the handled message into the `CorrelationId` property of the reply message and delivers the message to the destination indicated by the `ReplyTo` property. One message can yield multiple replies, depending on the application context.
- **Multicast request/reply:** As a variation of the prior pattern, a publisher sends the message into a topic and multiple subscribers become eligible to consume the message. Each of the subscribers might respond in the fashion described previously. This pattern is used in discovery or roll-call scenarios and the respondent typically identifies itself with a user property or inside the payload. If `ReplyTo` points to a topic, such a set of discovery responses can be distributed to an audience.
- **Multiplexing:** This session feature enables multiplexing of streams of related messages through a single queue or subscription such that each session (or group) of related messages, identified by matching `SessionId` values, are routed to a specific receiver while the receiver holds the session under lock. Read more about the details of sessions [here](#).
- **Multiplexed request/reply:** This session feature enables multiplexed replies, allowing several publishers to share a reply queue. By setting `ReplyToSessionId`, the publisher can instruct the consumer(s) to copy that value into the `SessionId` property of the reply message. The publishing queue or topic does not need to be session-aware. As the message is sent, the publisher can then specifically wait for a session with the given `SessionId` to materialize on the queue by conditionally accepting a session receiver.

Routing inside of a Service Bus namespace can be realized using auto-forward chaining and topic subscription rules. Routing across namespaces can be realized using Azure LogicApps. As indicated in the previous list, the `To` property is reserved for future use and may eventually be interpreted by the broker with a specially enabled feature. Applications that wish to implement routing should do so based on user properties and not lean on the `To` property; however, doing so now will not cause compatibility issues.

Payload serialization

When in transit or stored inside of Service Bus, the payload is always an opaque, binary block. The `ContentType` property enables applications to describe the payload, with the suggested format for the property values being a MIME content-type description according to IETF RFC2045; for example, `application/json; charset=utf-8`.

Unlike the Java or .NET Standard variants, the .NET Framework version of the Service Bus API supports creating `BrokeredMessage` instances by passing arbitrary .NET objects into the constructor.

When using the legacy SBMP protocol, those objects are then serialized with the default binary serializer, or with a serializer that is externally supplied. When using the AMQP protocol, the object is serialized into an AMQP object. The receiver can retrieve those objects with the `GetBody<T>()` method, supplying the

expected type. With AMQP, the objects are serialized into an AMQP graph of `ArrayList` and `IDictionary<string, object>` objects, and any AMQP client can decode them.

While this hidden serialization magic is convenient, applications should take explicit control of object serialization and turn their object graphs into streams before including them into a message, and do the reverse on the receiver side. This yields interoperable results. It should also be noted that while AMQP has a powerful binary encoding model, it is tied to the AMQP messaging ecosystem and HTTP clients will have trouble decoding such payloads.

Exercise: Send and receive message from a Service Bus queue by using .NET.

In this exercise you will learn how to:

- Create a Service Bus namespace, and queue, using the Azure CLI.
- Create a .NET Core console application to send a set of messages to the queue.
- Create a .NET Core console application to receive those messages from the queue.

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**⁹ on one of the **supported platforms**¹⁰.
- The **C# extension**¹¹ for Visual Studio Code.

Login to Azure

In this section you will open your terminal and create some variables that will be used throughout the rest of the exercise to make command entry, and unique resource name creation, a bit easier.

1. Launch the **Azure Cloud Shell**¹² and select **Bash** and the environment.
2. Create variables used in the Azure CLI commands. Replace `<myLocation>` with a region near you.

```
myLocation=<myLocation>
myNameSpaceName=az204svcbus$RANDOM
```

Create Azure resources

1. Create a resource group to hold the Azure resources you will be creating.
`az group create --name az204-svcbus-rg --location $myLocation`
2. Create a Service Bus messaging namespace. The command below will create a namespace using the variable you created earlier. The operation will take a few minutes to complete.

⁹ <https://code.visualstudio.com/>

¹⁰ https://code.visualstudio.com/docs/supporting/requirements#_platforms

¹¹ <https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp>

¹² <https://shell.azure.com>

```
az servicebus namespace create \
--resource-group az204-svcbus-rg \
--name $myNameSpaceName \
--location $myLocation
```

3. Create a Service Bus queue

```
az servicebus queue create --resource-group az204-svcbus-rg \
--namespace-name $myNameSpaceName \
--name az204-queue
```

Retrieve the connection string for the Service Bus Namespace

1. Open the Azure portal and navigate to the **az204-svcbus-rg** resource group.
2. Select the **az204svibus** resource you just created.
3. Select **Shared access policies** in the **Settings** section, then select the **RootManageSharedAccessKey** policy.
4. Copy the **Primary Connection String** from the dialog box that opens up and save it to a file, or leave the portal open and copy the key when needed.

Create console app to send messages to the queue

1. Open a local terminal and create, and change in to, a directory named *az204svibus* and then run the command to launch Visual Studio Code.

```
code .
```

2. Open the terminal in VS Code by selecting **Terminal > New Terminal** in the menu bar and run the following commands to create the console app and add the **Azure.Messaging.ServiceBus** package.

```
dotnet new console
dotnet add package Azure.Messaging.ServiceBus
```

3. In *Program.cs*, add the following `using` statements at the top of the file after the current `using` statement.

```
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
```

4. In the *Program* class, add the following two static properties. Set the `ServiceBusConnectionString` variable to the connection string that you obtained earlier.

```
// connection string to your Service Bus namespace
static string connectionString = "<NAMESPACE CONNECTION STRING>";

// name of your Service Bus topic
static string queueName = "az204-queue";
```

5. Declare the following static properties in the `Program` class. See code comments for details.

```
// the client that owns the connection and can be used to create senders and receivers  
static ServiceBusClient client;  
  
// the sender used to publish messages to the queue  
static ServiceBusSender sender;  
  
// number of messages to be sent to the queue  
private const int numMessages = 3;
```

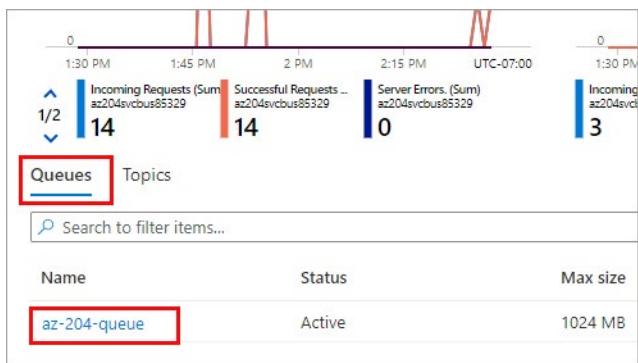
6. Replace the `Main()` method with the following `asyncMain` method.

```
static async Task Main()  
{  
    // Create the clients that we'll use for sending and processing messages.  
    client = new ServiceBusClient(connectionString);  
    sender = client.CreateSender(queueName);  
  
    // create a batch  
    using ServiceBusMessageBatch messageBatch = await sender.CreateMessageBatchAsync();  
  
    for (int i = 1; i <= 3; i++)  
    {  
        // try adding a message to the batch  
        if (!messageBatch.TryAddMessage(new ServiceBusMessage($"Message {i}")))  
        {  
            // if an exception occurs  
            throw new Exception($"Exception {i} has occurred.");  
        }  
    }  
  
    try  
    {  
        // Use the producer client to send the batch of messages to the Service Bus queue  
        await sender.SendMessagesAsync(messageBatch);  
        Console.WriteLine($"A batch of {numMessages} messages has been published to the  
queue.");  
    }  
    finally  
    {  
        // Calling DisposeAsync on client types is required to ensure that network  
        // resources and other unmanaged objects are properly cleaned up.  
        await sender.DisposeAsync();  
        await client.DisposeAsync();  
    }  
  
    Console.WriteLine("Press any key to end the application");  
    Console.ReadKey();  
}
```

7. Save the file and run the `dotnet build` command to ensure there are no errors.
8. Run the program and wait for the confirmation message.

A batch of 3 messages has been published to the queue.

9. Login to the Azure portal and navigate to your Service Bus namespace. On the Overview page, select the `az204-queue` queue in the bottom-middle pane.



Notice the following values in the Essentials section:

- The **Active** message count value for the queue is now **3**. Each time you run this sender app without retrieving the messages, this value increases by 3.
- The **current size** of the queue increments each time the app adds messages to the queue.
- In the **Messages** chart in the bottom **Metrics** section, you can see that there are three incoming messages for the queue.

Update project to receive messages to the queue

In this section you'll modify the program to receive messages from the queue.

1. In the `Program` class, delete the static properties that follow `ServiceBusClient`. We'll keep using `connectionString`, `queueName`, and `ServiceBusClient` for the rest of the exercise. Add the following after the `ServiceBusClient` static property.

```
// the processor that reads and processes messages from the queue
static ServiceBusProcessor processor;
```

2. Add the following methods to the `Program` class to handle messages and any errors.

```
// handle received messages
static async Task MessageHandler(ProcessMessageEventArgs args)
{
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Received: {body}");

    // complete the message. message is deleted from the queue.
    await args.CompleteMessageAsync(args.Message);
}

// handle any errors when receiving messages
```

```
static Task ErrorHandler(ProcessErrorEventArgs args)
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
}
```

3. Replace the `Main()` method. It calls the `ReceiveMessages` method to receive messages from the queue.

```
static async Task Main()
{
    // Create the client object that will be used to create sender and receiver objects
    client = new ServiceBusClient(connectionString);

    // create a processor that we can use to process the messages
    processor = client.CreateProcessor(queueName, new ServiceBusProcessorOptions());

    try
    {
        // add handler to process messages
        processor.ProcessMessageAsync += MessageHandler;

        // add handler to process any errors
        processor.ProcessErrorAsync += ErrorHandler;

        // start processing
        await processor.StartProcessingAsync();

        Console.WriteLine("Wait for a minute and then press any key to end the processing");
        Console.ReadKey();

        // stop processing
        Console.WriteLine("\nStopping the receiver...");
        await processor.StopProcessingAsync();
        Console.WriteLine("Stopped receiving messages");
    }
    finally
    {
        // Calling DisposeAsync on client types is required to ensure that network
        // resources and other unmanaged objects are properly cleaned up.
        await processor.DisposeAsync();
        await client.DisposeAsync();
    }
}
```

4. Use the `dotnet build` command to ensure there are no errors.
5. Use the `dotnet run` command to run the application. You should see the received messages. Press any key to stop the receiver and the application.

Wait for a minute and then press any key to end the processing
Received: Message 1

Received: Message 2

Received: Message 3

Stopping the receiver...

Stopped receiving messages

6. Check the portal again. Notice that the **Active Message Count** value is now 0. You may need to refresh the portal page.

Clean up resources

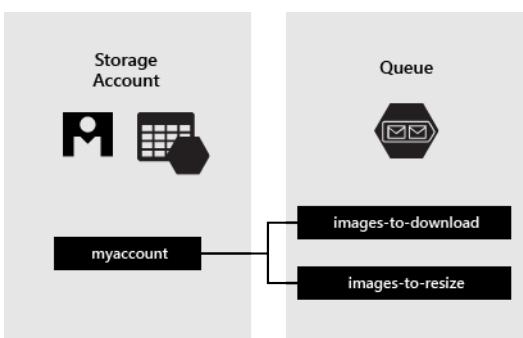
When the resources are no longer needed, you can use the `az group delete` command in the Azure Cloud Shell to remove the resource group.

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

Explore Azure Queue Storage

Azure Queue Storage is a service for storing large numbers of messages. You access messages from anywhere in the world via authenticated calls using HTTP or HTTPS. A queue message can be up to 64 KB in size. A queue may contain millions of messages, up to the total capacity limit of a storage account. Queues are commonly used to create a backlog of work to process asynchronously.

The Queue service contains the following components:



- **URL format:** Queues are addressable using the URL format `https://<storage account>.queue.core.windows.net/<queue>`. For example, the following URL addresses a queue in the diagram above `https://myaccount.queue.core.windows.net/images-to-download`
- **Storage account:** All access to Azure Storage is done through a storage account.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase.
- **Message:** A message, in any format, of up to 64 KB. For version 2017-07-29 or later, the maximum time-to-live can be any positive number, or -1 indicating that the message doesn't expire. If this parameter is omitted, the default time-to-live is seven days.

Create and manage Azure Queue Storage queues and messages by using .NET

In this unit we'll be covering how to create queues and manage messages in Azure Queue Storage by showing code snippets from a .NET project.

The code examples rely on the following NuGet packages:

- **Azure.Core library for .NET¹³:** This package provides shared primitives, abstractions, and helpers for modern .NET Azure SDK client libraries.
- **Azure.Storage.Common client library for .NET¹⁴:** This package provides infrastructure shared by the other Azure Storage client libraries.
- **Azure.Storage.Queues client library for .NET¹⁵:** This package enables working with Azure Queue Storage for storing messages that may be accessed by a client.
- **System.Configuration.ConfigurationManager library for .NET¹⁶:** This package provides access to configuration files for client applications.

Create the Queue service client

The `QueueClient` class enables you to retrieve queues stored in Queue storage. Here's one way to create the service client:

```
QueueClient queueClient = new QueueClient(connectionString, queueName);
```

Create a queue

This example shows how to create a queue if it does not already exist:

```
// Get the connection string from app settings
string connectionString = ConfigurationManager.AppSettings["StorageConnectionString"];

// Instantiate a QueueClient which will be used to create and manipulate
// the queue
QueueClient queueClient = new QueueClient(connectionString, queueName);

// Create the queue
queueClient.CreateIfNotExists();
```

Insert a message into a queue

To insert a message into an existing queue, call the `SendMessage` method. A message can be either a string (in UTF-8 format) or a byte array. The following code creates a queue (if it doesn't exist) and inserts a message:

¹³ <https://www.nuget.org/packages/azure.core/>
¹⁴ <https://www.nuget.org/packages/azure.storage.common/>
¹⁵ <https://www.nuget.org/packages/azure.storage.queues/>
¹⁶ <https://www.nuget.org/packages/system.configuration.configurationmanager/>

```
// Get the connection string from app settings
string connectionString = ConfigurationManager.AppSettings["StorageConnectionString"];
```

```
// Instantiate a QueueClient which will be used to create and manipulate
// the queue
QueueClient queueClient = new QueueClient(connectionString, queueName);
```

```
// Create the queue if it doesn't already exist
queueClient.CreateIfNotExists();
```

```
if (queueClient.Exists())
{
    // Send a message to the queue
    queueClient.SendMessage(message);
}
```

Peek at the next message

You can peek at the messages in the queue without removing them from the queue by calling the `PeekMessages` method. If you don't pass a value for the `maxMessages` parameter, the default is to peek at one message.

```
// Get the connection string from app settings
string connectionString = ConfigurationManager.AppSettings["StorageConnectionString"];
```

```
// Instantiate a QueueClient which will be used to manipulate the queue
QueueClient queueClient = new QueueClient(connectionString, queueName);
```

```
if (queueClient.Exists())
{
    // Peek at the next message
    PeakedMessage[] peekedMessage = queueClient.PeekMessages();
}
```

Change the contents of a queued message

You can change the contents of a message in-place in the queue. If the message represents a work task, you could use this feature to update the status of the work task. The following code updates the queue message with new contents, and sets the visibility timeout to extend another 60 seconds. This saves the state of work associated with the message, and gives the client another minute to continue working on the message.

```
// Get the connection string from app settings
string connectionString = ConfigurationManager.AppSettings["StorageConnectionString"];
```

```
// Instantiate a QueueClient which will be used to manipulate the queue
QueueClient queueClient = new QueueClient(connectionString, queueName);
```

```
if (queueClient.Exists())
{
    // Get the message from the queue
    QueueMessage[] message = queueClient.ReceiveMessages();

    // Update the message contents
    queueClient.UpdateMessage(message[0].MessageId,
        message[0].PopReceipt,
        "Updated contents",
        TimeSpan.FromSeconds(60.0) // Make it invisible for another 60
seconds
    );
}
```

De-queue the next message

Dequeue a message from a queue in two steps. When you call `ReceiveMessages`, you get the next message in a queue. A message returned from `ReceiveMessages` becomes invisible to any other code reading messages from this queue. By default, this message stays invisible for 30 seconds. To finish removing the message from the queue, you must also call `DeleteMessage`. This two-step process of removing a message assures that if your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls `DeleteMessage` right after the message has been processed.

```
// Get the connection string from app settings
string connectionString = ConfigurationManager.AppSettings["StorageConnectionString"];

// Instantiate a QueueClient which will be used to manipulate the queue
QueueClient queueClient = new QueueClient(connectionString, queueName);

if (queueClient.Exists())
{
    // Get the next message
    QueueMessage[] retrievedMessage = queueClient.ReceiveMessages();

    // Process (i.e. print) the message in less than 30 seconds
    Console.WriteLine($"Dequeued message: '{retrievedMessage[0].MessageText}'");

    // Delete the message
    queueClient.DeleteMessage(retrievedMessage[0].MessageId, retrievedMessage[0].PopReceipt);
}
```

Get the queue length

You can get an estimate of the number of messages in a queue. The `GetProperties` method returns queue properties including the message count. The `ApproximateMessagesCount` property contains the approximate number of messages in the queue. This number is not lower than the actual number of messages in the queue, but could be higher.

```
// Instantiate a QueueClient which will be used to manipulate the queue
QueueClient queueClient = new QueueClient(connectionString, queueName);

if (queueClient.Exists())
{
    QueueProperties properties = queueClient.GetProperties();

    // Retrieve the cached approximate message count.
    int cachedMessagesCount = properties.ApproximateMessagesCount;

    // Display number of messages.
    Console.WriteLine($"Number of messages in queue: {cachedMessagesCount}");
}
```

Delete a queue

To delete a queue and all the messages contained in it, call the `Delete` method on the queue object.

```
// Get the connection string from app settings
string connectionString = ConfigurationManager.AppSettings["StorageConnectionString"];

// Instantiate a QueueClient which will be used to manipulate the queue
QueueClient queueClient = new QueueClient(connectionString, queueName);

if (queueClient.Exists())
{
    // Delete the queue
    queueClient.Delete();
}
```

Knowledge check

Multiple choice

Which of the following advanced features of Azure Service Bus creates a first-in, first-out (FIFO) guarantee?

- Transactions
- Scheduled delivery
- Message sessions

Multiple choice

In Azure Service Bus messages are durably stored which enables a load-leveling benefit. Which of the below correctly describes the load-leveling benefit relative to a consuming application's performance?

- Performance needs to handle peak load
- Performance needs to handle average load
- Performance needs to handle low loads

Summary

In this module, you learned how to:

- Choose the appropriate queue mechanism for your solution.
- Explain how the messaging entities that form the core capabilities of Service Bus operate.
- Send and receive message from a Service Bus queue by using .NET.
- Identify the key components of Azure Queue Storage
- Create queues and manage messages in Azure Queue Storage by using .NET.

Answers

Multiple choice

Which of the following advanced features of Azure Service Bus creates a first-in, first-out (FIFO) guarantee?

- Transactions
- Scheduled delivery
- Message sessions

Explanation

That's correct. To create a first-in, first-out (FIFO) guarantee in Service Bus, use sessions. Message sessions enable joint and ordered handling of unbounded sequences of related messages.

Multiple choice

In Azure Service Bus messages are durably stored which enables a load-leveling benefit. Which of the below correctly describes the load-leveling benefit relative to a consuming application's performance?

- Performance needs to handle peak load
- Performance needs to handle average load
- Performance needs to handle low loads

Explanation

That's correct. Intermediating message producers and consumers with a queue means that the consuming application only has to be able to handle average load instead of peak load.

Module 11 Instrument solutions to support monitoring and logging

Monitor app performance

Introduction

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

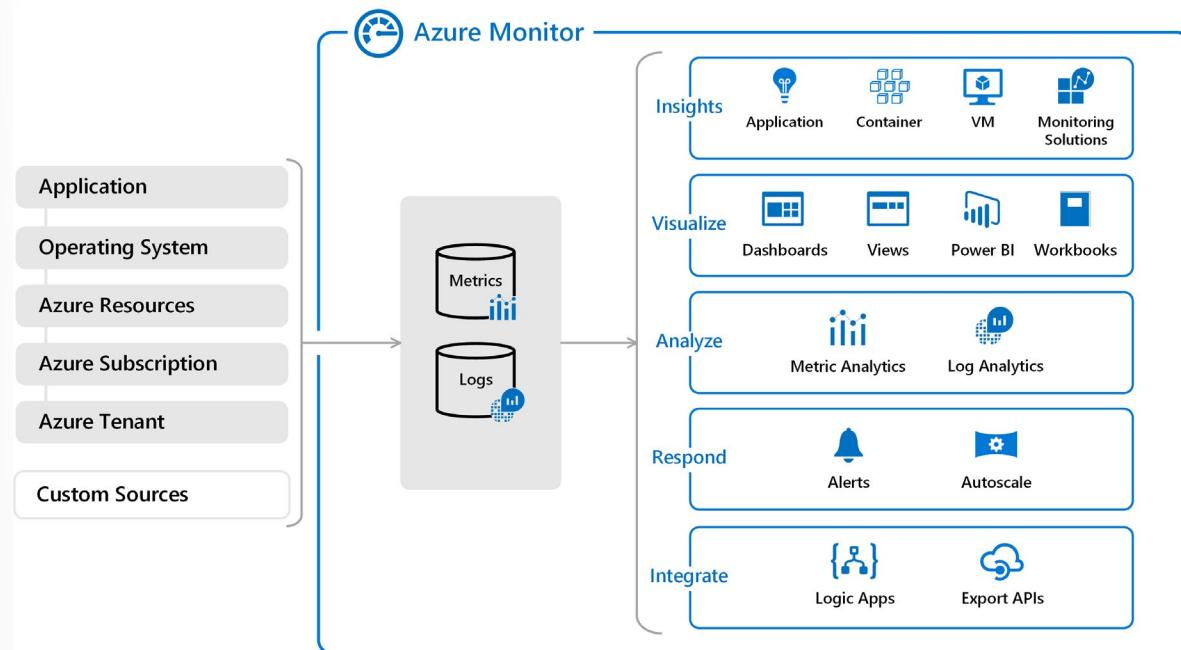
After completing this module, you'll be able to:

- Explain how Azure Monitor operates as the center of monitoring in Azure.
- 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 Azure Monitor

Azure Monitor delivers a comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments. This information helps you understand how your applications are performing and proactively identify issues affecting them and the resources they depend on.

The following diagram gives a high-level view of Azure Monitor. At the center of the diagram are the data stores for metrics and logs, which are the two fundamental types of data used by Azure Monitor. On the left are the sources of monitoring data that populate these data stores. On the right are the different functions that Azure Monitor performs with this collected data. This includes such actions as analysis, alerting, and streaming to external systems.



What data does Azure Monitor collect?

Azure Monitor can collect data from a variety of sources. This ranges from your application, any operating system and services it relies on, down to the platform itself. Azure Monitor collects data from each of the following tiers:

- **Application monitoring data:** Data about the performance and functionality of the code you have written, regardless of its platform.
- **Guest OS monitoring data:** Data about the operating system on which your application is running. This could be running in Azure, another cloud, or on-premises.
- **Azure resource monitoring data:** Data about the operation of an Azure resource. For a complete list of the resources that have metrics or logs, visit [What can you monitor with Azure Monitor?](#)¹
- **Azure subscription monitoring data:** Data about the operation and management of an Azure subscription, as well as data about the health and operation of Azure itself.
- **Azure tenant monitoring data:** Data about the operation of tenant-level Azure services, such as Azure Active Directory.

Monitoring data platform

All data collected by Azure Monitor fits into one of two fundamental types, **metrics** and **logs**. Metrics are numerical values that describe some aspect of a system at a particular point in time. They are lightweight and capable of supporting near real-time scenarios. Logs contain different kinds of data organized into records with different sets of properties for each type. Telemetry such as events and traces are stored as logs in addition to performance data so that it can all be combined for analysis.

For many Azure resources, you'll see metric data collected by Azure Monitor right in their Overview page in the Azure portal. Log data collected by Azure Monitor can be analyzed with queries to quickly retrieve,

¹ <https://docs.microsoft.com/azure/azure-monitor/monitor-reference#list-of-azure-monitor-supported-services>

consolidate, and analyze collected data. You can create and test queries using Log Analytics in the Azure portal.

Insights and curated visualizations

Monitoring data is only useful if it can increase your visibility into the operation of your computing environment. Some Azure resource providers have a "curated visualization" which gives you a customized monitoring experience for that particular service or set of services. They generally require minimal configuration. Larger scalable curated visualizations are known as "insights" and marked with that name in the documentation and Azure portal. Some examples are:

- **Application Insights:** Application Insights monitors the availability, performance, and usage of your web applications whether they're hosted in the cloud or on-premises. It leverages the powerful data analysis platform in Azure Monitor to provide you with deep insights into your application's operations. It enables you to diagnose errors without waiting for a user to report them.
- **Container Insights:** Container Insights monitors the performance of container workloads that are deployed to managed Kubernetes clusters hosted on Azure Kubernetes Service (AKS) and Azure Container Instances. It gives you performance visibility by collecting metrics from controllers, nodes, and containers that are available in Kubernetes through the Metrics API. Container logs are also collected.
- **VM Insights:** VM Insights monitors your Azure virtual machines (VM) at scale. It analyzes the performance and health of your Windows and Linux VMs and identifies their different processes and interconnected dependencies on external processes.

Explore Application Insights

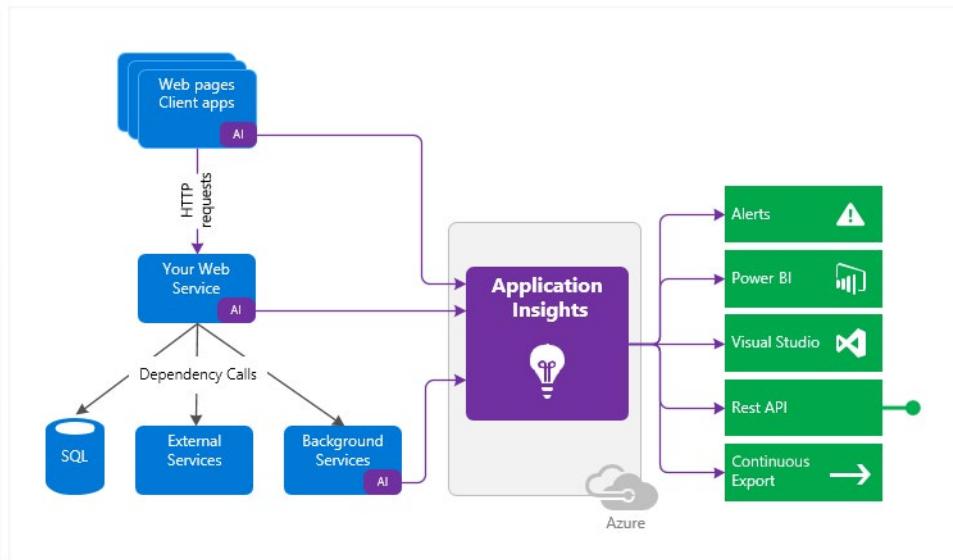
Application Insights, a feature of Azure Monitor, is an extensible Application Performance Management (APM) service for developers and DevOps professionals. Use it to monitor your live applications. It will automatically detect performance anomalies, and includes powerful analytics tools to help you diagnose issues and to understand what users actually do with your app. It's designed to help you continuously improve performance and usability.

How Application Insights works

You install a small instrumentation package (SDK) in your application or enable Application Insights using the Application Insights Agent when **supported**². The instrumentation monitors your app and directs the telemetry data to an Azure Application Insights Resource using a unique GUID that we refer to as an Instrumentation Key.

You can instrument not only the web service application, but also any background components, and the JavaScript in the web pages themselves. The application and its components can run anywhere - it doesn't have to be hosted in Azure.

² <https://docs.microsoft.com/azure/azure-monitor/app/platforms>



In addition, you can pull in telemetry from the host environments such as performance counters, Azure diagnostics, or Docker logs. You can also set up web tests that periodically send synthetic requests to your web service.

All these telemetry streams are integrated into Azure Monitor. In the Azure portal, you can apply powerful analytic and search tools to the raw data. The impact on your app's performance is small. Tracking calls are non-blocking, and are batched and sent in a separate thread.

What Application Insights monitors

Application Insights is aimed at the development team, to help you understand how your app is performing and how it's being used. It monitors:

- **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.

Use Application Insights

Application Insights is one of the many services hosted within Microsoft Azure, and telemetry is sent there for analysis and presentation. It is 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 additional 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

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 queries³** 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. This makes them 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 selector⁴** to switch between log-based and standard metrics in **metrics explorer⁵**.

Log-based metrics

Developers can use the SDK to either emit these 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 reduce 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.

³ <https://docs.microsoft.com/azure/kusto/query/>

⁴ <https://docs.microsoft.com/azure/azure-monitor/essentials/metrics-getting-started#create-your-first-metric-chart>

⁵ <https://docs.microsoft.com/azure/azure-monitor/essentials/metrics-getting-started>

Pre-aggregated metrics

The pre-aggregated metrics are not stored as individual events with lots of properties. Instead, they are stored as pre-aggregated time series, and only with key dimensions. This makes the new metrics superior at query time: retrieving data happens much faster and requires less compute power. This consequently 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 2.7⁶** SDK or later for .NET) pre-aggregate metrics during collection. This applies to **standard metrics sent by default⁷** so the accuracy isn't affected by sampling or filtering. It also applies to custom metrics sent using **GetMetric⁸** resulting in less data ingestion and lower cost.

For the SDKs that don't implement pre-aggregation (that is, older versions of Application Insights SDKs or for browser instrumentation) the Application Insights backend still populates the new metrics by aggregating the events received by the Application Insights event collection endpoint. This means that 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 is worth mentioning that the collection endpoint pre-aggregates events before ingestion sampling, which means that **ingestion sampling⁹** will never impact the accuracy of pre-aggregated metrics, regardless of the SDK version you use with your application.

Instrument an app for monitoring

Earlier in this module we highlighted that an app can be monitored in Application Insights with, or without, adding code. In this unit we'll cover that in more detail.

Auto-instrumentation

Auto-instrumentation allows you to enable application monitoring with Application Insights without changing your code.

Application Insights is integrated with various resource providers and works on different environments. In essence, all you have to do is enable and - in some cases - configure the agent, which will collect the telemetry automatically. In no time, you'll see the metrics, requests, and dependencies in your Application Insights resource, which will allow you to spot the source of potential problems before they occur, and analyze the root cause with end-to-end transaction view.

The list of services that are supported by auto-instrumentation changes rapidly, visit this **page¹⁰** for a list of what is currently supported.

⁶ <https://www.nuget.org/packages/Microsoft.ApplicationInsights/2.7.2>

⁷ <https://docs.microsoft.com/azure/azure-monitor/essentials/metrics-supported#microsoftinsightscomponents>

⁸ <https://docs.microsoft.com/azure/azure-monitor/app/api-custom-events-metrics#getmetric>

⁹ <https://docs.microsoft.com/azure/azure-monitor/app/sampling>

¹⁰ <https://docs.microsoft.com/azure/azure-monitor/app/codeless-overview#supported-environments-languages-and-resource-providers>

Instrumenting for distributed tracing

Distributed tracing is the equivalent of call stacks for modern cloud and microservices architectures, with the addition of a simplistic performance profiler thrown in. In Azure Monitor, we provide two experiences for consuming distributed trace data. The first is our transaction diagnostics view, which is like a call stack with a time dimension added in. The transaction diagnostics view provides visibility into one single transaction/request, and is helpful for finding the root cause of reliability issues and performance bottlenecks on a per request basis.

Azure Monitor also offers an application map view which aggregates many transactions to show a topological view of how the systems interact, and what the average performance and error rates are.

How to enable distributed tracing

Enabling distributed tracing across the services in an application is as simple as adding the proper SDK or library to each service, based on the language the service was implemented in.

Enabling via Application Insights SDKs

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

With the proper Application Insights SDK installed and configured, tracing information is automatically collected for popular frameworks, libraries, and technologies by SDK dependency auto-collectors. The full list of supported technologies is available in the Dependency auto-collection documentation.

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

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)**¹¹: You can create this simple test through the portal to validate whether an endpoint is responding and measure performance associated with that response. You can also set

¹¹ <https://docs.microsoft.com/azure/azure-monitor/app/monitor-web-app-availability>

custom success criteria coupled with more advanced features, like parsing dependent requests and allowing for retries.

- **Standard test (Preview)**¹²: 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**¹³: If you decide to create a custom application to run availability tests, you can use the **TrackAvailability()**¹⁴ 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](#)¹⁵ article for guidance on diagnosing availability issues.

Troubleshoot app performance by using Application Map

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 are 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 will update with the number of components in your application as they are discovered.

¹² <https://docs.microsoft.com/azure/azure-monitor/app/availability-standard-tests>

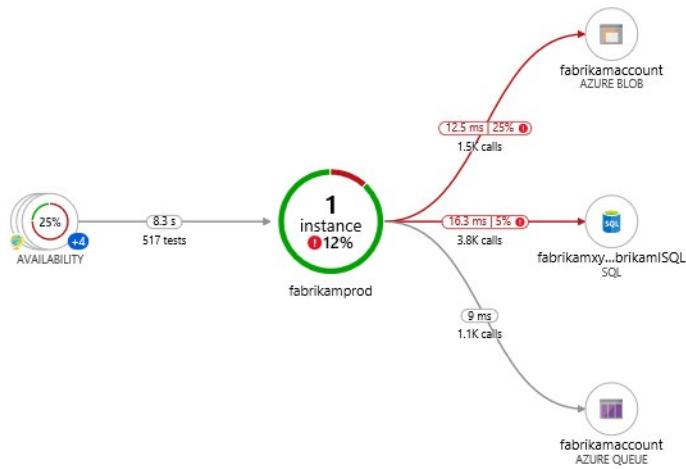
¹³ <https://docs.microsoft.com/azure/azure-monitor/app/availability-azure-functions>

¹⁴ <https://docs.microsoft.com/dotnet/api/microsoft.applicationinsights.telemetryclient.trackavailability>

¹⁵ <https://docs.microsoft.com/azure/azure-monitor/app/troubleshoot-availability>

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 is not required. The initial load for such an application will have 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.

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

NAME	DURATION (AVG)
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.

Knowledge check

Multiple choice

Which of the following availability tests is recommended for authentication tests?

- URL ping
- Standard
- Custom TrackAvailability

Multiple choice

Which of the following metric collection types below provides near real-time querying and alerting on dimensions of metrics, and more responsive dashboards?

- Log-based
- Pre-aggregated
- Azure Service Bus

Summary

In this module, you learned how to:

- Explain how Azure Monitor operates as the center of monitoring in Azure.
- 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.

Answers

Multiple choice

Which of the following availability tests is recommended for authentication tests?

- URL ping
- Standard
- Custom TrackAvailability

Explanation

That's correct. Custom TrackAvailability test is the long term supported solution for multi request or authentication test scenarios.

Multiple choice

Which of the following metric collection types below provides near real-time querying and alerting on dimensions of metrics, and more responsive dashboards?

- Log-based
- Pre-aggregated
- Azure Service Bus

Explanation

That's correct. Pre-aggregated metrics are stored as a time series and only with key dimensions which enables near real-time alerting on dimensions of metrics, more responsive dashboards.

Module 12 Integrate caching and content delivery within solutions

Develop for Azure Cache for Redis

Introduction

Caching is a common technique that aims to improve the performance and scalability of a system. It does this by temporarily copying frequently accessed data to fast storage that's located close to the application. If this fast data storage is located closer to the application than the original source, then caching can significantly improve response times for client applications by serving data more quickly.

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

- Explain the key scenarios Azure Cache for Redis covers and its service tiers
- Identify the key parameters for creating an Azure Cache for Redis instance and interact with the cache
- Connect an app to Azure Cache for Redis by using .NET Core

Explore Azure Cache for Redis

Azure Cache for Redis provides an in-memory data store based on the **Redis¹** software. Redis improves the performance and scalability of an application that uses backend data stores heavily. It's able to process large volumes of application requests by keeping frequently accessed data in the server memory, which can be written to and read from quickly. Redis brings a critical low-latency and high-throughput data storage solution to modern applications.

Azure Cache for Redis offers both the Redis open-source (OSS Redis) and a commercial product from Redis Labs (Redis Enterprise) as a managed service. It provides secure and dedicated Redis server instances and full Redis API compatibility. The service is operated by Microsoft, hosted on Azure, and usable by any application within or outside of Azure.

¹ <https://redis.io/>

Key scenarios

Azure Cache for Redis improves application performance by supporting common application architecture patterns. Some of the most common include the following patterns:

Pattern	Description
Data cache	Databases are often too large to load directly into a cache. It's common to use the cache-aside (https://docs.microsoft.com/azure/architecture/patterns/cache-aside) pattern to load data into the cache only as needed. When the system makes changes to the data, the system can also update the cache, which is then distributed to other clients.
Content cache	Many web pages are generated from templates that use static content such as headers, footers, banners. These static items shouldn't change often. Using an in-memory cache provides quick access to static content compared to backend datastores.
Session store	This pattern is commonly used with shopping carts and other user history data that a web application might associate with user cookies. Storing too much in a cookie can have a negative effect on performance as the cookie size grows and is passed and validated with every request. A typical solution uses the cookie as a key to query the data in a database. Using an in-memory cache, like Azure Cache for Redis, to associate information with a user is much faster than interacting with a full relational database.
Job and message queuing	Applications often add tasks to a queue when the operations associated with the request take time to execute. Longer running operations are queued to be processed in sequence, often by another server. This method of deferring work is called task queuing.
Distributed transactions	Applications sometimes require a series of commands against a backend data-store to execute as a single atomic operation. All commands must succeed, or all must be rolled back to the initial state. Azure Cache for Redis supports executing a batch of commands as a single transaction (https://redis.io/topics/transactions).

Service tiers

Azure Cache for Redis is available in these tiers:

Tier	Description
Basic	An OSS Redis cache running on a single VM. This tier has no service-level agreement (SLA) and is ideal for development/test and non-critical workloads.
Standard	An OSS Redis cache running on two VMs in a replicated configuration.
Premium	High-performance OSS Redis caches. This tier offers higher throughput, lower latency, better availability, and more features. Premium caches are deployed on more powerful VMs compared to the VMs for Basic or Standard caches.
Enterprise	High-performance caches powered by Redis Labs' Redis Enterprise software. This tier supports Redis modules including RediSearch, RedisBloom, and RedisTimeSeries. Also, it offers even higher availability than the Premium tier.
Enterprise Flash	Cost-effective large caches powered by Redis Labs' Redis Enterprise software. This tier extends Redis data storage to non-volatile memory, which is cheaper than DRAM, on a VM. It reduces the overall per-GB memory cost.

Configure Azure Cache for Redis

You can create a Redis cache using the Azure portal, the Azure CLI, or Azure PowerShell.

Create and configure an Azure Cache for Redis instance

There are several parameters you will need to decide in order to configure the cache properly for your purposes.

Name

The Redis cache will need a globally unique name. The name has to be unique within Azure because it is used to generate a public-facing URL to connect and communicate with the service.

The name must be between 1 and 63 characters, composed of numbers, letters, and the '-' character. The cache name can't start or end with the '-' character, and consecutive '-' characters aren't valid.

Location

You will need to decide where the Redis cache will be physically located by selecting an Azure region. You should always place your cache instance and your application in the same region. Connecting to a cache in a different region can significantly increase latency and reduce reliability. If you are connecting to the cache outside of Azure, then select a location close to where the application consuming the data is running.

Important: Put the Redis cache as close to the data consumer as you can.

Pricing tier

As mentioned in the last unit, there are three pricing tiers available for an Azure Cache for Redis.

- **Basic:** Basic cache ideal for development/testing. Is limited to a single server, 53 GB of memory, and 20,000 connections. There is no SLA for this service tier.
- **Standard:** Production cache which supports replication and includes an SLA. It supports two servers, and has the same memory/connection limits as the Basic tier.
- **Premium:** Enterprise tier which builds on the Standard tier and includes persistence, clustering, and scale-out cache support. This is the highest performing tier with up to 530 GB of memory and 40,000 simultaneous connections.

You can control the amount of cache memory available on each tier - this is selected by choosing a cache level from C0-C6 for Basic/Standard and P0-P4 for Premium. Check the [pricing page²](#) for full details.

Tip: Microsoft recommends you always use Standard or Premium Tier for production systems. The Basic Tier is a single node system with no data replication and no SLA.

The Premium tier allows you to persist data in two ways to provide disaster recovery:

- RDB persistence takes a periodic snapshot and can rebuild the cache using the snapshot in case of failure.
- AOF persistence saves every write operation to a log that is saved at least once per second. This creates bigger files than RDB but has less data loss.

There are several other settings which are only available to the **Premium** tier.

Virtual Network support

If you create a premium tier Redis cache, you can deploy it to a virtual network in the cloud. Your cache will be available to only other virtual machines and applications in the same virtual network. This provides a higher level of security when your service and cache are both hosted in Azure, or are connected through an Azure virtual network VPN.

Clustering support

With a premium tier Redis cache, you can implement clustering to automatically split your dataset among multiple nodes. To implement clustering, you specify the number of shards to a maximum of 10. The cost incurred is the cost of the original node, multiplied by the number of shards.

Accessing the Redis instance

Redis has a command-line tool for interacting with an Azure Cache for Redis as a client. The tool is available for Windows platforms by downloading the [Redis command-line tools for Windows³](#). If you want to run the command-line tool on another platform, download Azure Cache for Redis from <https://redis.io/download>.

Redis supports a set of known commands. A command is typically issued as `COMMAND parameter1 parameter2 parameter3`.

Here are some common commands you can use:

² <https://azure.microsoft.com/pricing/details/cache/>

³ <https://github.com/MSOpenTech/redis/releases/>

Command	Description
ping	Ping the server. Returns "PONG".
set [key] [value]	Sets a key/value in the cache. Returns "OK" on success.
get [key]	Gets a value from the cache.
exists [key]	Returns '1' if the key exists in the cache, '0' if it doesn't.
type [key]	Returns the type associated to the value for the given key .
incr [key]	Increment the given value associated with key by '1'. The value must be an integer or double value. This returns the new value.
incrby [key] [amount]	Increment the given value associated with key by the specified amount. The value must be an integer or double value. This returns the new value.
del [key]	Deletes the value associated with the key .
flushdb	Delete <i>all</i> keys and values in the database.

Below is an example of a command:

```
> set somekey somevalue
OK
> get somekey
"somevalue"
> exists somekey
(string) 1
> del somekey
(string) 1
> exists somekey
(string) 0
```

Adding an expiration time to values

Caching is important because it allows us to store commonly used values in memory. However, we also need a way to expire values when they are stale. In Redis this is done by applying a time to live (TTL) to a key.

When the TTL elapses, the key is automatically deleted, exactly as if the DEL command were issued. Here are some notes on TTL expirations.

- Expirations can be set using seconds or milliseconds precision.
- The expire time resolution is always 1 millisecond.
- Information about expires are replicated and persisted on disk, the time virtually passes when your Redis server remains stopped (this means that Redis saves the date at which a key will expire).

Here is an example of an expiration:

```
> set counter 100
OK
> expire counter 5
```

```
(integer) 1
> get counter
100
... wait ...
> get counter
(nil)
```

Accessing a Redis cache from a client

To connect to an Azure Cache for Redis instance, you'll need several pieces of information. Clients need the host name, port, and an access key for the cache. You can retrieve this information in the Azure portal through the **Settings > Access Keys** page.

- The host name is the public Internet address of your cache, which was created using the name of the cache. For example `sportsresults.redis.cache.windows.net`.
- The access key acts as a password for your cache. There are two keys created: primary and secondary. You can use either key, two are provided in case you need to change the primary key. You can switch all of your clients to the secondary key, and regenerate the primary key. This would block any applications using the original primary key. Microsoft recommends periodically regenerating the keys - much like you would your personal passwords.

Warning: Your access keys should be considered confidential information, treat them like you would a password. Anyone who has an access key can perform any operation on your cache!

Interact with Azure Cache for Redis by using .NET

Typically, a client application will use a client library to form requests and execute commands on a Redis cache. You can get a list of client libraries directly from the Redis clients page.

Executing commands on the Redis cache

A popular high-performance Redis client for the .NET language is **StackExchange.Redis**⁴. The package is available through NuGet and can be added to your .NET code using the command line or IDE. Below are examples of how to use the client.

Connecting to your Redis cache with StackExchange.Redis

Recall that we use the host address, port number, and an access key to connect to a Redis server. Azure also offers a connection string for some Redis clients which bundles this data together into a single string. It will look something like the following (with the `cache-name` and `password-here` fields filled in with real values):

```
[cache-name].redis.cache.windows.net:6380,password=[password-here],ssl=True,abortConnect=False
```

You can pass this string to **StackExchange.Redis** to create a connection to the server.

⁴ <https://github.com/StackExchange/StackExchange.Redis>

Notice that there are two additional parameters at the end:

- **ssl** - ensures that communication is encrypted.
- **abortConnection** - allows a connection to be created even if the server is unavailable at that moment.

There are several other **optional parameters**⁵ you can append to the string to configure the client library.

Creating a connection

The main connection object in **StackExchange.Redis** is the `StackExchange.Redis.ConnectionMultiplexer` class. This object abstracts the process of connecting to a Redis server (or group of servers). It's optimized to manage connections efficiently and intended to be kept around while you need access to the cache.

You create a `ConnectionMultiplexer` instance using the static `ConnectionMultiplexer.Connect` or `ConnectionMultiplexer.ConnectAsync` method, passing in either a connection string or a `ConfigurationOptions` object.

Here's a simple example:

```
using StackExchange.Redis;  
...  
var connectionString = "[cache-name].redis.cache.windows.net:6380,password=[password-here],ssl=True,abortConnect=False";  
var redisConnection = ConnectionMultiplexer.Connect(connectionString);
```

Once you have a `ConnectionMultiplexer`, there are 3 primary things you might want to do:

- Access a Redis Database. This is what we will focus on here.
- Make use of the publisher/subscript features of Redis. This is outside the scope of this module.
- Access an individual server for maintenance or monitoring purposes.

Accessing a Redis database

The Redis database is represented by the `IDatabase` type. You can retrieve one using the `GetDatabase()` method:

```
IDatabase db = redisConnection.GetDatabase();
```

Tip: The object returned from `GetDatabase` is a lightweight object, and does not need to be stored. Only the `ConnectionMultiplexer` needs to be kept alive.

Once you have a `IDatabase` object, you can execute methods to interact with the cache. All methods have synchronous and asynchronous versions which return `Task` objects to make them compatible with the `async` and `await` keywords.

Here is an example of storing a key/value in the cache:

```
bool wasSet = db.StringSet("favorite:flavor", "i-love-rocky-road");
```

⁵ <https://github.com/StackExchange/StackExchange.Redis/blob/master/docs/Configuration.md#configuration-options>

The `StringSet` method returns a `bool` indicating whether the value was set (`true`) or not (`false`). We can then retrieve the value with the `StringGet` method:

```
string value = db.StringGet("favorite:flavor");
Console.WriteLine(value); // displays: ""i-love-rocky-road""
```

Getting and Setting binary values

Recall that Redis keys and values are binary safe. These same methods can be used to store binary data. There are implicit conversion operators to work with `byte[]` types so you can work with the data naturally:

```
byte[] key = ...;
byte[] value = ...;

db.StringSet(key, value);

byte[] key = ...;
byte[] value = db.StringGet(key);
```

StackExchange.Redis represents keys using the `RedisKey` type. This class has implicit conversions to and from both `string` and `byte[]`, allowing both text and binary keys to be used without any complication. Values are represented by the `RedisValuetype`. As with `RedisKey`, there are implicit conversions in place to allow you to pass `string` or `byte[]`.

Other common operations

The `IDatabase` interface includes several other methods to work with the Redis cache. There are methods to work with hashes, lists, sets, and ordered sets.

Here are some of the more common ones that work with single keys, you can [read the source code⁶](#) for the interface to see the full list.

Method	Description
<code>CreateBatch</code>	Creates a group of operations that will be sent to the server as a single unit, but not necessarily processed as a unit.
<code>CreateTransaction</code>	Creates a group of operations that will be sent to the server as a single unit and processed on the server as a single unit.
<code>KeyDelete</code>	Delete the key/value.
<code>KeyExists</code>	Returns whether the given key exists in cache.
<code>KeyExpire</code>	Sets a time-to-live (TTL) expiration on a key.
<code>KeyRename</code>	Renames a key.
<code>KeyTimeToLive</code>	Returns the TTL for a key.

⁶ <https://github.com/StackExchange/StackExchange.Redis/blob/master/src/StackExchange.Redis/Interfaces/IDatabase.cs>

Method	Description
KeyType	Returns the string representation of the type of the value stored at key. The different types that can be returned are: string, list, set, zset and hash.

Executing other commands

The `IDatabase` object has an `Execute` and `ExecuteAsync` method which can be used to pass textual commands to the Redis server. For example:

```
var result = db.Execute("ping");
Console.WriteLine(result.ToString()); // displays: "PONG"
```

The `Execute` and `ExecuteAsync` methods return a `RedisResult` object which is a data holder that includes two properties:

- `Type` which returns a string indicating the type of the result - "STRING", "INTEGER", etc.
- `IsNull` a true/false value to detect when the result is null.

You can then use `ToString()` on the `RedisResult` to get the actual return value.

You can use `Execute` to perform any supported commands - for example, we can get all the clients connected to the cache ("CLIENT LIST"):

```
var result = await db.ExecuteAsync("client", "list");
Console.WriteLine($"Type = {result.Type}\r\nResult = {result}");
```

This would output all the connected clients:

```
Type = BulkString
Result = id=9469 addr=16.183.122.154:54961 fd=18 name=DESKTOP-AAAAAA age=0
idle=0 flags=N db=0 sub=1 psub=0 multi=-1 qbuf=0 qbuf-free=0 obl=0 oll=0
omem=0 ow=0 owmem=0 events=r cmd=subscribe numops=5
id=9470 addr=16.183.122.155:54967 fd=13 name=DESKTOP-BBBBBB age=0 idle=0
flags=N db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-free=32768 obl=0 oll=0 omem=0
ow=0 owmem=0 events=r cmd=client numops=17
```

Storing more complex values

Redis is oriented around binary safe strings, but you can cache off object graphs by serializing them to a textual format - typically XML or JSON. For example, perhaps for our statistics, we have a `GameStats` object which looks like:

```
public class GameStat
{
    public string Id { get; set; }
    public string Sport { get; set; }
    public DateTimeOffset DatePlayed { get; set; }
    public string Game { get; set; }
    public IReadOnlyList<string> Teams { get; set; }
    public IReadOnlyList<(string team, int score)> Results { get; set; }
```

```
    public GameStat(string sport, DateTimeOffset datePlayed, string game,
string[] teams, IEnumerable<(string team, int score)> results)
    {
        Id = Guid.NewGuid().ToString();
        Sport = sport;
        DatePlayed = datePlayed;
        Game = game;
        Teams = teams.ToList();
        Results = results.ToList();
    }

    public override string ToString()
    {
        return $"{Sport} {Game} played on {DatePlayed.Date.ToShortDateString()} - " +
            $"{String.Join(',', Teams)}\r\n\t" +
            $"{String.Join('\t', Results.Select(r => $"{r.team} - {r.score}\r\n"))}";
    }
}
```

We could use the **Newtonsoft.Json** library to turn an instance of this object into a string:

```
var stat = new GameStat("Soccer", new DateTime(2019, 7, 16), "Local Game",
    new[] { "Team 1", "Team 2" },
    new[] { ("Team 1", 2), ("Team 2", 1) });

string serializedValue = Newtonsoft.Json.JsonConvert.SerializeObject(stat);
bool added = db.StringSet("event:1950-world-cup", serializedValue);
```

We could retrieve it and turn it back into an object using the reverse process:

```
var result = db.StringGet("event:2019-local-game");
var stat = Newtonsoft.Json.JsonConvert.DeserializeObject<GameStat>(result.
    ToString());
Console.WriteLine(stat.Sport); // displays "Soccer"
```

Cleaning up the connection

Once you are done with the Redis connection, you can `Dispose` the `ConnectionMultiplexer`. This will close all connections and shutdown the communication to the server.

```
redisConnection.Dispose();
redisConnection = null;
```

Exercise: Connect an app to Azure Cache for Redis by using .NET Core

In this exercise you will learn how to:

- Create a new Redis Cache instance by using Azure CLI commands.
- Create a .NET Core console app to add and retrieve values from the cache by using the **StackExchange.Redis** package.

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**⁷ on one of the **supported platforms**⁸.
- The **C# extension**⁹ for Visual Studio Code.
- **.NET Core 3.1**¹⁰ is the target framework for the steps below.

Create Azure resources

1. Sign in to the portal: <https://portal.azure.com> and open the Cloud Shell, and select **Bash** as the shell.
2. Create a resource group for Azure resources. Replace <myLocation> with a region near you.

```
az group create --name az204-redis-rg --location <myLocation>
```
3. Create an Azure Cache for Redis instance by using the `az redis create` command. The instance name needs to be unique and the script below will attempt to generate one for you, replace <myLocation> with the region you used in the previous step. This command will take a few minutes to complete.

```
redisName=az204redis$RANDOM  
az redis create --location <myLocation> \  
  --resource-group az204-redis-rg \  
  --name $redisName \  
  --sku Basic --vm-size c0
```

4. In the Azure portal navigate to the new Redis Cache you created.
5. Select **Access keys** in the **Settings** section of the Navigation Pane and leave the portal open. We'll copy the **Primary connection string (StackExchange.Redis)** value to use in the app later.

Create the console application

1. Create a console app by running the command below in the Visual Studio Code terminal.

⁷ <https://code.visualstudio.com/>

⁸ https://code.visualstudio.com/docs/supporting/requirements#_platforms

⁹ <https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp>

¹⁰ <https://dotnet.microsoft.com/download/dotnet/3.1>

```
dotnet new console -o Rediscache
```

2. Open the app in Visual Studio Code by selecting **File > Open Folder** and choosing the folder for the app.

3. Add the `StackExchange.Redis` package to the project.

```
dotnet add package StackExchange.Redis
```

4. In the `Program.cs` file add the `using` statement below at the top.

```
using StackExchange.Redis;  
using System.Threading.Tasks;
```

5. Add the following variable to the `Program` class, replace `<REDIS_CONNECTION_STRING>` with the **Primary connection string (StackExchange.Redis)** from the portal.

```
// connection string to your Redis Cache  
static string connectionString = "REDIS_CONNECTION_STRING";
```

6. Replace the `Main` method with the following code.

```
static async Task Main(string[] args)  
{  
    // The connection to the Azure Cache for Redis is managed by the ConnectionMultiplexer class.  
    using (var cache = ConnectionMultiplexer.Connect(connectionString))  
    {  
        IDatabase db = cache.GetDatabase();  
  
        // Snippet below executes a PING to test the server connection  
        var result = await db.ExecuteAsync("ping");  
        Console.WriteLine($"PING = {result.Type} : {result}");  
  
        // Call StringSetAsync on the IDatabase object to set the key "test:key" to the value "100"  
        bool setValue = await db.StringSetAsync("test:key", "100");  
        Console.WriteLine($"SET: {setValue}");  
  
        // StringGetAsync takes the key to retrieve and return the value  
        string getValue = await db.StringGetAsync("test:key");  
        Console.WriteLine($"GET: {getValue}");  
  
    }  
}
```

7. In the Visual Studio Code terminal run the commands below to build the app to check for errors, and then run the app using the commands below

```
dotnet build  
dotnet run
```

The output should be similar to the following:

```
PING = SimpleString : PONG
SET: True
GET: 100
```

8. Return to the portal and select **Activity log** in the **Azure Cache for Redis** blade. You can view the operations in the log.

Clean up resources

When the resources are no longer needed, you can use the `az group delete` command to remove the resource group.

```
az group delete -n az204-redis-rg --no-wait
```

Knowledge check

Multiple choice

Which of the Azure Cache for Redis service tiers is the lowest tier recommended for use in production scenarios?

- Basic
- Standard
- Premium

Multiple choice

Caching is important because it allows us to store commonly used values in memory. However, we also need a way to expire values when they are stale. In Redis this is done by applying a time to live (TTL) to a key. Which value represents the expire time resolution?

- 1 millisecond
- 10 milliseconds
- seconds or milliseconds

Summary

In this module, you learned how to:

- Explain the key scenarios Azure Cache for Redis covers and its service tiers
- Identify the key parameters for creating an Azure Cache for Redis instance and interact with the cache
- Connect an app to Azure Cache for Redis by using .NET Core

Develop for storage on CDNs

Introduction

A content delivery network (CDN) is a distributed network of servers that can efficiently deliver web content to users. CDNs' store cached content on edge servers in point-of-presence (POP) locations that are close to end users, to minimize latency.

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

- Explain how the Azure Content Delivery Network works and how it can improve the user experience.
- Control caching behavior and purge content.
- Perform actions on Azure CDN by using the Azure CDN Library for .NET.

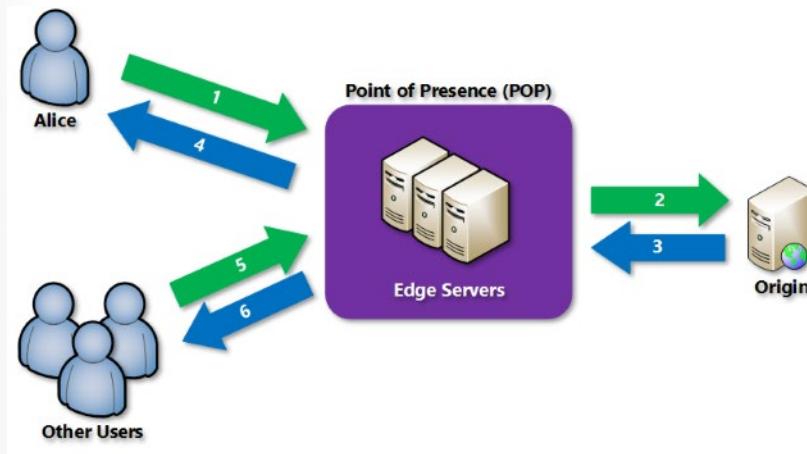
Explore Azure Content Delivery Networks

Azure Content Delivery Network (CDN) offers developers a global solution for rapidly delivering high-bandwidth content to users by caching their content at strategically placed physical nodes across the world. Azure CDN can also accelerate dynamic content, which cannot be cached, by leveraging various network optimizations using CDN POPs. For example, route optimization to bypass Border Gateway Protocol (BGP).

The benefits of using Azure CDN to deliver web site assets include:

- Better performance and improved user experience for end users, especially when using applications in which multiple round-trips are required to load content.
- Large scaling to better handle instantaneous high loads, such as the start of a product launch event.
- Distribution of user requests and serving of content directly from edge servers so that less traffic is sent to the origin server.

How Azure Content Delivery Network works



1. A user (Alice) requests a file (also called an asset) by using a URL with a special domain name, such as <endpoint name>.azureedge.net. This name can be an endpoint hostname or a custom domain. The DNS routes the request to the best performing POP location, which is usually the POP that is geographically closest to the user.

2. If no edge servers in the POP have the file in their cache, the POP requests the file from the origin server. The origin server can be an Azure Web App, Azure Cloud Service, Azure Storage account, or any publicly accessible web server.
3. The origin server returns the file to an edge server in the POP.
4. An edge server in the POP caches the file and returns the file to the original requestor (Alice). The file remains cached on the edge server in the POP until the time-to-live (TTL) specified by its HTTP headers expires. If the origin server didn't specify a TTL, the default TTL is seven days.
5. Additional users can then request the same file by using the same URL that Alice used, and can also be directed to the same POP.
6. If the TTL for the file hasn't expired, the POP edge server returns the file directly from the cache. This process results in a faster, more responsive user experience.

Requirements

To use Azure CDN you need to create at least one CDN profile, which is a collection of CDN endpoints. Every CDN endpoint represents a specific configuration of content deliver behavior and access. To organize your CDN endpoints by internet domain, web application, or some other criteria, you can use multiple profiles. Because **Azure CDN pricing¹¹** is applied at the CDN profile level, you must create multiple CDN profiles if you want to use a mix of pricing tiers.

Limitations

Each Azure subscription has default limits for the following resources:

- The number of CDN profiles that can be created.
- The number of endpoints that can be created in a CDN profile.
- The number of custom domains that can be mapped to an endpoint.

For more information about CDN subscription limits, visit **CDN limits¹²**.

Azure Content Delivery Network products

Azure Content Delivery Network (CDN) includes four products:

- Azure CDN Standard from Microsoft
- Azure CDN Standard from Akamai
- Azure CDN Standard from Verizon
- Azure CDN Premium from Verizon

Visit the **product comparison¹³** for a detailed feature comparison between the four products.

¹¹ <https://azure.microsoft.com/pricing/details/cdn/>

¹² <https://docs.microsoft.com/azure/azure-resource-manager/management/azure-subscription-service-limits>

¹³ <https://docs.microsoft.com/azure/cdn/cdn-features>

Control cache behavior on Azure Content Delivery Networks

Because a cached resource can potentially be out-of-date or stale (compared to the corresponding resource on the origin server), it is important for any caching mechanism to control when content is refreshed. To save time and bandwidth consumption, a cached resource is not compared to the version on the origin server every time it is accessed. Instead, as long as a cached resource is considered to be fresh, it is assumed to be the most current version and is sent directly to the client. A cached resource is considered to be fresh when its age is less than the age or period defined by a cache setting. For example, when a browser reloads a webpage, it verifies that each cached resource on your hard drive is fresh and loads it. If the resource is not fresh (stale), an up-to-date copy is loaded from the server.

Controlling caching behavior

Azure CDNs provide two mechanisms for caching files. However, these configuration settings depend on the tier you've selected. Caching rules in Azure CDN Standard for Microsoft are set at the endpoint level and provide three configuration options. Other tiers provide additional configuration options, which include:

- **Caching rules.** Caching rules can be either global (apply to all content from a specified endpoint) or custom. Custom rules apply to specific paths and file extensions.
- **Query string caching.** Query string caching enables you to configure how Azure CDN responds to a query string. Query string caching has no effect on files that can't be cached.

With the Azure CDN Standard for Microsoft Tier, caching rules are as simple as the following three options:

- Ignore query strings. This option is the default mode. A CDN POP simply passes the request and any query strings directly to the origin server on the first request and caches the asset. New requests for the same asset will ignore any query strings until the TTL expires.
- Bypass caching for query strings. Each query request from the client is passed directly to the origin server with no caching.
- Cache every unique URL. Every time a requesting client generates a unique URL, that URL is passed back to the origin server and the response cached with its own TTL. This final method is inefficient where each request is a unique URL, as the cache-hit ratio becomes low.

To change these settings, in the Endpoint pane, select **Caching rules** and then select the caching option that you want to apply to the endpoint and select **Save**.

Caching and time to live

If you publish a website through Azure CDN, the files on that site are cached until their TTL expires. The Cache-Control header contained in the HTTP response from origin server determines the TTL duration.

If you don't set a TTL on a file, Azure CDN sets a default value. However, this default may be overridden if you have set up caching rules in Azure. Default TTL values are as follows:

- Generalized web delivery optimizations: seven days
- Large file optimizations: one day
- Media streaming optimizations: one year

Content updating

In normal operation, an Azure CDN edge node will serve an asset until its TTL expires. The edge node reconnects to the origin server when the TTL expires and a client makes a request to the same asset. The node will fetch another copy of the asset, resetting the TTL in the process.

To ensure that users always receive the latest version of an asset, consider including a version string in the asset URL. This approach causes the CDN to retrieve the new asset immediately.

Alternatively, you can purge cached content from the edge nodes, which refreshes the content on the next client request. You might purge cached content when publishing a new version of a web app or to replace any out-of-date assets.

You can purge content in several ways.

- On an endpoint by endpoint basis, or all endpoints simultaneously should you want to update everything on your CDN at once.
- Specify a file, by including the path to that file or all assets on the selected endpoint by checking the **Purge All** checkbox in the Azure portal.
- Based on wildcards (*) or using the root (/).

The Azure CLI provides a special purge verb that will unpublish cached assets from an endpoint. This is very useful if you have an application scenario where a large amount of data is invalidated and should be updated in the cache. To unpublish assets, you must specify either a file path, a wildcard directory, or both:

```
az cdn endpoint purge \
    --content-paths '/css/*' '/js/app.js' \
    --name ContosoEndpoint \
    --profile-name DemoProfile \
    --resource-group ExampleGroup
```

You can also preload assets into an endpoint. This is useful for scenarios where your application creates a large number of assets, and you want to improve the user experience by prepopulating the cache before any actual requests occur:

```
az cdn endpoint load \
    --content-paths '/img/*' '/js/module.js' \
    --name ContosoEndpoint \
    --profile-name DemoProfile \
    --resource-group ExampleGroup
```

Geo-filtering

Geo-filtering enables you to allow or block content in specific countries, based on the country code. In the Azure CDN Standard for Microsoft Tier, you can only allow or block the entire site. With the Verizon and Akamai tiers, you can also set up restrictions on directory paths. For more information, see the further reading section in the Summary unit.

Interact with Azure Content Delivery Networks by using .NET

You can use the Azure CDN Library for .NET to automate creation and management of CDN profiles and endpoints. Install the **Microsoft.Azure.Management.Cdn.Fluent**¹⁴ directly from the Visual Studio Package Manager console or with the .NET Core CLI.

In this unit you will see code examples illustrating common actions.

Create a CDN client

The example below shows creating a client by using the `CdnManagementClient` class.

```
static void Main(string[] args)
{
    // Create CDN client
    CdnManagementClient cdn = new CdnManagementClient(new TokenCreden-
    tials(authResult.AccessToken))
        { SubscriptionId = subscriptionId };
}
```

List CDN profiles and endpoints

The first thing the method below does is list all the profiles and endpoints in our resource group, and if it finds a match for the profile and endpoint names specified in our constants, makes a note of that for later so we don't try to create duplicates.

```
private static void ListProfilesAndEndpoints(CdnManagementClient cdn)
{
    // List all the CDN profiles in this resource group
    var profileList = cdn.Profiles.ListByResourceGroup(resourceGroupName);
    foreach (Profile p in profileList)
    {
        Console.WriteLine("CDN profile {0}", p.Name);
        if (p.Name.Equals(profileName, StringComparison.OrdinalIgnoreCase))
        {
            // Hey, that's the name of the CDN profile we want to create!
            profileAlreadyExists = true;
        }

        //List all the CDN endpoints on this CDN profile
        Console.WriteLine("Endpoints:");
        var endpointList = cdn.Endpoints.ListByProfile(p.Name, resource-
        GroupName);
        foreach (Endpoint e in endpointList)
        {
            Console.WriteLine("-{0} ({1})", e.Name, e.HostName);
            if (e.Name.Equals(endpointName, StringComparison.OrdinalIgnoreCase))

```

¹⁴ <https://www.nuget.org/packages/Microsoft.Azure.Management.Cdn.Fluent>

```
        {
            // The unique endpoint name already exists.
            endpointAlreadyExists = true;
        }
    }
    Console.WriteLine();
}
}
```

Create CDN profiles and endpoints

The example below shows creating an Azure CDN profile.

```
private static void CreateCdnProfile(CdnManagementClient cdn)
{
    if (profileAlreadyExists)
    {
        //Check to see if the profile already exists
    }
    else
    {
        //Create the new profile
        ProfileCreateParameters profileParms =
            new ProfileCreateParameters() { Location = resourceLocation, Sku =
= new Sku(SkuName.StandardVerizon) };
        cdn.Profiles.Create(profileName, profileParms, resourceGroupName);
    }
}
```

Once the profile is created, we'll create an endpoint.

```
private static void CreateCdnEndpoint(CdnManagementClient cdn)
{
    if (endpointAlreadyExists)
    {
        //Check to see if the endpoint already exists
    }
    else
    {
        //Create the new endpoint
        EndpointCreateParameters endpointParms =
            new EndpointCreateParameters()
            {
                Origins = new List<DeepCreatedOrigin>() { new DeepCreate-
dOrigin("Contoso", "www.contoso.com") },
                IsHttpAllowed = true,
                IsHttpsAllowed = true,
                Location = resourceLocation
            };
        cdn.Endpoints.Create(endpointName, endpointParms, profileName,
resourceGroupName);
```

```
        }  
    }
```

Purge an endpoint

A common task that we might want to perform is purging the content in our endpoint.

```
private static void PromptPurgeCdnEndpoint(CdnManagementClient cdn)  
{  
    if (PromptUser(String.Format("Purge CDN endpoint {0}?", endpointName)))  
    {  
        Console.WriteLine("Purging endpoint. Please wait...");  
        cdn.Endpoints.PurgeContent(resourceGroupName, profileName, endpoint-  
Name, new List<string>() { /* */ });  
        Console.WriteLine("Done.");  
        Console.WriteLine();  
    }  
}
```

Knowledge check

Multiple choice

Each Azure subscription has default limits on resources needed for an Azure Content Delivery Network. Which of the following resources has subscription limitations that may impact your solution?

- Resource group
- CDN profiles
- Storage account

Multiple choice

When publishing a website through Azure CDN, the files on that site are cached until their time-to-live (TTL) expires. What is the default TTL for large file optimizations?

- One day
- One week
- One year

Summary

In this module, you learned how to:

- Explain how the Azure Content Delivery Network works and how it can improve the user experience.
- Control caching behavior and purge content.
- Perform actions on Azure CDN by using the Azure CDN Library for .NET.

Answers

Multiple choice

Which of the Azure Cache for Redis service tiers is the lowest tier recommended for use in production scenarios?

- Basic
- Standard
- Premium

Explanation

That's correct. The standard tier is the lowest tier that offers replication which is always recommended for production scenarios.

Multiple choice

Caching is important because it allows us to store commonly used values in memory. However, we also need a way to expire values when they are stale. In Redis this is done by applying a time to live (TTL) to a key. Which value represents the expire time resolution?

- 1 millisecond
- 10 milliseconds
- seconds or milliseconds

Explanation

That's correct. The expire time resolution is always 1 millisecond.

Multiple choice

Each Azure subscription has default limits on resources needed for an Azure Content Delivery Network. Which of the following resources has subscription limitations that may impact your solution?

- Resource group
- CDN profiles
- Storage account

Explanation

That's correct. The number of CDN profiles that can be created is limited by the type of Azure subscription.

Multiple choice

When publishing a website through Azure CDN, the files on that site are cached until their time-to-live (TTL) expires. What is the default TTL for large file optimizations?

- One day
- One week
- One year

Explanation

That's correct. The default TTL for large file optimizations is one day.