

Azure for Architects

Third Edition

Create secure, scalable, high-availability applications on the cloud



Packt

www.packt.com

Ritesh Modi, Jack Lee, and Rithin Skaria

Azure for Architects

Third Edition

Create secure, scalable, high-availability
applications on the cloud

Ritesh Modi, Jack Lee, and Rithin Skaria

Packt»

Azure for Architects Third Edition

Copyright © 2020 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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: Ritesh Modi, Jack Lee, and Rithin Skaria

Technical Reviewers: Melony Qin and Sanjeev Kumar

Managing Editors: Aditya Datar and Afzal Shaikh

Acquisitions Editor: Shrilekha Inani

Production Editors: Ganesh Bhadwalkar and Deepak Chavan

Editorial Board: Vishal Bodwani, Ben Renow-Clarke, Edward Doxey, Joanne Lovell, Arijit Sarkar, and Dominic Shakeshaft

First Edition: October 2017

Second Edition: January 2019

Third Edition: June 2020

Production Reference: 3260620

ISBN: 978-1-83921-586-5

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham B3 2PB, UK



startHere(Azure);

**Get hands-on
in the cloud**

Learn Azure. Experiment with more than 100 services.

- Free services
- \$200 credit
- Free training

[Try Azure for free >](#)

Get help with your project.
[Talk to a sales specialist >](#)

Table of Contents

Preface	i
Chapter 1: Getting started with Azure	1
Cloud computing	2
The advantages of cloud computing	3
Why cloud computing?	3
Deployment paradigms in Azure	5
Understanding Azure	6
Azure as an intelligent cloud	8
Azure Resource Manager	8
The ARM architecture	9
Why ARM?	9
ARM advantages	10
ARM concepts	11
Virtualization	14
Containers	15
Docker	17
Interacting with the intelligent cloud	17
The Azure portal	17
PowerShell	18
The Azure CLI	18
The Azure REST API	19
ARM templates	19
Summary	20

Chapter 2: Azure solution availability, scalability, and monitoring	23
High availability	24
Azure high availability	25
Concepts	26
Load balancing	29
VM high availability	30
Compute high availability	30
High-availability platforms	32
Load balancers in Azure	32
The Azure Application Gateway	35
Azure Traffic Manager	36
Azure Front Door	38
Architectural considerations for high availability	38
High availability within Azure regions	39
High availability across Azure regions	40
Scalability	42
Scalability versus performance	43
Azure scalability	44
PaaS scalability	46
IaaS scalability	49
VM scale sets	50
VMSS architecture	51
VMSS scaling	51

Upgrades and maintenance	54
Application updates	56
Guest updates	56
Image updates	56
Best practices of scaling for VMSSes	57
Monitoring	58
Azure monitoring	59
Azure activity logs	59
Azure diagnostic logs	60
Azure application logs	60
Guest and host OS logs	60
Azure Monitor	61
Azure Application Insights	61
Azure Log Analytics	61
Solutions	62
Alerts	63
Summary	67
Chapter 3: Design pattern – Networks, storage, messaging, and events	69
Azure Availability Zones and Regions	70
Availability of resources	70
Data and privacy compliance	70
Application performance	71
Cost of running applications	71

Virtual networks	71
Architectural considerations for virtual networks	72
Benefits of virtual networks	76
Virtual network design	76
Connecting to resources within the same region and subscription	77
Connecting to resources within the same region in another subscription	77
Connecting to resources in different regions in another subscription	79
Connecting to on-premises datacenters	80
Storage	84
Storage categories	84
Storage types	84
Storage features	85
Architectural considerations for storage accounts	86
Cloud design patterns	88
Messaging patterns	89
Performance and scalability patterns	93
Summary	101
<hr/> Chapter 4: Automating architecture on Azure	103
Automation	104
Azure Automation	105
Azure Automation architecture	105
Process automation	107
Configuration management	108
Update management	109

Concepts related to Azure Automation	109
Runbook	109
Run As accounts	110
Jobs	111
Assets	112
Credentials	112
Certificates	113
Creating a service principal using certificate credentials	115
Connections	116
Runbook authoring and execution	118
Parent and child runbooks	119
Creating a runbook	120
Using Az modules	122
Webhooks	125
Invoking a webhook	127
Invoking a runbook from Azure Monitor	129
Hybrid Workers	134
Azure Automation State Configuration	136
Azure Automation pricing	141
Comparison with serverless automation	141
Summary	142

Chapter 5: Designing policies, locks, and tags for Azure deployments	145
Azure management groups	146
Azure tags	147
Tags with PowerShell	150
Tags with Azure Resource Manager templates	150
Tagging resource groups versus resources	151
Azure Policy	152
Built-in policies	153
Policy language	153
Allowed fields	156
Azure locks	156
Azure RBAC	158
Custom roles	161
How are locks different from RBAC?	162
Azure Blueprints	162
An example of implementing Azure governance features	163
Background	163
RBAC for Company Inc	163
Azure Policy	164
Azure locks	165
Summary	165

Chapter 6: Cost management for Azure solutions	167
Azure offer details	168
Understanding billing	169
Invoicing	176
The Modern Commerce experience	177
Usage and quotas	179
Resource providers and resource types	180
Usage and Billing APIs	182
Azure Enterprise Billing APIs	182
Azure Consumption APIs	183
Azure Cost Management APIs	184
Azure pricing calculator	184
Best practices	187
Azure Governance	187
Compute best practices	188
Storage best practices	189
PaaS best practices	190
General best practices	191
Summary	191

Chapter 7: Azure OLTP solutions	193
OLTP applications	194
Relational databases	195
Azure cloud services	195
Deployment models	196
Databases on Azure Virtual Machines	197
Databases hosted as managed services	198
Azure SQL Database	198
Application features	199
Security	204
Single Instance	210
Elastic pools	211
Managed Instance	213
SQL database pricing	215
DTU-based pricing	215
vCPU-based pricing	217
How to choose the appropriate pricing model	218
Azure Cosmos DB	219
Features	221
Use case scenarios	222
Summary	222

Chapter 8: Architecting secure applications on Azure	225
Security	226
 Security life cycle	228
 Azure security	230
IaaS security	231
 Network security groups	231
 Firewalls	234
 Application security groups	235
 Azure Firewall	236
 Reducing the attack surface area	237
 Implementing jump servers	238
 Azure Bastion	239
Application security	239
 SSL/TLS	239
 Managed identities	240
Azure Sentinel	244
PaaS security	245
 Azure Private Link	245
 Azure Application Gateway	245
 Azure Front Door	246
 Azure App Service Environment	247
 Log Analytics	247

Azure Storage	248
Azure SQL	252
Azure Key Vault	256
Authentication and authorization using OAuth	257
Security monitoring and auditing	265
Azure Monitor	265
Azure Security Center	267
Summary	268
Chapter 9: Azure Big Data solutions	271
<hr/>	
Big data	272
Process for big data	273
Big data tools	274
Azure Data Factory	274
Azure Data Lake Storage	274
Hadoop	275
Apache Spark	276
Databricks	276
Data integration	276
ETL	277
A primer on Azure Data Factory	278
A primer on Azure Data Lake Storage	279

Migrating data from Azure Storage to Data Lake Storage Gen2	280
Preparing the source storage account	280
Provisioning a new resource group	280
Provisioning a storage account	281
Provisioning the Data Lake Storage Gen2 service	283
Provisioning Azure Data Factory	284
Repository settings	285
Data Factory datasets	287
Creating the second dataset	289
Creating a third dataset	289
Creating a pipeline	291
Adding one more Copy Data activity	293
Creating a solution using Databricks	294
Loading data	297
Summary	303
<hr/>	
Chapter 10: Serverless in Azure – Working with Azure Functions	305
<hr/>	
Serverless	306
The advantages of Azure Functions	306
FaaS	308
The Azure Functions runtime	308
Azure Functions bindings and triggers	309
Azure Functions configuration	312
Azure Functions cost plans	314
Azure Functions destination hosts	316
Azure Functions use cases	316
Types of Azure functions	318

Creating an event-driven function	318
Function Proxies	321
Durable Functions	322
Steps for creating a durable function using Visual Studio	324
Creating a connected architecture with functions	329
Azure Event Grid	332
Event Grid	333
Resource events	335
Custom events	340
Summary	343
Chapter 11: Azure solutions using Azure Logic Apps, Event Grid, and Functions	345
<hr/>	
Azure Logic Apps	346
Activities	346
Connectors	346
The workings of a logic app	347
Creating an end-to-end solution using serverless technologies	355
The problem statement	355
Solution	355
Architecture	356
Prerequisites	357
Implementation	357
Testing	385
Summary	386

Chapter 12: Azure Big Data eventing solutions	389
Introducing events	390
Event streaming	391
Event Hubs	392
Event Hubs architecture	395
Consumer groups	402
Throughput	403
A primer on Stream Analytics	403
The hosting environment	407
Streaming units	408
A sample application using Event Hubs and Stream Analytics	408
Provisioning a new resource group	408
Creating an Event Hubs namespace	409
Creating an event hub	410
Provisioning a logic app	411
Provisioning the storage account	413
Creating a storage container	413
Creating Stream Analytics jobs	414
Running the application	416
Summary	418

Chapter 13: Integrating Azure DevOps	421
DevOps	422
The essence of DevOps	425
DevOps practices	427
Configuration management	428
Configuration management tools	429
Continuous integration	430
Continuous deployment	433
Continuous delivery	435
Continuous learning	435
Azure DevOps	436
TFVC	439
Git	439
Preparing for DevOps	440
Azure DevOps organizations	441
Provisioning Azure Key Vault	442
Provisioning a configuration-management server/service	442
Log Analytics	443
Azure Storage accounts	443
Docker and OS images	443
Management tools	443
DevOps for PaaS solutions	444
Azure App Service	445
Deployment slots	445
Azure SQL	446
The build and release pipelines	446

DevOps for IaaS	458
Azure virtual machines	458
Azure public load balancers	459
The build pipeline	459
The release pipeline	460
DevOps with containers	462
Containers	462
The build pipeline	463
The release pipeline	463
Azure DevOps and Jenkins	464
Azure Automation	466
Provisioning an Azure Automation account	467
Creating a DSC configuration	468
Importing the DSC configuration	469
Compiling the DSC configuration	470
Assigning configurations to nodes	470
Validation	471
Tools for DevOps	471
Summary	473
<hr/> Chapter 14: Architecting Azure Kubernetes solutions	475
Introduction to containers	476
Kubernetes fundamentals	477
Kubernetes architecture	479
Kubernetes clusters	480
Kubernetes components	481

Kubernetes primitives	484
Pod	485
Services	486
Deployments	488
Replication controller and ReplicaSet	490
ConfigMaps and Secrets	491
AKS architecture	492
Deploying an AKS cluster	493
Creating an AKS cluster	493
Kubectl	495
Connecting to the cluster	495
AKS networking	500
Kubenet	501
Azure CNI (advanced networking)	503
Access and identity for AKS	504
Virtual kubelet	505
Virtual nodes	506
Summary	507
Chapter 15: Cross-subscription deployments using ARM templates	509
<hr/>	
ARM templates	510
Deploying resource groups with ARM templates	513
Deploying ARM templates	515
Deployment of templates using Azure CLI	516
Deploying resources across subscriptions and resource groups	517
Another example of cross-subscription and resource group deployments	519

Deploying cross-subscription and resource group deployments using linked templates	522
Virtual machine solutions using ARM templates	526
PaaS solutions using ARM templates	532
Data-related solutions using ARM templates	534
Creating an IaaS solution on Azure with Active Directory and DNS	541
Summary	545
Chapter 16: ARM template modular design and implementation	547
Problems with the single template approach	548
Reduced flexibility in changing templates	548
Troubleshooting large templates	548
Dependency abuse	549
Reduced agility	549
No reusability	549
Understanding the Single Responsibility Principle	550
Faster troubleshooting and debugging	550
Modular templates	550
Deployment resources	551
Linked templates	552
Nested templates	554
Free-flow configurations	556
Known configurations	556
Understanding copy and copyIndex	567
Securing ARM templates	569
Using outputs between ARM templates	570
Summary	573

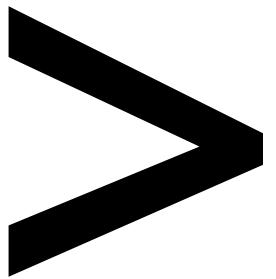
Chapter 17: Designing IoT solutions 575

IoT	576
IoT architecture	577
Connectivity	579
Identity	581
Capture	581
Ingestion	581
Storage	582
Transformation	582
Analytics	582
Presentation	583
Azure IoT	584
Connectivity	584
Identity	585
Capture	585
Ingestion	585
Storage	586
Transformation and analytics	586
Presentation	587
Azure IoT Hub	588
Protocols	589
Device registration	589
Message management	590
Security	593
Scalability	594
Azure IoT Edge	596
High availability	596

Azure IoT Central	597
Summary	598
Chapter 18: Azure Synapse Analytics for architects	601
Azure Synapse Analytics	602
A common scenario for architects	603
An overview of Azure Synapse Analytics	603
What is workload isolation?	604
Introduction to Synapse workspaces and Synapse Studio	605
Apache Spark for Synapse	607
Synapse SQL	608
Synapse pipelines	609
Azure Synapse Link for Cosmos DB	610
Migrating from existing legacy systems to Azure Synapse Analytics	611
Why you should migrate your legacy data warehouse to Azure Synapse Analytics	611
The three-step migration process	613
The two types of migration strategies	614
Reducing the complexity of your existing legacy data warehouse before migrating	615
Converting physical data marts to virtual data marts	615
Migrating existing data warehouse schemas to Azure Synapse Analytics	616
Migrating historical data from your legacy data warehouse to Azure Synapse Analytics	619
Migrating existing ETL processes to Azure Synapse Analytics	621
Re-developing scalable ETL processes using ADF	622
Recommendations for migrating queries, BI reports, dashboards, and other visualizations	622
Common migration issues and resolutions	623

Common SQL incompatibilities and resolutions	625
SQL DDL differences and resolutions	626
SQL DML differences and resolutions	627
SQL DCL differences and resolutions	627
Extended SQL differences and workarounds	631
Security considerations	632
Data encryption at rest	632
Data in motion	632
Tools to help migrate to Azure Synapse Analytics	633
ADF	633
Azure Data Warehouse Migration Utility	634
Microsoft Services for Physical Data Transfer	634
Microsoft Services for data ingestion	635
Summary	636
<hr/> Chapter 19: Architecting intelligent solutions	639
The evolution of AI	640
Azure AI processes	641
Data ingestion	641
Data transformation	641
Analysis	641
Data modeling	642
Validating the model	642
Deployment	642
Monitoring	642

Azure Cognitive Services	643
Vision	644
Search	644
Language	644
Speech	644
Decision	644
Understanding Cognitive Services	645
Consuming Cognitive Services	646
Building an OCR service	646
Using PowerShell	649
Using C#	650
The development process	652
Building a visual features service using the Cognitive Search .NET SDK	655
Using PowerShell	655
Using .NET	656
Safeguarding the Cognitive Services key	658
Using Azure Functions Proxies	658
Consuming Cognitive Services	659
Summary	659
Index	661



Preface

About

This section briefly introduces the authors, the coverage of this book, the technical skills you'll need to get started, and the hardware and software requirements required to architect solutions using Azure.

About Azure for Architects, Third Edition

Thanks to its support for high availability, scalability, security, performance, and disaster recovery, Azure has been widely adopted to create and deploy different types of application with ease. Updated for the latest developments, this third edition of *Azure for Architects* helps you get to grips with the core concepts of designing serverless architecture, including containers, Kubernetes deployments, and big data solutions. You'll learn how to architect solutions such as serverless functions, you'll discover deployment patterns for containers and Kubernetes, and you'll explore large-scale big data processing using Spark and Databricks. As you advance, you'll implement DevOps using Azure DevOps, work with intelligent solutions using Azure Cognitive Services, and integrate security, high availability, and scalability into each solution. Finally, you'll delve into Azure security concepts such as OAuth, OpenConnect, and managed identities.

By the end of this book, you'll have gained the confidence to design intelligent Azure solutions based on containers and serverless functions.

About the Authors

Ritesh Modi is a former Microsoft senior technology evangelist. He has been recognized as a Microsoft Regional Director for his contributions to Microsoft products, services, and communities. He is a cloud architect, a published author, a speaker, and a leader who is popular for his contributions to datacenters, Azure, Kubernetes, blockchain, cognitive services, DevOps, artificial intelligence, and automation. He is the author of eight books.

Ritesh has spoken at numerous national and international conferences and is a published author for MSDN magazine. He has more than a decade of experience in building and deploying enterprise solutions for customers, and has more than 25 technical certifications. His hobbies are writing books, playing with his daughter, watching movies, and learning new technologies. He currently lives in Hyderabad, India. You can follow him on Twitter at [@automationnext](#).

Jack Lee is a senior Azure certified consultant and an Azure practice lead with a passion for software development, cloud, and DevOps innovations. Jack has been recognized as a Microsoft MVP for his contributions to the tech community. He has presented at various user groups and conferences, including the Global Azure Bootcamp at Microsoft Canada. Jack is an experienced mentor and judge at hackathons and is also the president of a user group that focuses on Azure, DevOps, and software development. He is the co-author of *Cloud Analytics with Microsoft Azure*, published by Packt Publishing. You can follow Jack on Twitter at [@jlee_consulting](#).

Rithin Skaria is an open source evangelist with over 7 years of experience of managing open source workloads in Azure, AWS, and OpenStack. He is currently working for Microsoft and is a part of several open source community activities conducted within Microsoft. He is a Microsoft Certified Trainer, Linux Foundation Certified Engineer and Administrator, Kubernetes Application Developer and Administrator, and also a Certified OpenStack Administrator. When it comes to Azure, he has four certifications (solution architecture, Azure administration, DevOps, and security), and he is also certified in Office 365 administration. He has played a vital role in several open source deployments, and the administration and migration of these workloads to the cloud. He also co-authored *Linux Administration on Azure*, published by Packt Publishing. Connect with him on LinkedIn at [@rithin-skaria](#).

About the Reviewers

Melony Qin is a woman in STEM. Currently working as a Program Manager at Microsoft, she's a member of the **Association for Computing Machinery (ACM)** and **Project Management Institute (PMI)**. She has contributed to serverless computing, big data processing, DevOps, artificial intelligence, machine learning, and IoT with Microsoft Azure. She holds all the Azure certifications (both the Apps and Infrastructure and the Data and AI tracks) as well as **Certified Kubernetes Administrator (CKA)** and **Certified Kubernetes Application Developer (CKAD)**, and is mainly working on her contributions to **open-source software (OSS)**, DevOps, Kubernetes, serverless, big data analytics, and IoT on Microsoft Azure in the community. She's the author and co-author of two books, *Microsoft Azure Infrastructure* and *The Kubernetes Workshop*, both published by Packt Publishing. She can be reached out via Twitter at [@MelonyQ](#).

Sanjeev Kumar is a Cloud Solution Architect for SAP on Azure at Microsoft. He is currently based in Zurich, Switzerland. He has worked with SAP technology for over 19 years. He has been working with public cloud technologies for about 8 years, the last 2 years of which have been focused on Microsoft Azure.

In his SAP on Azure cloud architecture advisory role, Sanjeev Kumar has worked with a number of the world's top financial services and manufacturing companies. His focus areas include cloud architecture and design to help customers migrate their SAP systems to Azure and adopt Azure best practices for SAP deployments, especially by implementing Infrastructure as Code and DevOps. He has also worked in the areas of containerization and microservices using Docker and Azure Kubernetes Service, streaming data processing using Apache Kafka, and full stack application development using Node.js. He has worked on various product development initiatives spanning IaaS, PaaS, and SaaS. He is also interested in the emerging topics of artificial intelligence, machine learning, and large-scale data processing and analytics. He writes on topics related to SAP on Azure, DevOps, and Infrastructure as Code on LinkedIn, where you can find him at [@sanjeevkumarprofile](#).

Learning Objectives

By the end of this book, you will be able to:

- Understand the components of the Azure cloud platform
- Use cloud design patterns
- Use enterprise security guidelines for your Azure deployment
- Design and implement serverless and integration solutions
- Build efficient data solutions on Azure
- Understand container services on Azure

Audience

If you are a cloud architect, DevOps engineer, or a developer looking to learn about the key architectural aspects of the Azure cloud platform, this book is for you. A basic understanding of the Azure cloud platform will help you grasp the concepts covered in this book more effectively.

Approach

This book covers each topic with step-by-step explanations of essential concepts, practical examples, and self-assessment questions. By providing a balance of theory and practical experience of working through engaging projects, this book will help you understand how architects work in the real world.

Hardware Requirements

For the optimal experience, we recommend the following configuration:

- Minimum 4 GB RAM
- Minimum 32 GB of free memory

Software Requirements

- Visual Studio 2019
- Docker for Windows latest version
- AZ PowerShell module 1.7 and above
- Azure CLI latest version
- Azure subscription
- Windows Server 2016/2019
- Window 10 latest version - 64 bit

Conventions

Code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user inputs are shown as follows:

The DSC configuration still isn't known to Azure Automation. It's available on some local machines. It should be uploaded to Azure Automation DSC Configurations.

Azure Automation provides the **Import-AzureRmAutomationDscConfiguration** cmdlet to import the configuration to Azure Automation:

```
Import-AzureRmAutomationDscConfiguration -SourcePath "C:\DSC\AA\DSConfigurations\ConfigureSiteOnIIS.ps1" -ResourceGroupName "omsauto" -AutomationAccountName "datacenterautomation" -Published -Verbose
```

Download Resources

The code bundle for this book is also hosted on GitHub at: <https://github.com/PacktPublishing/Azure-for-Architects-Third-Edition>.

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

1

Getting started with Azure

Every few years, a technological innovation emerges that permanently changes the entire landscape and ecosystem around it. If we go back in time, the 1970s and 1980s were the time of mainframes. These mainframes were massive, often occupying large rooms, and were solely responsible for almost all computing work. Since the technology was difficult to procure and time-consuming to use, many enterprises used to place orders for mainframes one month in advance before they could have an operational mainframe set up.

Then, the early 1990s witnessed a boom in demand for personal computing and the internet. As a result, computers became much smaller in size and comparatively easy to procure for the general public. Consistent innovations on the personal computing and internet fronts eventually changed the entire computer industry. Many people had desktop computers that were capable of running multiple programs and connecting to the internet. The rise of the internet also propagated the rise of client-server deployments. Now there could be centralized servers hosting applications, and services could be reached by anyone who had a connection to the internet anywhere on the globe. This was also a time when server technology gained prominence; Windows NT was released during this time and was soon followed by Windows 2000 and Windows 2003 at the turn of the century.

The most remarkable innovation of the 2000s was the rise and adoption of portable devices, especially smartphones, and with these came a plethora of apps. Apps could connect to centralized servers on the internet and carry out business as usual. Users were no longer dependent on browsers to do this work; all servers were either self-hosted or hosted using a service provider, such as an **internet service provider (ISP)**.

Users did not have much control over their servers. Multiple customers and their deployments were part of the same server, even without customers knowing about it.

However, something else happened in the middle and latter parts of the first decade of the 2000s. This was the rise of cloud computing, and it again rewrote the entire landscape of the IT industry. Initially, adoption was slow, and people approached it with caution, either because the cloud was in its infancy and still had to mature, or because people had various negative notions about what it was.

To gain a better understanding of the disruptive technology, we will cover the following topics in this chapter:

- Cloud computing
- **Infrastructure as a service (IaaS), platform as a service (PaaS), and Software as a service (SaaS)**
- Understanding Azure
- **Azure Resource Manager (ARM)**
- Virtualization, containers, and Docker
- Interacting with the intelligent cloud

Cloud computing

Today, cloud computing is one of the most promising upcoming technologies, and enterprises, no matter how big or small, are adopting it as a part of their IT strategy. It is difficult these days to have any meaningful conversation about an IT strategy without including cloud computing in the overall solution discussions.

Cloud computing, or simply the cloud in layman's terms, refers to the availability of resources on the internet. These resources are made available to users on the internet as services. For example, storage is available on-demand through the internet for users to store their files, documents, and more. Here, storage is a service that is offered by a cloud provider.

A cloud provider is an enterprise or consortium of companies that provides cloud services to other enterprises and consumers. They host and manage these services on behalf of the user. They are responsible for enabling and maintaining the health of services. There are large datacenters across the globe that have been opened by cloud providers to cater to the IT demands of users.

Cloud resources consist of hosting services on on-demand infrastructures, such as computing infrastructures, networks, and storage facilities. This flavor of the cloud is known as IaaS.

The advantages of cloud computing

Cloud adoption is at an all-time high and is growing because of several advantages, such as these:

- **Pay-as-you-go model:** Customers do not need to purchase hardware and software for cloud resources. There is no capital expenditure for using a cloud resource; customers simply pay for the time that they use or reserve a resource.
- **Global access:** Cloud resources are available globally through the internet. Customers can access their resources on-demand from anywhere.
- **Unlimited resources:** The scaling capability of cloud technology is unlimited; customers can provision as many resources as they want, without any constraints. This is also known as unlimited scalability.
- **Managed services:** The cloud provider provides numerous services that are managed by them for customers. This takes away any technical and financial burden from the customer.

Why cloud computing?

To understand the need for cloud computing, we must understand the industry's perspective.

Flexibility and agility

Instead of creating a large monolithic application using a big-bang approach deployment methodology, today, applications comprise smaller services using the microservices paradigm. Microservices help to create services in an independent and autonomous manner that can be evolved in isolation without bringing the entire application down. They offer large amounts of flexibility and agility in bringing changes to production in a faster and better way. There are many microservices that come together to create an application and provide integrated solutions for customers. These microservices should be discoverable and have well-defined endpoints for integration. The number of integrations with the microservices approach is very high compared to traditional monolithic applications. These integrations add complexity in both the development and deployment of applications.

Speed, standardization, and consistency

It follows that the methodology for deployments should also undergo changes to adapt to the needs of these services, that is, frequent changes and frequent deployments. For frequent changes and deployments, it is important to use processes that help in bringing about these changes in a predictable and consistent manner. Automated agile processes should be used such that smaller changes can be deployed and tested in isolation.

Staying relevant

Finally, deployment targets should be redefined. Not only should deployment targets be easily creatable within seconds, but also the environment built should be consistent across versions, with appropriate binaries, runtimes, frameworks, and configuration. Virtual machines were used with monolithic applications but microservices need more agility, flexibility, and a more lightweight option than virtual machines. Container technology is the preferred mechanism for deployment targets for these services, and we will cover more about that later in this chapter.

Scalability

Some important tenets of using microservices are that they have an unlimited scaling capability in isolation, global high availability, disaster recovery with a near-zero recovery point, and time objectives. These qualities of microservices necessitate infrastructure that can scale in an unlimited fashion. There should not be any resource constraints. While this is the case, it is also important that an organization does not pay for resources up front when they are not utilized.

Cost-effectiveness

Paying for resources that are being consumed and using them optimally by increasing and decreasing the resource counts and capacity automatically is the fundamental tenet of cloud computing. These emerging application requirements demand the cloud as the preferred platform to scale easily, be highly available, be disaster-resistant, bring in changes easily, and achieve predictable and consistent automated deployments in a cost-effective manner.

Deployment paradigms in Azure

There are three different deployment patterns that are available in Azure; they are as follows:

- IaaS
- PaaS
- SaaS

The difference between these three deployment patterns is the level of control that is exercised by customers via Azure. *Figure 1.1* displays the different levels of control within each of these deployment patterns:

IaaS	PaaS	SaaS
Applications	Applications	Applications
Data	Data	Data
Runtime	Runtime	Runtime
Middleware	Middleware	Middleware
OS	OS	OS
Virtualization	Virtualization	Virtualization
Servers	Servers	Servers
Storage	Storage	Storage
Networking	Networking	Networking

Managed by Consumer
Managed by Vendor

Figure 1.1: Cloud services—IaaS, PaaS, and SaaS

It is clear from *Figure 1.1* that customers have more control when using IaaS deployments, and this level of control continually decreases as we progress from PaaS to SaaS deployments.

IaaS

IaaS is a type of deployment model that allows customers to provision their own infrastructure on Azure. Azure provides several infrastructure resources and customers can provision them on-demand. Customers are responsible for maintaining and governing their own infrastructure. Azure will ensure the maintenance of the physical infrastructure on which these virtual infrastructure resources are hosted. Under this approach, customers require active management and operations in the Azure environment.

PaaS

PaaS takes away infrastructure deployment and control from the customer. This is a higher-level abstraction compared to IaaS. In this approach, customers bring their own application, code, and data, and deploy them on the Azure-provided platform. These platforms are managed and governed by Azure and customers are solely responsible for their applications. Customers perform activities related to their application deployment only. This model provides faster and easier options for the deployment of applications compared to IaaS.

SaaS

SaaS is a higher-level abstraction compared to PaaS. In this approach, software and its services are available for customer consumption. Customers only bring their data into these services—they do not have any control over these services. Now that we have a basic understanding of service types in Azure, let's get into the details of Azure and understand it from the ground up.

Understanding Azure

Azure provides all the benefits of the cloud while remaining open and flexible. Azure supports a wide variety of operating systems, languages, tools, platforms, utilities, and frameworks. For example, it supports Linux and Windows, SQL Server, MySQL, and PostgreSQL. It supports most of the programming languages, including C#, Python, Java, Node.js, and Bash. It supports NoSQL databases, such as MongoDB and Cosmos DB, and it also supports continuous integration tools, such as Jenkins and Azure DevOps Services (formerly **Visual Studio Team Services (VSTS)**). The whole idea behind this ecosystem is to enable customers to have the freedom to choose their own language, platform, operating system, database, storage, and tools and utilities. Customers should not be constrained from a technology perspective; instead, they should be able to build and focus on their business solution, and Azure provides them with a world-class technology stack that they can use.

Azure is very much compatible with the customer's choice of technology stack. For example, Azure supports all popular (open-source and commercial) database environments. Azure provides Azure SQL, MySQL, and Postgres PaaS services. It provides the Hadoop ecosystem and offers HDInsight, a 100% Apache Hadoop-based PaaS. It also provides a Hadoop on Linux **virtual machine (VM)** implementation for customers who prefer the IaaS approach. Azure also provides the Redis Cache service and supports other popular database environments, such as Cassandra, Couchbase, and Oracle as an IaaS implementation.

The number of services is increasing by the day in Azure and the most up-to-date list of services can be found at <https://azure.microsoft.com/services>.

Azure also provides a unique cloud computing paradigm known as the hybrid cloud. The hybrid cloud refers to a deployment strategy in which a subset of services is deployed on a public cloud, while other services are deployed on an on-premises private cloud or datacenter. There is a **virtual private network (VPN)** connection between the public and private clouds. Azure offers customers the flexibility to divide and deploy their workload on both the public cloud and an on-premises datacenter.

Azure has datacenters across the globe and combines these datacenters into regions. Each region has multiple datacenters to ensure that recovery from disasters is quick and efficient. At the time of writing, there are 58 regions across the globe. This provides customers with the flexibility to deploy their services in their choice of location. They can also combine these regions to deploy a solution that is disaster-resistant and deployed near their customer base.

Note

In China and Germany, the Azure Cloud Services are separate for general use and for governmental use. This means that the cloud services are maintained in separate datacenters.

Azure as an intelligent cloud

Azure provides infrastructure and services to ingest billions of transactions using hyper-scale processing. It provides petabytes of storage for data, and it provides a host of interconnected services that can pass data among themselves. With such capabilities in place, data can be processed to generate meaningful knowledge and insights. There are multiple types of insights that can be generated through data analysis, which are as follows:

- **Descriptive:** This type of analysis provides details about what is happening or has happened in the past.
- **Predictive:** This type of analysis provides details about what is going to happen in the future.
- **Prescriptive:** This type of analysis provides details about what should be done to either enhance or prevent current or future events.
- **Cognitive:** This type of analysis actually executes actions that are determined by prescriptive analytics in an automated manner.

While deriving insights from data is good, it is equally important to act on them. Azure provides a rich platform to ingest large volumes of data, process and transform it, store and generate insights from it, and display it on real-time dashboards. It is also possible to take action on these insights automatically. These services are available to every customer of Azure and they provide a rich ecosystem in which customers can create solutions. Enterprises are creating numerous applications and services that are completely disrupting industries because of the easy availability of these intelligent services from Azure, which are combined to create meaningful value for end customers. Azure ensures that services that are commercially not viable to implement for small and medium companies can now be readily consumed and deployed in a few minutes.

Azure Resource Manager

Azure Resource Manager (ARM) is the technology platform and orchestration service from Microsoft that ties up all the components that were discussed earlier. It brings Azure's resource providers, resources, and resource groups together to form a cohesive cloud platform. It makes Azure services available as subscriptions, resource types available to resource groups, and resource and resource APIs accessible to the portal and other clients, and it authenticates access to these resources. It also enables features such as tagging, authentication, **role-based access control (RBAC)**, resource locking, and policy enforcement for subscriptions and their resource groups. It also provides deployment and management features using the Azure portal, Azure PowerShell, and **command-line interface (CLI)** tools.

The ARM architecture

The architecture of ARM and its components is shown in Figure 1.2. As we can see, an **Azure Subscription** comprises multiple resource groups. Each resource group contains resource instances that are created from resource types that are available in the resource provider:

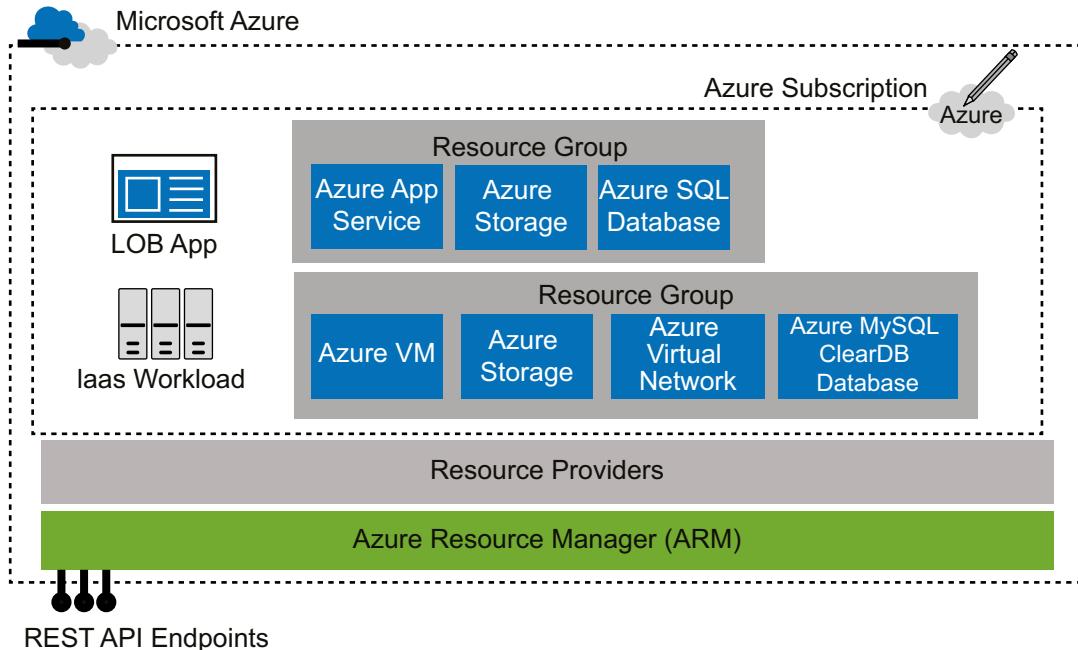


Figure 1.2: The ARM architecture

Why ARM?

Prior to ARM, the framework used by Azure was known as **Azure Service Manager (ASM)**. It is important to have a small introduction to it so that we can get a clear understanding of the emergence of ARM and the slow and steady deprecation of ASM.

Limitations of ASM

ASM has inherent constraints. For example, ASM deployments are slow and blocking—operations are blocked if an earlier operation is already in progress. Some of the limitations of ASM are as follows:

- **Parallelism:** Parallelism is a challenge in ASM. It is not possible to execute multiple transactions successfully in parallel. The operations in ASM are linear and so they are executed one after another. If multiple transactions are executed at the same time, there will either be parallel operation errors or the transactions will get blocked.
- **Resources:** Resources in ASM are provisioned and managed in isolation of each other; there is no relation between ASM resources. Grouping services and resources or configuring them together is not possible.
- **Cloud services:** Cloud services are the units of deployment in ASM. They are reliant on affinity groups and are not scalable due to their design and architecture.

Granular and discrete roles and permissions cannot be assigned to resources in ASM. Customers are either service administrators or co-administrators in the subscription. They either get full control over resources or do not have access to them at all. ASM provides no deployment support. Either deployments are done manually, or we need to resort to writing procedural scripts in .NET or PowerShell. ASM APIs are not consistent between resources.

ARM advantages

ARM provides distinct advantages and benefits over ASM, which are as follows:

- **Grouping:** ARM allows the grouping of resources together in a logical container. These resources can be managed together and go through a common life cycle as a group. This makes it easier to identify related and dependent resources.
- **Common life cycles:** Resources in a group have the same life cycle. These resources can evolve and be managed together as a unit.
- **RBAC:** Granular roles and permissions can be assigned to resources providing discrete access to customers. Customers can also have only those rights that are assigned to them.

- **Deployment support:** ARM provides deployment support in terms of templates, enabling DevOps and **infrastructure as code (IaC)**. These deployments are faster, consistent, and predictable.
- **Superior technology:** The cost and billing of resources can be managed as a unit. Each resource group can provide its usage and cost information.
- **Manageability:** ARM provides advanced features, such as security, monitoring, auditing, and tagging, for better manageability of resources. Resources can be queried based on tags. Tags also provide cost and billing information for resources that are tagged similarly.
- **Migration:** Migration and updating resources is easier within and across resource groups.

ARM concepts

With ARM, everything in Azure is a resource. Examples of resources are VMs, network interfaces, public IP addresses, storage accounts, and virtual networks. ARM is based on concepts that are related to resource providers and resource consumers. Azure provides resources and services through multiple resource providers that are consumed and deployed in groups.

Resource providers

These are services that are responsible for providing resource types through ARM. The top-level concept in ARM is the resource provider. These providers are containers for resource types. Resource types are grouped into resource providers. They are responsible for deploying and managing resources. For example, a VM resource type is provided by a resource provider called **Microsoft.Compute/virtualMachines** resource. **Representational state transfer (REST)** API operations are versioned to distinguish between them. The version naming is based on the dates on which they are released by Microsoft. It is necessary for a related resource provider to be available to a subscription to deploy a resource. Not all resource providers are available to a subscription out of the box. If a resource is not available to a subscription, then we need to check whether the required resource provider is available in each region. If it is available, the customer can explicitly register for the subscription.

Resource types

Resource types are an actual resource specification defining the resource's public API interface and implementation. They implement the working and operations supported by the resource. Similar to resource providers, resource types also evolve over time in terms of their internal implementation, and there are multiple versions of their schemas and public API interfaces. The version names are based on the dates that they are released by Microsoft as a preview or **general availability (GA)**. The resource types become available as a subscription after a resource provider is registered to them. Also, not every resource type is available in every Azure region. The availability of a resource is dependent on the availability and registration of a resource provider in an Azure region and must support the API version needed for provisioning it.

Resource groups

Resource groups are units of deployment in ARM. They are containers grouping multiple resource instances in a security and management boundary. A resource group is uniquely named in a subscription. Resources can be provisioned on different Azure regions and yet belong to the same resource group. Resource groups provide additional services to all the resources within them. Resource groups provide metadata services, such as tagging, which enables the categorization of resources; the policy-based management of resources; RBAC; the protection of resources from accidental deletion or updates; and more. As mentioned before, they have a security boundary, and users that don't have access to a resource group cannot access resources contained within it. Every resource instance needs to be part of a resource group; otherwise, it cannot be deployed.

Resources and resource instances

Resources are created from resource types and are an instance of a resource type. An instance can be unique globally or at a resource group level. The uniqueness is defined by both the name of the resource and its type. If we compare this with object-oriented programming constructs, resource instances can be seen as objects and resource types can be seen as classes. The services are consumed through the operations that are supported and implemented by resource instances. The resource type defines properties and each instance should configure mandatory properties during the provisioning of an instance. Some are mandatory properties, while others are optional. They inherit the security and access configuration from their parent resource group. These inherited permissions and role assignments can be overridden for each resource. A resource can be locked in such a way that some of its operations can be blocked and not made available to roles, users, and groups even though they have access to it. Resources can be tagged for easy discoverability and manageability.

ARM features

Here are some of the main features that are provided by ARM:

- **RBAC: Azure Active Directory (Azure AD)** authenticates users to provide access to subscriptions, resource groups, and resources. ARM implements OAuth and RBAC within the platform, enabling authorization and access control for resources, resource groups, and subscriptions based on roles assigned to a user or group. A permission defines access to the operations in a resource. These permissions can allow or deny access to the resource. A role definition is a collection of these permissions. Roles map Azure AD users and groups to particular permissions. Roles are subsequently assigned to a scope; this can be an individual, a collection of resources, a resource group, or the subscription. The Azure AD identities (users, groups, and service principals) that are added to a role gain access to the resource according to the permissions defined in the role. ARM provides multiple out-of-the-box roles. It provides system roles, such as the **owner**, **contributor**, and **reader**. It also provides resource-based roles, such as SQL DB contributor and VM contributor. ARM also allows the creation of custom roles.
- **Tags:** Tags are name-value pairs that add additional information and metadata to resources. Both resources and resource groups can be tagged with multiple tags. Tags help in the categorization of resources for better discoverability and manageability. Resources can be quickly searched for and easily identified. Billing and cost information can also be fetched for resources that have the same tags. While this feature is provided by ARM, an IT administrator defines its usage and taxonomy with regard to resources and resource groups. Taxonomy and tags, for example, can relate to departments, resource usage, location, projects, or any other criteria that are deemed fit from a cost, usage, billing, or search perspective. These tags can then be applied to resources. Tags that are defined at the resource group level are not inherited by their resources.
- **Policies:** Another security feature that is provided by ARM is custom policies. Custom policies can be created to control access to resources. Policies are defined as conventions and rules, and they must be adhered to while interacting with resources and resource groups. The policy definition contains an explicit denial of actions on resources or access to resources. By default, every access is allowed if it is not mentioned in the policy definition. These policy definitions are assigned to the resource, resource group, and subscription scope. It is important to note that these policies are not replacements or substitutes for RBAC. In fact, they complement and work together with RBAC. Policies are evaluated after a user is authenticated by Azure AD and authorized by the RBAC service. ARM provides a JSON-based policy definition language for defining policies. Some examples of policy definitions are that a policy must tag every provisioned resource, and resources can only be provisioned to specific Azure regions.

- **Locks:** Subscriptions, resource groups, and resources can be locked to prevent accidental deletions or updates by an authenticated user. Locks applied at higher levels flow downstream to the child resources. Alternatively, locks that are applied at the subscription level lock every resource group and the resources within it.
- **Multi-region:** Azure provides multiple regions for provisioning and hosting resources. ARM allows resources to be provisioned at different locations while still residing within the same resource group. A resource group can contain resources from different regions.
- **Idempotent:** This feature ensures predictability, standardization, and consistency in resource deployment by ensuring that every deployment will result in the same state of resources and configuration, no matter the number of times it is executed.
- **Extensible:** ARM provides an extensible architecture to allow the creation and plugging in of new resource providers and resource types on the platform.

Virtualization

Virtualization was a breakthrough innovation that completely changed the way that physical servers were looked at. It refers to the abstraction of a physical object into a logical object.

The virtualization of physical servers led to virtual servers known as VMs. These VMs consume and share the physical CPU, memory, storage, and other hardware of the physical server on which they are hosted. This enables the faster and easier provisioning of application environments on-demand, providing high availability and scalability with reduced cost. One physical server is enough to host multiple VMs, with each VM containing its own operating system and hosting services on it.

There was no longer any need to buy additional physical servers for deploying new applications and services. The existing physical servers were sufficient to host more VMs. Furthermore, as part of rationalization, many physical servers were consolidated into a few with the help of virtualization.

Each VM contains the entire operating system, and each VM is completely isolated from other VMs, including the physical hosts. Although a VM uses the hardware that is provided by the host physical server, it has full control over its assigned resources and its environment. These VMs can be hosted on a network such as a physical server with its own identity.

Azure can create Linux and Windows VMs in a few minutes. Microsoft provides its own images, along with images from its partners and the community; users can also provide their own images. VMs are created using these images.

Containers

Containers are also a virtualization technology; however, they do not virtualize a server. Instead, a container is operating system-level virtualization. What this means is that containers share the operating system kernel (which is provided by the host) among themselves along with the host. Multiple containers running on a host (physical or virtual) share the host operating system kernel. Containers ensure that they reuse the host kernel instead of each having a dedicated kernel to themselves.

Containers are completely isolated from their host or from other containers running on the host. Windows containers use Windows storage filter drivers and session isolation to isolate operating system services such as the file system, registry, processes, and networks. The same is true even for Linux containers running on Linux hosts. Linux containers use the Linux namespace, control groups, and union file system to virtualize the host operating system.

The container appears as if it has a completely new and untouched operating system and resources. This arrangement provides lots of benefits, such as the following:

- Containers are fast to provision and take less time to provision compared to virtual machines. Most of the operating system services in a container are provided by the host operating system.
- Containers are lightweight and require fewer computing resources than VMs. The operating system resource overhead is no longer required with containers.
- Containers are much smaller than VMs.
- Containers can help solve problems related to managing multiple application dependencies in an intuitive, automated, and simple manner.
- Containers provide infrastructure in order to define all application dependencies in a single place.

Containers are an inherent feature of Windows Server 2016 and Windows 10; however, they are managed and accessed using a Docker client and a Docker daemon. Containers can be created on Azure with a Windows Server 2016 SKU as an image. Each container has a single main process that must be running for the container to exist. A container will stop when this process ends. Additionally, a container can either run in interactive mode or in detached mode like a service:

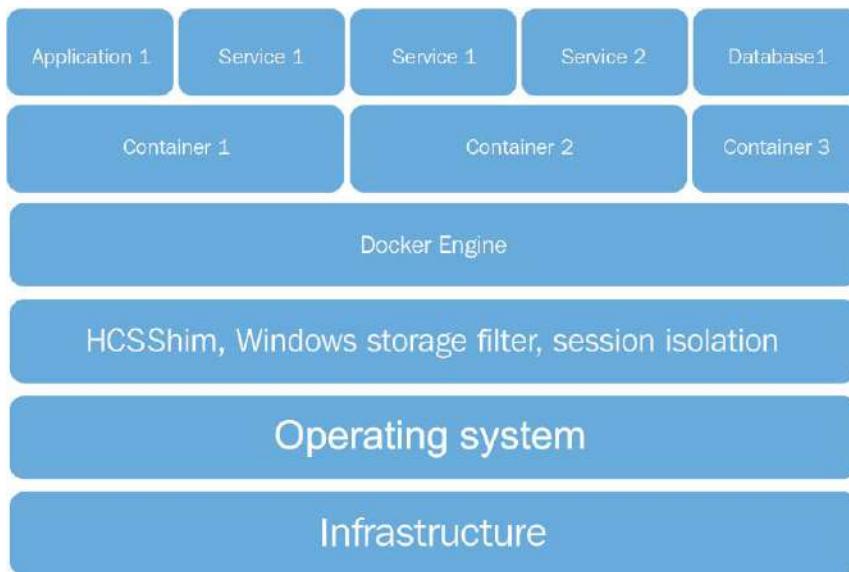


Figure 1.3: Container architecture

Figure 1.3 shows all the technical layers that enable containers. The bottom-most layer provides the core infrastructure in terms of network, storage, load balancers, and network cards. At the top of the infrastructure is the compute layer, consisting of either a physical server or both physical and virtual servers on top of a physical server. This layer contains the operating system with the ability to host containers. The operating system provides the execution driver that the layers above use to call the kernel code and objects to execute containers. Microsoft created **Host Container System Shim (HCSShim)** for managing and creating containers and uses Windows storage filter drivers for image and file management.

Container environment isolation is enabled for the Windows session. Windows Server 2016 and Nano Server provide the operating system, enable the container features, and execute the user-level Docker client and Docker Engine. Docker Engine uses the services of HCSShim, storage filter drivers, and sessions to spawn multiple containers on the server, with each containing a service, application, or database.

Docker

Docker provides management features to Windows containers. It comprises the following two executables:

- The Docker daemon
- The Docker client

The Docker daemon is the workhorse for managing containers. It is a Windows service responsible for managing all activities on the host that are related to containers. The Docker client interacts with the Docker daemon and is responsible for capturing inputs and sending them across to the Docker daemon. The Docker daemon provides the runtime, libraries, graph drivers, and engine to create, manage, and monitor containers and images on the host server. It also has the ability to create custom images that are used for building and shipping applications to multiple environments.

Interacting with the intelligent cloud

Azure provides multiple ways to connect, automate, and interact with the intelligent cloud. All these methods require users to be authenticated with valid credentials before they can be used. The different ways to connect to Azure are as follows:

- The Azure portal
- PowerShell
- The Azure CLI
- The Azure REST API

The Azure portal

The Azure portal is a great place to get started. With the Azure portal, users can log in and start creating and managing Azure resources manually. The portal provides an intuitive and user-friendly user interface through the browser. The Azure portal provides an easy way to navigate to resources using **blades**. The blades display all the properties of a resource, including its logs, cost, relationship with other resources, tags, security options, and more. An entire cloud deployment can be managed through the portal.

PowerShell

PowerShell is an object-based command-line shell and scripting language that is used for the administration, configuration, and management of infrastructure and environments. It is built on top of .NET Framework and provides automation capabilities. PowerShell has truly become a first-class citizen among IT administrators and automation developers for managing and controlling the Windows environment. Today, almost every Windows environment and many Linux environments can be managed by PowerShell. In fact, almost every aspect of Azure can also be managed by PowerShell. Azure provides rich support for PowerShell. It provides a PowerShell module for each resource provider containing hundreds of cmdlets. Users can use these cmdlets in their scripts to automate interaction with Azure. The Azure PowerShell module is available through the web platform installer and through the **PowerShell Gallery**. Windows Server 2016 and Windows 10 provide package management and **PowerShellGet** modules for the quick and easy downloading and installation of PowerShell modules from the PowerShell Gallery. The **PowerShellGet** module provides the **Install-Module** cmdlet for downloading and installing modules on the system.

Installing a module is a simple act of copying the module files at well-defined module locations, which can be done as follows:

```
Import-Module PowerShellGet  
Install-Module -Name az -verbose
```

The **Import-module** command imports a module and its related functions within the current execution scope and **Install-Module** helps in installing modules.

The Azure CLI

Azure also provides Azure CLI 2.0, which can be deployed on Linux, Windows, and macOS operating systems. Azure CLI 2.0 is Azure's new command-line utility for managing Azure resources. Azure CLI 2.0 is optimized for managing and administering Azure resources from the command line, and for building automation scripts that work against ARM. The CLI can be used to execute commands using the Bash shell or the Windows command line. The Azure CLI is very famous among non-Windows users as it allows you to talk to Azure on Linux and macOS. The steps for installing Azure CLI 2.0 are available at <https://docs.microsoft.com/cli/azure/install-azure-cli?view=azure-cli-latest>.

The Azure REST API

All Azure resources are exposed to users through REST endpoints. REST APIs are service endpoints that implement HTTP operations (or methods) by providing **create**, **retrieve**, **update**, or **delete** (**CRUD**) access to the service's resources. Users can consume these APIs to create and manage resources. In fact, the CLI and PowerShell mechanisms use these REST APIs internally to interact with resources on Azure.

ARM templates

In an earlier section, we looked at deployment features such as multi-service, multi-region, extensible, and idempotent features that are provided by ARM. ARM templates are the primary means of provisioning resources in ARM. ARM templates provide implementation support for ARM's deployment features.

ARM templates provide a declarative model through which resources, their configuration, scripts, and extensions are specified. ARM templates are based on the **JavaScript Object Notation (JSON)** format. They use JSON syntax and conventions to declare and configure resources. JSON files are text-based, user-friendly, and easily readable files.

They can be stored in a source code repository and have version control. They are also a means to represent IaC that can be used to provision resources in an Azure resource group again and again, predictably and uniformly. A template needs a resource group for deployment. It can only be deployed to a resource group, and the resource group should exist before executing a template deployment. A template is not capable of creating a resource group.

Templates provide the flexibility to be generic and modular in their design and implementation. Templates provide the ability to accept parameters from users, declare internal variables, define dependencies between resources, link resources within the same resource group or different resource groups, and execute other templates. They also provide scripting language type expressions and functions that make them dynamic and customizable at runtime.

Deployments

PowerShell allows the following two modes for the deployment of templates:

- **Incremental:** Incremental deployment adds resources declared in the template that don't exist in a resource group, leaves resources unchanged in a resource group that is not part of a template definition, and leaves resources unchanged in a resource group that exists in both the template and resource group with the same configuration state.
- **Complete:** Complete deployment, on the other hand, adds resources declared in a template to the resource group, deletes resources that do not exist in the template from the resource group, and leaves resources unchanged that exist in both the resource group and template with the same configuration state.

Summary

The cloud is a relatively new paradigm and is still in its nascent stage. There will be a lot of innovation and capabilities added over time. Azure is one of the top cloud providers today and it provides rich capabilities through IaaS, PaaS, SaaS, and hybrid deployments. In fact, Azure Stack, which is an implementation of the private cloud from Microsoft, will be released soon. This will have the same features available on a private cloud as on the public cloud. They both will, in fact, connect and work seamlessly and transparently together.

It is very easy to get started with Azure, but developers and architects can also fall into a trap if they do not design and architect their solutions appropriately. This book is an attempt to provide guidance and directions for architecting solutions the right way, using appropriate services and resources. Every service on Azure is a resource. It is important to understand how these resources are organized and managed in Azure. This chapter provided context around ARM and groups—which are the core frameworks that provide the building blocks for resources. ARM offers a set of services to resources that help provide uniformity, standardization, and consistency in managing them. The services, such as RBAC, tags, policies, and locks, are available to every resource provider and resource. Azure also provides rich automation features to automate and interact with resources. Tools such as PowerShell, ARM templates, and the Azure CLI can be incorporated as part of release pipelines, continuous deployment, and delivery. Users can connect to Azure from heterogeneous environments using these automation tools.

The next chapter will discuss some of the important architectural concerns that help to solve common cloud-based deployment problems and ensure applications are secure, available, scalable, and maintainable in the long run.

2

Azure solution availability, scalability, and monitoring

Architectural concerns, such as high availability and scalability, are some of the highest-priority items for any architect. This is common across many projects and solutions. However, this becomes even more important when deploying applications to the cloud because of the complexity involved. Most of the time, the complexity does not come from the application, but from the choices available in terms of similar resources on the cloud. The other complex issue that arises from the cloud is the constant availability of new features. These new features can almost make an architect's decisions completely redundant in hindsight.

In this chapter, we will look at an architect's perspective in terms of deploying highly available and scalable applications on Azure.

Azure is a mature platform that provides a number of options for implementing high availability and scalability at multiple levels. It is vital for an architect to know about them, including the differences between them and the costs involved, and finally, be in a position to choose an appropriate solution that meets the best solution requirements. There is no one solution for everything, but there is a good one for each project.

Running applications and systems that are available to users for consumption whenever they need them is one of the topmost priorities for organizations. They want their applications to be operational and functional, and to continue to be available to their customers even when some untoward events occur. High availability is the primary theme of this chapter. Keeping the lights on is the common metaphor that is used for high availability. Achieving high availability for applications is not an easy task, and organizations have to spend considerable time, energy, resources, and money in doing so. Additionally, there is still the risk that an organization's implementation will not produce the desired results. Azure provides a lot of high-availability features for **virtual machines (VMs)** and the **Platform as a Service (PaaS)** service. In this chapter, we will go through the architectural and design features that are provided by Azure to ensure high availability for running applications and services.

In this chapter, we will cover the following topics:

- High availability
- Azure high availability
- Architectural considerations for high availability
- Scalability
- Upgrades and maintenance

High availability

High availability forms one of the core non-functional technical requirements for any business-critical service and its deployment. High availability refers to the feature of a service or application that keeps it operational on a continuous basis; it does so by meeting or surpassing its promised **service level agreement (SLA)**. Users are promised a certain SLA based on the service type. The service should be available for consumption based on its SLA. For example, an SLA can define 99% availability for an application for the entire year. This means that it should be available for consumption by users for 361.35 days. If it fails to remain available for this period, that constitutes a breach of the SLA. Most mission-critical applications define their high-availability SLA as 99.999% for a year. This means the application should be up, running, and available throughout the year, but it can only be down and unavailable for 5.2 hours. If the downtime goes beyond that, you are eligible for credit, which will be calculated based on the total uptime percentage.

It is important to note here that high availability is defined in terms of time (yearly, monthly, weekly, or a combination of these).

A service or application is made up of multiple components and these components are deployed on separate tiers and layers. Moreover, a service or application is deployed on an **operating system (OS)** and hosted on a physical machine or VM. It consumes network and storage services for various purposes. It might even be dependent on external systems. For these services or applications to be highly available, it is important that networks, storage, OSes, VMs or physical machines, and each component of the application is designed with the SLA and high availability in mind. A definite application life cycle process is used to ensure that high availability should be baked in from the start of application planning until its introduction to operations. This also involves introducing redundancy. Redundant resources should be included in the overall application and deployment architecture to ensure that if one resource goes down, another takes over and serves the requests of the customer.

Some of the major factors affecting the high availability of an application are as follows:

- Planned maintenance
- Unplanned maintenance
- Application deployment architecture

We will be looking into each of these factors in the following sections. Let's take a closer look at how high availability is ensured for deployments in Azure.

Azure high availability

Achieving high availability and meeting high SLA requirements is tough. Azure provides lots of features that enable high availability for applications, from the host and guest OS to applications using its PaaS. Architects can use these features to get high availability in their applications using configuration instead of building these features from scratch or depending on third-party tools.

In this section, we will look at the features and capabilities provided by Azure to make applications highly available. Before we get into the architectural and configuration details, it is important to understand concepts related to Azure's high availability.

Concepts

The fundamental concepts provided by Azure to attain high availability are as follows:

- Availability sets
- The fault domain
- The update domain
- Availability zones

As you know, it's very important that we design solutions to be highly available. The workloads might be mission-critical and require highly available architecture. We will take a closer look at each of the concepts of high availability in Azure now. Let's start with availability sets.

Availability sets

High availability in Azure is primarily achieved through redundancy. Redundancy means that there is more than one resource instance of the same type that takes control in the event of a primary resource failure. However, just having more similar resources does not make them highly available. For example, there could be multiple VMs provisioned within a subscription, but simply having multiple VMs does not make them highly available. Azure provides a resource known as an availability set, and having multiple VMs associated with it makes them highly available. A minimum of two VMs should be hosted within the availability set to make them highly available. All VMs in the availability set become highly available because they are placed on separate physical racks in the Azure datacenter. During updates, these VMs are updated one at a time, instead of all at the same time. Availability sets provide a fault domain and an update domain to achieve this, and we will discuss this more in the next section. In short, availability sets provide redundancy at the datacenter level, similar to locally redundant storage.

It is important to note that availability sets provide high availability within a datacenter. If an entire datacenter is down, then the availability of the application will be impacted. To ensure that applications are still available when a datacenter goes down, Azure has introduced a new feature known as availability zones, which we will learn about shortly.

If you recall the list of fundamental concepts, the next one in the list is the fault domain. The fault domain is often denoted by the acronym FD. In the next section, we will discuss what the FD is and how it is relevant while designing highly available solutions.

The fault domain

Fault domains (FDs) represent a group of VMs that share a common power source and network switch. When a VM is provisioned and assigned to an availability set, it is hosted within an FD. Each availability set has either two or three FDs by default, depending on the Azure region. Some regions provide two, while others provide three FDs in an availability set. FDs are non-configurable by users.

When multiple VMs are created, they are placed on separate FDs. If the number of VMs is more than the FDs, the additional VMs are placed on existing FDs. For example, if there are five VMs, there will be FDs hosted on more than one VM.

FDs are related to physical racks in the Azure datacenter. FDs provide high availability in the case of unplanned downtime due to hardware, power, and network failure. Since each VM is placed on a different rack with different hardware, a different power supply, and a different network, other VMs continue running if a rack snaps off.

The next one in the list is the update domain.

The update domain

An FD takes care of unplanned downtime, while an update domain handles downtime from planned maintenance. Each VM is also assigned an update domain and all the VMs within that update domain will reboot together. There can be as many as 20 update domains in a single availability set. Update domains are non-configurable by users. When multiple VMs are created, they are placed on separate update domains. If more than 20 VMs are provisioned on an availability set, they are placed in a round-robin fashion on these update domains. Update domains take care of planned maintenance. From **Service Health** in the Azure portal, you can check the planned maintenance details and set alerts.

In the next section, we will be covering availability zones.

Availability zones

This is a relatively new concept introduced by Azure and is very similar to zone redundancy for storage accounts. Availability zones provide high availability within a region by placing VM instances on separate datacenters within the region. Availability zones are applicable to many resources in Azure, including VMs, managed disks, VM scale sets, and load balancers. The complete list of resources that are supported by availability zones can be found at <https://docs.microsoft.com/azure/availability-zones/az-overview#services-that-support-availability-zones>. Being unable to configure availability across zones was a gap in Azure for a long time, and it was eventually fixed with the introduction of availability zones.

Each Azure region comprises multiple datacenters equipped with independent power, cooling, and networking. Some regions have more datacenters, while others have less. These datacenters within the region are known as zones. To ensure resiliency, there's a minimum of three separate zones in all enabled regions. Deploying VMs in an availability zone ensures that these VMs are in different datacenters and are on different racks and networks. These datacenters in a region relate to high-speed networks and there is no lag in communication between these VMs. Figure 2.1 shows how availability zones are set up in a region:

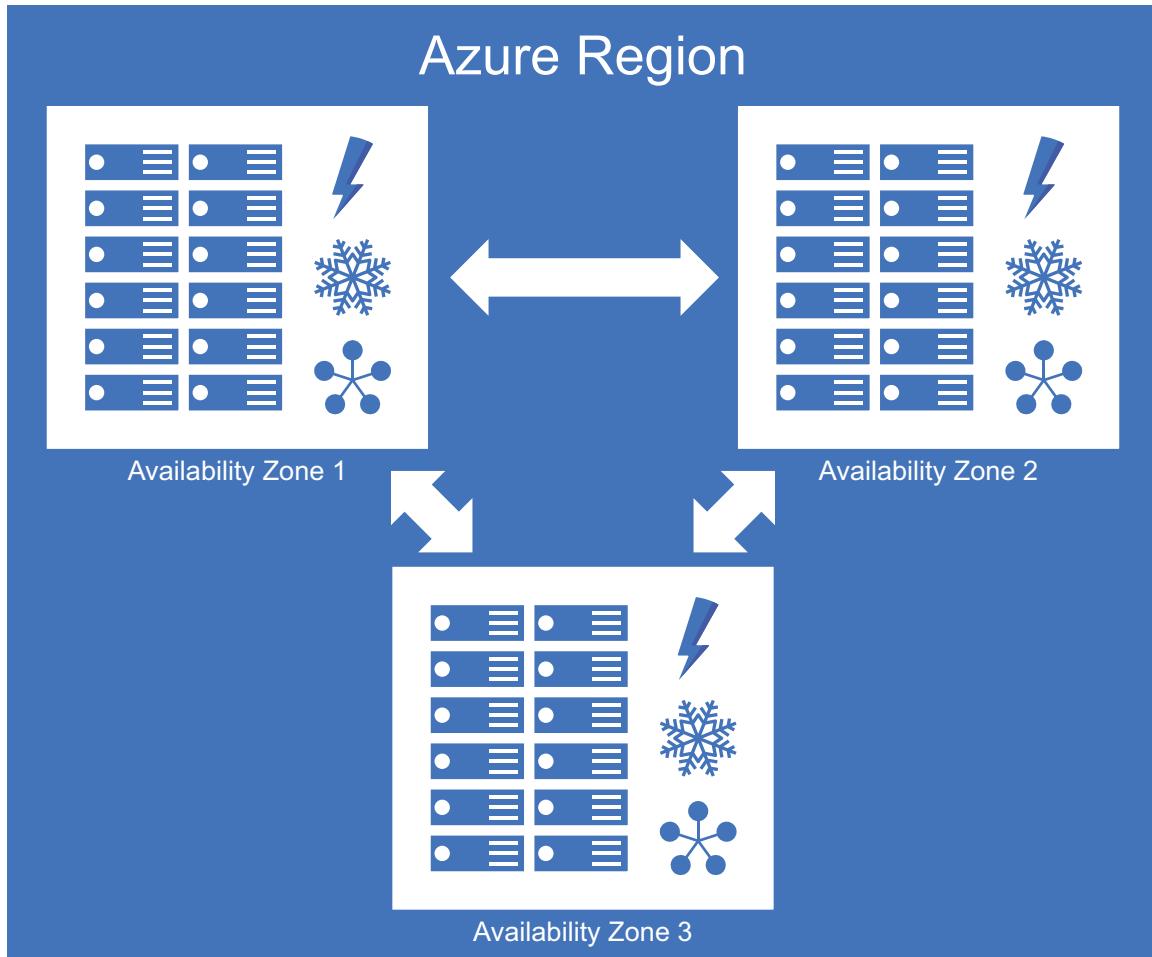


Figure 2.1: Availability zones in a region

You can find more information about availability zones at <https://docs.microsoft.com/azure/availability-zones/az-overview>.

Zone-redundant services replicate your applications and data across availability zones to protect from single points of failure.

If an application needs higher availability and you want to ensure that it is available even if an entire Azure region is down, the next rung of the ladder for availability is the Traffic Manager feature, which will be discussed later in this chapter. Let's now move on to understanding Azure's take on load balancing for VMs.

Load balancing

Load balancing, as the name suggests, refers to the process of balancing a load among VMs and applications. With one VM, there is no need for a load balancer because the entire load is on a single VM and there is no other VM to share the load. However, with multiple VMs containing the same application and service, it is possible to distribute the load among them through load balancing. Azure provides a few resources to enable load balancing:

- **Load balancers:** The Azure load balancer helps to design solutions with high availability. Within the **Transmission Control Protocol (TCP)** stack, it is a layer 4 transport-level load balancer. This is a layer 4 load balancer that distributes incoming traffic among healthy instances of services that are defined in a load-balanced set. Level 4 load balancers work at the transport level and have network-level information, such as an IP address and port, to decide the target for the incoming request. Load balancers are discussed in more detail later in this chapter.
- **Application gateways:** An Azure Application Gateway delivers high availability to your applications. They are layer 7 load balancers that distribute the incoming traffic among healthy instances of services. Level 7 load balancers can work at the application level and have application-level information, such as cookies, HTTP, HTTPS, and sessions for the incoming request. Application gateways are discussed in more detail later in this chapter. Application gateways are also used when deploying Azure Kubernetes Service, specifically for scenarios in which ingress traffic from the internet should be routed to the Kubernetes services in the cluster.
- **Azure Front Door:** Azure Front Door is very similar to application gateways; however, it does not work at the region or datacenter level. Instead, it helps in routing requests across regions globally. It has the same feature set as that provided by application gateways, but at the global level. It also provides a web application firewall for the filtering of requests and provides other security-related protection. It provides session affinity, TLS termination, and URL-based routing as some of its features.
- **Traffic Manager:** Traffic Manager helps in the routing of requests at the global level across multiple regions based on the health and availability of regional endpoints. It supports doing so using DNS redirect entries. It is highly resilient and has no service impact during region failures as well.

Since we've explored the methods and services that can be used to achieve load balancing, we'll go ahead and discuss how to make VMs highly available.

VM high availability

VMs provide compute capabilities. They provide processing power and hosting for applications and services. If an application is deployed on a single VM and that machine is down, then the application will not be available. If the application is composed of multiple tiers and each tier is deployed in its own single instance of a VM, even downtime for a single instance of VM can render the entire application unavailable. Azure tries to make even single VM instances highly available for 99.9% of the time, particularly if these single-instance VMs use premium storage for their disks. Azure provides a higher SLA for those VMs that are grouped together in an availability set. It provides a 99.95% SLA for VMs that are part of an availability set with two or more VMs. The SLA is 99.99% if VMs are placed in availability zones. In the next section, we will be discussing high availability for compute resources.

Compute high availability

Applications demanding high availability should be deployed on multiple VMs in the same availability set. If applications are composed of multiple tiers, then each tier should have a group of VMs on their dedicated availability set. In short, if there are three tiers of an application, there should be three availability sets and a minimum of six VMs (two in each availability set) to make the entire application highly available.

So, how does Azure provide an SLA and high availability to VMs in an availability set with multiple VMs in each availability set? This is the question that might come to mind for you.

Here, the use of concepts that we considered before comes into play—that is, the fault and update domains. When Azure sees multiple VMs in an availability set, it places those VMs on a separate FD. In other words, these VMs are placed on separate physical racks instead of the same rack. This ensures that at least one VM continues to be available even if there is a power, hardware, or rack failure. There are two or three FDs in an availability set and, depending on the number of VMs in an availability set, the VMs are placed in separate FDs or repeated in a round-robin fashion. This ensures that high availability is not impacted because of the failure of the rack.

Azure also places these VMs on a separate update domain. In other words, Azure tags these VMs internally in such a way that these VMs are patched and updated one after another, such that any reboot in an update domain does not affect the availability of the application. This ensures that high availability is not impacted because of the VM and host maintenance. It is important to note that Azure is not responsible for OS-level and application maintenance.

With the placement of VMs in separate fault and update domains, Azure ensures that all VMs are never down at the same time and that they are alive and available for serving requests, even though they might be undergoing maintenance or facing physical downtime challenges:

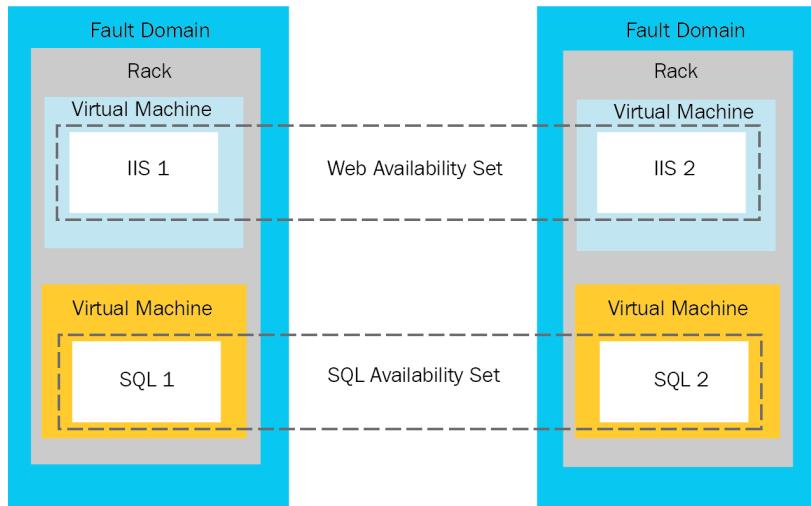


Figure 2.2: VM distribution across fault and update domains

Figure 2.2 shows four VMs (two have **Internet Information Services (IIS)** and the other two have SQL Server installed on them). Both the IIS and SQL VMs are part of availability sets. The IIS and SQL VMs are in separate FDs and different racks in the datacenter. They are also in separate update domains.

Figure 2.3 shows the relationship between fault and update domains:

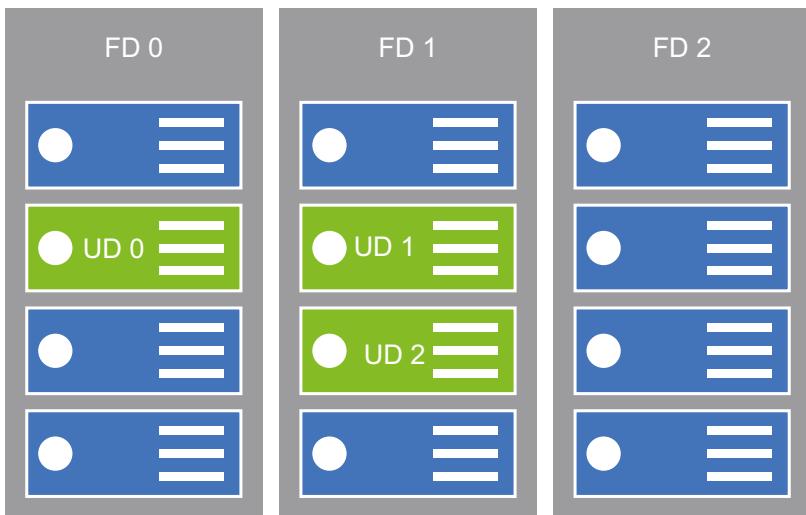


Figure 2.3: Layout of update domains and FDs in an availability set

So far, we have discussed achieving high availability for compute resources. In the next section, you will learn how high availability can be implemented for PaaS.

High-availability platforms

Azure has provided a lot of new features to ensure high availability for PaaS. Some of them are listed here:

- Containers in app services
- Azure Container Instances groups
- Azure Kubernetes Service
- Other container orchestrators, such as DC/OS and Swarm

Another important platform that brings high availability is **Service Fabric**. Both Service Fabric and container orchestrators that include Kubernetes ensure that the desired number of application instances are always up and running in an environment. What this means is that even if one of the instances goes down in the environment, the orchestrator will know about it by means of active monitoring and will spin up a new instance on a different node, thereby maintaining the ideal and desired number of instances. It does this without any manual or automated interference from the administrator.

While Service Fabric allows any type of application to become highly available, orchestrators such as Kubernetes, DC/OS, and Swarm are specific to containers. Also, it is important to understand that these platforms provide features that help in rolling updates, rather than a big bank update that might affect the availability of the application.

When we were discussing high availability for VMs, we took a brief look at what load balancing is. Let's take a closer look at it to better understand how it works in Azure.

Load balancers in Azure

Azure provides two resources that have the functionality of a load balancer. It provides a level 4 load balancer, which works at the transport layer within the TCP OSI stack, and a level 7 load balancer (application gateway), which works at the application and session levels.

Although both application gateways and load balancers provide the basic features of balancing a load, they serve different purposes. There are a number of use cases in which it makes more sense to deploy an application gateway than a load balancer.

An application gateway provides the following features that are not available with Azure load balancers:

- **Web application firewall:** This is an additional firewall on top of the OS firewall and it gives the ability to peek into incoming messages. This helps in identifying and preventing common web-based attacks, such as SQL injection, cross-site scripting attacks, and session hijacks.
- **Cookie-based session affinity:** Load balancers distribute incoming traffic to service instances that are healthy and relatively free. A request can be served by any service instance. However, there are applications that need advanced features in which all subsequent requests following the first request should be processed by the same service instance. This is known as cookie-based session affinity. An application gateway provides cookie-based session affinity to keep a user session on the same service instance using cookies.
- **Secure Sockets Layer (SSL) offload:** The encryption and decryption of request and response data is performed by SSL and is generally a costly operation. Web servers should ideally be spending their resources on processing and serving requests, rather than the encryption and decryption of traffic. SSL offload helps in transferring this cryptography process from the web server to the load balancer, thereby providing more resources to web servers serving users. The request from the user is encrypted but gets decrypted at the application gateway instead of the web server. The request from the application gateway to the web server is unencrypted.
- **End-to-end SSL:** While SSL offload is a nice feature for certain applications, there are certain mission-critical secure applications that need complete SSL encryption and decryption even if traffic passes through load balancers. An application gateway can be configured for end-to-end SSL cryptography as well.
- **URL-based content routing:** Application gateways are also useful for redirecting traffic to different servers based on the URL content of incoming requests. This helps in hosting multiple services alongside other applications.

Azure load balancers

An Azure load balancer distributes incoming traffic based on the transport-level information that is available to it. It relies on the following features:

- An originating IP address
- A target IP address
- An originating port number
- A target port number
- A type of protocol—either TCP or HTTP

An Azure load balancer can be a private load balancer or a public load balancer. A private load balancer can be used to distribute traffic within the internal network. As this is internal, there won't be any public IPs assigned and they cannot be accessed from the internet. A public load balancer has an external public IP attached to it and can be accessed via the internet. In *Figure 2.4*, you can see how internal (private) and public load balancers are incorporated into a single solution to handle internal and external traffic, respectively:

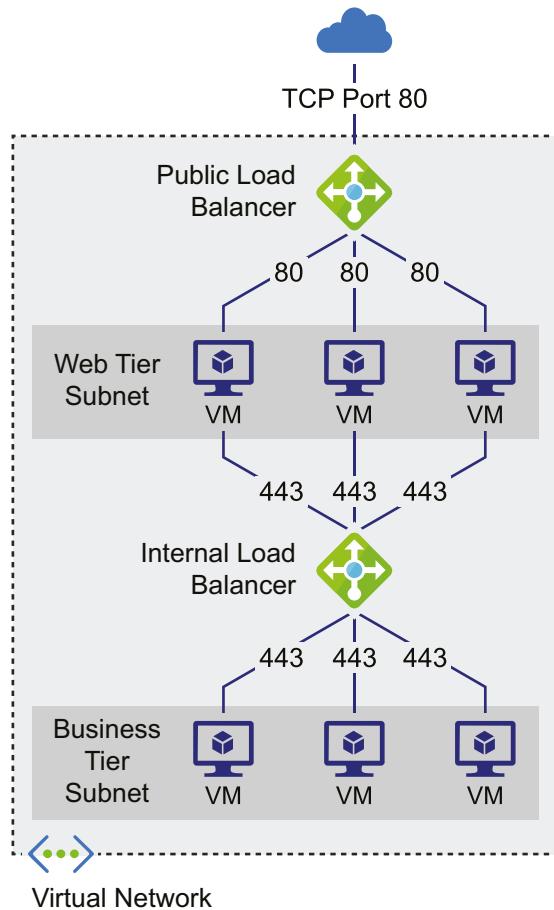


Figure 2.4: Distributing traffic using Azure load balancers

In *Figure 2.4*, you can see that external users are accessing the VMs via the public load balancer, and then the traffic from the VM is distributed across another set of VMs using an internal load balancer.

We have done a comparison of how Azure load balancers differ from Application Gateways. In the next section, we will discuss application gateways in more detail.

The Azure Application Gateway

An Azure load balancer helps us to enable solutions at the infrastructure level. However, there are times when using a load balancer requires advanced services and features. These advanced services include SSL termination, sticky sessions, advanced security, and more. An Azure application gateway provides these additional features; the Azure application gateway is a level 7 load balancer that works with the application and session payload in a TCP OSI stack.

Application gateways have more information compared to Azure load balancers in order to make decisions on request routing and load balancing between servers. Application gateways are managed by Azure and are highly available.

An application gateway sits between the users and the VMs, as shown in *Figure 2.5*:

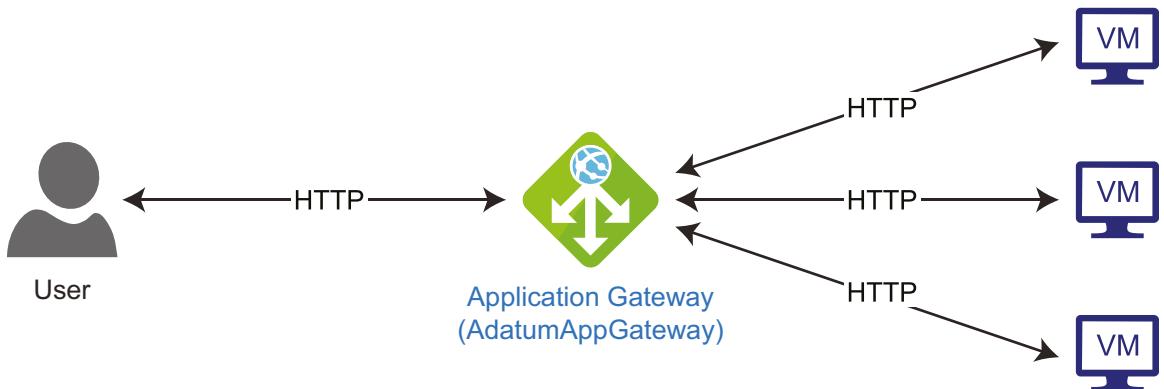


Figure 2.5: An Azure application gateway

Application gateways are a managed service. They use **Application Request Routing (ARR)** to route requests to different services and endpoints. Creating an application gateway requires a private or public IP address. The application gateway then routes the HTTP/HTTPS traffic to configured endpoints.

An application gateway is similar to an Azure load balancer from a configuration perspective, with additional constructs and features. Application gateways can be configured with a front-end IP address, a certificate, a port configuration, a back-end pool, session affinity, and protocol information.

Another service that we discussed in relation to high availability for VMs was Azure Traffic Manager. Let's try to understand more about this service in the next section.

Azure Traffic Manager

After gaining a good understanding of both Azure load balancers and application gateways, it's time to get into the details of Traffic Manager. Azure load balancers and application gateways are much-needed resources for high availability within a datacenter or region; however, to achieve high availability across regions and datacenters, there is a need for another resource, and Traffic Manager helps us in this regard.

Traffic Manager helps us to create highly available solutions that span multiple geographies, regions, and datacenters. Traffic Manager is not similar to load balancers. It uses the **Domain Name Service (DNS)** to redirect requests to an appropriate endpoint determined by the health and configuration of the endpoint. Traffic Manager is not a proxy or a gateway, and it does not see the traffic passing between the client and the service. It simply redirects requests based on the most appropriate endpoints.

Azure Traffic Manager helps to control the traffic that is distributed across application endpoints. An endpoint can be termed as any internet-facing service hosted inside or outside of Azure.

Endpoints are internet-facing, reachable public URLs. Applications are provisioned within multiple geographies and Azure regions. Applications deployed to each region have a unique endpoint referred to by **DNS CNAME**. These endpoints are mapped to the Traffic Manager endpoint. When a Traffic Manager instance is provisioned, it gets an endpoint by default with a **.trafficmanager.net** URL extension.

When a request arrives at the Traffic Manager URL, it finds the most appropriate endpoint in its list and redirects the request to it. In short, Azure Traffic Manager acts as a global DNS to identify the region that will serve the request.

However, how does Traffic Manager know which endpoints to use and redirect client requests to? There are two aspects that Traffic Manager considers to determine the most appropriate endpoint and region.

Firstly, Traffic Manager actively monitors the health of all endpoints. It can monitor the health of VMs, cloud services, and app services. If it determines that the health of an application deployed to a region is not suitable for redirecting traffic, it redirects the requests to a healthy endpoint.

Secondly, Traffic Manager can be configured with routing information. There are six traffic routing methods available in Traffic Manager, which are as follows:

- **Priority:** This should be used when all traffic should go to a default endpoint, and backups are available in case the primary endpoints are unavailable.
- **Weighted:** This should be used to distribute traffic across endpoints evenly, or according to defined weights.
- **Performance:** This should be used for endpoints in different regions, and users should be redirected to the closest endpoint based on their location. This has a direct impact on network latency.
- **Geographic:** This should be used to redirect users to an endpoint (Azure, external, or nested) based on the nearest geographical location. This can help in adhering to compliance related to data protection, localization, and region-based traffic collection.
- **Subnet:** This is a new routing method and it helps in providing clients with different endpoints based on their IP addresses. In this method, a range of IP addresses are assigned to each endpoint. These IP address ranges are mapped to the client IP address to determine an appropriate returning endpoint. Using this routing method, it is possible to provide different content to different people based on their originating IP address.
- **Multivalue:** This is also a new method added in Azure. In this method, multiple endpoints are returned to the client and any of them can be used. This ensures that if one endpoint is unhealthy, then other endpoints can be used instead. This helps in increasing the overall availability of the solution.

It should be noted that after Traffic Manager determines a valid healthy endpoint, clients connect directly to the application. Let's now move on to understand Azure's capabilities in routing user requests globally.

In the next section, we will be discussing another service, called Azure Front Door. This service is like Azure Application Gateway; however, there is a small difference that makes this service distinct. Let's go ahead and learn more about Azure Front Door.

Azure Front Door

Azure Front Door is the latest offering in Azure that helps route requests to services at a global level instead of a local region or datacenter level, as in the case of Azure Application Gateway and load balancers. Azure Front Door is like Application Gateway, with the difference being in the scope. It is a layer 7 load balancer that helps in routing requests to the nearest best-performing service endpoint deployed in multiple regions. It provides features such as TLS termination, session affinity, URL-based routing, and multiple site hosting, along with a web application firewall. It is similar to Traffic Manager in that it is, by default, resilient to entire region failures and it provides routing capabilities. It also conducts endpoint health probes periodically to ensure that requests are routed to healthy endpoints only.

It provides four different routing methods:

- **Latency:** Requests will route to endpoints that will have the least latency end to end.
- **Priority:** Requests will route to a primary endpoint and to a secondary endpoint in the case of the failure of the primary.
- **Weighted:** Requests will route based on weights assigned to the endpoints.
- **Session Affinity:** Requests in a session will end up with the same endpoint to make use of session data from prior requests. The original request can end up with any available endpoint.

Deployments looking for resilience at the global level should include Azure Front Door in their architecture, alongside application gateways and load balancers. In the next section, you will see some of the architectural considerations that you should account for while designing highly available solutions.

Architectural considerations for high availability

Azure provides high availability through various means and at various levels. High availability can be at the datacenter level, the region level, or even across Azure. In this section, we will go through some of the architectures for high availability.

High availability within Azure regions

The architecture shown in *Figure 2.6* shows a high-availability deployment within a single Azure region. High availability is designed at the individual resource level. In this architecture, there are multiple VMs at each tier connected through either an application gateway or a load balancer, and they are each part of an availability set. Each tier is associated with an availability set. These VMs are placed on separate fault and update domains. While the web servers are connected to application gateways, the rest of the tiers, such as the application and database tiers, have internal load balancers:

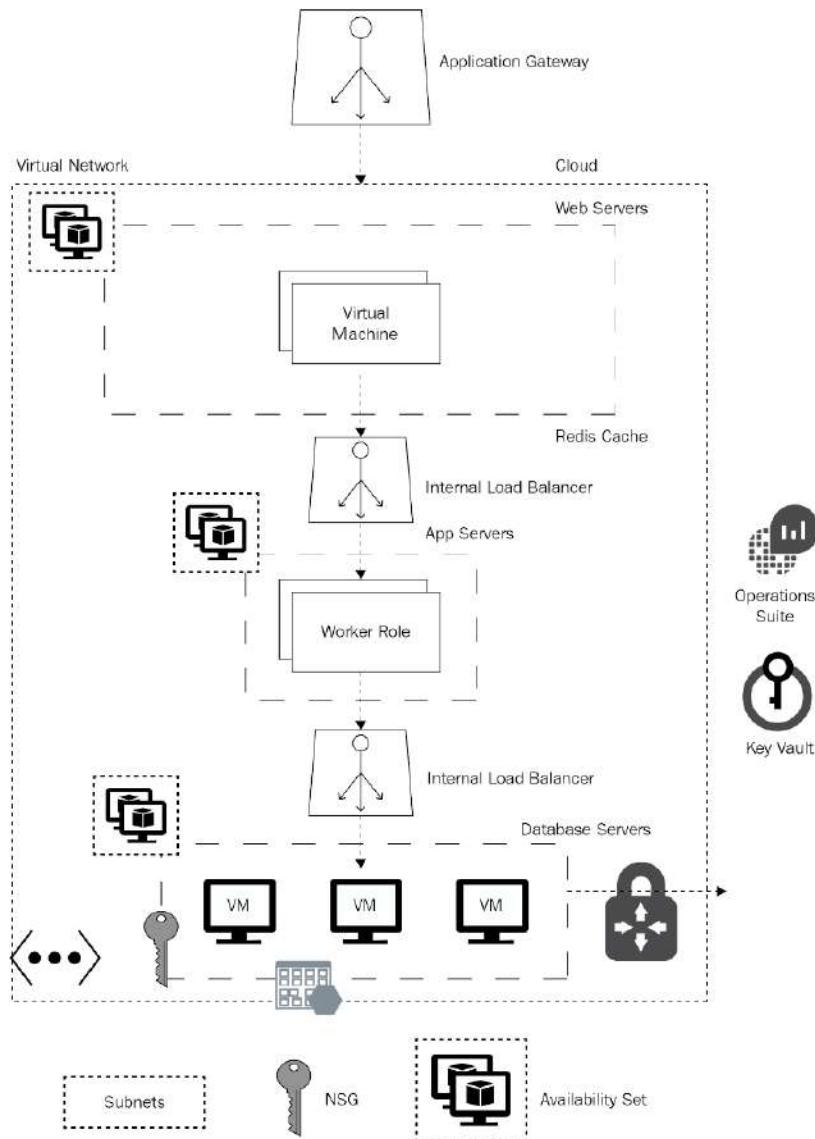


Figure 2.6: Designing high availability within a region

Now that you know how to design highly available solutions in the same region, let's discuss how an architecture that is similar, but spread across Azure regions, can be designed.

High availability across Azure regions

This architecture shows similar deployments in two different Azure regions. As shown in *Figure 2.7*, both regions have the same resources deployed. High availability is designed at the individual resource level within these regions. There are multiple VMs at each tier, connected through load balancers, and they are part of an availability set. These VMs are placed on separate fault and update domains. While the web servers are connected to external load balancers, the rest of the tiers, such as the application and database tiers, have internal load balancers. It should be noted that application load balancers can be used for web servers and the application tier (instead of Azure load balancers) if there is a need for advanced services, such as session affinity, SSL termination, advanced security using a **web application firewall (WAF)**, and path-based routing. The databases in both regions are connected to each other using virtual network peering and gateways. This is helpful in configuring log shipping, SQL Server Always On, and other data synchronization techniques.

The endpoints of the load balancers from both regions are used to configure Traffic Manager endpoints, and traffic is routed based on the priority load-balancing method. Traffic Manager helps in routing all requests to the East US region and, after failover, to West Europe in the case of the non-availability of the first region:

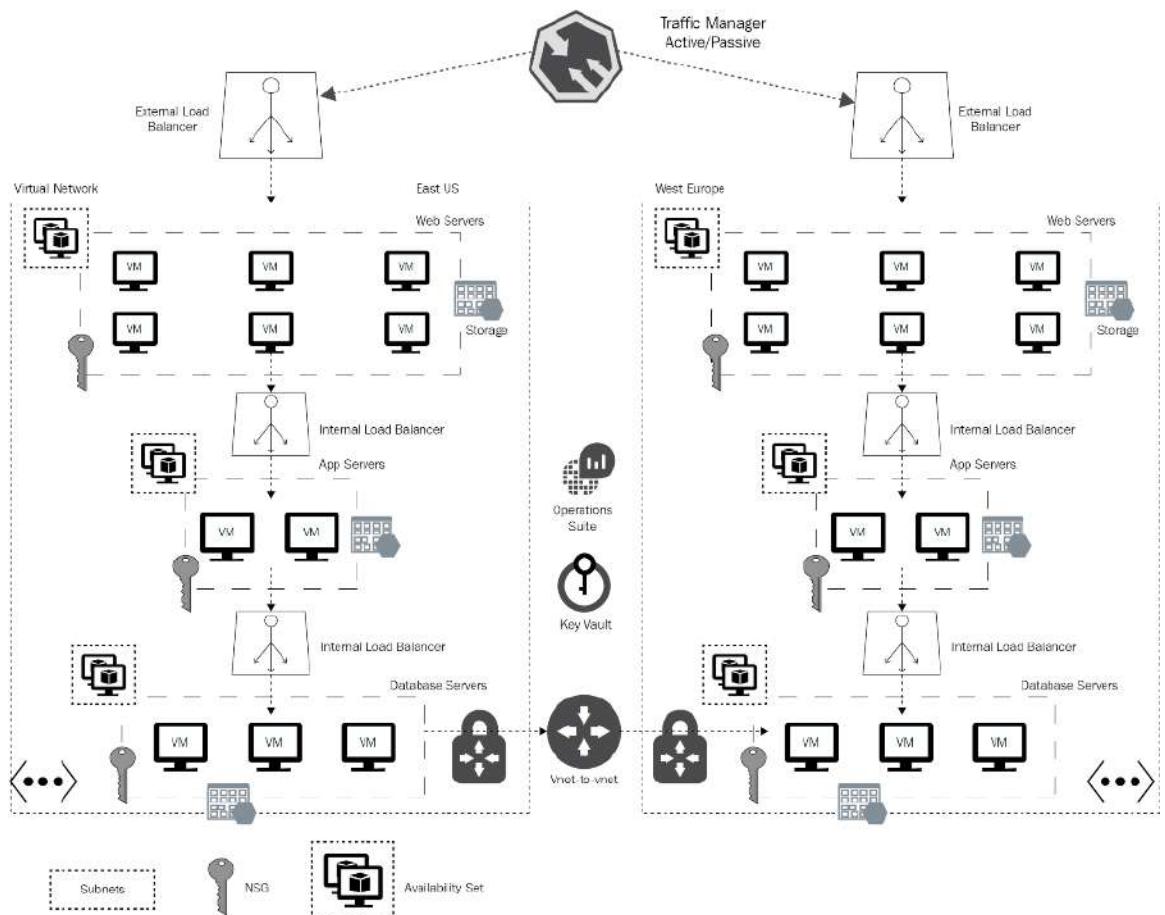


Figure 2.7: Designing high availability across Azure regions

In the next section, we will be exploring scalability, which is another advantage of the cloud.

Scalability

Running applications and systems that are available to users for consumption is important for architects of any business-critical application. However, there is another equally important application feature that is one of the top priorities for architects, and this is the scalability of the application.

Imagine a situation in which an application is deployed and obtains great performance and availability with a few users, but both availability and performance decrease as the number of users begin to increase. There are times when an application performs well under a normal load, but suffers a drop in performance with an increase in the number of users. This can happen if there is a sudden increase in the number of users and the environment is not built for such a large number of users.

To accommodate such spikes in the number of users, you might provision the hardware and bandwidth for handling spikes. The challenge with this is that the additional capacity is not used for the majority of the year, and so does not provide any return on investment. It is provisioned for use only during the holiday season or sales. I hope that by now you are becoming familiar with the problems that architects are trying to solve. All these problems are related to capacity sizing and the scalability of an application. The focus of this chapter is to understand scalability as an architectural concern and to check out the services that are provided by Azure for implementing scalability.

Capacity planning and sizing are a couple of the top priorities for architects and their applications and services. Architects must find a balance between buying and provisioning too many resources and buying and provisioning too few resources. Having too few resources can lead to you not being able to serve all users, resulting in them turning to a competitor. On the other hand, having too many resources can hurt your budget and return on investment because most of the resources will remain unused most of the time. Moreover, the problem is amplified by the varying level of demand at different times. It is almost impossible to predict the number of users of an application over a day, let alone a year. However, it is possible to find an approximate number using past information and continuous monitoring.

Scalability refers to the ability to handle a growing number of users and provide them with the same level of performance as when there are fewer users utilizing resources for application deployment, processes, and technology. Scalability might mean serving more requests without a decrease in performance, or it might mean handling larger and more time-consuming work without any loss of performance in both cases.

Capacity planning and sizing exercises should be undertaken by architects at the very beginning of a project and during the planning phase to provide scalability to applications.

Some applications have stable demand patterns, while it is difficult to predict others. Scalability requirements are known for stable-demand applications, while discerning them can be a more involved process for variable-demand applications. Autoscaling, a concept that we will review in the next section, should be used for applications whose demands cannot be predicted.

People often tend to confuse scalability with performance. In the next section, you will see a quick comparison of these two terms.

Scalability versus performance

It is quite easy to get confused between scalability and performance when it comes to architectural concerns, because scalability is all about ensuring that irrespective of the number of users consuming the application, all users receive the same predetermined level of performance.

Performance relates to ensuring that an application caters to predefined response times and throughput. Scalability refers to having provisions for more resources when needed in order to accommodate more users without sacrificing performance.

It is better to understand this using an analogy: the speed of a train directly relates to the performance of a railway network. However, getting more trains to run in parallel at the same or at higher speeds represents the scalability of the railway network.

Now that you know what the difference between scalability and performance is, let's discuss how Azure provides scalability.

Azure scalability

In this section, we will look at the features and capabilities provided by Azure to make applications highly available. Before we get into the architecture and configuration details, it is important to understand Azure's high-availability concepts, in other words, scaling.

Scaling refers to either increasing or decreasing the amount of resources that are used to serve requests from users. Scaling can be automatic or manual. Manual scaling requires an administrator to manually initiate the scaling process, while automatic scaling refers to an automatic increase or decrease in resources based on the events available from the environment and ecosystem, such as memory and CPU availability. Resources can be scaled up or down, or out and in, which will be explained later in this section.

In addition to rolling updates, the fundamental constructs provided by Azure to achieve high availability are as follows:

- Scaling up and down
- Scaling out and in
- Autoscaling

Scaling up

Scaling a VM or service up entails the addition of further resources to existing servers, such as CPU, memory, and disks. It aims to increase the capacity of existing physical hardware and resources.

Scaling down

Scaling a VM or service down entails the removal of existing resources from existing servers, such as CPU, memory, and disks. It aims to decrease the capacity of existing physical and virtual hardware and resources.

Scaling out

Scaling out entails adding further hardware, such as additional servers and capacity. This typically involves adding new servers, assigning them IP addresses, deploying applications on them, and making them part of the existing load balancers such that traffic can be routed to them. Scaling out can be automatic or manual as well. However, for better results, automation should be used:



Figure 2.8: Scaling out

Scaling in

Scaling in refers to the process of removing the existing hardware in terms of existing servers and capacity. This typically involves removing existing servers, deallocated their IP addresses, and removing them from the existing load balancer configuration such that traffic cannot be routed to them. Like scaling out, scaling in can be automatic or manual.

Autoscaling

Autoscaling refers to the process of either scaling up/down or scaling out/in dynamically based on application demand, and this happens using automation. Autoscaling is useful because it ensures that a deployment always consists of an ideal number of server instances. Autoscaling helps in building applications that are fault tolerant. It not only supports scalability, but also makes applications highly available. Finally, it provides the best cost management. Autoscaling makes it possible to have the optimal configuration for server instances based on demand. It helps in not over-provisioning servers, only for them to end up being underutilized, and removes servers that are no longer required after scaling out.

So far, we've discussed scalability in Azure. Azure offers scalability options for most of its services. Let's explore scalability for PaaS in Azure in the next section.

PaaS scalability

Azure provides App Service for hosting managed applications. App Service is a PaaS offering from Azure. It provides services for the web and mobile platforms. Behind the web and mobile platforms is a managed infrastructure that is managed by Azure on behalf of its users. Users do not see or manage any infrastructure; however, they have the ability to extend the platform and deploy their applications on top of it. In doing so, architects and developers can concentrate on their business problems instead of worrying about the base platform and infrastructure provisioning, configuration, and troubleshooting. Developers have the flexibility to choose any language, OS, and framework to develop their applications. App Service provides multiple plans and, based on the plans chosen, various degrees of scalability are available. App Service provides the following five plans:

- **Free:** This uses shared infrastructure. It means that multiple applications will be deployed on the same infrastructure from the same or multiple tenants. It provides 1 GB of storage free of charge. However, there is no scaling facility in this plan.
- **Shared:** This also uses shared infrastructure and provides 1 GB of storage free of charge. Additionally, custom domains are also provided as an extra feature. However, there is no scaling facility in this plan.
- **Basic:** This has three different **stock keeping units (SKUs)**: B1, B2, and B3. They each have increasing units of resources available to them in terms of CPU and memory. In short, they provide improved configuration of the VMs backing these services. Additionally, they provide storage, custom domains, and SSL support. The basic plan provides basic features for manual scaling. There is no autoscaling available in this plan. A maximum of three instances can be used to scale out an application.
- **Standard:** This also has three different SKUs: S1, S2, and S3. They each have increasing units of resources available to them in terms of CPU and memory. In short, they provide improved configuration of the VMs backing these services. Additionally, they provide storage, custom domains, and SSL support that is similar to that of the basic plan. This plan also provides a Traffic Manager instance, staging slots, and one daily backup as an additional feature on top of the basic plan. The standard plan provides features for automatic scaling. A maximum of 10 instances can be used to scale out the application.

- Premium:** This also has three different SKUs: P1, P2, and P3. They each have increasing units of resources available to them in terms of CPU and memory. In short, they provide improved configuration of the VMs backing these services. Additionally, they provide storage, custom domains, and SSL support that is similar to the basic plan. This plan also provides a Traffic Manager instance, staging slots, and 50 daily backups as an additional feature on top of the basic plan. The standard plan provides features for autoscaling. A maximum of 20 instances can be used to scale out the application.

We have explored the scalability tiers available for PaaS services. Now, let's see how scaling can be done in the case of an App Service plan.

PaaS – scaling up and down

Scaling up and down services that are hosted by App Service is quite simple. The Azure app services Scale Up menu opens a new pane with all plans and their SKUs listed. Choosing a plan and SKU will scale a service up or down, as shown in Figure 2.9:

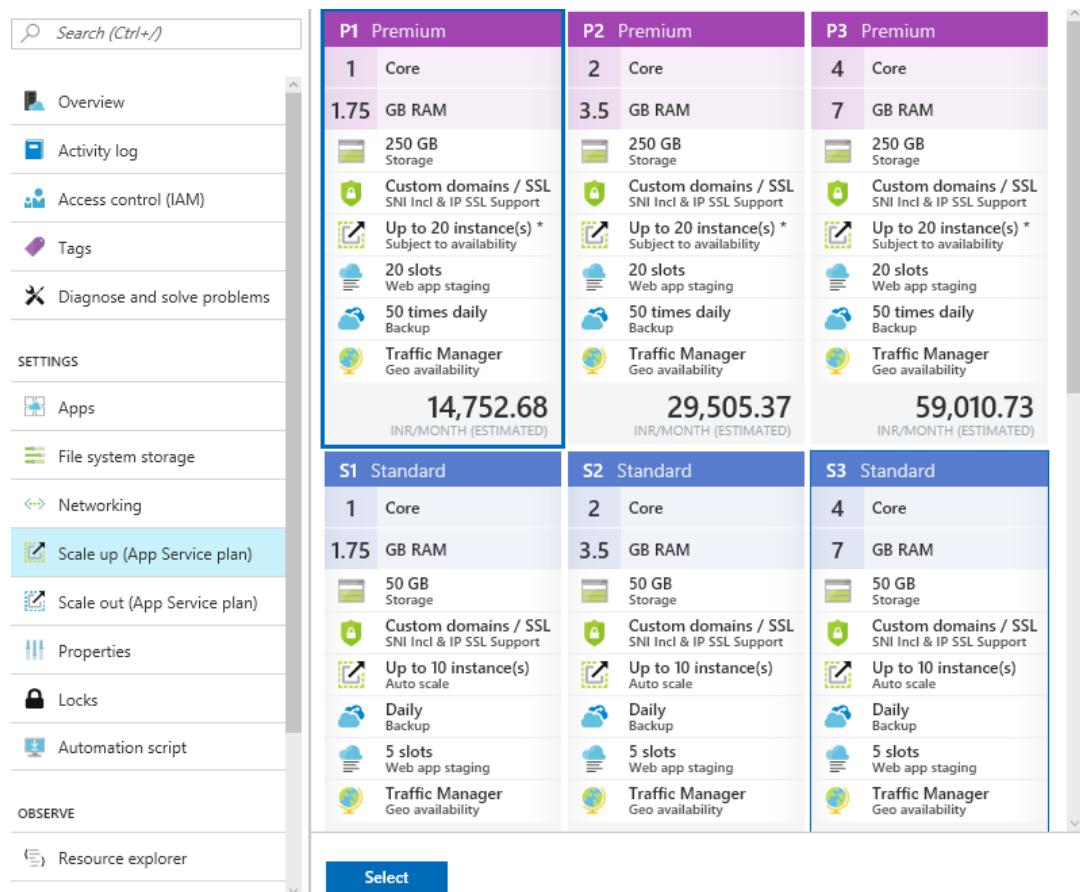


Figure 2.9: Different plans with their SKUs

PaaS – scaling out and in

Scaling out and in services hosted in App Service is also quite simple. The Azure app services Scale Out menu item opens a new pane with scaling configuration options.

By default, autoscaling is disabled for both premium and standard plans. It can be enabled using the **Scale Out** menu item and by clicking on the **Enable autoscale** button, as shown in Figure 2.10:

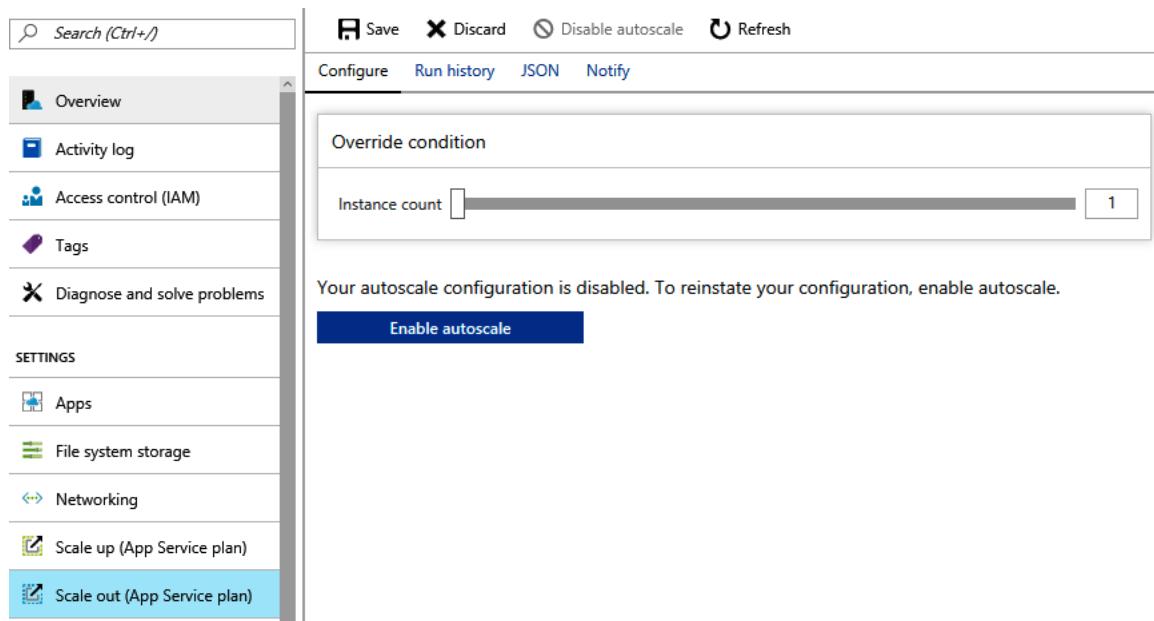


Figure 2.10: Enabling the autoscale option

Manual scaling does not require configuration, but autoscaling helps in configuring with the aid of the following properties:

- **Mode of scaling:** This is based on a performance metric such as CPU or memory usage, or users can simply specify a number of instances for scaling.
- **When to scale:** Multiple rules can be added that determine when to scale out and in. Each rule can determine criteria such as CPU or memory consumption, whether to increase or decrease the number of instances, and how many instances to increase or decrease to at a time. At least one rule for scaling out and one rule for scaling in should be configured. Threshold definitions help in defining the upper and lower limits that should trigger the autoscale—by either increasing or decreasing the number of instances.
- **How to scale:** This specifies how many instances to create or remove in each scale-out or scale-in operation:

The screenshot shows the Azure portal interface for configuring an App Service plan's scaling settings. On the left, a sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. The main panel is titled 'Configure' and shows the 'Scale out (App Service plan)' section. It includes fields for 'Autoscale setting name' (set to 'destRG') and 'Resource group'. Under 'Default' scale condition, it specifies 'Scale based on a metric'. A rule is defined: 'Scale out and scale in your instances based on metric. For example, add a rule that increases instance count is above 70%'. The rule details are: Metric name: CPU Percentage, Time grain statistic: Average, Operator: Greater than, Threshold: 70, Duration (in minutes): 10. The 'Action' section shows an operation to 'Increase count by 1'. A blue 'Add' button is located at the bottom right.

Figure 2.11: Setting the instance limits

This is quite a good feature to enable in any deployment. However, you should enable both scaling out and scaling in together to ensure that your environment is back to normal capacity after scaling out.

Since we have covered the scalability in PaaS, let's move on and discuss scalability in IaaS next.

IaaS scalability

There are users who will want to have complete control over their base infrastructure, platform, and application. They will prefer to consume IaaS solutions rather than PaaS solutions. When such customers create VMs, they are also responsible for capacity sizing and scaling. There is no out-of-the-box configuration for manually scaling or autoscaling VMs. These customers will have to write their own automation scripts, triggers, and rules to achieve autoscaling. With VMs comes the responsibility of maintaining them. The patching, updating, and upgrading of VMs is the responsibility of owners. Architects should think about both planned and unplanned maintenance. How these VMs should be patched, the order, grouping, and other factors must be considered to ensure that neither the scalability nor the availability of an application is compromised. To help alleviate such problems, Azure provides **VM scale sets (VMSS)** as a solution, which we will discuss next.

VM scale sets

VMSSes are Azure compute resources that you can use to deploy and manage a set of identical VMs. With all VMs configured in the same way, scale sets are designed to support true autoscaling, and no pre-provisioning of VMs is required. It helps in provisioning multiple identical VMs that are connected to each other through a virtual network and subnet.

A VMSS consists of multiple VMs, but they are managed at the VMSS level. All VMs are part of this unit and any changes made are applied to the unit, which, in turn, applies it to those VMs that are using a predetermined algorithm:

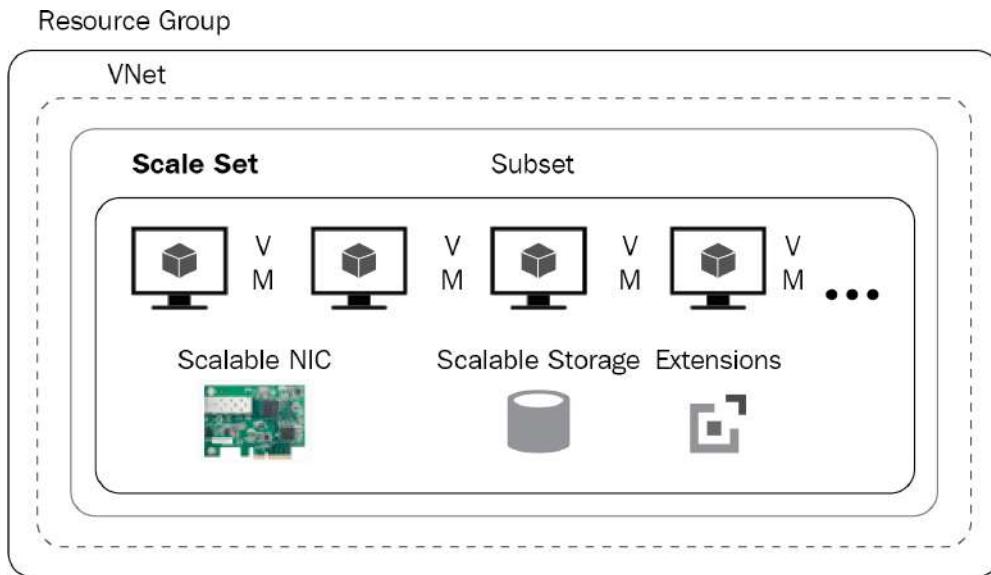


Figure 2.12: A VM scale set

This enables these VMs to be load balanced using an Azure load balancer or an application gateway. The VMs could be either Windows or Linux VMs. They can run automated scripts using a PowerShell extension and they can be managed centrally using a state configuration. They can be monitored as a unit, or individually using Log Analytics.

VMSSes can be provisioned from the Azure portal, the Azure CLI, Azure Resource Manager templates, REST APIs, and PowerShell cmdlets. It is possible to invoke REST APIs and the Azure CLI from any platform, environment, or OS, and in any language.

Many of Azure's services already use VMSSes as their underlying architecture. Among them are Azure Batch, Azure Service Fabric, and Azure Container Service. Azure Container Service, in turn, provisions Kubernetes and DC/OS on these VMSSes.

VMSS architecture

VMSSes allow the creation of up to 1,000 VMs in a scale set when using a platform image, and 100 VMs if using a custom image. If the number of VMs is less than 100 in a scale set, they are placed in a single availability set; however, if the number is greater than 100, multiple availability sets are created (known as placement groups), and VMs are distributed among these availability sets. We know from *Chapter 1, Getting started with Azure*, that VMs in an availability set are placed on separate fault and update domains. Availability sets related to VMSSes have five fault and update domains by default. VMSSes provide a model that holds metadata information for the entire set. Changing this model and applying changes impacts all VM instances. This information includes the maximum and minimum number of VM instances, the OS SKU and version, the current number of VMs, fault and update domains, and more. This is demonstrated in *Figure 2.13*:



Figure 2.13: VMs in an availability set

VMSS scaling

Scaling refers to increasing or decreasing compute and storage resources. A VMSS is a feature-rich resource that makes scaling easy and efficient. It provides autoscaling, which helps in scaling up or down based on external events and data such as CPU and memory usage. Some of the VMSS scaling features are given here.

Horizontal versus vertical scaling

Scaling can be horizontal or vertical, or both. Horizontal scaling is another name for scaling out and in, while vertical scaling refers to scaling up and down.

Capacity

VMSSes have a **capacity** property that determines the number of VMs in a scale set. A VMSS can be deployed with zero as a value for this property. It will not create a single VM; however, if you provision a VMSS by providing a number for the **capacity** property, that number of VMs are created.

Autoscaling

The autoscaling of VMs in a VMSS refers to the addition or removal of VM instances based on the configured environment in order to meet the performance and scalability demands of an application. Generally, in the absence of a VMSS, this is achieved using automation scripts and runbooks.

VMSSes help in this automation process with the support of configuration. Instead of writing scripts, a VMSS can be configured for autoscaling up and down.

Autoscaling uses multiple integrated components to achieve its end goal. Autoscaling entails continuously monitoring VMs and collecting telemetry data about them. This data is stored, combined, and then evaluated against a set of rules to determine whether autoscaling should be triggered. The trigger could be to scale out or scale in. It could also be to scale up or down.

The autoscaling mechanism uses diagnostic logs for collecting telemetry data from VMs. These logs are stored in storage accounts as diagnostic metrics. The autoscaling mechanism also uses the Application Insights monitoring service, which reads these metrics, combines them, and stores them in a storage account.

Background autoscaling jobs run continually to read Application Insights' storage data, evaluate it based on all the rules configured for autoscaling, and, if any of the rules or combination of rules are met, run the process of autoscaling. The rules can take into consideration the metrics from guest VMs and the host server.

The rules defined using the property descriptions are available at <https://docs.microsoft.com/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-autoscale-overview>.

The VMSS autoscale architecture is shown in Figure 2.14:

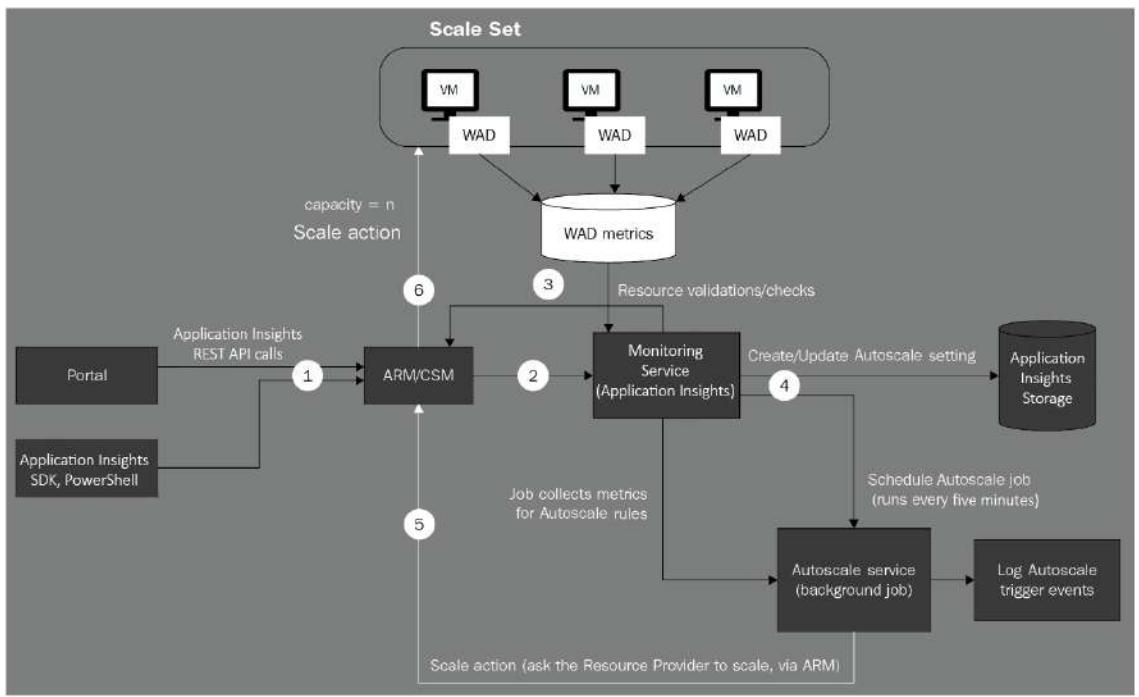


Figure 2.14: VMSS autoscale architecture

Autoscaling can be configured for scenarios that are more complex than general metrics available from environments. For example, scaling could be based on any of the following:

- A specific day
- A recurring schedule such as weekends
- Weekdays versus weekends
- Holidays and one-off events
- Multiple resource metrics

These can be configured using the **schedule** property of Application Insights resources, which help in registering rules.

Architects should ensure that at least two actions—scale out and scale in—are configured together. Scaling a configuration in or out will not help in achieving the scaling benefits provided by VMSSes.

To summarize, we have covered the scalability options in Azure and the detailed scaling features in the case of IaaS and PaaS to meet your business requirements. If you recall the shared responsibility model, you'll remember that platform upgrades and maintenance should be done by the cloud provider. In this case, Microsoft takes care of upgrades and maintenance related to the platform. Let's see how this is achieved in the next section.

Upgrades and maintenance

After a VMSS and applications are deployed, they need to be actively maintained. Planned maintenance should be conducted periodically to ensure that both the environment and application are up to date with the latest features, from a security and resilience point of view.

Upgrades can be associated with applications, the guest VM instance, or the image itself. Upgrades can be quite complex because they should happen without affecting the availability, scalability, and performance of environments and applications. To ensure that updates can take place one instance at a time using rolling upgrade methods, it is important that a VMSS supports and provides capabilities for these advanced scenarios.

There is a utility provided by the Azure team to manage updates for VMSSes. It's a Python-based utility that can be downloaded from <https://github.com/gbowerman/vmssdashboard>. It makes REST API calls to Azure to manage scale sets. This utility can be used to start, stop, upgrade, and reimagine VMs on in an FD or group of VMs, as shown in *Figure 2.15*:

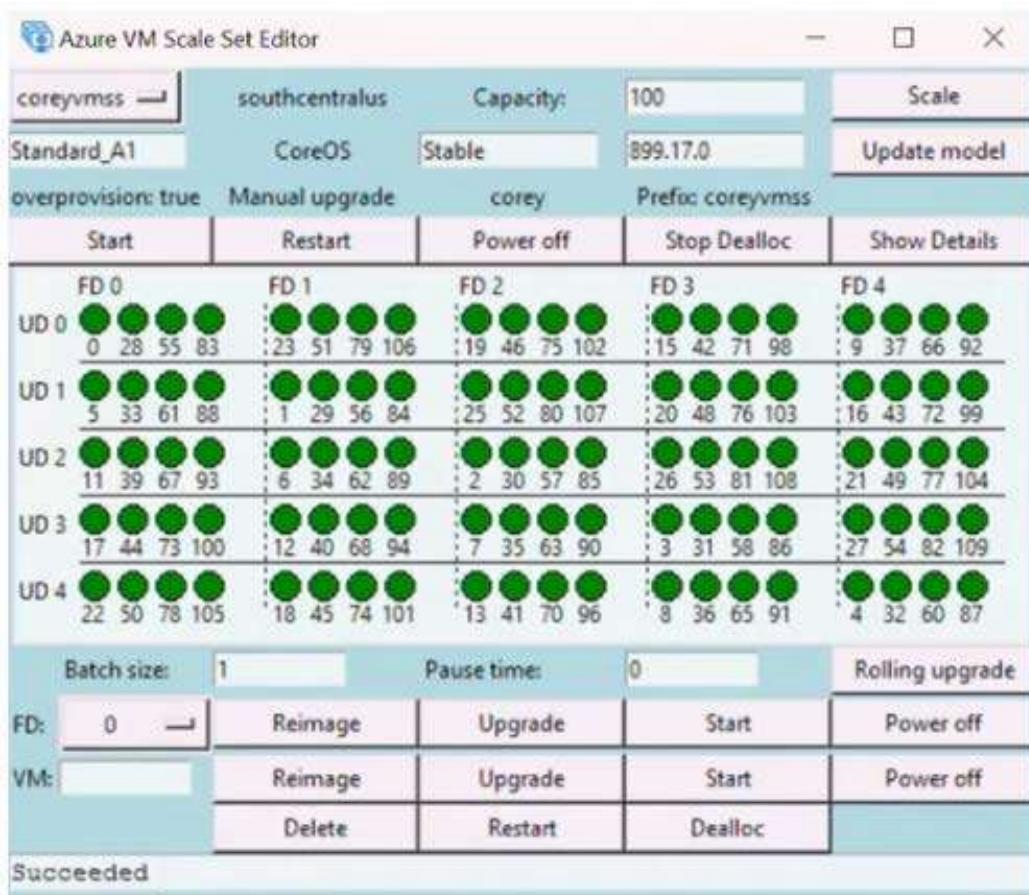


Figure 2.15: Utility for managing VMSS updates

Since you have a basic understanding of upgrade and maintenance, let's see how application updates are done in VMSSes.

Application updates

Application updates in VMSSes should not be executed manually. They must be run as part of the release management and pipelines that use automation. Moreover, an update should happen one application instance at a time and not affect the overall availability and scalability of an application. Configuration management tools, such as **Desired State Configuration (DSC)**, should be deployed to manage application updates. The DSC pull server can be configured with the latest version of the application configuration and it should be applied on a rolling basis to each instance.

In the next section, we will focus on how the updates are done on the guest OS.

Guest updates

Updates to VMs are the responsibility of the administrator. Azure is not responsible for patching guest VMs. Guest updates are in preview mode and users should control patching manually or use custom automation methods, such as runbooks and scripts. However, rolling patch upgrades are in preview mode and can be configured in the Azure Resource Manager template using an upgrade policy, as follows:

```
"upgradePolicy": {  
    "mode": "Rolling",  
    "automaticOSUpgrade": "true" or "false",  
    "rollingUpgradePolicy": {  
        "batchInstancePercent": 20,  
        "maxUnhealthyUpgradedInstanceCount": 0,  
        "pauseTimeBetweenBatches": "PT0S"  
    }  
}
```

Now that we know how guest updates are managed in Azure, let's see how image updates are accomplished.

Image updates

A VMSS can update the OS version without any downtime. OS updates involve changing the version or SKU of the OS or changing the URI of a custom image. Updating without downtime means updating VMs one at a time or in groups (such as one FD at a time) rather than all at once. By doing so, any VMs that are not being upgraded can keep running.

So far, we have discussed updates and maintenance. Let's now examine what the best practices of scaling for VMSSes are.

Best practices of scaling for VMSSes

In this section, we will go through some of the best practices that applications should implement to take advantage of the scaling capability provided by VMSSes.

The preference for scaling out

Scaling out is a better scaling solution than scaling up. Scaling up or down means resizing VM instances. When a VM is resized, it generally needs to be restarted, which has its own disadvantages. First, there is downtime for the machine. Second, if there are active users connected to the application on that instance, they might face a lack of availability of the application, or they might even lose transactions. Scaling out does not impact existing VMs; rather, it provisions newer machines and adds them to the group.

New instances versus dormant instances

Scaling new instances can take two broad approaches: creating the new instance from scratch, which requires installing applications, configuring, and testing them; or starting the dormant, sleeping instances when they are needed due to scalability pressure on other servers.

Configuring the maximum and minimum number of instances appropriately

Setting a value of two for both the minimum and maximum instance counts, with the current instance count being two, means no scaling action can occur. There should be an adequate difference between the maximum and minimum instance counts, which are inclusive. Autoscaling always scales between these limits.

Concurrency

Applications are designed for scalability to focus on concurrency. Applications should use asynchronous patterns to ensure that client requests do not wait indefinitely to acquire resources if resources are busy serving other requests. Implementing asynchronous patterns in code ensures that threads do not wait for resources and that systems are exhausted of all available threads. Applications should implement the concept of timeouts if intermittent failures are expected.

Designing stateless applications

Applications and services should be designed to be stateless. Scalability can become a challenge to achieve with stateful services, and it is quite easy to scale stateless services. With states comes the requirement for additional components and implementations, such as replication, centralized or decentralized repository, maintenance, and sticky sessions. All these are impediments on the path to scalability. Imagine a service maintaining an active state on a local server. Irrespective of the number of requests on the overall application or the individual server, the subsequent requests must be served by the same server. Subsequent requests cannot be processed by other servers. This makes scalability implementation a challenge.

Caching and the Content Distribution Network (CDN)

Applications and services should take advantage of caching. Caching helps eliminate multiple subsequent calls to either databases or filesystems. This helps in making resources available and free for more requests. The CDN is another mechanism that is used to cache static files, such as images and JavaScript libraries. They are available on servers across the globe. They also make resources available and free for additional client requests—this makes applications highly scalable.

N+1 design

N+1 design refers to building redundancy within the overall deployment for each component. It means to plan for some redundancy even when it is not required. This could mean additional VMs, storage, and network interfaces.

Considering the preceding best practices while designing workloads using VMSSes will improve the scalability of your applications. In the next section, we will explore monitoring.

Monitoring

Monitoring is an important architectural concern that should be part of any solution, big or small, mission-critical or not, cloud-based or not—it should not be neglected.

Monitoring refers to the act of keeping track of solutions and capturing various telemetry information, processing it, identifying the information that qualifies for alerts based on rules, and raising them. Generally, an agent is deployed within the environment and monitors it, sending telemetry information to a centralized server, where the rest of the processing of generating alerts and notifying stakeholders takes place.

Monitoring takes both proactive and reactive actions and measures against a solution. It is also the first step toward auditing a solution. Without the ability to monitor log records, it is difficult to audit a system from various perspectives, such as security, performance, and availability.

Monitoring helps us identify availability, performance, and scalability issues before they arise. Hardware failure, software misconfiguration, and patch update challenges can be discovered well before they impact users through monitoring, and performance degradation can be fixed before it happens.

Monitoring reactively logs pinpoint areas and locations that are causing issues, identifies the issues, and enables faster and better repairs.

Teams can identify patterns of issues using monitoring telemetry information and eliminate them by innovating new solutions and features.

Azure is a rich cloud environment that provides multiple rich monitoring features and resources to monitor not only cloud-based deployment but also on-premises deployment.

Azure monitoring

The first question that should be answered is, "What must we monitor?" This question becomes more important for solutions that are deployed on the cloud because of the constrained control over them.

There are some important components that should be monitored. They include the following:

- Custom applications
- Azure resources
- Guest OSes (VMs)
- Host OSes (Azure physical servers)
- Azure infrastructure

There are different Azure logging and monitoring services for these components, and they are discussed in the following sections.

Azure activity logs

Previously known as audit logs and operational logs, activity logs are control-plane events on the Azure platform. They provide information and telemetry information at the subscription level, instead of the individual resource level. They track information about all changes that happen at the subscription level, such as creating, deleting, and updating resources using **Azure Resource Manager (ARM)**. Activity logs help us discover the identity of (such as service principal, users, or groups), and perform actions on (such as write or update), resources (for example, storage, virtual machines, or SQL databases) at any given point in time. They provide information about resources that are modified in their configuration, but not their inner workings and execution. For example, you can get the logs for starting a VM, resizing a VM, or stopping a VM.

The next topic that we are going to discuss is diagnostic logs.

Azure diagnostic logs

The information originating within the inner workings of Azure resources is captured in what are known as **diagnostic logs**. They provide telemetry information about the operations of resources that are inherent to the resources. Not every resource provides diagnostic logs, and resources that provide logs on their own content are completely different from other resources. Diagnostic logs are configured individually for each resource. Examples of diagnostic logs include storing a file in a container in a blob in a storage account.

The next type of log that we are going to discuss is application logs.

Azure application logs

Application logs can be captured by Application Insights resources and can be managed centrally. They get information about the inner workings of custom applications, such as their performance metrics and availability, and users can get insights from them in order to manage them better.

Lastly, we have guest and host OS logs. Let's understand what these are.

Guest and host OS logs

Both guest and host OS logs are offered to users using Azure Monitor. They provide information about the statuses of host and guest OSes:

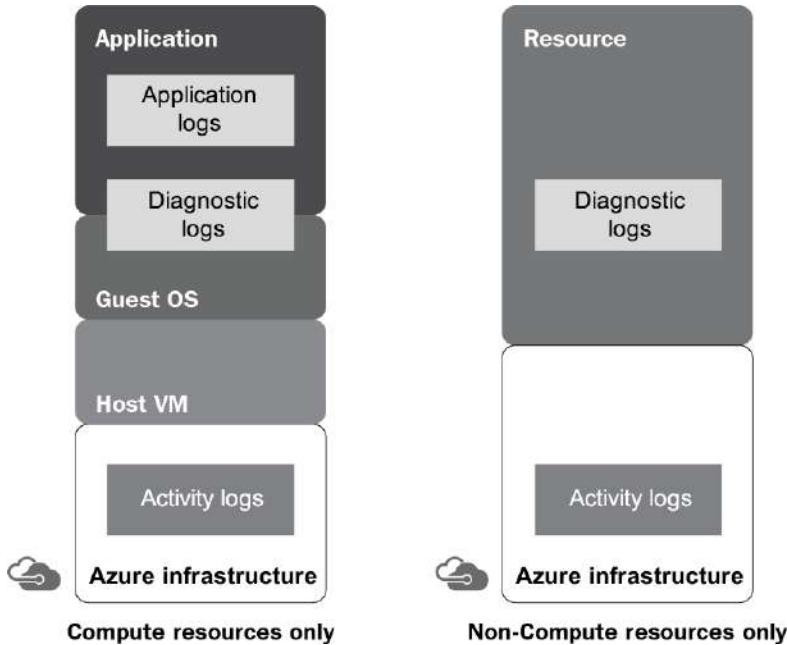


Figure 2.16: Logging in Azure

The important Azure resources related to monitoring are Azure Monitor, Azure Application Insights, and Log Analytics, previously known as **Operational Insights**.

There are other tools, such as **System Center Operations Manager (SCOM)**, that are not part of the cloud feature but can be deployed on IaaS-based VMs to monitor any workload on Azure or an on-premises datacenter. Let's discuss the three monitoring resources in the following section.

Azure Monitor

Azure Monitor is a central tool and resource that provides complete management features that allow you to monitor an Azure subscription. It provides management features for activity logs, diagnostic logs, metrics, Application Insights, and Log Analytics. It should be treated as a dashboard and management resource for all other monitoring capabilities.

Our next topic is Azure Application Insights.

Azure Application Insights

Azure Application Insights provides centralized, Azure-scale monitoring, logs, and metrics capabilities to custom applications. Custom applications can send metrics, logs, and other telemetry information to Azure Application Insights. It also provides rich reporting, dashboarding, and analytics capabilities to get insights from incoming data and act on them.

Now that we have covered Application Insights, let's look at another similar service called Azure Log Analytics.

Azure Log Analytics

Azure Log Analytics enables the centralized processing of logs and generates insights and alerts from them. Activity logs, diagnostic logs, application logs, event logs, and even custom logs can send information to Log Analytics, which can further provide rich reporting, dashboarding, and analytics capabilities to get insights from incoming data and act on them.

Now that we know the purpose of Log Analytics, let's discuss how logs are stored in a Log Analytics workspace and how they can be queried.

Logs

A Log Analytics workspace provides search capabilities to search for specific log entries, export all telemetry data to Excel and/or Power BI, and search a query language called **Kusto Query Language (KQL)**, which is similar to SQL.

The **Log Search** screen is shown here:

The screenshot shows the Log Analytics workspace interface. At the top, there's a navigation bar with 'Logs' and a search bar containing 'Heartbeat'. Below the search bar are tabs for 'Tables', 'Queries', and 'Filter'. A sidebar on the left lists 'Favorites' and two main categories: 'Change Tracking' and 'LogManagement'. Under 'Change Tracking', there are links for 'ConfigurationChange' and 'ConfigurationData'. Under 'LogManagement', there are links for 'Alert', 'AppCenterError', 'ComputerGroup', 'Heartbeat', 'Operation', 'ReservedCommonFields', and 'Usage'. The main pane shows a results table with one row: 'Completed. Showing results from the last 24 hours.' followed by 'NO RESULTS FOUND (last 24 hours)'. It also includes a note that '0 records matched for the selected time range' and a 'Need Help?' section with links to 'Select another time range...' and 'Add a custom time filter to your query...'.

Figure 2.17: Log search in a Log Analytics workspace

In the next section, we will be covering Log Analytics solutions, which are like additional capabilities in a Log Analytics workspace.

Solutions

Solutions in Log Analytics are further capabilities that can be added to a workspace, capturing additional telemetry data that is not captured by default. When these solutions are added to a workspace, appropriate management packs are sent to all the agents connected to the workspace so that they can configure themselves to capture solution-specific data from VMs and containers and then send it to the Log Analytics workspace. Monitoring solutions from Microsoft and partners are available from Azure Marketplace.

Azure provides lots of Log Analytics solutions for tracking and monitoring different aspects of environments and applications. At a minimum, a set of solutions that are generic and applicable to almost any environment should be added to the workspace:

- Capacity and performance
- Agent health
- Change tracking
- Containers
- Security and audit
- Update management
- Network performance monitoring

Another key aspect of monitoring is alerts. Alerts help to notify the right people during any monitored event. In the next section, we will cover alerts.

Alerts

Log Analytics allows us to generate alerts in relation to ingested data. It does so by running a pre-defined query composed of conditions for incoming data. If it finds any records that fall within the ambit of the query results, it generates an alert. Log Analytics provides a highly configurable environment for determining the conditions for generating alerts, time windows in which the query should return the records, time windows in which the query should be executed, and actions to be taken when the query returns an alert:

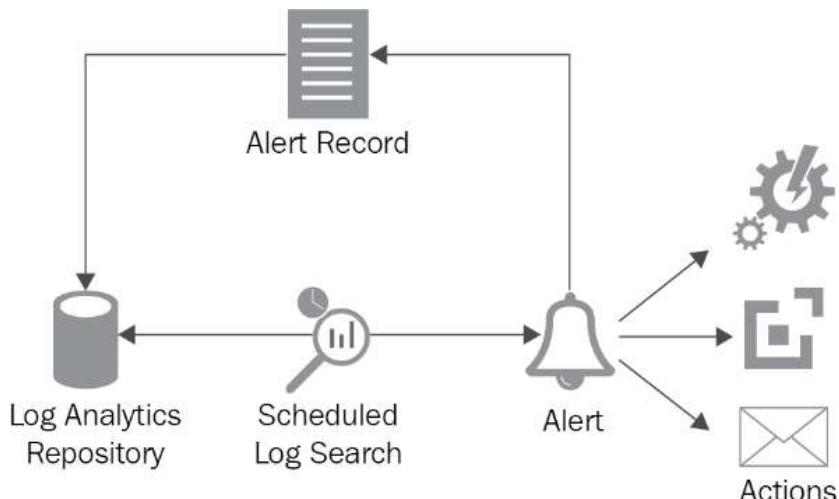


Figure 2.18: Configuring alerts through Log Analytics

Let's go through the steps for configuring alerts through Log Analytics:

1. The first step in configuring an alert is to add a new alert rule from the Azure portal or automation from the alert menu of the Log Analytics resource.
2. From the resultant panel, select a scope for the alert rule. The scope determines which resource should be monitored for alerts—it could be a resource instance, such as an Azure storage account, a resource type, such as an Azure VM, a resource group, or a subscription:

Select a resource X

Select the resource(s) you want to monitor. Available signal types for your selection will show up on the bottom right.

Filter by subscription * ⓘ RiteshSubscription

Filter by resource type ⓘ Virtual machines

Filter by location ⓘ All

Search to filter items...

Resource	Resource type	Location
<input type="checkbox"/> RiteshSubscription	Subscription	
<input type="checkbox"/> onlinetuesday	Resource group	West Europe
<input type="checkbox"/> vs2019docker	Virtual machine	West Europe
<input type="checkbox"/> rg-harvestingclouds-infra101	Resource group	East US
<input type="checkbox"/> vmAccounts101	Virtual machine	East US
<input type="checkbox"/> spark	Resource group	West Europe
<input type="checkbox"/> spark	Virtual machine	West Europe

Figure 2.19: Selecting a resource for the alert

3. Following resource selection, conditions must be set for the alert. The condition determines the rule that is evaluated against the logs and metrics on the selected resource, and only after the condition turns true is an alert generated. There are a ton of metrics and logs available for generating conditions. In the following example, an alert is created with a static threshold value of 80% for **Percentage CPU (Avg)** and the data is to be collected every five minutes and evaluated every minute:

Percentage CPU (Avg)
vs2019docker
--

Alert logic

Threshold ⓘ

Static Dynamic

Operator ⓘ

Greater than

Aggregation type * ⓘ

Average

Threshold value * ⓘ

80 %

Condition preview

Whenever the average percentage cpu is greater than 80 %

Evaluated based on

Aggregation granularity (Period) * ⓘ

5 minutes

Frequency of evaluation ⓘ

Every 1 Minute

Figure 2.20: Creating an alert for Percentage CPU (Avg)

Alerts also support dynamic thresholds, which use machine learning to learn the historical behavior of metrics and detect irregularities that could indicate service issues.

4. Finally, create an action group or reuse an existing group that determines notifications regarding alerts to stakeholders. The **Action Groups** section allows you to configure things that should follow an alert. Generally, there should be a remedial and/or notification action. Log Analytics provides eight different ways to create a new action. They can be combined in any way you like. An alert will execute any or all of the following configured actions:
 - **Email/SMS/push/voice notification:** This sends an email/SMS/push/voice notification to the configured recipients.
 - **Webhooks:** A webhook runs an arbitrary external process using an HTTP POST mechanism. For example, a REST API can be executed, or the Service Manager/ServiceNow APIs can be invoked to create a ticket.
 - **Azure Functions:** This runs an Azure function, passing the necessary payload and running the logic that the payload contains.
 - **Logic Apps:** This executes a custom Logic Apps workflow.
 - **Email Azure Resource Manager Role:** This emails a holder of an Azure Resource Manager role, such as an owner, contributor, or reader.
 - **Secure webhook:** A webhook runs an arbitrary external process using an HTTP POST mechanism. Webhooks are protected using an identity provider, such as Azure Active Directory.
 - **Automation runbooks:** This action executes Azure Automation runbooks.
 - **ITSM:** ITSM solutions should be provisioned before using this option. It helps with connecting and sending information to ITSM systems.
5. After all of this configuration, you need to provide the **Name**, **Description**, and **Severity** values for the alert rule to generate it.

As mentioned at the beginning of this section, alerts play a vital role in monitoring that helps authorized personnel to take necessary actions based on the alert that's triggered.

Summary

High availability and scalability are crucially important architectural concerns. Almost every application and every architect try to implement high availability. Azure is a mature platform that understands the need for these architectural concerns in applications and provides resources to implement them at multiple levels. These architectural concerns are not an afterthought, and they should be part of the application development life cycle, starting from the planning phase itself.

Monitoring is an important architectural aspect of any solution. It is also the first step toward being able to audit an application properly. It enables operations to manage a solution, both reactively and proactively. It provides the necessary records for troubleshooting and fixing the issues that might arise from platforms and applications. There are many resources in Azure that are specific to implementing monitoring for Azure, other clouds, and on-premises datacenters. Application Insights and Log Analytics are two of the most important resources in this regard. Needless to say, monitoring is a must for making your solutions and products better by innovating based on insights derived from monitoring data.

This chapter was purely about the availability, scalability, and monitoring of solutions; the next chapter is about design patterns related to virtual networks, storage accounts, regions, availability zones, and availability sets. While designing solutions in the cloud, these principles are very important in building cost-effective solutions with increased productivity and availability.

3

Design pattern – Networks, storage, messaging, and events

In the previous chapter, you got an overview of the Azure cloud and learned about some of the important concepts related to it. This chapter is about Azure cloud patterns that are related to virtual networks, storage accounts, regions, Availability Zones, and Availability Sets. These are important constructs that affect the final architecture delivered to customers in terms of cost, efficiencies, and overall productivity. The chapter also briefly discusses the cloud patterns that help us to implement scalability and performance for an architecture.

In this chapter, we'll cover the following topics:

- Azure Virtual Network design
- Azure Storage design
- Azure Availability Zones, regions, and Availability Sets
- Azure design patterns related to messaging, performance, and scalability

Azure Availability Zones and Regions

Azure is backed up by large datacenters interconnected into a single large network. The datacenters are grouped together, based on their physical proximity, into Azure regions. For example, datacenters in Western Europe are available to Azure users in the West Europe region. Users cannot choose their preferred datacenter. They can select their Azure region and Azure will allocate an appropriate datacenter.

Choosing an appropriate region is an important architectural decision as it affects:

- The availability of resources
- Data and privacy compliance
- The performance of the application
- The cost of running applications

Let's discuss each of these points in detail.

Availability of resources

Not all resources are available in every Azure region. If your application architecture demands a resource that is not available in a region, choosing that region will not help. Instead, a region should be chosen based on the availability of the resources required by the application. It might be that the resource is not available while developing the application architecture, and it could be on Azure's roadmap to make it available subsequently.

For example, Log Analytics is not available in all regions. If your data sources are in Region A and the Log Analytics workspace is in Region B, you need to pay for the bandwidth, which is the data egress charges from Region A to B. Similarly, some services can work with resources that are located in the same region. For instance, if you would like to encrypt the disks of your virtual machine that is deployed in Region A, you need to have Azure Key Vault deployed in Region A to store the encryption keys. Before deploying any services, you need to check whether your dependency services are available in that region. A good source to check the availability of Azure products across regions is this product page: <https://azure.microsoft.com/global-infrastructure/services>.

Data and privacy compliance

Each country has its own rules for data and privacy compliance. Some countries are very specific about storing their citizens' data in their own territories. Hence, such legal requirements should be taken into consideration for every application's architecture.

Application performance

The performance of an application is dependent on the network route taken by requests and responses to get to their destinations and back again. The location that is geographically closer to you may not always be the region with the lowest latency. We calculate distance in kilometers or miles, but latency is based on the route the packet takes. For example, an application deployed in Western Europe for Southeast Asian users will not perform as well as an application deployed to the East Asia region for users in that region. So, it's very important that you architect your solutions in the closest region to provide the lowest latency and thus the best performance.

Cost of running applications

The cost of Azure services differs from region to region. A region with an overall lower cost should be chosen. There is a complete chapter on cost management in this book (*Chapter 6, Cost management for Azure solutions*), and it should be referred to for more details on cost.

So far, we have discussed how to choose the right region to architect our solution. Now that we have a suitable region in mind for our solution, let's discuss how to design our virtual networks in Azure.

Virtual networks

Virtual networks should be thought of like a physical office or home LAN network setup. Conceptually, they are the same, although **Azure Virtual Network (VNet)** is implemented as a software-defined network backed up by a giant physical network infrastructure.

A VNet is required to host a virtual machine. It provides a secure communication mechanism between Azure resources so that they can connect to each other. The VNets provide internal IP addresses to the resources, facilitate access and connectivity to other resources (including virtual machines on the same virtual network), route requests, and provide connectivity to other networks.

A virtual network is contained within a resource group and is hosted within a region, for example, West Europe. It cannot span multiple regions but can span all datacenters within a region, which means we can span virtual networks across multiple Availability Zones in a region. For connectivity across regions, virtual networks can be connected using VNet-to-VNet connectivity.

Virtual networks also provide connectivity to on-premises datacenters, enabling hybrid clouds. There are multiple types of VPN technologies that you can use to extend your on-premises datacenters to the cloud, such as site-to-site VPN and point-to-site VPN. There is also dedicated connectivity between Azure VNet and on-premises networks through the use of ExpressRoute.

Virtual networks are free of charge. Every subscription can create up to 50 virtual networks across all regions. However, this number can be increased by reaching out to Azure Support. You will not be charged if data does not leave the region of deployment. At the time of writing, inbound and outbound data transfers within Availability Zones from the same region don't incur charges; however, billing will commence from July 1, 2020.

Information about networking limits is available in the Microsoft documentation at <https://docs.microsoft.com/azure/azure-resource-manager/management/azure-subscription-service-limits>.

Architectural considerations for virtual networks

Virtual networks, like any other resource, can be provisioned using ARM templates, REST APIs, PowerShell, and the CLI. It is quite important to plan the network topology as early as possible to avoid troubles later in the development life cycle. This is because once a network is provisioned and resources start using it, it is difficult to change it without having downtime. For example, moving a virtual machine from one network to another will require the virtual machine to be shut down.

Let's look at some of the key architectural considerations while designing a virtual network.

Regions

VNet is an Azure resource and is provisioned within a region, such as West Europe. Applications spanning multiple regions will need separate virtual networks, one per region, and they also need to be connected using VNet-to-VNet connectivity. There is a cost associated with VNet-to-VNet connectivity for both inbound and outbound traffic. There are no charges for inbound (ingress) data, but there are charges associated with outbound data.

Dedicated DNS

VNet by default uses Azure's DNS to resolve names within a virtual network, and it also allows name resolution on the internet. If an application wants a dedicated name resolution service or wants to connect to on-premises datacenters, it should provision its own DNS server, which should be configured within the virtual network for successful name resolution. Also, you can host your public domain in Azure and completely manage the records from the Azure portal, without the need to manage additional DNS servers.

Number of virtual networks

The number of virtual networks is affected by the number of regions, bandwidth usage by services, cross-region connectivity, and security. Having fewer but larger VNets instead of multiple smaller VNets will eliminate the management overhead.

Number of subnets in each virtual network

Subnets provide isolation within a virtual network. They can also provide a security boundary. **Network security groups (NSGs)** can be associated with subnets, thereby restricting or allowing specific access to IP addresses and ports. Application components with separate security and accessibility requirements should be placed within separate subnets.

IP ranges for networks and subnets

Each subnet has an IP range. The IP range should not be so large that IPs are underutilized, but conversely shouldn't be so small that subnets become suffocated because of a lack of IP addresses. This should be considered after understanding the future IP address needs of the deployment.

Planning should be done for IP addresses and ranges for Azure networks, subnets, and on-premises datacenters. There should not be an overlap to ensure seamless connectivity and accessibility.

Monitoring

Monitoring is an important architectural facet and must be included within the overall deployment. Azure Network Watcher provides logging and diagnostic capabilities with insights on network performance and health. Some of the capabilities of the Azure Network Watcher are:

- Diagnosing network traffic filtering problems to or from a virtual machine
- Understanding the next hop of user-defined routes
- Viewing the resources in a virtual network and their relationships
- Communication monitoring between a virtual machine and an endpoint
- Traffic capture from a virtual machine
- NSG flow logs, which log information related to traffic flowing through an NSG. This data will be stored in Azure Storage for further analysis

It also provides diagnostic logs for all the network resources in a resource group.

Network performance can be monitored through Log Analytics. The Network Performance Monitor management solution provides network monitoring capability. It monitors the health, availability, and reachability of networks. It is also used to monitor connectivity between public cloud and on-premises subnets hosting various tiers of a multi-tiered application.

Security considerations

Virtual networks are among the first components that are accessed by any resource on Azure. Security plays an important role in allowing or denying access to a resource. NSGs are the primary means of enabling security for virtual networks. They can be attached to virtual network subnets, and every inbound and outbound flow is constrained, filtered, and allowed by them.

User-defined routing (UDR) and IP forwarding also helps in filtering and routing requests to resources on Azure. You can read more about UDR and forced tunneling at <https://docs.microsoft.com/azure/virtual-network/virtual-networks-udr-overview>.

Azure Firewall is a fully managed Firewall as a Service offering from Azure. It can help you protect the resources in your virtual network. Azure Firewall can be used for packet filtering in both inbound and outbound traffic, among other things. Additionally, the threat intelligence feature of Azure Firewall can be used to alert and deny traffic from or to malicious domains or IP addresses. The data source for IP addresses and domains is Microsoft's threat intelligence feed.

Resources can also be secured and protected by deploying network appliances (<https://azure.microsoft.com/solutions/network-appliances>) such as Barracuda, F5, and other third-party components.

Deployment

Virtual networks should be deployed in their own dedicated resource groups. Network administrators should have the owner's permission to use this resource group, while developers or team members should have contributor permissions to allow them to create other Azure resources in other resource groups that consume services from the virtual network.

It is also a good practice to deploy resources with static IP addresses in a dedicated subnet, while dynamic IP address-related resources can be on another subnet.

Policies should not only be created so that only network administrators can delete the virtual network, but also should also be tagged for billing purposes.

Connectivity

Resources in a region on a virtual network can talk seamlessly. Even resources on other subnets within a virtual network can talk to each other without any explicit configuration. Resources in multiple regions cannot use the same virtual network. The boundary of a virtual network is within a region. To make a resource communicate across regions, we need dedicated gateways at both ends to facilitate conversation.

Having said that, if you would like to initiate a private connection between two networks in different regions, you can use Global VNet peering. With Global VNet peering, the communication is done via Microsoft's backbone network, which means no public internet, gateway, or encryption is required during the communication. If your virtual networks are in the same region with different address spaces, resources in one network will not be able to communicate with the other. Since they are in the same region, we can use virtual network peering, which is similar to Global VNet peering; the only difference is that the source and destination virtual networks are deployed in the same region.

As many organizations have a hybrid cloud, Azure resources sometimes need to communicate or connect with on-premises datacenters or vice versa. Azure virtual networks can connect to on-premises datacenters using VPN technology and ExpressRoute. In fact, one virtual network is capable of connecting to multiple on-premises datacenters and other Azure regions in parallel. As a best practice, each of these connections should be in their dedicated subnets within a virtual network.

Now that we have explored several aspects of virtual networking, let's go ahead and discuss the benefits of virtual networks.

Benefits of virtual networks

Virtual networks are a must for deploying any meaningful IaaS solution. Virtual machines cannot be provisioned without virtual networks. Apart from being almost a mandatory component in IaaS solutions, they provide great architectural benefits, some of which are outlined here:

- **Isolation:** Most application components have separate security and bandwidth requirements and have different life cycle management. Virtual networks help to create isolated pockets for these components that can be managed independently of other components with the help of virtual networks and subnets.
- **Security:** Filtering and tracking the users that are accessing resources is an important feature provided by virtual networks. They can stop access to malicious IP addresses and ports.
- **Extensibility:** Virtual networks act like a private LAN on the cloud. They can also be extended into a **Wide Area Network (WAN)** by connecting other virtual networks across the globe and can be extensions to on-premises datacenters.

We have explored the benefits of virtual networks. Now the question is how we can leverage these benefits and design a virtual network to host our solution. In the next section, we will look at the design of virtual networks.

Virtual network design

In this section, we will consider some of the popular designs and use case scenarios of virtual networks.

There can be multiple usages of virtual networks. A gateway can be deployed at each virtual network endpoint to enable security and transmit packets with integrity and confidentiality. A gateway is a must when connecting to on-premises networks; however, it is optional when using Azure VNet peering. Additionally, you can make use of the Gateway Transit feature to simplify the process of extending your on-premises datacenter without deploying multiple gateways. Gateway Transit allows you to share an ExpressRoute or VPN gateway with all peered virtual networks. This will make it easy to manage and reduce the cost of deploying multiple gateways.

In the previous section, we touched on peering and mentioned that we don't use gateways or the public internet to establish communication between peered networks. Let's move on and explore some of the design aspects of peering, and which peering needs to be used in particular scenarios.

Connecting to resources within the same region and subscription

Multiple virtual networks within the same region and subscription can be connected to each other. With the help of VNet peering, both networks can be connected and use the Azure private network backbone to transmit packets to each other. Virtual machines and services on these networks can talk to each other, subject to network traffic constraints. In the following diagram, VNet1 and VNet2 both are deployed in the West US region. However, the address space for VNet1 is 172.16.0.0/16, and for VNet2 it is 10.0.0.0/16. By default, resources in VNet1 will not be able to communicate with resources in VNet2. Since we have established VNet peering between the two, the resources will be able to communicate with each other via the Microsoft backbone network:

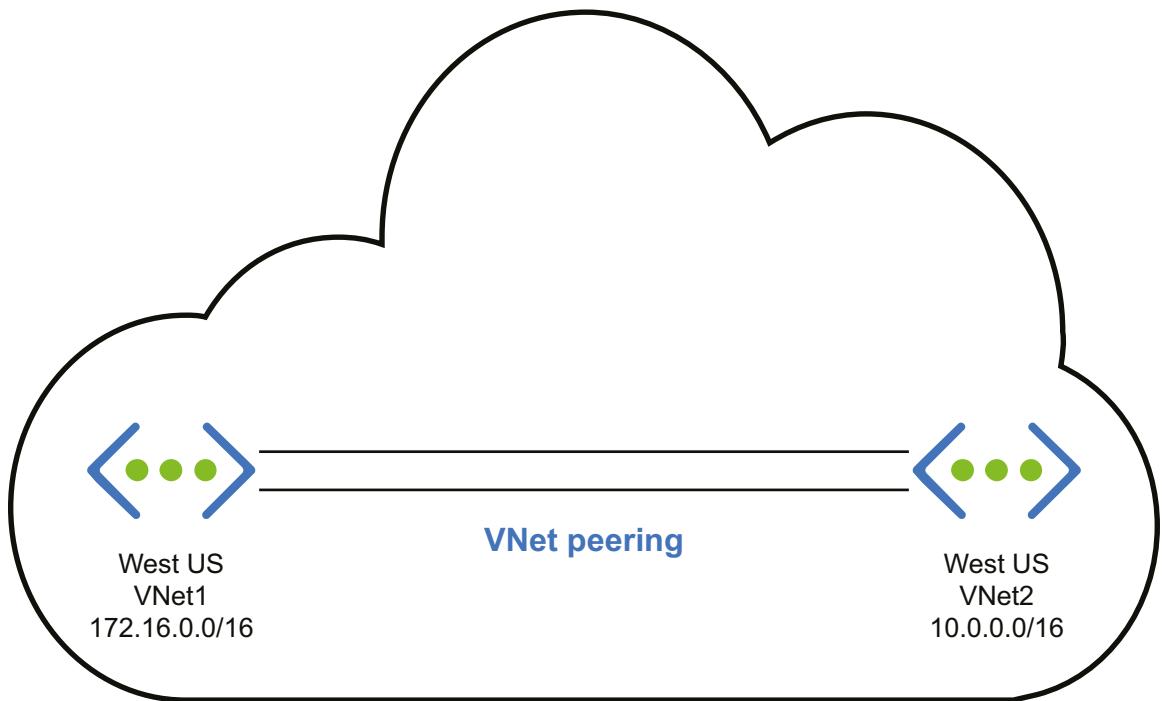


Figure 3.1: VNet peering for resources with the same subscription

Connecting to resources within the same region in another subscription

This scenario is very similar to the previous one except that the virtual networks are hosted in two different subscriptions. The subscriptions can be part of the same tenant or from multiple tenants. If both the resources are part of the same subscription and from the same region, the previous scenario applies. This scenario can be implemented in two ways: by using gateways or by using virtual network peering.

If we are using gateways in this scenario, we need to deploy a gateway at both ends to facilitate communication. Here is the architectural representation of using gateways to connect two resources with different subscriptions:

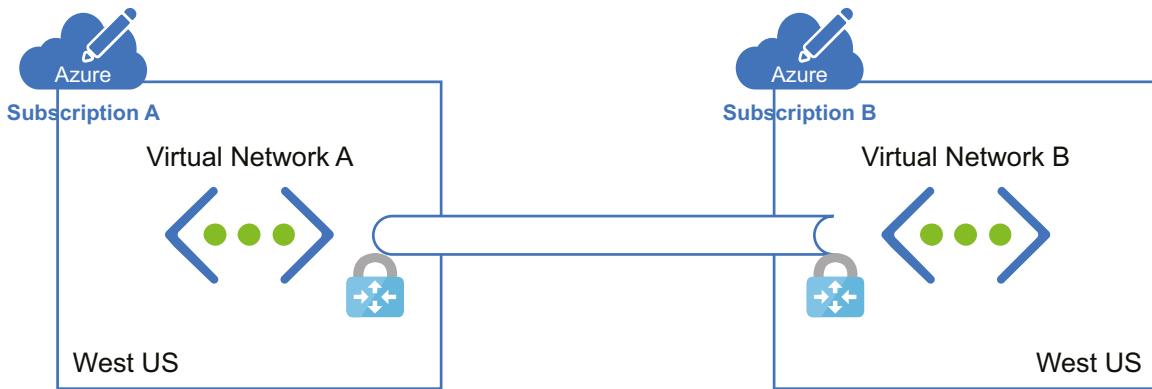


Figure 3.2: VNet peering for resources with different subscriptions using gateways

However, the deployment of gateways incurs some charges. We will discuss VNet peering, and after that we will compare these two implementations to see which is best for our solution.

While using peering, we are not deploying any gateways. Figure 3.3 represents how peering is done:

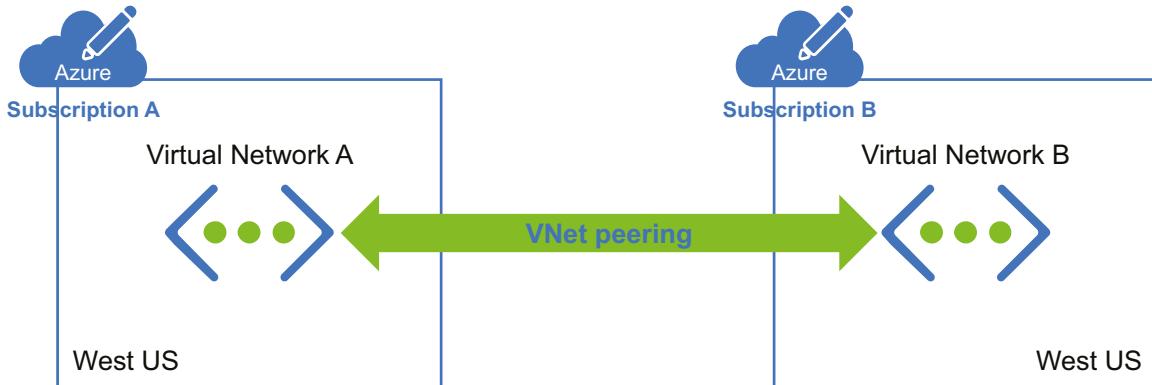


Figure 3.3: VNet peering across subscriptions

VNet peering provides a low-latency, high-bandwidth connection, and, as shown in the diagram, we are not deploying any gateways to make the communication happen. This is useful for scenarios such as data replication or failover. As mentioned earlier, peering uses the Microsoft backbone network, which eliminates the need for the public internet.

Gateways are used in scenarios where encryption is needed and bandwidth is not a concern, as this will be a limited-bandwidth connection. However, this doesn't mean that there is a constraint on bandwidth. Also, this approach is used where customers are not so latency-sensitive.

So far, we have looked at resources in the same region across subscriptions. In the next section, we will explore how to establish a connection between virtual networks in two different regions.

Connecting to resources in different regions in another subscription

In this scenario, we have two implementations again. One uses a gateway and the other uses Global VNet peering.

Traffic will pass through the public network, and we will have gateways deployed at both ends to facilitate an encrypted connection. Figure 3.4 explains how it's done:

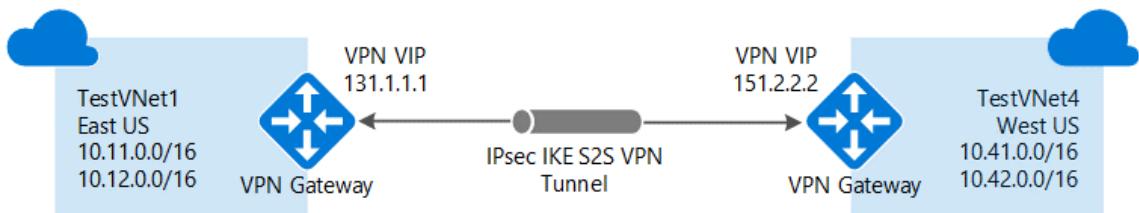


Figure 3.4: Connecting resources in different regions with different subscriptions

We will take a similar approach using Global VNet peering. Figure 3.5 shows how Global VNet peering is done:



Figure 3.5: Connecting resources in different regions using Global VNet peering

The considerations in choosing gateways or peering have already been discussed. These considerations are applicable in this scenario as well. So far, we have been connecting virtual networks across regions and subscriptions; we haven't talked about connecting an on-premises datacenter to the cloud yet. In the next section, we will discuss ways to do this.

Connecting to on-premises datacenters

Virtual networks can be connected to on-premises datacenters so that both Azure and on-premises datacenters become a single WAN. An on-premises network needs to be deployed on gateways and VPNs on both sides of the network. There are three different technologies available for this purpose.

Site-to-site VPN

This should be used when both the Azure network and the on-premises datacenter are connected to form a WAN, where any resource on both networks can access any other resource on the networks irrespective of whether they are deployed on Azure or an on-premises datacenter. VPN gateways are required to be available on both sides of networks for security reasons. Also, Azure gateways should be deployed on their own subnets on virtual networks connected to on-premises datacenters. Public IP addresses must be assigned to on-premises gateways for Azure to connect to them over the public network:

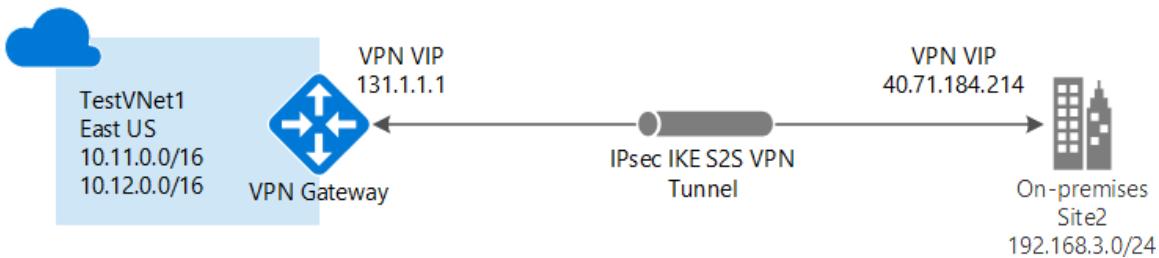


Figure 3.6: Site-to-site VPN architecture

Point-to-site VPN

This is similar to site-to-site VPN connectivity, but there is a single server or computer attached to the on-premises datacenter. It should be used when there are very few users or clients that would connect to Azure securely from remote locations. Also, there is no need for public IPs and gateways on the on-premises side in this case:

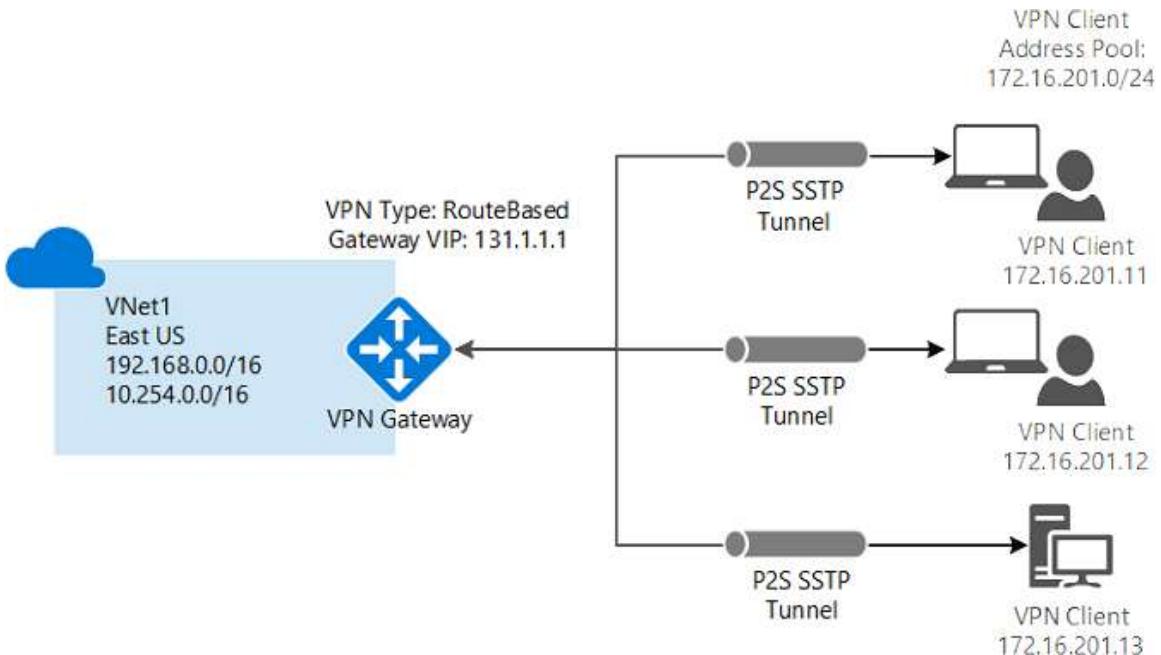


Figure 3.7: Point-to-site VPN architecture

ExpressRoute

Both site-to-site and point-to-site VPNs work using the public internet. They encrypt the traffic on the networks using VPN and certificates technology. However, there are applications that want to be deployed using hybrid technologies—some components on Azure, with others on an on-premises datacenter—and at the same time do not want to use the public internet to connect to Azure and on-premises datacenters. Azure ExpressRoute is the best solution for them, although it's a costly option compared to the two other types of connection. It is also the most secure and reliable provider, with higher speed and reduced latency because the traffic never hits the public internet. Azure ExpressRoute can help to extend on-premises networks into Azure over a dedicated private connection facilitated by a connectivity provider. If your solution is network intensive, for example, a transactional enterprise application such as SAP, use of ExpressRoute is highly recommended.

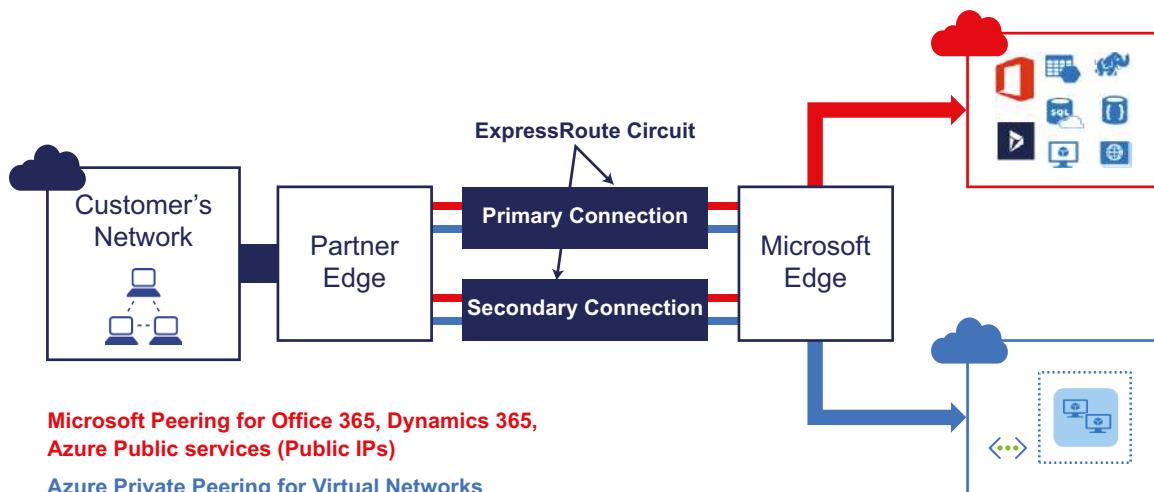


Figure 3.8: ExpressRoute network architecture

Figure 3.9 shows all three types of hybrid networks:

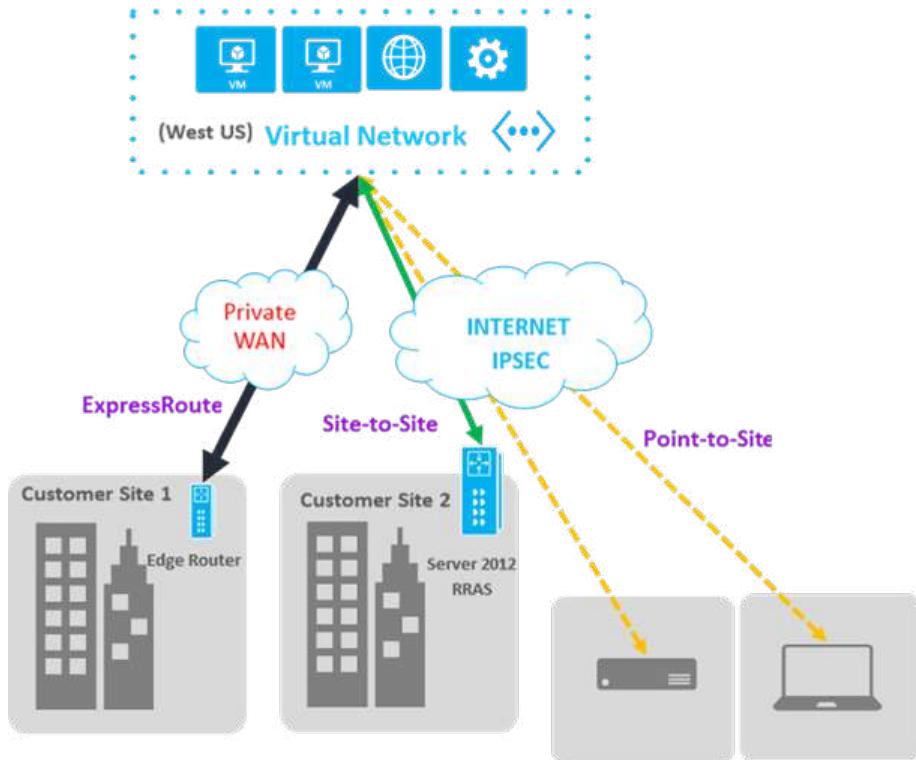


Figure 3.9: Different types of hybrid networks

It is a good practice for virtual networks to have separate subnets for each logical component with separate deployments, from a security and isolation perspective.

All the resources we deploy in Azure require networking in one way or another, so a deep understanding of networking is required when architecting solutions in Azure. Another key element is storage. In the next section, you will be learning more about storage.

Storage

Azure provides a durable, highly available, and scalable storage solution through storage services.

Storage is used to persist data for long-term needs. Azure Storage is available on the internet for almost every programming language.

Storage categories

Storage has two categories of storage accounts:

- A standard storage performance tier that allows you to store tables, queues, files, blobs, and Azure virtual machine disks.
- A premium storage performance tier supporting Azure virtual machine disks, at the time of writing. Premium storage provides higher performance and IOPS than standard general storage. Premium storage is currently available as data disks for virtual machines backed up by SSDs.

Depending on the kind of data that is being stored, the storage is classified into different types. Let's look at the storage types and learn more about them.

Storage types

Azure provides four types of general storage services:

- **Azure Blob storage:** This type of storage is most suitable for unstructured data, such as documents, images, and other kinds of files. Blob storage can be in the Hot, Cool, or Archive tier. The Hot tier is meant for storing data that needs to be accessed very frequently. The Cool tier is for data that is less frequently accessed than data in the Hot tier and is stored for 30 days. Finally, the Archive tier is for archival purposes where the access frequency is very low.
- **Azure Table storage:** This is a NoSQL key-attribute data store. It should be used for structured data. The data is stored as entities.
- **Azure Queue storage:** This provides reliable message storage for storing large numbers of messages. These messages can be accessed from anywhere via HTTP or HTTPS calls. A queue message can be up to 64 KB in size.
- **Azure Files:** This is shared storage based on the SMB protocol. It is typically used for storing and sharing files. It also stores unstructured data, but its main distinction is that it is sharable via the SMB protocol.
- **Azure disks:** This is block-level storage for Azure Virtual Machines.

These five storage types cater to different architectural requirements and cover almost all types of data storage facilities.

Storage features

Azure Storage is elastic. This means that you can store as little as a few megabytes or as much as petabytes of data. You do not need to pre-block the capacity, and it will grow and shrink automatically. Consumers just need to pay for the actual usage of storage. Here are some of the key benefits of using Azure Storage:

- Azure Storage is secure. It can only be accessed using the SSL protocol. Moreover, access should be authenticated.
- Azure Storage provides the facility to generate an account-level **Secure Access Signature (SAS)** token that can be used by storage clients to authenticate themselves. It is also possible to generate individual service-level SAS tokens for blobs, queues, tables, and files.
- Data stored in Azure storage can be encrypted. This is known as secure data at rest.
- Azure Disk Encryption is used to encrypt the OS and data disks in IaaS virtual machines. **Client-Side Encryption (CSE)** and **Storage Service Encryption (SSE)** are both used to encrypt data in Azure Storage. SSE is an Azure Storage setting that ensures that data is encrypted while data is being written to storage and decrypted while it is read by the storage engine. This ensures that no application changes are required to enable SSE. In CSE, client applications can use the Storage SDK to encrypt data before it is sent and written to Azure Storage. The client application can later decrypt this data while it is read. This provides security for both data in transit and data at rest. CSE is dependent on secrets from Azure Key Vault.
- Azure Storage is highly available and durable. What this means is that Azure always maintains multiple copies of Azure accounts. The location and number of copies depend on the replication configuration.

Azure provides the following replication settings and data redundancy options:

- **Locally redundant storage (LRS):** Within a single physical location in the primary region, there will be three replicas of your data synchronously. From a billing standpoint, this is the cheapest option; however, it's not recommended for solutions that require high availability. LRS provides a durability level of 99.99999999% for objects over a given year.
- **Zone-redundant storage (ZRS):** In the case of LRS, the replicas were stored in the same physical location. In the case of ZRS, the data will be replicated synchronously across the Availability Zones in the primary region. As each of these Availability Zones is a separate physical location in the primary region, ZRS provides better durability and higher availability than LRS.
- **Geo-redundant storage (GRS):** GRS increases the high availability by synchronously replicating three copies of data within a single primary region using LRS. It also copies the data to a single physical location in the secondary region.
- **Geo-zone-redundant storage (GZRS):** This is very similar to GRS, but instead of replicating data within a single physical location in the primary region, GZRS replicates it synchronously across three Availability Zones. As we discussed in the case of ZRS, since the Availability Zones are isolated physical locations within the primary region, GZRS has better durability and can be included in highly available designs.
- **Read-access geo-redundant storage (RA-GRS) and read-access geo-zone-redundant storage:** The data replicated to the secondary region by GZRS or GRS is not available for read or write. This data will be used by the secondary region in the case of the failover of the primary datacenter. RA-GRS and RA-GZRS follow the same replication pattern as GRS and GZRS respectively; the only difference is that the data replicated to the secondary region via RA-GRS or RA-GZRS can be read.

Now that we have understood the various storage and connection options available on Azure, let's learn about the underlying architecture of the technology.

Architectural considerations for storage accounts

Storage accounts should be provisioned within the same region as other application components. This would mean using the same datacenter network backbone without incurring any network charges.

Azure Storage services have scalability targets for capacity, transaction rate, and bandwidth associated with each of them. A general storage account allows 500 TB of data to be stored. If there is a need to store more than 500 TB of data, then either multiple storage accounts should be created, or premium storage should be used.

General storage performs at a maximum of 20,000 IOPS or 60 MB of data per second. Any requirements for higher IOPS or data managed per second will be throttled. If this is not enough for your applications from a performance perspective, either premium storage or multiple storage accounts should be used. For an account, the scalability limit for accessing tables is up to 20,000 (1 KB each) entries. The count of entities being inserted, updated, deleted, or scanned will contribute toward the target. A single queue can process approximately 2,000 messages (1 KB each) per second, and each of the **AddMessage**, **GetMessage**, and **DeleteMessage** counts will be treated as a message. If these values aren't sufficient for your application, you should spread the messages across multiple queues.

The size of virtual machines determines the size and capacity of the available data disks. While larger virtual machines have data disks with higher IOPS capacity, the maximum capacity will still be limited to 20,000 IOPS and 60 MB per second. It is to be noted that these are maximum numbers and so generally lower levels should be taken into consideration when finalizing storage architecture.

At the time of writing, GRS accounts offer a 10 Gbps bandwidth target in the US for ingress and 20 Gbps if RA-GRS/GRS is enabled. When it comes to LRS accounts, the limits are on the higher side compared to GRS. For LRS accounts, ingress is 20 Gbps and egress is 30 Gbps. Outside the US, the values are lower: the bandwidth target is 10 Gbps and 5 Gbps for egress. If there is a requirement for a higher bandwidth, you can reach out to Azure Support and they will be able to help you with further options.

Storage accounts should be enabled for authentication using SAS tokens. They should not allow anonymous access. Moreover, for blob storage, different containers should be created with separate SAS tokens generated based on the different types and categories of clients accessing those containers. These SAS tokens should be periodically regenerated to ensure that the keys are not at risk of being cracked or guessed. You will learn more about SAS tokens and other security options in *Chapter 8, Architecting secure applications on Azure*.

Generally, blobs fetched for blob storage accounts should be cached. We can determine whether the cache is stale by comparing its last modified property to re-fetch the latest blob.

Storage accounts provide concurrency features to ensure that the same file and data is not modified simultaneously by multiple users. They offer the following:

- **Optimistic concurrency:** This allows multiple users to modify data simultaneously, but while writing, it checks whether the file or data has changed. If it has, it tells the users to re-fetch the data and perform the update again. This is the default concurrency for tables.
- **Pessimistic concurrency:** When an application tries to update a file, it places a lock, which explicitly denies any updates to it by other users. This is the default concurrency for files when accessed using the SMB protocol.
- **Last writer wins:** The updates are not constrained, and the last user updates the file irrespective of what was read initially. This is the default concurrency for queues, blobs, and files (when accessed using REST).

By this point, you should know what the different storage services are and how they can be leveraged in your solutions. In the next section, we will look at design patterns and see how they relate to architectural designs.

Cloud design patterns

Design patterns are proven solutions to known design problems. They are reusable solutions that can be applied to problems. They are not reusable code or designs that can be incorporated as is within a solution. They are documented descriptions and guidance for solving a problem. A problem might manifest itself in different contexts, and design patterns can help to solve it. Azure provides numerous services, with each service providing specific features and capabilities. Using these services is straightforward, but creating solutions by weaving multiple services together can be a challenge. Moreover, achieving high availability, super scalability, reliability, performance, and security for a solution is not a trivial task.

Azure design patterns provide ready solutions that can be tailored to individual problems. They help us to make highly available, scalable, reliable, secure, and performance-centric solutions on Azure. Although there are many patterns and some of the patterns are covered in detail in subsequent chapters, some of the messaging, performance, and scalability patterns are mentioned in this chapter. Also, links are provided for detailed descriptions of these patterns. These design patterns deserve a complete book by themselves. They have been mentioned here to make you aware of their existence and to provide references for further information.

Messaging patterns

Messaging patterns help connect services in a loosely coupled manner. What this means is that services never talk to each other directly. Instead, a service generates and sends a message to a broker (generally a queue) and any other service that is interested in that message can pick it and process it. There is no direct communication between the sender and receiver service. This decoupling not only makes services and the overall application more reliable but also more robust and fault tolerant. Receivers can receive and read messages at their own speed.

Messaging helps the creation of asynchronous patterns. Messaging involves sending messages from one entity to another. These messages are created and forwarded by a sender, stored in durable storage, and finally consumed by recipients.

The top architectural concerns addressed by messaging patterns are as follows:

- **Durability:** Messages are stored in durable storage, and applications can read them after they are received in case of a failover.
- **Reliability:** Messages help implement reliability as they are persisted on disk and never lost.
- **Availability of messages:** The messages are available for consumption by applications after the restoration of connectivity and before downtime.

Azure provides Service Bus queues and topics to implement messaging patterns within applications. Azure Queue storage can also be used for the same purpose.

Choosing between Azure Service Bus queues and Queue storage is about deciding on how long the message should be stored, the size of the message, latency, and cost. Azure Service Bus provides support for 256 KB messages, while Queue storage provides support for 64 KB messages. Azure Service Bus can store messages for an unlimited period, while Queue storage can store messages for 7 days. The cost and latency are higher with Service Bus queues.

Depending on your application's requirements and needs, the preceding factors should be considered before deciding on the best queue. In the next section, we will be discussing different types of messaging patterns.

The Competing Consumers pattern

A single consumer of messages works in a synchronous manner unless the application implements the logic of reading messages asynchronously. The Competing Consumers pattern implements a solution in which multiple consumers are ready to process incoming messages, and they compete to process each message. This can lead to solutions that are highly available and scalable. This pattern is scalable because with multiple consumers, it is possible to process a higher number of messages in a smaller period. It is highly available because there should be at least one consumer to process messages even if some of the consumers crash.

This pattern should be used when each message is independent of other messages. The messages by themselves contain all the information required for a consumer to complete a task. This pattern should not be used if there is any dependency among messages. The consumers should be able to complete the tasks in isolation. Also, this pattern is applicable if there is variable demand for services. Additional consumers can be added or removed based on demand.

A message queue is required to implement the Competing Consumers pattern. Here, patterns from multiple sources pass through a single queue, which is connected to multiple consumers at the other end. These consumers should delete each message after reading so that they are not re-processed:

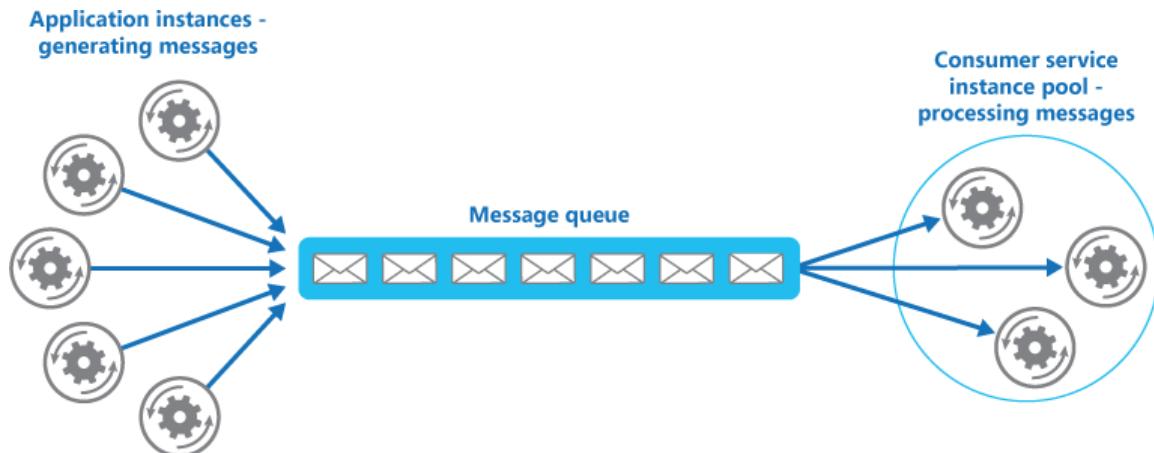


Figure 3.10: The Competing Consumers pattern

Refer to the Microsoft documentation at <https://docs.microsoft.com/azure/architecture/patterns/competing-consumers> to learn more about this pattern.

The Priority Queue pattern

There is often a need to prioritize some messages over others. This pattern is important for applications that provide different **service-level agreements (SLAs)** to consumers, which provide services based on differential plans and subscriptions.

Queues follow the first-in, first-out pattern. Messages are processed in a sequence. However, with the help of the Priority Queue pattern, it is possible to fast-track the processing of certain messages due to their higher priority. There are multiple ways to implement this. If the queue allows you to assign priority and re-order messages based on priority, then even a single queue is enough to implement this pattern:

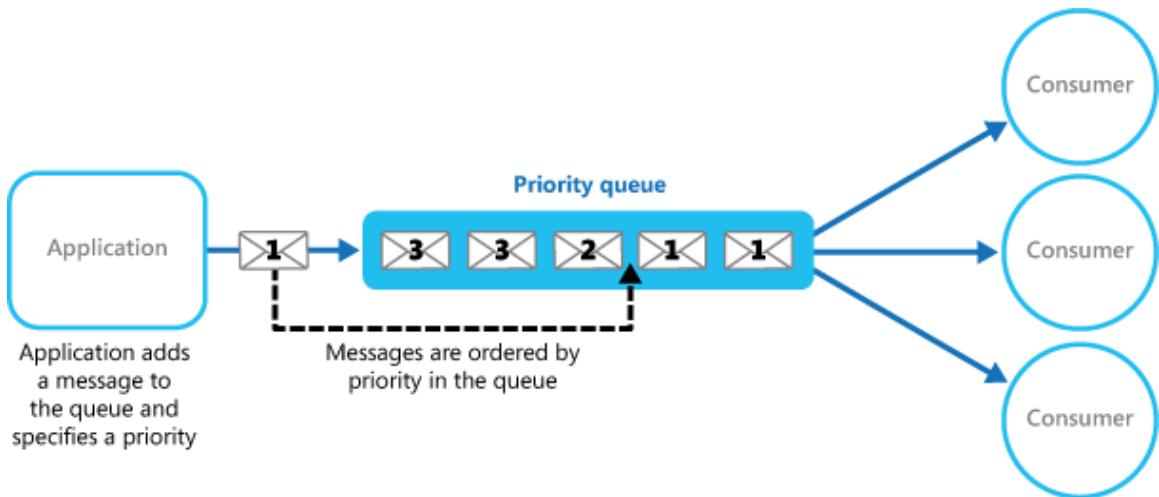


Figure 3.11: The single Priority Queue pattern

However, if the queue cannot re-order messages, then separate queues can be created for different priorities, and each queue can have separate consumers associated with it:

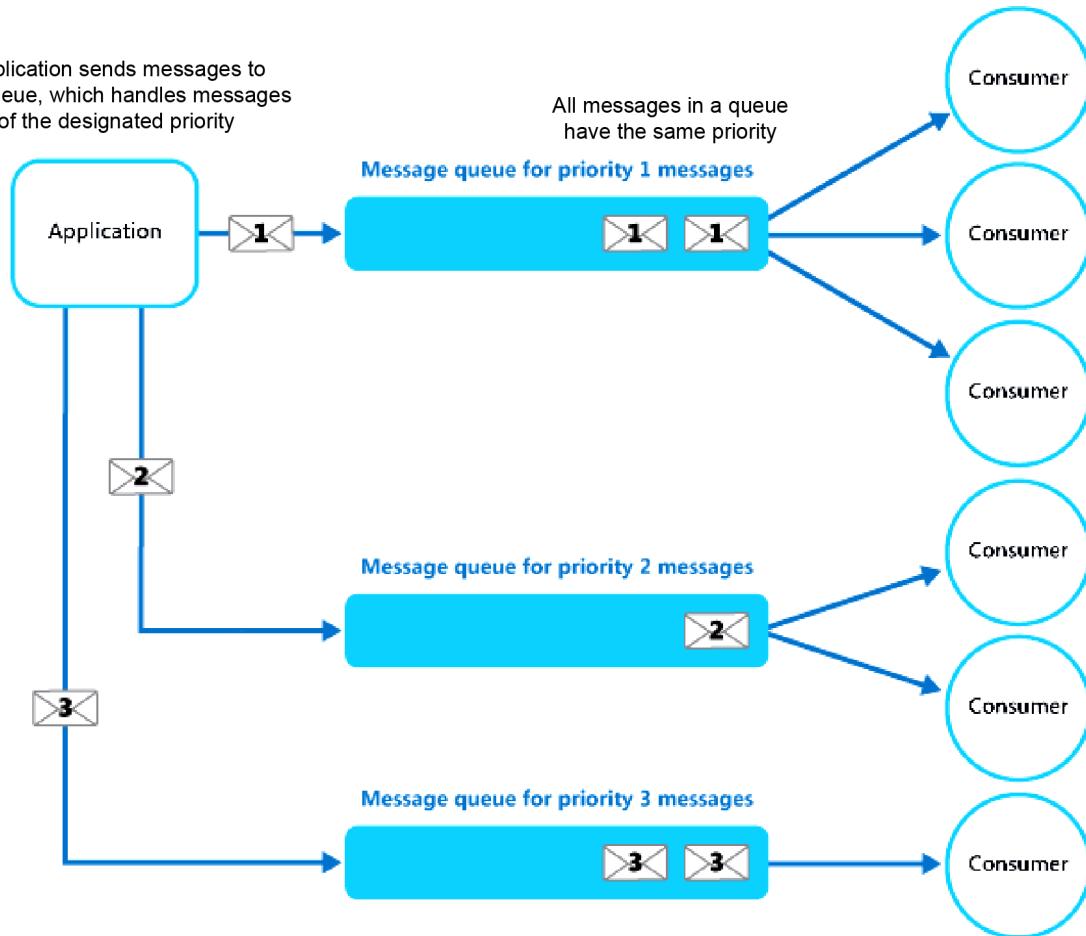


Figure 3.12: Using separate message queues for different priorities

In fact, this pattern can use the Competing Consumer pattern to fast-track the processing of messages from each queue using multiple consumers. Refer to the Microsoft documentation at <https://docs.microsoft.com/azure/architecture/patterns/priority-queue> to read more about the Priority Queue pattern.

The Queue-Based Load Leveling pattern

The Queue-Based Load Leveling pattern reduces the impact of peaks in demand on the availability and alertness of both tasks and services. Between a task and a service, a queue will act as a buffer. It can be invoked to handle the unexpected heavy loads that can cause service interruption or timeouts. This pattern helps to address performance and reliability issues. To prevent the service from getting overloaded, we will introduce a queue that will store a message until it's retrieved by the service. Messages will be taken from the queue by the service in a consistent manner and processed.

Figure 3.13 shows how the Queue-Based Load Leveling pattern works:

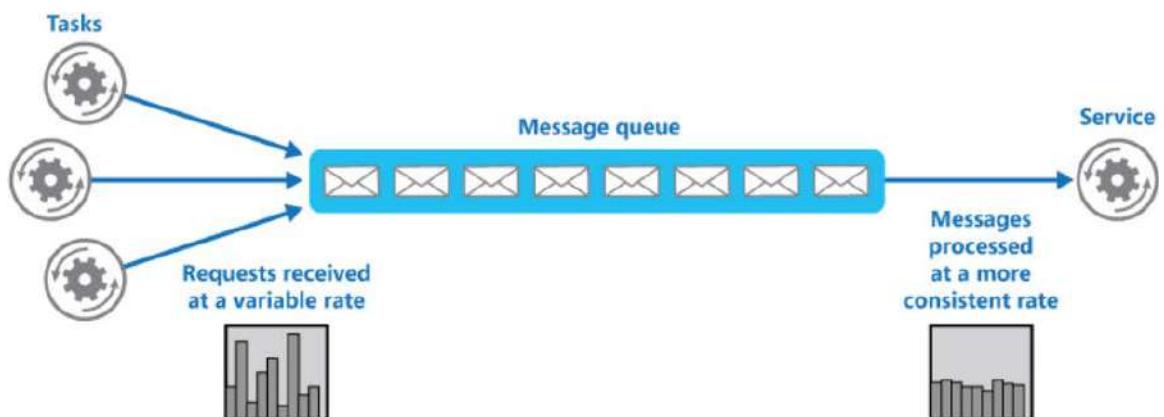


Figure 3.13: The Queue-Based Load Leveling pattern

Even though this pattern helps to handle spikes of unexpected demand, it is not the best choice when you are architecting a service with minimal latency. Talking of latency, which is a performance measurement, in the next section we will be focusing on performance and scalability patterns.

Performance and scalability patterns

Performance and scalability go together. Performance is the measure of how quickly a system can execute an action within a given time interval in a positive manner. On the other hand, scalability is the ability of a system to handle unexpected load without affecting the performance of the system, or how quickly the system can be expanded with the available resources. In this section, a couple of design patterns related to performance and scalability will be described.

The Command and Query Responsibility Segregation (CQRS) pattern

CQRS is not an Azure-specific pattern but a general pattern that can be applied in any application. It increases the overall performance and responsiveness of an application.

CQRS is a pattern that segregates the operations that read data (queries) from the operations that update data (commands) by using separate interfaces. This means that the data models used for querying and updates are different. The models can then be isolated, as shown in Figure 3.14, although that's not an absolute requirement.

This pattern should be used when there are large and complex business rules executed while updating and retrieving data. Also, this pattern has an excellent use case in which one team of developers can focus on the complex domain model that is part of the write model, and another team can focus on the read model and the user interfaces. It is also wise to use this pattern when the ratio of read to write is skewed. The performance of data reads should be fine-tuned separately from the performance of data writes.

CQRS not only improves the performance of an application, but it also helps the design and implementation of multiple teams. Due to its nature of using separate models, CQRS is not suitable if you are using model and scaffolding generation tools:

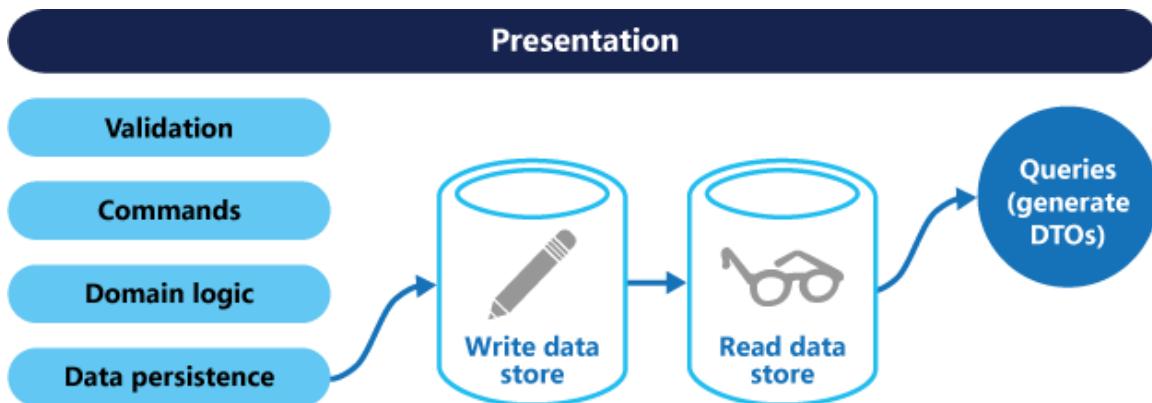


Figure 3.14: The CQRS pattern

Refer to the Microsoft documentation at <https://docs.microsoft.com/azure/architecture/patterns/cqrs> to read more about this pattern.

The Event Sourcing pattern

As most applications work with data and as the users are working with it, the classic approach for the application would be to maintain and update the current state of the data. Reading data from the source, modifying it, and updating the current state with the modified value is the typical data processing approach. However, there are some limitations:

- As the update operations are directly made against the data store, this will slow down the overall performance and responsiveness.
- If there are multiple users working on and updating the data, there may be conflicts and some of the relevant updates may fail.

The solution for this is to implement the Event Sourcing pattern, where the changes will be recorded in an append-only store. A series of events will be pushed by the application code to the event store, where they will be persisted. The events persisted in an event store act as a system of record about the current state of data. Consumers will be notified, and they can handle the events if needed once they are published.

The Event Sourcing pattern is shown in Figure 3.15:

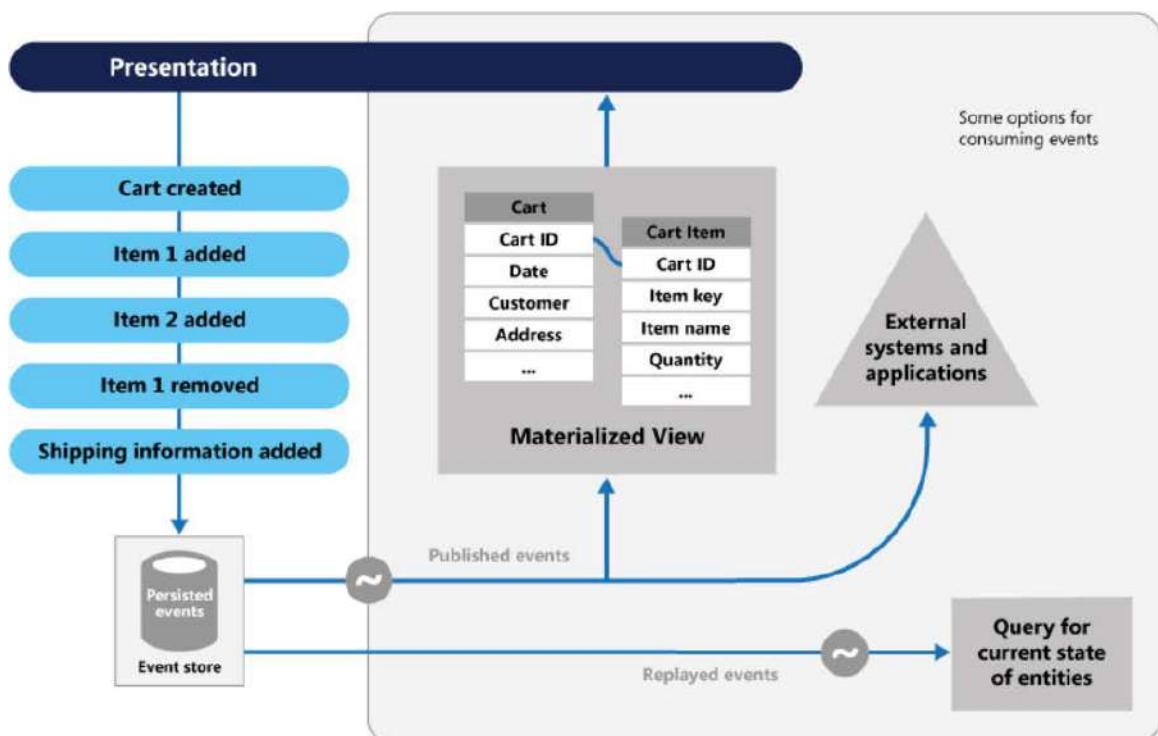


Figure 3.15: The Event Sourcing pattern

More information about this pattern is available at <https://docs.microsoft.com/azure/architecture/patterns/event-sourcing>.

The Throttling pattern

At times, there are applications that have very stringent SLA requirements from a performance and scalability perspective, irrespective of the number of users consuming the service. In these circumstances, it is important to implement the Throttling pattern because it can limit the number of requests that are allowed to be executed. The load on applications cannot be predicted accurately in all circumstances. When the load on an application spikes, throttling reduces pressure on the servers and services by controlling the resource consumption. The Azure infrastructure is a very good example of this pattern.

This pattern should be used when meeting the SLA is a priority for applications to prevent some users from consuming more resources than allocated, to optimize spikes and bursts in demand, and to optimize resource consumption in terms of cost. These are valid scenarios for applications that have been built to be deployed on the cloud.

There can be multiple strategies for handling throttling in an application. The Throttling strategy can reject new requests once the threshold is crossed, or it can let the user know that the request is in the queue and it will get the opportunity to be executed once the number of requests is reduced.

Figure 3.16 illustrates the implementation of the Throttling pattern in a multi-tenant system, where each tenant is allocated a fixed resource usage limit. Once they cross this limit, any additional demand for resources is constrained, thereby maintaining enough resources for other tenants:

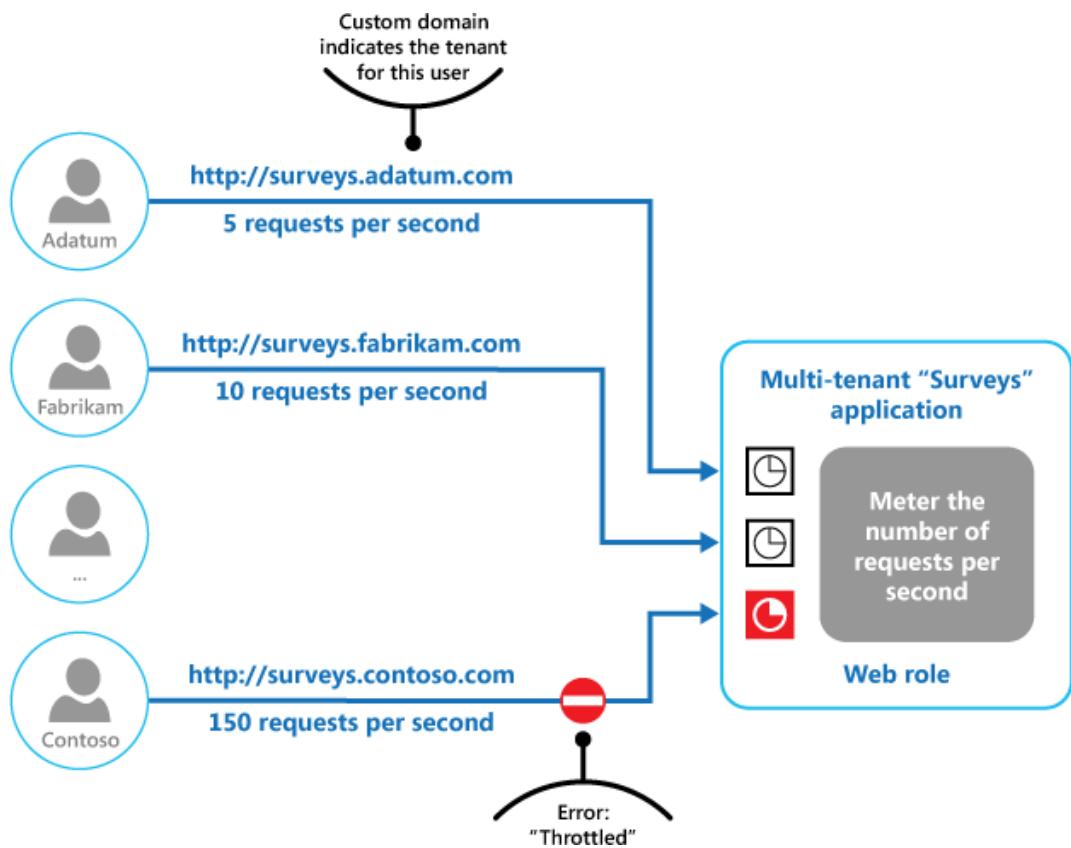


Figure 3.16: The Throttling pattern

Read more about this pattern at <https://docs.microsoft.com/azure/architecture/patterns/throttling>.

Retry pattern

The Retry pattern is an extremely important pattern that makes applications and services more resilient to transient failures. Imagine you are trying to connect to and use a service, and the service is not available for some reason. If the service is going to become available soon, it makes sense to keep trying to get a successful connection. This will make the application more robust, fault tolerant, and stable. In Azure, most of the components are running on the internet, and that internet connection can produce transient faults intermittently. Since these faults can be rectified within seconds, an application should not be allowed to crash. The application should be designed in a manner that means it can try to use the service again repeatedly in the case of failure and stop retrying when either it is successful or it eventually determines that there is a fault that will take time to rectify.

This pattern should be implemented when an application could experience transient faults as it interacts with a remote service or accesses a remote resource. These faults are expected to be short-lived, and repeating a request that has previously failed could succeed on a subsequent attempt.

The Retry pattern can adopt different retry strategies depending on the nature of the errors and the application:

- **Retry a fixed number of times:** This denotes that the application will try to communicate with the service a fixed number of times before determining that there's been a failure and raising an exception. For example, it will retry three times to connect to another service. If it is successful in connecting within these three tries, the entire operation will be successful; otherwise, it will raise an exception.
- **Retry based on schedule:** This denotes that the application will try to communicate with the service repeatedly for a fixed number of seconds or minutes and wait for a fixed number of seconds or minutes before retrying. For example, the application will try to connect to the service every three seconds for 60 seconds. If it is successful in connecting within this time, the entire operation will be successful. Otherwise, it will raise an exception.
- **Sliding and delaying the retry:** This denotes that the application will try to communicate with the service repeatedly based on the schedule and keep adding an incremental delay in subsequent tries. For example, for a total of 60 seconds, the first retry happens after a second, the second retry happens two seconds after the previous retry, the third retry happens four seconds after the previous retry, and so on. This reduces the overall number of retries.

Figure 3.17 illustrates the Retry pattern. The first request gets an HTTP 500 response, the second retry again gets an HTTP 500 response, and finally the request is successful and gets HTTP 200 as the response:

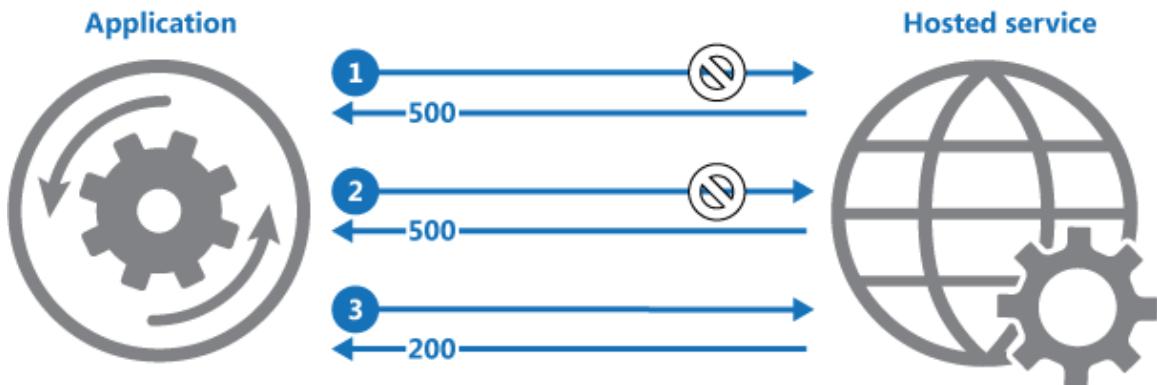


Figure 3.17: The Retry pattern

Refer to this Microsoft documentation at <https://docs.microsoft.com/azure/architecture/patterns/retry> to find out more about this pattern.

The Circuit Breaker pattern

This is an extremely useful pattern. Imagine again that you are trying to connect to and use a service, and the service is not available for some reason. If the service is not going to become available soon, there is no use continuing to retry the connection. Moreover, keeping other resources occupied while retrying wastes a lot of resources that could potentially be used elsewhere.

The Circuit Breaker pattern helps eliminate this waste of resources. It can prevent applications from repeatedly trying to connect to and use a service that is not available. It also helps applications to detect whether a service is up and running again, and allow applications to connect to it.

To implement the Circuit Breaker pattern, all requests to the service should pass through a service that acts as a proxy to the original service. The purpose of this proxy service is to maintain a state machine and act as a gateway to the original service. There are three states that it maintains. There could be more states included, depending on the application's requirements.

The minimal states needed to implement this pattern are as follows:

- **Open:** This denotes that the service is down and the application is shown as an exception immediately, instead of allowing it to retry or wait for a timeout. When the service is up again, the state is transitioned to Half-Open.
- **Closed:** This state denotes that the service is healthy and the application can go ahead and connect to it. Generally, a counter shows the number of failures before it can transition to the Open state.
- **Half-Open:** At some point, when the service is up and running, this state allows a limited number of requests to pass through it. This state is a litmus test that checks whether the requests that pass through are successful. If the requests are successful, the state is transitioned from Half-Open to Closed. This state can also implement a counter to allow a certain number of requests to be successful before it can transition to Closed.

The three states and their transitions are illustrated in *Figure 3.18*:

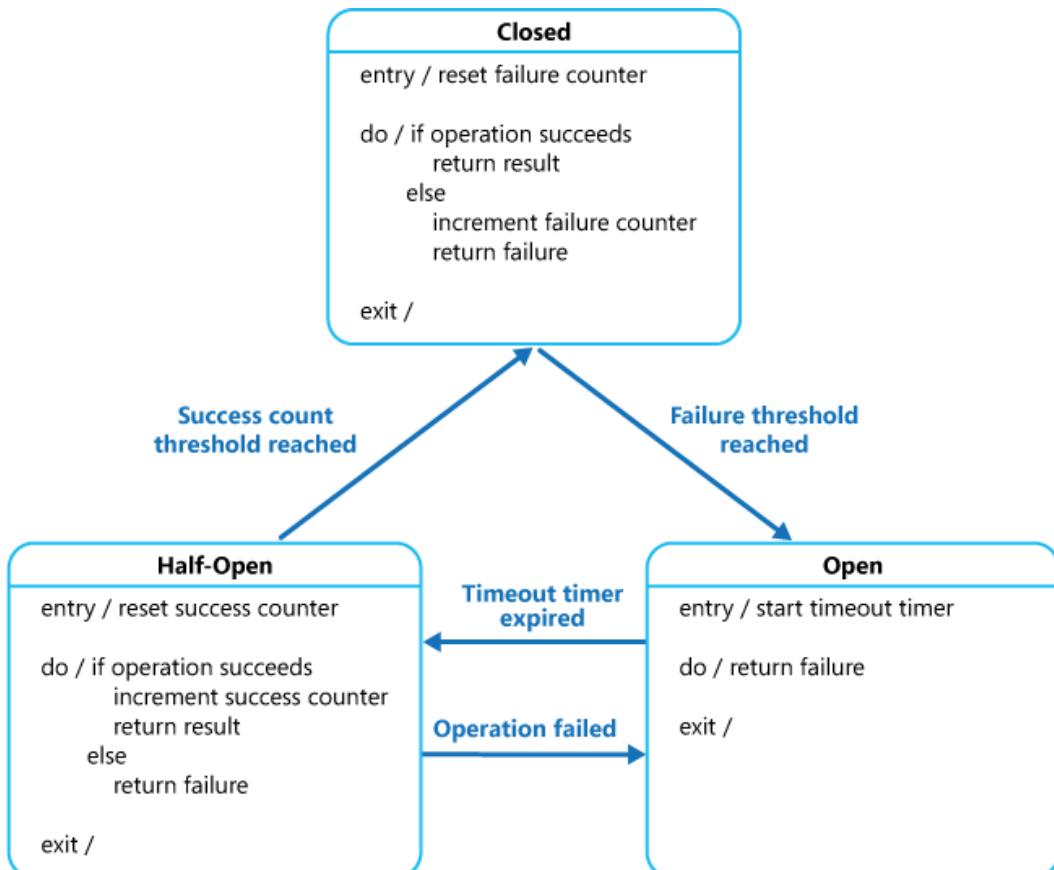


Figure 3.18: The Circuit Breaker pattern

Read more this pattern in the Microsoft documentation at <https://docs.microsoft.com/azure/architecture/patterns/circuit-breaker>.

In this section, we discussed design patterns that can be used to architect reliable, scalable, and secure applications in the cloud. There are other patterns, though, which you can explore at <https://docs.microsoft.com/azure/architecture/patterns>.

Summary

There are numerous services available on Azure, and most of them can be combined to create real solutions. This chapter explained the three most important services provided by Azure—regions, storage, and networks. They form the backbone of the majority of solutions deployed on any cloud. This chapter provided details about these services and how their configuration and provisioning can affect design decisions.

Important considerations for both storage and networks were detailed in this chapter. Both networks and storage provide lots of choices, and it is important to choose an appropriate configuration based on your requirements.

Finally, some of the important design patterns related to messaging, such as Competing Consumers, Priority Queue, and Load Leveling, were described. Patterns such as CQRS and Throttling were illustrated, and other patterns, such as Retry and Circuit Breaker, were also discussed. We will keep these patterns as the baseline when we deploy our solutions.

In the next chapter, we will be discussing how to automate the solutions we are going to architect. As we move ahead in the world of automation, every organization wants to eliminate the overhead of creating resources one by one, which is very demanding. Since automation is the solution for this, in the next chapter you will learn more about it.

4

Automating architecture on Azure

Every organization wants to reduce manual effort and error in their pursuits, and automation plays an important role in bringing about predictability, standardization, and consistency in both building a product and in operations. Automation has been the focus of almost every **Chief information officer (CIO)** and digital officer to ensure that their systems are highly available, scalable, reliable, and able to cater to the needs of their customers.

Automation became more prominent with the advent of the cloud because new resources can be provisioned on the fly without the procurement of hardware resources. Hence, cloud companies want automation in almost all of their activities to reduce misuse, errors, governance, maintenance, and administration.

In this chapter, we will evaluate Azure Automation as a major service that provides automation capabilities, along with its differentiating capabilities compared to other apparently similar-looking services. This chapter will cover the following:

- The Azure Automation landscape
- The Azure Automation service
- Resources for Azure Automation services
- Writing Azure Automation runbooks
- Webhooks
- Hybrid Workers

Let's get started with Azure Automation, a cloud service for process automation.

Automation

Automation is needed for the provisioning, operations, management, and deprovisioning of IT resources within an organization. Figure 4.1 gives you a closer look at what each of these use cases represents:

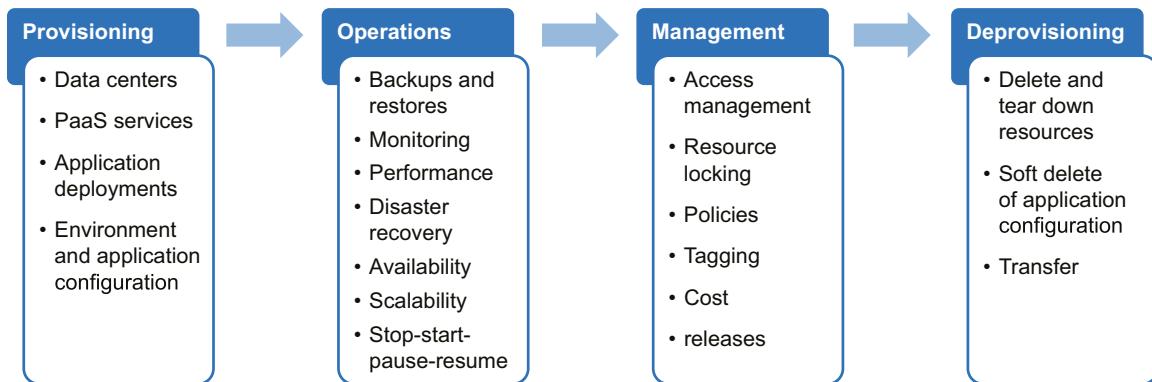


Figure 4.1: Use cases of automation

Before the advent of the cloud, IT resources were primarily on-premises, and manual processes were often used for these activities. However, since cloud adoption has increased, automation has found increased focus and attention. The primary reason is that cloud technology's agility and flexibility provide an opportunity to provision, deprovision, and manage these resources on the fly in a tiny fraction of the time it used to take. Along with this flexibility and agility come the requirements to be more predictable and consistent with the cloud because it has become easy for organizations to create resources.

Microsoft has a great tool for IT automation known as System Center Orchestrator. It is a great tool for automation for on-premises and cloud environments, but it is a product and not a service. It should be licensed and deployed on servers, and then runbooks can be executed to effect changes on cloud and on-premises environments.

Microsoft realized that an automation solution was required that could be provided to customers as a service rather than bought and deployed as a product. Enter Azure Automation.

Azure Automation

Azure provides a service called **Azure Automation**, which is an essential service for the automation of processes, activities, and tasks not only on Azure but also on-premises as well. Using Azure Automation, organizations can automate their processes and tasks related to processing, tear-down, operations, and the management of their resources across the cloud, IT environments, platforms, and languages. In *Figure 4.2*, we can see some features of Azure Automation:

Cross-cloud	Cross-environment	Cross-platform	Cross-language
<ul style="list-style-type: none">• Azure• Other clouds• Any combination	<ul style="list-style-type: none">• Cloud• On-premises• Hybrid	<ul style="list-style-type: none">• Linux• Windows	<ul style="list-style-type: none">• PowerShell• Python• Bash

Figure 4.2: Features of Azure Automation

Azure Automation architecture

Azure Automation comprises multiple components, and each of these components is completely decoupled from the others. Most of the integration happens at the data store level, and no components talk to each other directly.

When an Automation account is created on Azure, it is managed by a management service. The management service is a single point of contact for all activities within Azure Automation. All requests from the portal, including saving, publishing, and creating runbooks, to execution, stopping, suspending, starting, and testing are sent to the automation management service and the service writes the request data to its data store. It also creates a job record in the data store and, based on the status of the runbook workers, assigns it to a worker.

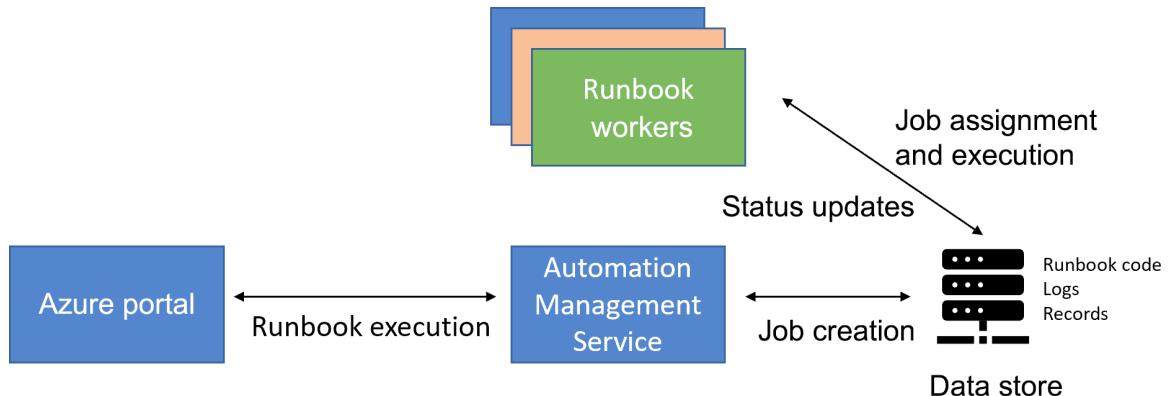


Figure 4.3: Azure Automation architecture

The worker keeps polling the database for any new jobs assigned to it. Once it finds a job assignment, it fetches the job information and starts executing the job using its execution engine. The results are written back to the database, read by the management service, and displayed back on the Azure portal.

The Hybrid Workers that we will read about later in this chapter are also runbook workers, although they're not shown in Figure 4.3.

The first step in getting started with Azure Automation is to create a new account. Once the account is created, all other artifacts are created within the account.

The account acts as the main top-level resource that can be managed using Azure resource groups and its own control plane.

The account should be created within a region, and all automation within this account gets executed on servers in that region.

It is important to choose the region wisely, preferably close to other Azure resources that the Automation account integrates or manages, to reduce the network traffic and latency between the regions.

The Automation account also supports a couple of **Run As** accounts, which can be created from the Automation account. As these Run As accounts are analogous to a service account, we mostly create them to execute actions. Even though we generally say Run As account, there are two types of Run As account: one is called the Azure Classic Run As account, and the other one is simply the Run As account, and both of them are used to connect to Azure subscriptions. The Azure Classic Run As account is for connecting to Azure using the **Azure Service Management API**, and the Run As account is for connecting to Azure using the **Azure Resource Management (ARM) API**.

Both of these accounts use certificates to authenticate with Azure. These accounts can be created while creating the Automation account, or you can opt to create them at a later stage from the Azure portal.

It is recommended to create these Run As accounts later instead of creating them while creating the Automation account because if they are created while setting up the Automation account, Automation will generate the certificates and service principals behind the scenes with the default configuration. If more control and custom configuration is needed for these Run As accounts, such as using an existing certificate or service principal, then the Run As accounts should be created after the Automation account.

Once the Automation account is created, it provides a dashboard through which multiple automation scenarios can be enabled.

Some of the important scenarios that can be enabled using an Automation account are related to:

- Process automation
- Configuration management
- Update management

Automation is about writing scripts that are reusable and generic so that they can be reused in multiple scenarios. For example, an automation script should be generic enough to start and stop any VM in any resource group in any subscription and management group. Hardcoding VM server information, along with resource group, subscription, and management group names, will result in the creation of multiple similar scripts, and any change in one will undoubtedly result in changing all the scripts. It is better to create a single script for this purpose by using scripting parameters and variables, and you should ensure that the values are supplied by the executor for these artifacts.

Let's take a closer look at each of the aforementioned scenarios.

Process automation

Process automation refers to the development of scripts that reflect real-world processes. Process automation comprises multiple activities, where each activity performs a discrete task. Together, these activities form a complete process. The activities might be executed on the basis of whether the previous activity executed successfully or not.

There are some requirements that any process automation requires from the infrastructure it is executed on. Some of them are as follows:

- The ability to create workflows
- The ability to execute for a long duration
- The ability to save the execution state when the workflow is not complete, which is also known as checkpointing and hydration
- The ability to resume from the last saved state instead of starting from the beginning

The next scenario we are going to explore is configuration management.

Configuration management

Configuration management refers to the process of managing the system configuration throughout its life cycle. Azure Automation State Configuration is the Azure configuration management service that allows users to write, manage, and compile PowerShell DSC configuration for cloud nodes and on-premises datacenters.

Azure Automation State Configuration lets us manage Azure VMs, Azure Classic VMs, and physical machines or VMs (Windows/Linux) on-premises, and it also provides support for VMs in other cloud providers.

One of the biggest advantages of Azure Automation State Configuration is it provides scalability. We can manage thousands of machines from a single central management interface. We can assign configurations to machines with ease and verify whether they are compliant with the desired configuration.

Another advantage is that Azure Automation can be used as a repository to store your **Desired State Configuration (DSC)** configurations, and at the time of need they can be used.

In the next section, we will be talking about update management.

Update management

As you already know, update management is the responsibility of the customer to manage updates and patches when it comes to IaaS. The Update Management feature of Azure Automation can be used to automate or manage updates and patches for your Azure VMs. There are multiple methods by which you can enable Update Management on your Azure VM:

- From your Automation account
- By browsing the Azure portal
- From a runbook
- From an Azure VM

Enabling it from an Azure VM is the easiest method. However, if you have a large number of VMs and need to enable Update Management, then you have to consider a scalable solution such as a runbook or from an Automation account.

Now that you are clear about the scenarios, let's explore the concepts related to Azure Automation.

Concepts related to Azure Automation

You now know that Azure Automation requires an account, which is called an Azure Automation account. Before we dive deeper, let's examine the concepts related to Azure Automation. Understanding the meaning of each of these terms is very important, as we are going to use these terms throughout this chapter. Let's start with runbook.

Runbook

An Azure Automation runbook is a collection of scripting statements representing a single step in process automation or a complete process automation. It is possible to invoke other runbooks from a parent runbook, and these runbooks can be authored in multiple scripting languages. The languages that support authoring runbooks are as follows:

- PowerShell
- Python 2 (at the time of writing)
- PowerShell workflows
- Graphical PowerShell
- Graphical PowerShell workflows

Creating an Automation account is very easy and can be done from the Azure portal. In the **All Services** blade, you can find **Automation Account**, or you can search for it in the Azure portal. As mentioned before, during creation you will get an option to create a Run As account. Figure 4.4 shows the inputs required to create an Automation account:

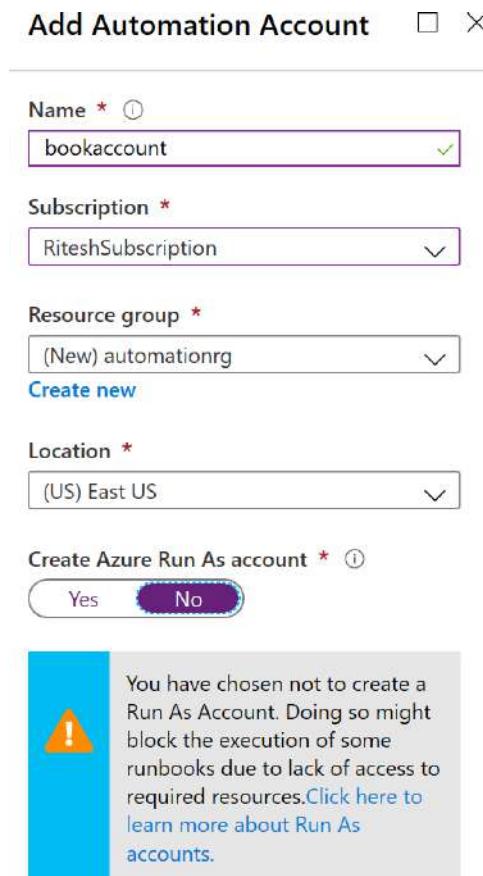


Figure 4.4: Creating an Automation account

Run As accounts

Azure Automation accounts, by default, do not have access to any resources included in any Azure subscription, including the subscription in which they are hosted. An account needs access to an Azure subscription and its resources in order to manage them. A Run As account is one way to provide access to subscriptions and the resources within them.

This is an optional exercise. There can be at most one Run As account for each classic and resource manager-based subscription; however, an Automation account might need to connect to numerous subscriptions. In such cases, it is advisable to create shared resources for each of the subscriptions and use them in runbooks.

After creating the Automation account, navigate to the **Run as accounts** view on the portal and you will see that two types of accounts can be created. In Figure 4.5, you can see that the option to create an **Azure Run As Account** and an **Azure Classic Run As Account** is available in the **Run as accounts** blade:

The screenshot shows the 'Run as accounts' blade for the 'bookaccount' Automation Account. The left sidebar has a 'Search (Ctrl+ /)' input field and a 'Run as accounts' tab selected. The main area has a heading 'Run As accounts in Azure Automation are used to provide authentication for managing resources' and links to 'Learn more about Run As accounts and how to manage them', 'Learn more about changing Run As role assignments', and 'Learn more about co-administrator permission required to configure Classic Run As accounts'. Below this are two sections: '+ Azure Run As Account' with a 'Create' button, and '+ Azure Classic Run As Account' with a 'Create' button.

Figure 4.5: Azure Run As Account options

These Run As accounts can be created using the Azure portal, PowerShell, and the CLI. For information about creating these accounts using PowerShell, visit <https://docs.microsoft.com/azure/automation/manage-runas-account>.

In the case of the ARM Run As account, this script creates a new Azure AD service principal and a new certificate and provides contributor RBAC permissions to the newly created service principal on the subscription.

Jobs

The submission of a job request is not linked directly to the execution of the job request because of Azure Automation's decoupled architecture. The linkage between them is indirect using a data store. When a request to execute a runbook is received by Automation, it creates a new record in its database with all the relevant information. There is another service running on multiple servers, known as Hybrid Runbook Worker, within Azure, which looks for any new entries added to the database for the execution of a runbook. Once it sees a new record, it locks the record so that no other service can read it and then executes the runbook.

Assets

Azure Automation assets refer to shared artifacts that can be used across runbooks. They are shown in *Figure 4.6*:

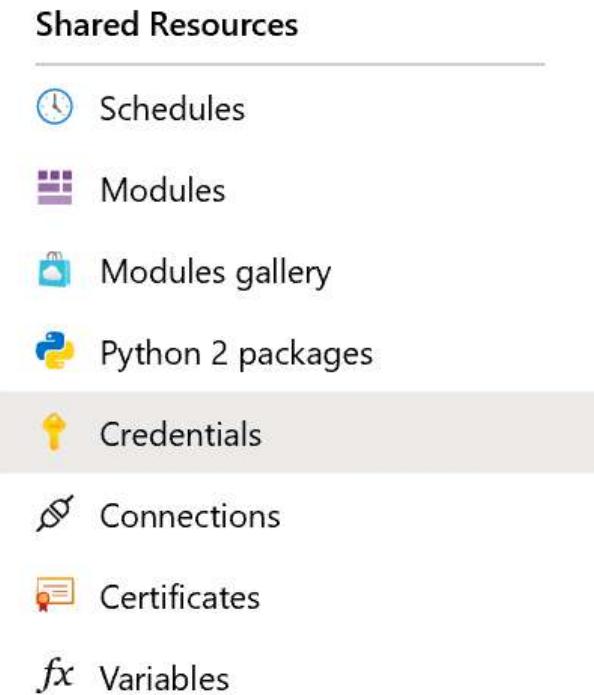


Figure 4.6: Shared artifacts in Azure Automation

Credentials

Credentials refers to the secrets, such as the username/password combination, that can be used to connect to other integration services that need authentication. These credentials can be used within runbooks using the **Get-AutomationPSCredential** PowerShell cmdlet along with its associated name:

```
$myCredential = Get-AutomationPSCredential -Name 'MyCredential'
```

The Python syntax requires that we import the **automationassets** module and use the **get_automation_credential** function along with the associated credential name:

```
import automationassets  
cred = automationassets.get_automation_credential("credtest")
```

Certificates

Certificates refers to the X.509 certificate that can be purchased from certificate authorities or can be self-signed. Certificates are used for identification purposes in Azure Automation. Every certificate has a pair of keys known as private/public keys. The private key is used for creating a certificate asset in Azure Automation, and the public key should be available in the target service. Using the private key, the Automation account can create a digital signature and append it to the request before sending it to the target service. The target service can fetch the details (the hash) from the digital signature using the already available public key and ascertain the identity of the sender of the request.

Certificate assets store certificate information and keys in Azure Automation. These certificates can be used directly within runbooks, and they are also used by the connection's assets. The next section shows the way to consume certificates in a connection asset. The Azure service principal connection asset uses a certificate thumbprint to identify the certificate it wants to use, while other types of connection use the name of the certificate asset to access the certificate.

A certificate asset can be created by providing a name and uploading a certificate. It is possible to upload public certificates (.cer files) as well as private certificates (.pfx files). The private part of the certificate also has a password that should be used before accessing the certificate.

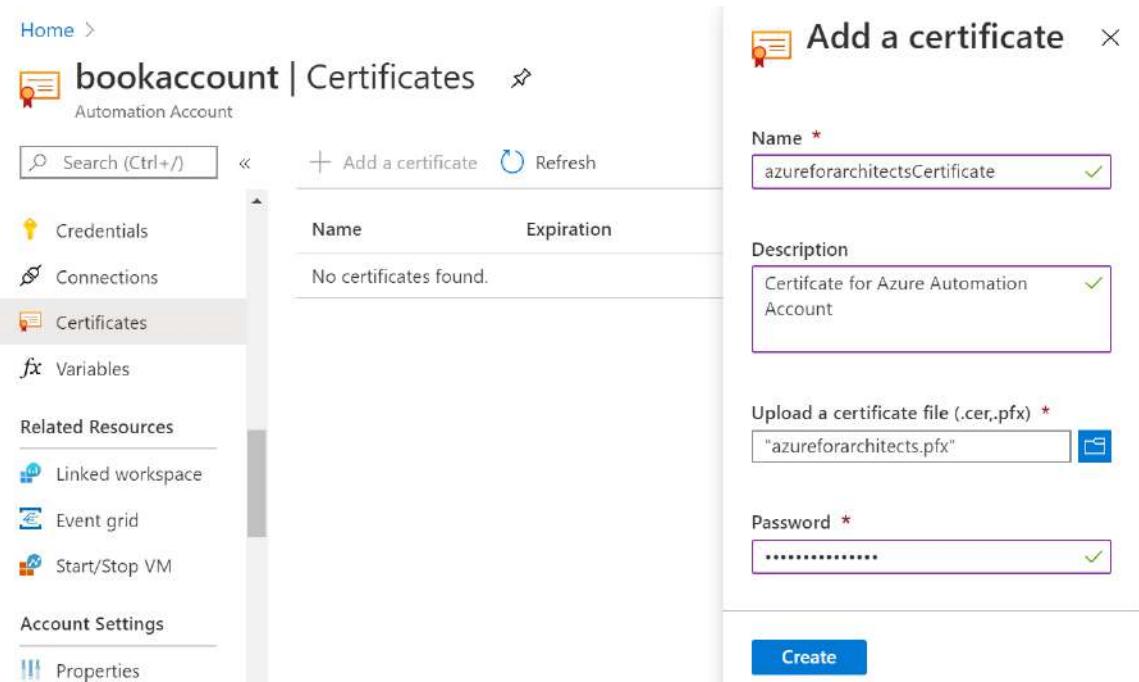


Figure 4.7: Adding a certificate to Azure Automation

Creating a certificate involves providing a name and a description, uploading the certificate, providing a password (in the case of .pfx files), and informing the user whether the certificate is exportable or not.

There should be a certificate available before this certificate asset can be created. Certificates can be purchased from certificate authorities or can be generated. Generated certificates are known as self-signed certificates. It is always a good practice to use certificates from certificate authorities for important environments such as production environments. It is fine to use self-signing certificates for development purposes.

To generate a self-signed certificate using PowerShell, use this command:

```
$cert = New-SelfSignedCertificate -CertStoreLocation "Cert:\CurrentUser\my"
-KeySpec KeyExchange -Subject "cn=azureforarchitects"
```

This will create a new certificate in the current user certificate store in your personal folder. Since this certificate also needs to be uploaded to the Azure Automation certificate asset, it should be exported to the local file system, as shown in *Figure 4.8*:

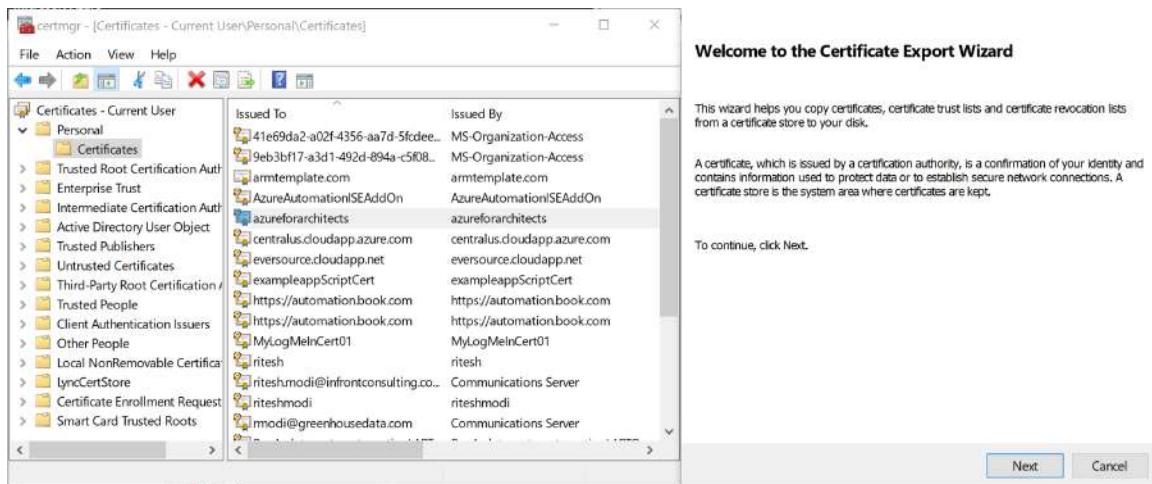


Figure 4.8: Exporting the certificate

When exporting the certificate, the private key should also be exported, so **Yes, export the private key** should be selected.

Select the **Personal Information Exchange** option, and the rest of the values should remain as the defaults.

Provide a password and the filename **C:\azureforarchitects.pfx**, and the export should be successful.

Connecting to Azure can be done in multiple ways. However, the most secure is by way of a certificate. A service principal is created on Azure using the certificate. The service principal can be authenticated against using the certificate. The private key of the certificate is with the user and the public part is with Azure. In the next section, a service principal will be created using the certificate created in this section.

Creating a service principal using certificate credentials

A service principal can be created using the Azure portal, Azure CLI, or Azure PowerShell. The script for creating a service principal using Azure PowerShell is available in this section.

After logging into Azure, the certificate created in the previous section is converted into base64 encoding. A new service principal, **azureforarchitects**, is created, and the certificate credential is associated with the newly created service principal. Finally, the new service principal is provided contributor role-based access control permissions on the subscription:

```
Login-AzAccount

$certKey = [system.Convert]::ToBase64String($cert.GetRawCertData())

$sp = New-AzADServicePrincipal -DisplayName "azureforarchitects"

New-AzADSpCredential -ObjectId $sp.Id -CertValue $certKey -StartDate
$cert.NotBefore -EndDate $cert.NotAfter

New-AzRoleAssignment -RoleDefinitionName contributor -ServicePrincipalName
$sp.ApplicationId

Get-AzADServicePrincipal -ObjectId $sp.Id

$cert.Thumbprint

Get-AzSubscription
```

To create a connection asset, the application ID can be obtained using the **Get-AzADServicePrincipal** cmdlet, and the result is shown in Figure 4.9:

```
ServicePrincipalNames : {http://azureforarchitects, ef52538d-9eb6-45e0-bf67-7f484b84cd25}
ApplicationId        : ef52538d-9eb6-45e0-bf67-7f484b84cd25
ObjectType          : ServicePrincipal
DisplayName         : azureforarchitects
Id                  : 15ee7335-9b96-4ae7-957f-62e4997e7b4d
Type                :
```

Figure 4.9: Checking the service principal

The certificate thumbprint can be obtained using the certificate reference along with **SubscriptionId**, which can be obtained using the **Get-AzSubscription** cmdlet.

Connections

Connection assets are used for creating connection information to external services. In this regard, even Azure is considered as an external service. Connection assets hold all the necessary information needed for successfully connecting to a service. There are three connection types provided out of the box by Azure Automation:

- Azure
- Azure classic certificate
- Azure service principal

It is a good practice to use Azure service principal to connect to Azure Resource Manager resources and to use the Azure classic certificate for Azure classic resources. It is important to note that Azure Automation does not provide any connection type to connect to Azure using credentials such as a username and password.

Azure and Azure classic certificates are similar in nature. They both help us connect to Azure Service management API-based resources. In fact, Azure Automation creates an Azure classic certificate connection while creating a Classic Run As account.

Azure service principal is used internally by Run As accounts to connect to Azure Resource Manager-based resources.

A new connection asset of type **AzureServicePrincipal** is shown in *Figure 4.10*. It needs:

- The name of the connection. It is mandatory to provide a name.
- A description of the connection. This value is optional.
- Select an appropriate **Type**. It is mandatory to select an option; **AzureServicePrincipal** is selected for creating a connection asset for all purposes in this chapter.
- **ApplicationId**, also known as **clientId**, is the application ID generated during the creation of a service principal. The next section shows the process of creating a service principal using Azure PowerShell. It is mandatory to provide an application ID.
- **TenantId** is the unique identifier of the tenant. This information is available from the Azure portal or by using the **Get-AzSubscription** cmdlet. It is mandatory to provide a tenant identifier.
- **CertificateThumbprint** is the certificate identifier. This certificate should already be uploaded to Azure Automation using the certificate asset. It is mandatory to provide a certificate thumbprint.
- **SubscriptionId** is the identifier of the subscription. It is mandatory to provide a subscription ID.

You can add a new connection using the **Connections** blade in the Automation account, as shown in *Figure 4.10*:

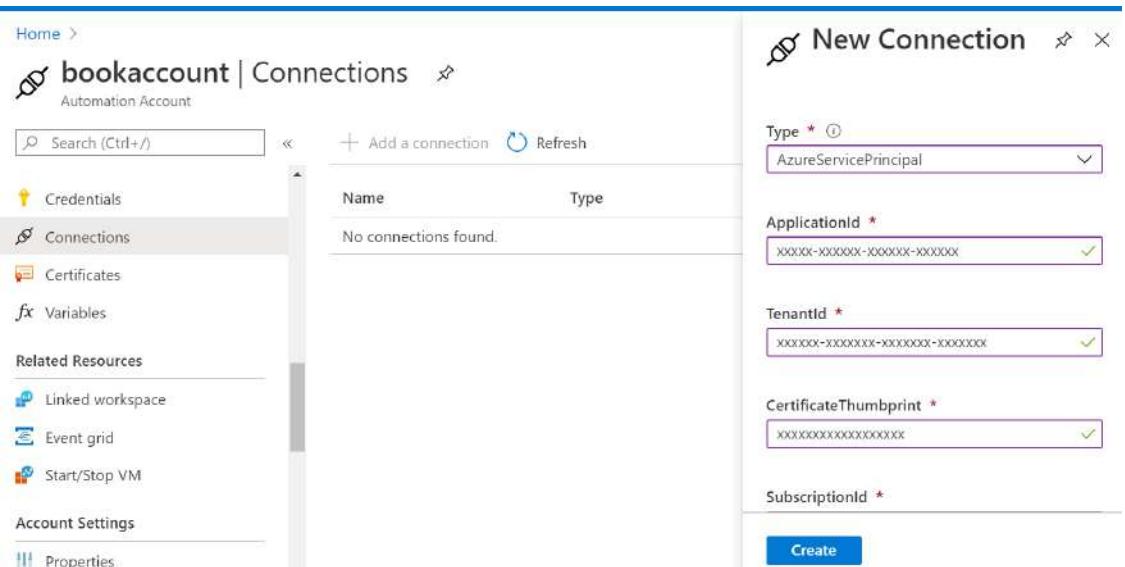


Figure 4.10: Adding a new connection to the Automation Account

Runbook authoring and execution

Azure Automation allows the creation of automation scripts known as runbooks. Multiple runbooks can be created using the Azure portal or PowerShell ISE. They can also be imported from **Runbook Gallery**. The gallery can be searched for specific functionality, and the entire code is displayed within the runbook.

A runbook can accept parameter values just like a normal PowerShell script. The next example takes a single parameter named **connectionName** of type **string**. It is mandatory to supply a value for this parameter when executing this runbook:

```
param(  
    [parameter(mandatory=$true)]  
    [string] $connectionName  
)  
  
$connection = Get-AutomationConnection -name $connectionName  
$subscriptionid = $connection.subscriptionid  
$tenantid = $connection.tenantid  
$applicationid = $connection.applicationid  
$cretThumbprint = $connection.CertificateThumbprint  
  
Login-AzureRMAccount -CertificateThumbprint $cretThumbprint  
-ApplicationId $applicationid -ServicePrincipal -Tenant $tenantid  
  
Get-AzureRMVM
```

The runbook uses the **Get-AutomationConnection** cmdlet to reference the shared connection asset. The name of the asset is contained within the parameter value. Once the reference to the connection asset has been made, the values from the connection reference are populated into the **\$connection** variable, and subsequently, they are assigned to multiple other variables.

The **Login-AzureRMAccount** cmdlet authenticates with Azure, and it supplies the values obtained from the connection object. It uses the service principal created earlier in this chapter for authentication.

Finally, the runbook invokes the **Get-AzureRMVm** cmdlet to list all the VMs in the subscription.

By default, Azure Automation still provides **AzureRM** modules for working with Azure. It does not install **Az** modules by default. Later we will install an **Az** module manually in the Azure Automation account and use cmdlets in runbooks.

Parent and child runbooks

Runbooks have a life cycle, from being authored to being executed. These life cycles can be divided into authoring status and execution status.

The authoring life cycle is shown in *Figure 4.11*.

When a new runbook is created, it has the **New** status and as it is edited and saved multiple times, it takes the **In edit** status, and finally, when it is published, the status changes to **Published**. It is also possible to edit a published runbook, and in that case, it goes back to the **In edit** status.



Figure 4.11: Authoring life cycle

The execution life cycle is described next.

The life cycle starts with the beginning of a runbook execution request. A runbook can be executed in multiple ways:

- Manually from the Azure portal
- By using a parent runbook as a child runbook
- By means of a webhook

It does not matter how a runbook is initiated; the life cycle remains the same. A request to execute the runbook is received by the Automation engine. The Automation engine creates a job and assigns it to a runbook worker. Currently, the runbook has a status of **Queued**.

There are multiple runbook workers, and the chosen one picks up the job request and changes the status to **Starting**. At this stage, if there are any scripting and parsing issues in the script, the status changes to **Failed** and the execution is halted.

Once the runbook execution is started by the worker, the status is changed to **Running**. The runbook can have multiple different statuses once it is running.

The runbook will change its status to **Completed** if the execution happens without any unhandled and terminating exceptions.

The running runbook can be manually stopped by the user, and it will have the **Stopped** status.

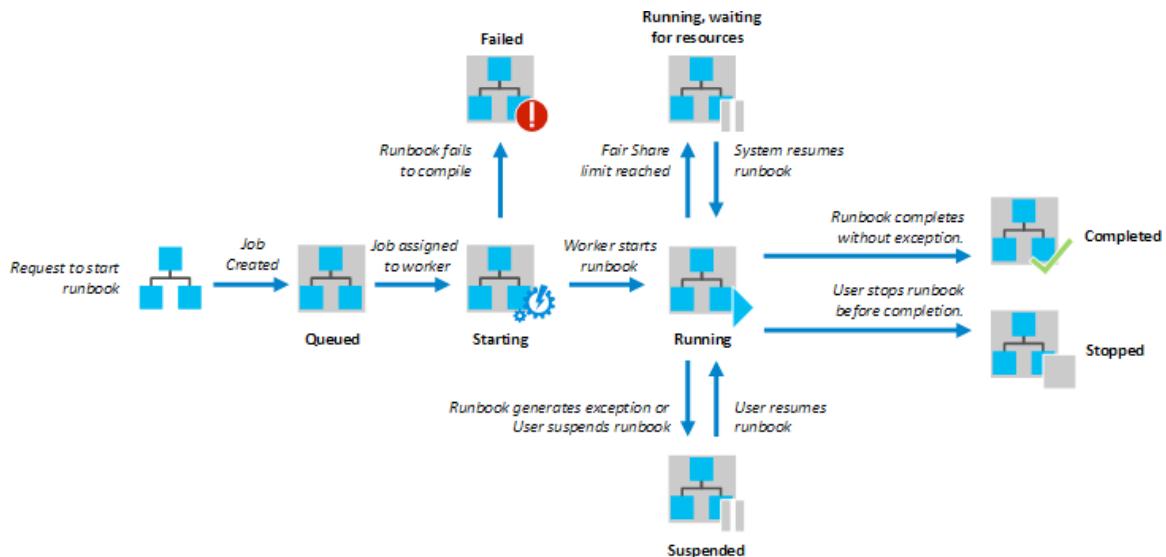


Figure 4.12: The execution life cycle for runbooks

The user can also suspend and resume the execution of the runbook.

Creating a runbook

A runbook can be created from the Azure portal by going to the **Runbook** menu item in the left navigation pane. A runbook has a name and type. The type determines the scripting language used for creating the runbook. We have already discussed the possible languages, and in this chapter, PowerShell will be used primarily for all examples.

Creating a PowerShell runbook is exactly the same as creating a PowerShell script. It can declare and accept multiple parameters—the parameters can have attributes such as data types, which are mandatory (just like any PowerShell parameter attributes). It can invoke PowerShell cmdlets whose modules are available and already loaded and declared, and it can invoke functions and return output.

A runbook can also invoke another runbook. It can invoke a child runbook inline within the original process and context or in a separate process and context.

Invoking a runbook inline is similar to invoking a PowerShell script. The next example invokes a child runbook using the inline approach:

```
.\ConnectAzure.ps1 -connectionName "azureforarchitectsconnection"
```

Get-AzSqlServer

In the preceding code, we saw how the **ConnectAzure** runbook accepts a parameter named **connectionName** and an appropriate value is supplied to it. This runbook creates a connection to Azure after authenticating with it using a service principal. Check out the syntax for invoking the child runbook. It is very similar to invoking a general PowerShell script along with parameters.

The next line of code, **Get-AzVm**, fetches the relevant information from Azure and lists the VM details. You will notice that although the authentication happens within a child runbook, the **Get-AzVm** cmdlet succeeds and lists all the VMs in the subscription because the child runbook executes in the same job as that of the parent runbook, and they share the context.

Alternatively, a child runbook can be invoked using the **Start-AzurermAutomationRunbook** cmdlet provided by Azure Automation. This cmdlet accepts the name of the Automation account, the resource group name, and the name of the runbook along with parameters, as mentioned here:

```
$params = @{"connectionName"="azureforarchitectsconnection"}

$job = Start-AzurermAutomationRunbook 'ConnectAzure' -AutomationAccountName 'bookaccount' -Name 'ConnectAzure' -ResourceGroupName 'automationrg' -parameters $params

if($job -ne $null) {
    Start-Sleep -s 100
    $job = Get-AzureAutomationJob -Id $job.Id -AutomationAccountName 'bookaccount'

    if ($job.Status -match "Completed") {
        $jobout = Get-AzureAutomationJobOutput -Id $job.Id -AutomationAccountName 'bookaccount' -Stream Output

        if ($jobout) {Write-Output $jobout.Text}
    }
}
```

Using this approach creates a new job that's different from the parent job, and they run in different contexts.

Using Az modules

So far, all examples have used **AzureRM** modules. The previously shown runbooks will be re-written to use cmdlets from the **Az** module.

As mentioned before, **Az** modules are not installed by default. They can be installed using the **Modules gallery** menu item in Azure Automation.

Search for **Az** in the gallery and the results will show multiple modules related to it. If the **Az** module is selected to be imported and installed, it will throw an error saying that its dependent modules are not installed and that they should be installed before installing the current module. The module can be found on the **Modules gallery** blade by searching for **Az**, as shown in *Figure 4.13*:

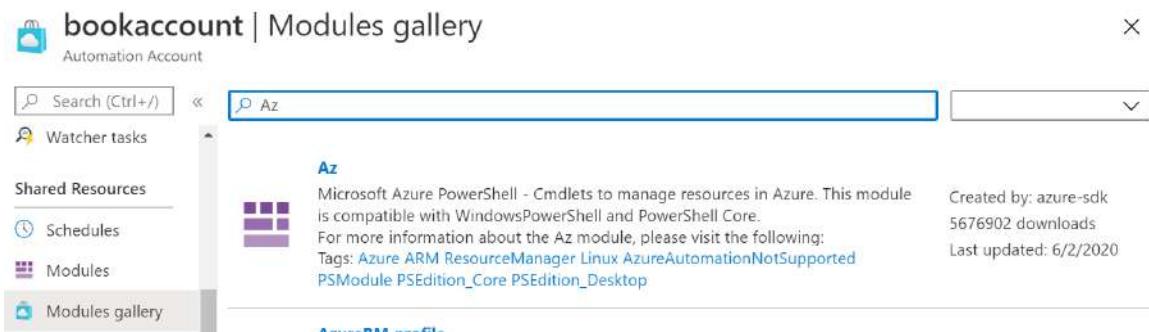


Figure 4.13: Finding the Az module on the Modules gallery blade

Instead of selecting the **Az** module, select **Az.Accounts** and import the module by following the wizard, as shown in *Figure 4.14*:

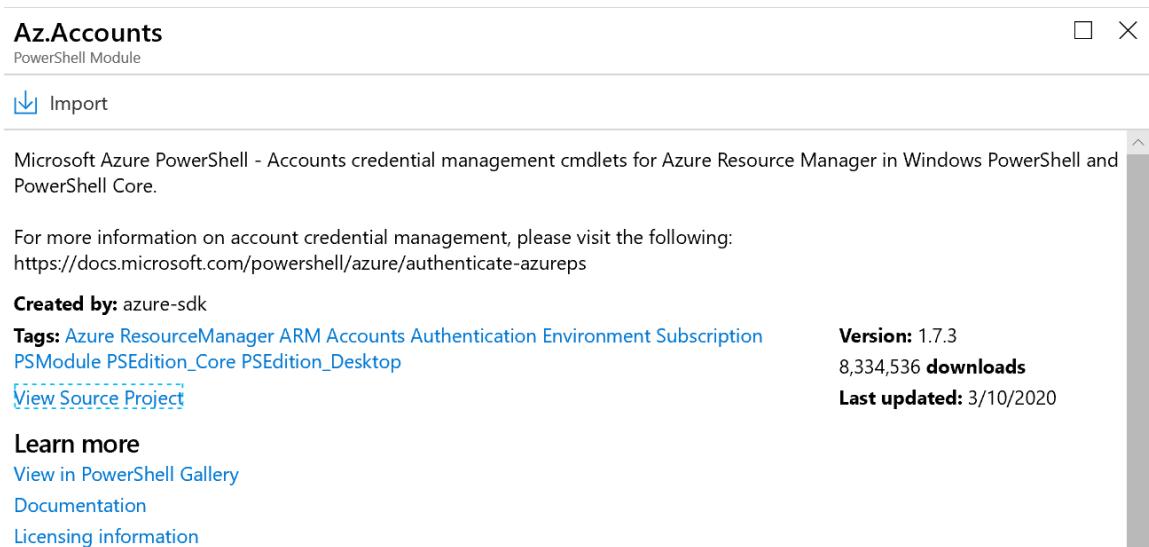


Figure 4.14: Importing the Az.Accounts module

After installing **Az.Accounts**, the **Az.Resources** module can be imported. Azure virtual machine-related cmdlets are available in the **Az.Compute** module, and it can also be imported using the same method as we used to import **Az.Accounts**.

Once these modules are imported, the runbooks can use the cmdlets provided by these modules. The previously shown **ConnectAzure** runbook has been modified to use the **Az** module:

```
param(  
    [parameter(mandatory=$true)]  
    [string] $connectionName  
)  
  
$connection = Get-AutomationConnection -name $connectionName  
$subscriptionid = $connection.subscriptionid  
$tenantid = $connection.tenantid  
$applicationid = $connection.applicationid  
$cretThumbprint = $connection.CertificateThumbprint  
  
Login-AzAccount -CertificateThumbprint $cretThumbprint  
-ApplicationId $applicationid -ServicePrincipal  
-Tenant $tenantid -SubscriptionId $subscriptionid  
  
Get-AzVm
```

The last two lines of the code are important. They are using **Az** cmdlets instead of **AzureRM** cmdlets.

Executing this runbook will give results similar to this:

The screenshot shows the Azure Runbooks interface. At the top, it displays the navigation path: Home > bookaccount | Runbooks > ConnectAzure (bookaccount/ConnectAzure) > ConnectAzure 3/26/2020, 7:19 AM. Below this, the title "ConnectAzure 3/26/2020, 7:19 AM" is shown with a "Job" icon. A toolbar below the title includes "Resume", "Stop", "Suspend", and "Refresh" buttons. The main content area shows the runbook's properties: Id (587f7695-fa49-4f99-8a9c-772f1a6c4788), Status (Completed), Ran on (localrunbookexecutionengine), Ran As (User), Created (3/26/2020, 7:19:25 AM), Last Update (3/26/2020, 7:20:05 AM), Runbook (ConnectAzure), and Source snapshot (View source snapshot). Below these details is a tabs section with "Input", "Output" (which is selected and highlighted in blue), "Errors", "Warnings", "All Logs", and "Exception". The "Output" tab displays the runbook's output, which includes several lines of JSON-like configuration for a virtual machine named "spark" in the "SPARK" resource group. The configuration includes properties like VmId, Name, Type, Location, LicenseType, Tags, AvailabilitySetReference, DiagnosticsProfile, Extensions, HardwareProfile, InstanceView, NetworkProfile, OSProfile, BillingProfile, Plan, ProvisioningState, StorageProfile, DisplayHint, Identity, Zones, FullyQualifiedDomainName, AdditionalCapabilities, and ProximityPlacementGroup.

```

9755ffce-e94b-4332-9be8-1ade15e78909
771f1cf4-b1ac-4f2e-ad21-de39ea201e7e
ef52538d-9eb6-45e0-bf67-7f484b84cd25
43A06770461183DE1725548BD760A85B7FB9171A

Environments
-----
{[AzureChinaCloud, AzureChinaCloud], [AzureCloud, AzureCloud], [AzureGermanCloud, AzureGermanCloud], [AzureUSGovernme...}

ResourceGroupName : SPARK
Id : /subscriptions/9755ffce-e94b-4332-9be8-1ade15e78909/resourceGroups/SPARK/providers/Microsoft.Compute/virtualMachines/spark
VmId : 3a10b076-559d-41c2-b0c6-ab60674f2ae
Name : spark
Type : Microsoft.Compute/virtualMachines
Location : westeurope
LicenseType :
Tags :
AvailabilitySetReference :
DiagnosticsProfile :
Extensions :
HardwareProfile : Microsoft.Azure.Management.Compute.Models.HardwareProfile
InstanceView :
NetworkProfile : Microsoft.Azure.Management.Compute.Models.NetworkProfile
OSProfile : Microsoft.Azure.Management.Compute.Models.OSProfile
BillingProfile :
Plan :
ProvisioningState : Succeeded
StorageProfile : Microsoft.Azure.Management.Compute.Models.StorageProfile
DisplayHint : Compact
Identity :
Zones :
FullyQualifiedDomainName :
AdditionalCapabilities :
ProximityPlacementGroup :

```

Figure 4.15: The Az.Accounts module successfully imported

In the next section, we will work with webhooks.

Webhooks

Webhooks became famous after the advent of REST endpoints and JSON data payloads. Webhooks are an important concept and architectural decision in the extensibility of any application. Webhooks are placeholders that are left within special areas of an application so that the user of the application can fill those placeholders with endpoint URLs containing custom logic. The application will invoke the endpoint URL, automatically passing in the necessary parameters, and then execute the logic available therein.

Azure Automation runbooks can be invoked manually from the Azure portal. They can also be invoked using PowerShell cmdlets and the Azure CLI. There are SDKs available in multiple languages that are capable of invoking runbooks.

Webhooks are one of the most powerful ways to invoke a runbook. It is important to note that runbooks containing the main logic should never be exposed directly as a webhook. They should be called using a parent runbook, and the parent runbook should be exposed as a webhook. The parent runbook should ensure that appropriate checks are made before invoking the main child runbook.

The first step in creating a webhook is to author a runbook normally, as done previously. After a runbook has been authored, it will be exposed as a webhook.

A new PowerShell-based runbook named **exposedrunbook** is created. This runbook takes a single parameter, **\$WebhookData**, of the object type. It should be named **verbatim**. This object is created by the Azure Automation runtime and is supplied to the runbook. The Azure Automation runtime constructs this object after obtaining the HTTP request header values and body content and fills in the **RequestHeader** and **RequestBody** properties of this object:

```
param(  
    [parameter(mandatory=$true)]  
    [object] $WebhookData  
  
)  
  
$webhookname = $WebhookData.WebhookName  
$headers = $WebhookData.RequestHeader  
$body = $WebhookData.RequestBody  
  
Write-output "webhook header data"
```

```

Write-Output $webhookname
Write-output $headers.message
Write-output $headers.subject

$connectionname = (ConvertFrom-Json -InputObject $body)

./connectAzure.ps1 -connectionName $connectionname[0].name

```

The three important properties of this object are **WebhookName**, **RequestHeader**, and **RequestBody**. The values are retrieved from these properties and sent to the output stream by the runbook.

The header and body content can be anything that the user supplies when invoking the webhook. These values get filled up into the respective properties and become available within the runbook. In the previous example, there are two headers set by the caller, namely **message** and **status** header. The caller will also supply the name of the shared connection to be used as part of the body content.

After the runbook is created, it should be published before a webhook can be created. After publishing the runbook, clicking on the **Webhook** menu at the top starts the process of creating a new webhook for the runbook, as shown in Figure 4.16:

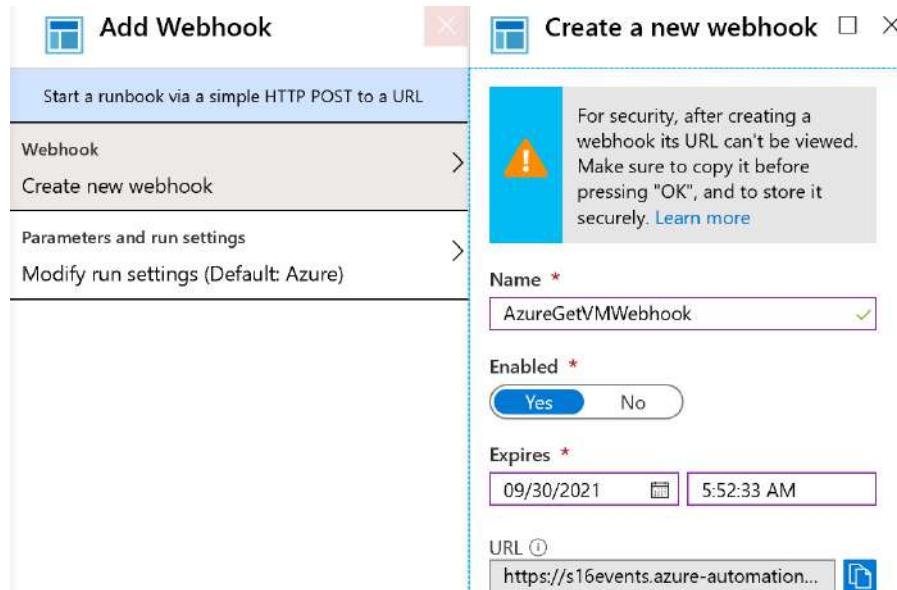


Figure 4.16: Creating a webhook

A name for the webhook should be provided. This value is available within the runbook using the **WebhookData** parameter with the **WebhookName** property name.

The webhook can be in the **enabled** or **disabled** state, and it can expire at a given date and time. It also generates a URL that is unique for this webhook and runbook. This URL should be provided to anyone who wishes to invoke the webhook.

Invoking a webhook

Webhooks are invoked as HTTP requests using the **POST** method. When a webhook is invoked, the HTTP request lands up with Azure Automation to start a runbook. It creates the **WebHookData** object, filling it with the incoming HTTP header and body data, and creates a job to be picked up by a runbook worker. This call uses the webhook URL generated in the previous step.

The webhook can be invoked using Postman, by any code having the capability of calling a **REST** endpoint using the **POST** method. In the next example, PowerShell will be used to invoke the webhook:

```
$uri = "https://s16events.azure-automation.net/
webhooks?token=rp0w93L60fAPYZQ4vryxl%2aN%2bS1Hz4F3qVdUaKUDzgM%3d"

$connection = @(
    @{ name="azureforarchitectsconnection" }

)

$body = ConvertTo-Json -InputObject $connection
$header = @{ subject="VMS specific to Ritesh";message="Get all virtual
machine details"}

$response = Invoke-WebRequest -Method Post -Uri $uri -Body $body -Headers
$header
$jobid = (ConvertFrom-Json ($response.Content)).jobids[0]
```

The PowerShell code declares the URL for the webhook and constructs the body in JSON format, with **name** set to **azureforarchitectsconnection** and a header with two header name-value pairs – **subject** and **message**. Both the header and body data can be retrieved in the runbook using the **WebhookData** parameter.

The **Invoke-WebRequest** cmdlet raises the request on the previously mentioned endpoint using the **POST** method, supplying both the header and the body.

The request is asynchronous in nature, and instead of the actual runbook output, the job identifier is returned as an HTTP response. It is also available within the response content. The job is shown in Figure 4.17:

The screenshot shows the Azure Automation Job details page. At the top, it displays the job name "exposedRunbook" and the run time "3/26/2020, 6:25 AM". Below this are buttons for "Resume" (with a play icon), "Stop" (with a square icon), "Suspend" (with a pause icon), and "Refresh" (with a circular arrow icon). The job status is listed as "Completed". Below the status, it says "Ran on: Azure" and "Ran As: User". There are tabs for "Input" (which is selected), "Output", "Errors", "Warnings", "All Logs", and "Exception". Under the "Input" tab, there is a table with one row labeled "Name" and the value "WEBHOOKDATA".

Figure 4.17: Checking the job

Clicking on **WEBHOOKDATA** shows the values that arrived in the runbook automation service in the HTTP request:

The screenshot shows a modal window titled "Input parameter" with the sub-label "WEBHOOKDATA". The content area contains a JSON string representing the webhook data:

```
{"WebhookName": "AzureGetVMWebhook", "RequestBody": "[\r\n  {\r\n    \"name\": \"azureforarchitectsconnection\"\r\n  }\r\n]", "RequestHeader": {"Host": "s16events.azure-automation.net", "User-Agent": "Mozilla/5.0", "message": "Get all virtual machine details", "subject": "VMS specific to Ritesh", "x-ms-request-id": "34da1397-f0d1-4ec4-b86a-fd705bfaaf3"}}
```

Figure 4.18: Verifying the output

Clicking on the output menu shows the list of VMs and SQL Server in the subscription.

The next important concepts in Azure Automation are Azure Monitor and Hybrid Workers, and the next sections will explain them in detail.

Invoking a runbook from Azure Monitor

Azure Automation runbooks can be invoked as responses to alerts generated within Azure. Azure Monitor is the central service that manages logs and metrics across resources and resource groups in a subscription. You can use Azure Monitor to create new alert rules and definitions that, when triggered, can execute Azure Automation runbooks. They can invoke an Azure Automation runbook in its default form or a webhook that in turn can execute its associated runbook. This integration between Azure Monitor and the ability to invoke runbooks opens numerous automation opportunities to autocorrect the environment, scale up and down compute resources, or take corrective actions without any manual intervention.

Azure alerts can be created and configured in individual resources and resource levels, but it is always a good practice to centralize alert definitions for easy and better maintenance and administration.

Let's go through the process of associating a runbook with an alert and invoking the runbook as part of the alert being raised.

The first step is to create a new alert, as shown in *Figure 4.19*:

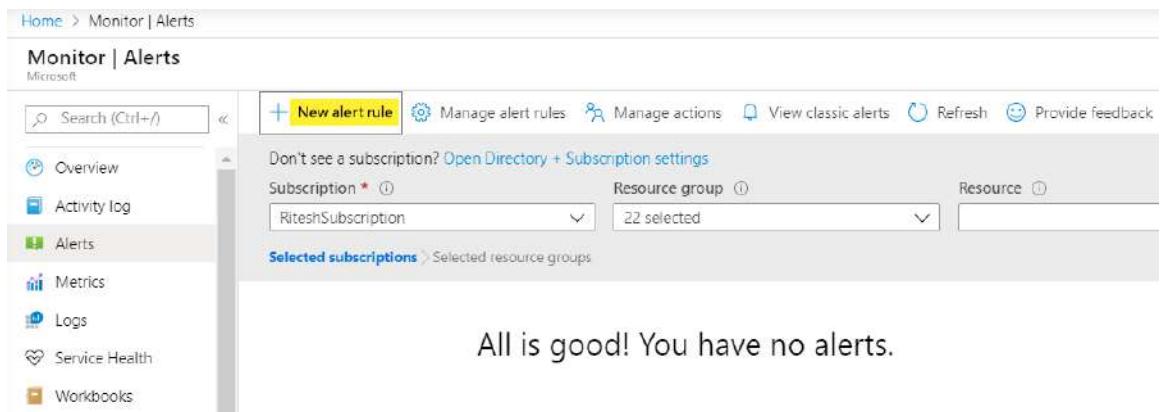


Figure 4.19: Creating an alert rule

Select a resource that should be monitored and evaluated for alert generation. A resource group has been selected from the list, and it automatically enables all resources within the resource group. It is possible to remove the resource selections from the resource group:

Select a resource

Select the resource(s) you want to monitor. Available signal types for your selection will show up on the bottom right.

Filter by subscription * ⓘ Filter by resource type ⓘ Filter by location ⓘ

RiteshSubscription	Virtual machines	All
<input type="text"/> Search to filter items...		
Resource	Location	
<input type="checkbox"/> RiteshSubscription		
<input type="checkbox"/> rg-harvestingclouds-infra101	East US	
<input type="checkbox"/> vmAccounts101	East US	
<input type="checkbox"/> spark	West Europe	
<input type="checkbox"/> spark	West Europe	
<input checked="" type="checkbox"/> visualstudio2017	West Europe	
<input checked="" type="checkbox"/> visualstudio	West Europe	

Figure 4.20: Selecting the scope of the alert

Configure the condition and rules that should get evaluated. Select the **Power Off Virtual Machine** signal name after selecting **Activity Log** as the **Signal type**:

Configure signal logic

Choose a signal below and configure the logic on the next screen to define the alert condition.

Signal type ⓘ Monitor service ⓘ

Activity Log	All
--------------	-----

Displaying 1 - 18 signals out of total 18 signals

Search by signal name

Signal name	↑↓	Signal type	↑↓	Monitor service	↑↓
All Administrative operations		Activity Log		Administrative	
Create or Update Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Delete Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Start Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Power Off Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Redeploy Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Restart Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Deallocate Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Generalize Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Capture Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Run Command on Virtual Machine (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	
Convert Virtual Machine disks to Managed Disks (Microsoft.Compute/virtualM...		Activity Log		Administrative	
Perform Maintenance Redeploy (Microsoft.Compute/virtualMachines)		Activity Log		Administrative	

Figure 4.21: Selecting the signal type

The resultant window will allow you to configure the **Alert logic/condition**. Select **critical** for **Event Level**, and set **Status** to **Succeeded**:

Alert logic

Event Level ⓘ	Status ⓘ	Event initiated by ⓘ
Critical	Succeeded	* (All services and users)

Condition preview

Whenever the Activity Log has an event with Category='Administrative', Signal name='Power Off Virtual Machine (Microsoft.Compute/virtualMachines)', Level='critical', Status='succeeded'

Figure 4.22: Setting up the alert logic

After determining the alert condition comes the most important configuration, which configures the response to the alert by invoking a runbook. We can use **Action groups** to configure the response to an alert. It provides numerous options to invoke an Azure function, webhook, or Azure Automation runbook, as well as to send emails and SMS.

Create an action group by providing a name, a short name, its hosting subscription, a resource group, and an **Action name**. Corresponding to **Action name** select the **Automation Runbook** option as **Action Type**:

Save Discard Refresh Delete

Short name ⓘ	startvm			
Action group name ⓘ	StartVirtualMachine			
Resource group ⓘ	default-activitylogalerts			
Subscription ⓘ	RiteshSubscription			
Actions				
Action name *	Action Type *	Status	Configure	Actions
RemoveVM	Automation Runbook	-	Edit details	X
Unique name for the acti...	Select an action type			

[Azure Privacy Statement](#)

[Azure Alerts Pricing](#)

Figure 4.23 Configuring the action group

Selecting an automation runbook will open another blade for selecting an appropriate Azure Automation account and runbook. Several runbooks are available out of the box, and one of them has been used here:

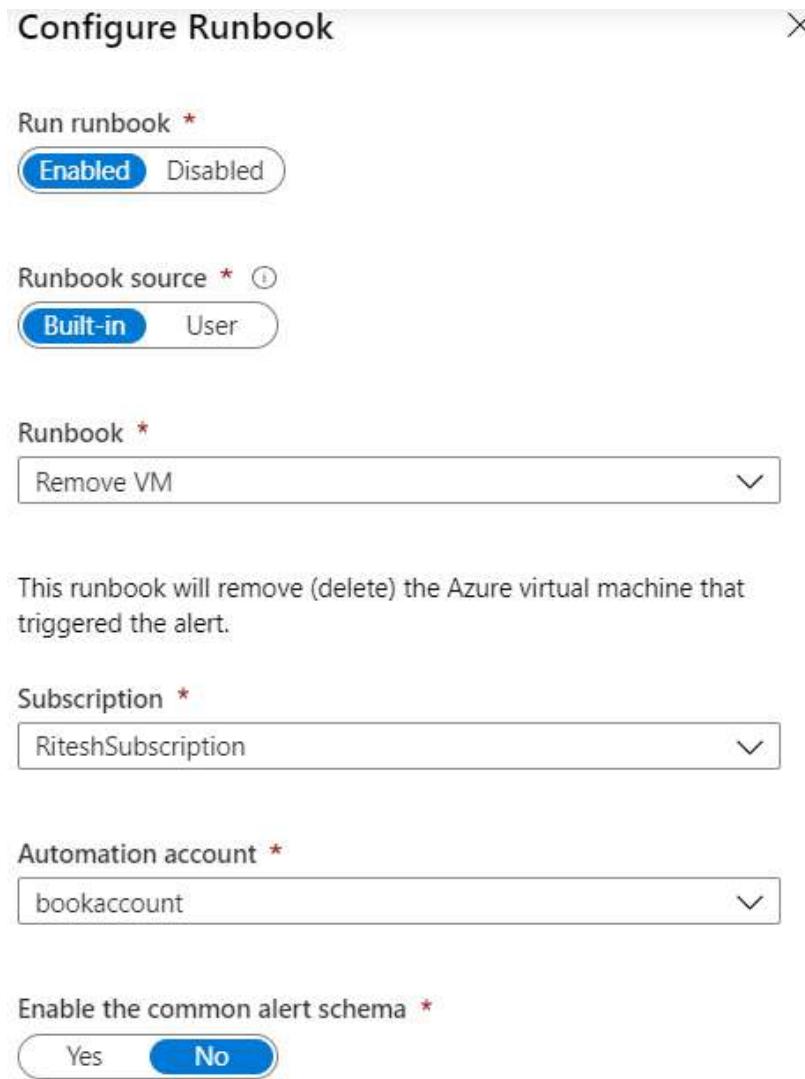


Figure 4.24 Creating the runbook

Finally, provide a name and hosting resource group to create a new alert.

If the VM is deallocated manually, the alert condition gets satisfied and it will raise an alert:

The screenshot shows the 'All Alerts' blade in the Azure portal. At the top, there are filters for Subscription (RiteshSubscription), Resource group (22 selected), Resource type (19 selected), Monitor condition (2 selected), Severity (5 selected), Alert state (3 selected), and Time range (Past 24 hours). Below the filters, a section titled 'Selected subscriptions' shows 'Selected resource group'. Under 'All Alerts', there is a note about action rules (preview). A table lists two alerts for 'startVMAlert' with severity Sev4, monitor condition Fired, and alert state New. Both alerts are associated with the 'visualstudio' resource, 'ActivityLog Administr...', Log signal type, and fired at 5/10/2020, 8:32:12 PM. The subscription is RiteshSubscription.

Name	Severity	Monitor Condition	Alert State	Affected resource	Monitor Service	Signal Type	Fired time	Subscription
startVMAlert	Sev4	Fired	New	visualstudio	ActivityLog Administr...	Log	5/10/2020, 8:32:12 PM	RiteshSubscription
startVMAlert	Sev4	Fired	New	visualstudio	ActivityLog Administr...	Log	5/10/2020, 8:32:05 PM	RiteshSubscription

Figure 4.25 Testing alerts

If you check the details of the VM after a few seconds, you should see that the VM is being deleted:

The screenshot shows the 'Virtual Machines' blade in the Azure portal. At the top, there are buttons for Connect, Start, Restart, Stop, Capture, Delete, and Refresh. A blue bar at the top says 'Advisors (1 of 5): Internet-facing virtual machines should be protected with Network Security Groups →'. Below the bar, the VM details are listed. The VM is named 'visualStudio2017', located in 'West Europe', and is part of the 'RiteshSubscription' and 'visualStudio2017' resource groups. It has a Public IP address of 52.174.123.111 and a Private IP address of 10.0.0.4. The status is 'Deleting'. The operating system is Windows, and the size is Standard DS1 v2 (1 vcpus, 3.5 GiB memory). There are no tags listed.

Figure 4.26 Verifying the results

Hybrid Workers

So far, all the execution of runbooks has primarily been on infrastructure provided by Azure. The runbook workers are Azure compute resources that are provisioned by Azure with appropriate modules and assets deployed in them. Any execution of runbooks happens on this compute. However, it is possible for users to bring their own compute and execute the runbook on this user-provided compute rather than on default Azure compute.

This has multiple advantages. The first and foremost is that the entire execution and its logs are owned by the user with Azure having no visibility of it. Second, the user-provided compute could be on any cloud, as well as on-premises.

Adding a Hybrid Worker involves multiple steps

- First and foremost, an agent needs to be installed on the user-provided compute. Microsoft provides a script that can download and configure the agent automatically. This script is available from <https://www.powershellgallery.com/packages/New-OnPremiseHybridWorker/1.6>.

The script can also be executed from PowerShell ISE as an administrator from within the server that should be part of the Hybrid Worker using the following command:

```
Install-Script -Name New-OnPremiseHybridWorker -verbose
```

- After the script is installed, it can be executed along with parameters related to the Azure Automation account details. A name is also provided for the Hybrid Worker. If the name does not exist already, it will be created; if it exists, the server will be added to the existing Hybrid Worker. It is possible to have multiple servers within a single Hybrid Worker, and it is possible to have multiple Hybrid Workers as well:

```
New-OnPremiseHybridWorker.ps1 -AutomationAccountName bookaccount  
-AAResourceGroupName automationrg '  
-HybridGroupName "localrunbookexecutionengine" '  
-SubscriptionID xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

- Once the execution finishes, navigating back to the portal will show an entry for a Hybrid Worker, as shown in Figure 4.27:

The screenshot shows the Azure portal interface for managing hybrid worker groups. At the top, there are two tabs: "User hybrid worker groups" (which is selected and highlighted with a blue border) and "System hybrid worker groups". Below the tabs, a search bar contains the placeholder "Search to filter items...". The main area displays a table with three columns: "Group name", "Number of workers", and "Last registration time". There is one entry in the table:

Group name	Number of workers	Last registration time
localrunbookexecutionengine	1	3/26/2020, 7:05 AM

Figure 4.27: Checking user Hybrid Worker groups

- If, at this time, an Azure runbook is executed that has a dependency on the **Az** module and a custom certificate uploaded to the certificate asset, it will fail with errors related to the **Az** module and the certificate not being found:

The screenshot shows the Azure portal interface for viewing runbook execution details. At the top, it says "ConnectAzure 3/26/2020, 7:08 AM". Below that, there are buttons for "Resume", "Stop", "Suspend", and "Refresh". The runbook details are listed as follows:

- Id:** a8de935e-9745-43b5-b614-32b8483dac48
- Status:** Completed
- Ran on:** localrunbookexecutionengine
- Ran As:** User
- Created:** 3/26/2020, 7:08:00 AM
- Last Update:** 3/26/2020, 7:08:36 AM
- Runbook:** ConnectAzure
- Source snapshot:** View source snapshot

Below this, there is a section titled "Errors" with a count of 2. The error log table has columns: Time, Type, and Details.

Time	Type	Details
3/26/2020, 7:08:34 AM	Error	No certificate was found in the certificate store with thumbprint 43A06770461183DE1725548BD76DA85B7FB9171A
3/26/2020, 7:08:35 AM	Error	System.Management.Automation.CommandNotFoundException: The term 'Get-azvm' is not recognized as the name of a cmdlet, function, script file, or operable program.

Figure 4.28: Checking errors

- Install the **Az** module using the following command on the server:

```
Install-Module -name Az -AllowClobber -verbose
```

It is also important to have the **.pfx** certificate available on this server. The previously exported certificate should be copied to the server and installed manually.

- After installation of the `Az` module and certificate, re-executing the runbook on the Hybrid Worker is shown in *Figure 4.29*, and it should show the list of VMs in the subscription:

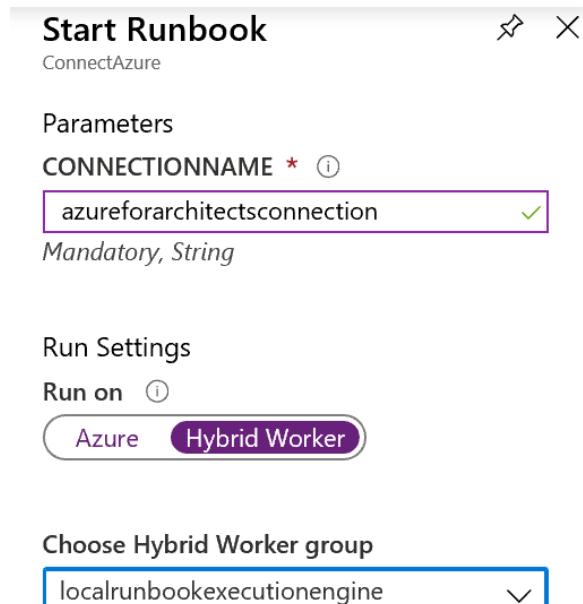


Figure 4.29: Setting up a runbook to run on a Hybrid Worker

When we discussed different scenarios, we talked about configuration management. In the next section, we will be discussing configuration management with Azure Automation in more detail.

Azure Automation State Configuration

Azure Automation provides a **Desired State Configuration (DSC)** pull server along with every Azure Automation account. The pull server can hold configuration scripts that can be pulled by servers across clouds and on-premises. This means that Azure Automation can be used to configure any server hosted anywhere in the world.

The DSC needs a local agent on these servers, also known as a **local configuration manager (LCM)**. It should be configured with the Azure Automation DSC pull server so it can download the required configuration and autoconfigure the server.

The autoconfiguration can be scheduled to be periodic (by default it is half an hour), and if the agent finds any deviation in the server configuration compared to the one available in the DSC script, it will autocorrect and bring back the server to the desired and expected state.

In this section, we will configure one server hosted on Azure, and the process will remain the same irrespective of whether the server is on a cloud or on-premises.

The first step is to create a DSC configuration. A sample configuration is shown here, and complex configurations can be authored similarly:

```
configuration ensureiis {
    import-dscresource -modulename psdesiredstateconfiguration

    node localhost {
        WindowsFeature iis {
            Name = "web-server"
            Ensure = "Present"

        }
    }
}
```

The configuration is quite simple. It imports the **PSDesiredStateConfiguration** base DSC module and declares a single-node configuration. This configuration is not associated with any specific node and can be used to configure any server. The configuration is supposed to configure an IIS web server and ensure that it is present on any server to which it is applied.

This configuration is not yet available on the Azure Automation DSC pull server, and so the first step is to import the configuration into the pull server. This can be done using the Automation account **Import-AzAutomationDscConfiguration** cmdlet, as shown next:

```
Import-AzAutomationDscConfiguration -SourcePath "C:\Ritesh\ensureiis.ps1"
    -AutomationAccountName bookaccount -ResourceGroupName automationrg -Force
    -Published
```

There are a few important things to note here. The name of the configuration should match the filename, and it must only contain alphanumeric characters and underscores. A good naming convention is to use verb/noun combinations. The cmdlets need the path of the configuration file and the Azure Automation account details to import the configuration script.

At this stage, the configuration is visible on the portal:

The screenshot shows the Azure portal's State configuration (DSC) blade for the 'bookaccount' automation account. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Configuration Management (Inventory, Change tracking, State configuration (DSC)), and Update management. The main area has tabs for Nodes, Configurations (which is selected and highlighted in yellow), Compiled configurations, and Gallery. Under Configuration, there is a search bar labeled 'Search configurations...'. A table lists the configuration 'ensureiis' with columns for Configuration, Compiled Configuration Count (0), and Last Modified (3/26/2020, 8:34 AM).

Figure 4.30: Adding configuration

Once the configuration script is imported, it is compiled and stored within the DSC pull server using the **Start-AzAutomationDscCompilationJob** cmdlet, as shown next:

```
Start-AzAutomationDscCompilationJob -ConfigurationName 'ensureiis'
-ResourceGroupName 'automationrg' -AutomationAccountName 'bookaccount'
```

The name of the configuration should match the one that was recently uploaded, and the compiled configuration should be available now on the **Compiled configurations** tab, as shown in Figure 4.31:

The screenshot shows the Azure portal's State configuration (DSC) blade for the 'bookaccount' automation account. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Configuration Management (Inventory, Change tracking, State configuration (DSC)), and Update management. The main area has tabs for Nodes, Configurations (which is selected and highlighted in yellow), Compiled configurations (which is selected and highlighted in blue), and Gallery. Under Node configuration, there is a search bar labeled 'Search node configurations...'. A table lists the compiled configuration 'ensureiis' under the node 'ensureiis.localhost' with columns for Node Configuration, Configuration, Node Count (0), Created (3/27/2020, 4:09 AM), and Last Modified (3/27/2020, 4:09 AM).

Figure 4.31: Listing compiled configurations

It is important to note that the **Node Count** in Figure 4.31 is **0**. It means that a node configuration called `ensureiss.localhost` exists but it is not assigned to any node. The next step is to assign the configuration to the node.

By now, we have a compiled DSC configuration available on the DSC pull server, but there are no nodes to manage. The next step is to onboard the VMs and associate them with the DSC pull server. This is done using the **Register-AzAutomationDscNode** cmdlet:

```
Register-AzAutomationDscNode -ResourceGroupName 'automationrg'
    -AutomationAccountName 'bookaccount' -AzureVMLocation "west
    Europe" -AzureVMResourceGroup 'spark' -AzureVMName 'spark'
    -ConfigurationModeFrequencyMins 30 -ConfigurationMode 'ApplyAndAutoCorrect'
```

This cmdlet takes the name of the resource group for both the VM and the Azure Automation account. It also configures the configuration mode and the **configurationModeFrequencyMins** property of the local configuration manager of the VM. This configuration will check and autocorrect any deviation from the configuration applied to it every 30 minutes.

If **VMresourcegroup** is not specified, the cmdlet tries to find the VM in the same resource group as the Azure Automation account, and if the VM location value is not provided, it tries to find the VM in the Azure Automation region. It is always better to provide values for them. Notice that this command can only be used for Azure VMs as it asks for **AzureVMname** explicitly. For servers on other clouds and on-premises, use the **Get-AzAutomationDscOnboardingMetaconfig** cmdlet.

Now, a new node configuration entry can also be found in the portal, as shown in Figure 4.32:

Node	Status	Node configuration	Last seen	Version
spark	Compliant		3/27/2020, 4:38 AM	2.80.0

Figure 4.32: Verifying node status

The node information can be obtained as follows:

```
$node = Get-AzAutomationDscNode -ResourceGroupName 'automationrg'
-AutomationAccountName 'bookaccount' -Name 'spark'
```

And a configuration can be assigned to the node:

```
Set-AzAutomationDscNode -ResourceGroupName 'automationrg'
-AutomationAccountName 'bookaccount' -NodeConfigurationName 'ensureiis.
localhost' -NodeId $node.Id
```

Once the compilation is complete, it can be assigned to the nodes. The initial status is **Pending**, as shown in Figure 4.33:

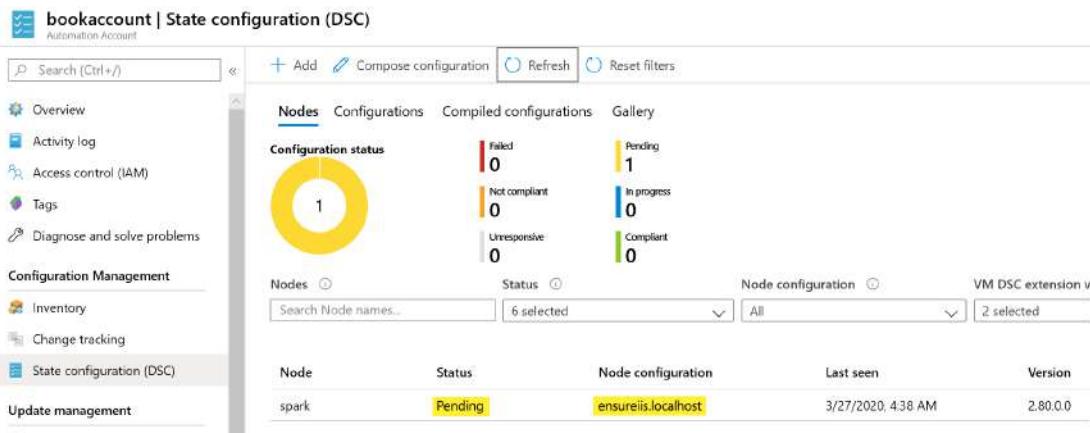


Figure 4.33: Verifying node status

After a few minutes, the configuration is applied to the node, the node becomes **Compliant**, and the status becomes **Completed**:

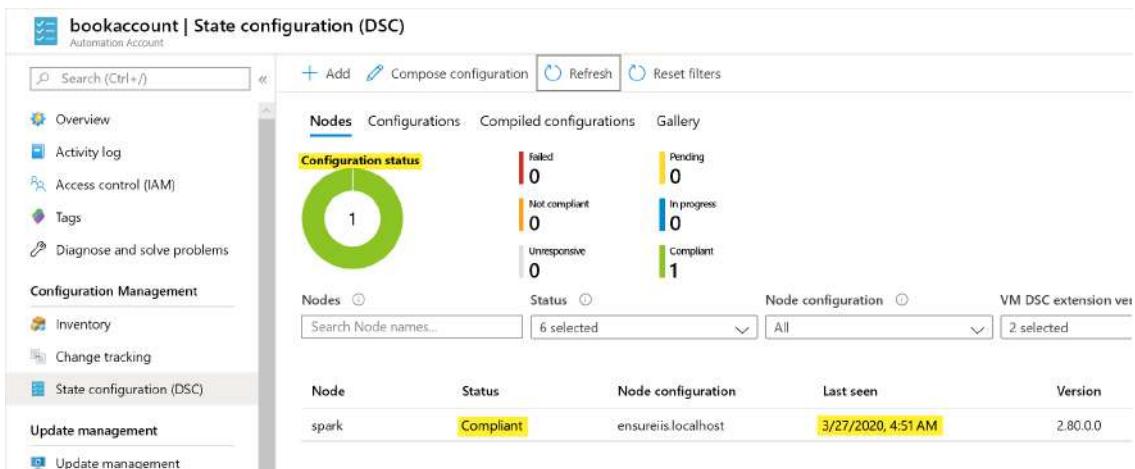


Figure 4.34: Verifying if the node is compliant

Later, logging into the server and checking if the web server (IIS) is installed confirms that it is installed, as you can see in *Figure 4.35*:

PS C:\Users\citynextadmin> Get-WindowsFeature -Name web-server		
Display Name	Name	Install State
[X] Web Server (IIS)	Web-Server	Installed

Figure 4.35: Checking whether the desired state has been achieved

In the next section, Azure Automation pricing will be discussed.

Azure Automation pricing

There is no cost for Azure Automation if no runbooks are executed on it. The cost of Azure Automation is charged per minute for execution of runbook jobs. This means that if the total number of runbook execution minutes is 10,000, the cost of Azure Automation would be \$0.002 per minute multiplied by 9,500, as the first 500 minutes are free.

There are other costs involved in Azure Automation depending on features consumed. For example, a DSC pull server does not cost anything within Azure Automation; neither does onboarding Azure VMs on to the pull server. However, if non-Azure servers are onboarded, typically from other clouds or on-premises, then the first five servers are free and anything on top of that costs \$6 per server per month in the West US region.

Pricing may vary from region to region, and it's always a good practice to verify the pricing on the official pricing page: <https://azure.microsoft.com/pricing/details/automation>.

You might ask, why do we need an Automation account when we can deploy serverless applications via Azure Functions? In the next section, we will explore the key differences between Azure Automation and serverless automation.

Comparison with serverless automation

Azure Automation and Azure serverless technologies, especially Azure Functions, are quite similar and overlap in terms of functionality. However, these are separate services with different capabilities and pricing.

It is important to understand that Azure Automation is a complete suite for process automation and configuration management, while Azure Functions is meant for implementing business functionality.

Azure Automation is used for automating the processes of provisioning, deprovisioning, management, and operations of infrastructure and configuration management thereafter. On the other hand, Azure Functions is meant for the creation of services, implementing functionality that can be part of microservices and other APIs.

Azure Automation is not meant for unlimited scale, and the load is expected to be moderate, while Azure Functions can handle unlimited traffic and scale automatically.

There are a host of shared assets, such as connections, variables, and modules, that can be reused across runbooks in Azure Automation; however, there is no out-of-the-box shared concept in Azure Functions.

Azure Automation can manage intermediate state by way of checkpointing and continue from the last saved state, while Azure functions are generally stateless and do not maintain any state.

Summary

Azure Automation is an important service within Azure and the only service for process automation and configuration management. This chapter covered a lot of important concepts related to Azure Automation and process automation, including shared assets such as connection, certificates, and modules.

It covered the creation of runbooks, including invoking runbooks in different ways, such as parent-child relationships, webhooks, and using the portal. The chapter also discussed the architecture and life cycle of runbooks.

We also looked at the usage of Hybrid Workers and, toward the end of the chapter, explored configuration management using a DSC pull server and a local configuration manager. Finally, we made comparisons with other technologies, such as Azure Functions.

In the next chapter, we will explore designing policies, locks, and tags for Azure deployments.

5

Designing policies, locks, and tags for Azure deployments

Azure is a versatile cloud platform. Customers can not only create and deploy their applications; they can also actively manage and govern their environments. Clouds generally follow a pay-as-you-go paradigm, where a customer subscribes and can then deploy virtually anything to the cloud. It could be as small as a basic virtual machine, or it could be thousands of virtual machines with higher **stock-keeping units (SKUs)**. Azure will not stop any customer from provisioning the resources they want to provision. Within an organization, there could be a large number of people with access to the organization's Azure subscription. There needs to be a governance model in place so that only necessary resources are provisioned by people who have the right to create them. Azure provides resource management features, such as **Azure Role-Based Access Control (RBAC)**, Azure Policy, management groups, blueprints, and resource locks, for managing and providing governance for resources.

Other major aspects of governance include cost, usage, and information management. An organization's management team always wants to be kept up to date about cloud consumption and costs. They would like to identify what team, department, or unit is using what percentage of their total cost. In short, they want to have reports based on various dimensions of consumption and cost. Azure provides a tagging feature that can help provide this kind of information on the fly.

In this chapter, we will cover the following topics:

- Azure management groups
- Azure tags
- Azure Policy
- Azure locks
- Azure RBAC
- Azure Blueprints
- Implementing Azure governance features

Azure management groups

We are starting with Azure management groups because, in most of the upcoming sections, we will be referencing or mentioning management groups. Management groups act as a level of scope for you to effectively assign or manage roles and policies. Management groups are very useful if you have multiple subscriptions.

Management groups act as a placeholder for organizing subscriptions. You can also have nested management groups. If you apply a policy or access at the management group level, it will be inherited by the underlying management groups and subscriptions. From the subscription level, that policy or access will be inherited by resource groups and then finally by the resources.

The hierarchy of management groups is shown here:

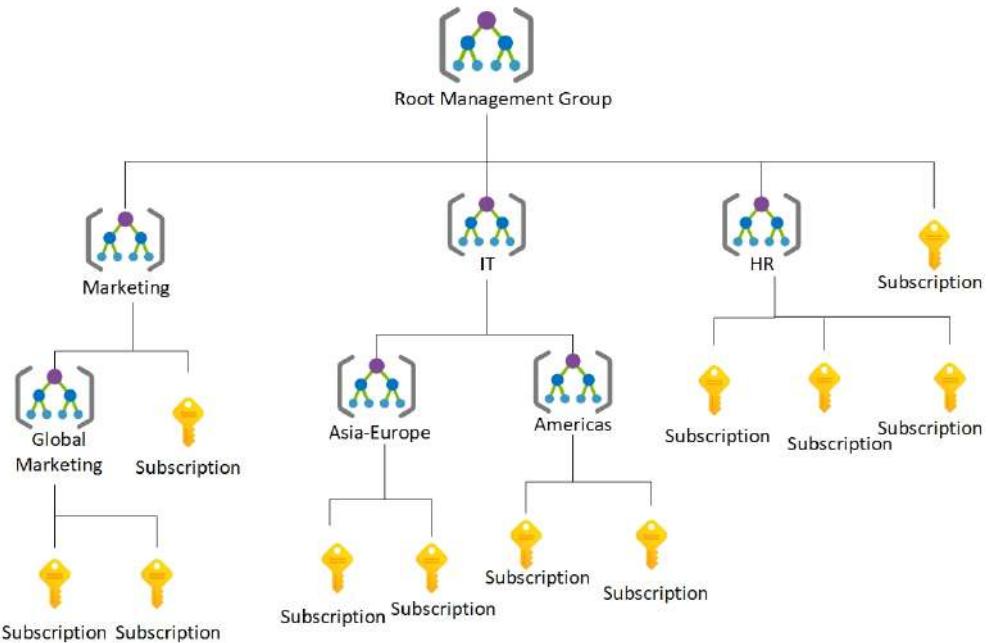


Figure 5.1: Hierarchy of Azure management groups

In Figure 5.1, we are using management groups to separate the operations of different departments, such as marketing, IT, and HR. Inside each of these departments, there are nested management groups and subscriptions, which helps to organize resources into a hierarchy for policy and access management. Later, you will see how management groups are used as a scope for governance, policy management, and access management.

In the next section, we will be discussing Azure tags, which play another vital role in the logical grouping of resources.

Azure tags

Azure allows the tagging of resource groups and resources with name-value pairs. Tagging helps in the logical organization and categorization of resources. Azure also allows the tagging of 50 name-value pairs for a resource group and its resources. Although a resource group acts as a container or a placeholder for resources, tagging a resource group does not mean the tagging of its constituent resources. Resource groups and resources should be tagged based on their usage, which will be explained later in this section. Tags are bound to a subscription, resource group, or resource. Azure accepts any name-value pair, and so it is important for an organization to define both the names and their possible values.

But why is tagging important? In other words, what problems can be solved using tagging? Tagging has the following benefits:

- **Categorization of resources:** An Azure subscription can be used by multiple departments within an organization. It is important for the management team to identify the owners of any resources. Tagging helps in assigning identifiers to resources that can be used to represent departments or roles.
- **Information management for Azure resources:** Again, Azure resources can be provisioned by anyone with access to the subscription. Organizations like to have a proper categorization of resources in place to comply with information management policies. Such policies can be based on application life cycle management, such as the management of development, testing, and production environments. They could also be based on usage or any other priorities. Each organization has its own way of defining information categories, and Azure caters for this with tags.
- **Cost management:** Tagging in Azure can help in identifying resources based on their categorization. Queries can be executed against Azure to identify cost per category, for instance. For example, the cost of resources in Azure for the development of an environment for the finance department and the marketing department can be easily ascertained. Moreover, Azure also provides billing information based on tags. This helps in identifying the consumption rates of teams, departments, or groups.

Tags in Azure do have certain limitations, however:

- Azure allows a maximum of 50 tag name-value pairs to be associated with resource groups.
- Tags are non-inheritable. Tags applied to a resource group do not apply to the individual resources within it. However, it is quite easy to forget to tag resources when provisioning them. Azure Policy provides the mechanism to use to ensure that tags are tagged with the appropriate value during provision time. We will consider the details of such policies later in this chapter.

Tags can be assigned to resources and resource groups using PowerShell, the Azure CLI 2.0, Azure Resource Manager templates, the Azure portal, and the Azure Resource Manager REST APIs.

An example of information management categorization using Azure tags is shown here:

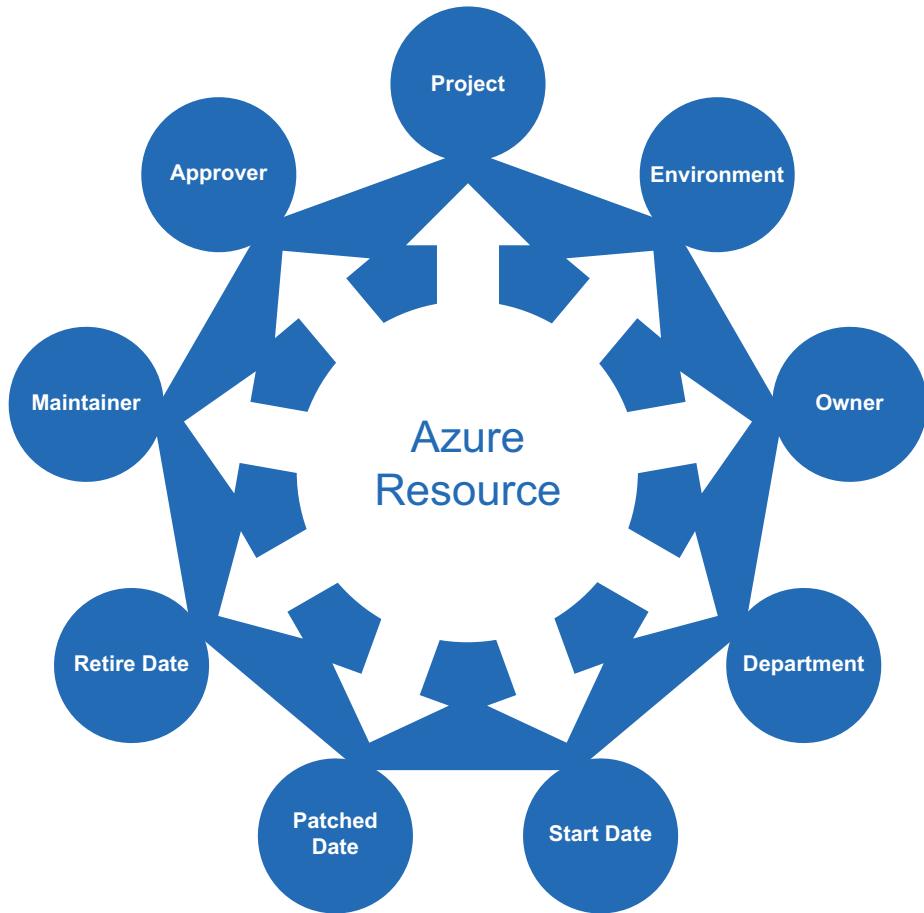


Figure 5.2: Information management categorization using Azure tags

In this example, the **Department**, **Project**, **Environment**, **Owner**, **Approver**, **Maintainer**, **Start Date**, **Retire Date**, and **Patched Date** name-value pairs are used to tag resources. It is extremely easy to find all the resources for a particular tag or a combination of tags using PowerShell, the Azure CLI, or REST APIs. The next section will discuss ways to use PowerShell to assign tags to resources.

Tags with PowerShell

Tags can be managed using PowerShell, Azure Resource Manager templates, the Azure portal, and REST APIs. In this section, PowerShell will be used to create and apply tags. PowerShell provides a cmdlet for retrieving and attaching tags to resource groups and resources:

- To retrieve tags associated with a resource using PowerShell, the **Get-AzResource** cmdlet can be used:

```
(Get-AzResource -Tag @{"Environment"="Production"}).Name
```

- To retrieve tags associated with a resource group using PowerShell, the following command can be used:

```
Get-AzResourceGroup -Tag {"Environment"="Production"}
```

- To set tags to a resource group, the **Update-AzTag** cmdlet can be used:

```
$tags = @{"Dept"="IT"; "Environment"="Production"}  
$resourceGroup = Get-AzResourceGroup -Name demoGroup  
New-AzTag -ResourceId $resourceGroup.ResourceId -Tag $tags
```

- To set tags to a resource, the same **Update-AzTag** cmdlet can be used:

```
$tags = @{"Dept"="Finance"; "Status"="Normal"}  
$resource = Get-AzResource -Name demoStorage -ResourceGroup demoGroup  
New-AzTag -ResourceId $resource.id -Tag $tags
```

- You can update existing tags using the **Update-AzTag** command; however, you need to specify the operation as **Merge** or **Replace**. **Merge** will append the new tags you are passing into the existing tags; however, the **Replace** operation will replace all the old tags with the new ones. Here is one example of updating the tags in a resource group without replacing the existing ones:

```
$tags = @{"Dept"="IT"; "Environment"="Production"}  
$resourceGroup = Get-AzResourceGroup -Name demoGroup  
Update-AzTag -ResourceId $resourceGroup.ResourceId -Tag $tags -Operation  
Merge
```

Let's now look at tags with Azure Resource Manager templates.

Tags with Azure Resource Manager templates

Azure Resource Manager templates also help in defining tags for each resource. They can be used to assign multiple tags to each resource, as follows:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/  
deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "resources": [  
        {  
            "apiVersion": "2019-06-01",  
            "type": "Microsoft.Storage/storageAccounts",  
            "name": "[concat('storage', uniqueString(resourceGroup().id))]",  
            "location": "[resourceGroup().location]",  
            "tags": {  
                "Dept": "Finance",  
                "Environment": "Production"  
            },  
            "sku": {  
                "name": "Standard_LRS"  
            },  
            "kind": "Storage",  
            "properties": { }  
        }  
    ]  
}
```

In the previous example, a couple of tags, **Dept** and **Environment**, were added to a storage account resource using Azure Resource Manager templates.

Tagging resource groups versus resources

It is a must for architects to decide the taxonomy and information architecture for Azure resources and resource groups. They should identify the categories by which resources will be classified based on the query requirements. However, they must also identify whether tags should be attached to individual resources or to resource groups.

If all resources within a resource group need the same tag, then it is better to tag the resource group, even though tags don't inherit the resources in the resource group. If your organization requires tags to be passed to all the underlying resources, then you can consider writing a PowerShell script to get the tags from the resource group and update the tags for the resources in the resource group. It is important to take queries on tags into consideration before deciding whether tags should be applied at the resource level or the resource group level. If the queries relate to individual resource types across a subscription and across resource groups, then assigning tags to individual resources makes more sense. However, if identifying resource groups is enough for your queries to be effective, then tags should be applied only to resource groups. If you are moving resources across resource groups, the tags applied at the resource group level will be lost. If you are moving resources, consider adding the tags again.

Azure Policy

In the previous section, we talked about applying tags for Azure deployments. Tags are great for organizing resources; however, there is one more thing that was not discussed: how do organizations ensure that tags are applied for every deployment? There should be automated enforcement of Azure tags to resources and resource groups. There is no check from Azure to ensure that appropriate tags are applied to resources and resource groups. This is not just specific to tags—this applies to the configuration of any resource on Azure. For example, you may wish to restrict where your resources can be provisioned geographically (to only the US-East region, for instance).

You might have guessed by now that this section is all about formulating a governance model on Azure. Governance is an important element in Azure because it ensures that everyone accessing the Azure environment is aware of organizational priorities and processes. It also helps to bring costs under control. It helps in defining organizational conventions for managing resources.

Each policy can be built using multiple rules, and multiple policies can be applied to a subscription or resource group. Based on whether the rules are satisfied, policies can execute various actions. An action could be to deny an ongoing transaction, to audit a transaction (which means writing to logs and allowing it to finish), or to append metadata to a transaction if it's found to be missing.

Policies could be related to the naming convention of resources, the tagging of resources, the types of resources that can be provisioned, the location of resources, or any combination of those.

Azure provides numerous built-in policies and it is possible to create custom policies. There is a policy language based on JSON that can be used to define custom policies.

Now that you know the purpose and use case of Azure Policy, let's go ahead and discuss built-in policies, policy language, and custom policies.

Built-in policies

Azure provides a service for the creation of custom policies; however, it also provides some out-of-the-box policies that can be used for governance. These policies relate to allowed locations, allowed resource types, and tags. More information for these built-in policies can be found at <https://docs.microsoft.com/azure/azure-resource-manager/resource-manager-policy>.

Policy language

Policies in Azure use JSON to define and describe policies. There are two steps in policy adoption. The policy should be defined and then it should be applied and assigned. Policies have scope and can be applied at the management group, subscription, or resource group level.

Policies are defined using **if...then** blocks, similar to any popular programming language. The **if** block is executed to evaluate the conditions, and based on the result of those conditions, the **then** block is executed:

```
{  
  "if": {  
    <condition> | <logical operator>  
  },  
  "then": {  
    "effect": "deny | audit | append"  
  }  
}
```

The policies not only allow simple **if** conditions but also allow multiple **if** conditions to be joined together logically to create complex rules. These conditions can be joined using **AND**, **OR**, and **NOT** operators:

- The **AND** syntax requires all conditions to be true.
- The **OR** syntax requires one of the conditions to be true.
- The **NOT** syntax inverts the result of the condition.

The **AND** syntax is shown next. It is represented by the **allOf** keyword:

```
"if": {  
  "allOf": [  
    {  
      "field": "tags",  
      "containsKey": "application"  
    },  
    {  
      "field": "type",  
      "equals": "Microsoft.Storage/storageAccounts"  
    }  
  ]  
},
```

The **OR** syntax is shown next. It is represented by the **anyOf** keyword:

```
"if": {  
  "anyOf": [  
    {  
      "field": "tags",  
      "containsKey": "application"  
    },  
    {  
      "field": "type",  
      "equals": "Microsoft.Storage/storageAccounts"  
    }  
  ]  
},
```

The **NOT** syntax is shown next. It is represented by the **not** keyword:

```
"if": {  
  "not": [  
    {  
      "field": "tags",  
    }  
  ]  
},
```

```

    "containsKey": "application"
},
{
  "field": "type",
  "equals": "Microsoft.Storage/storageAccounts"
}
]
},

```

In fact, these logical operators can be combined together, as follows:

```

"if": {
  "allOf": [
    {
      "not": {
        "field": "tags",
        "containsKey": "application"
      }
    },
    {
      "field": "type",
      "equals": "Microsoft.Storage/storageAccounts"
    }
  ]
},

```

This is very similar to the use of **if** conditions in popular programming languages such as C# and Node.js:

```

If ("type" == "Microsoft.Storage/storageAccounts") {
  Deny
}

```

It is important to note that there is no **allow** action, although there is a **Deny** action. This means that policy rules should be written with the possibility of denial in mind. Rules should evaluate conditions and **Deny** the action if the conditions are met.

Allowed fields

The fields that are allowed in conditions while defining policies are as follows:

- **Name:** The name of the resource for applying the policy to. This is very specific and applicable to a resource by its usage.
- **Type:** The type of resource, such as **Microsoft.Compute/VirtualMachines**. That would apply the policy to all instances of virtual machines, for example.
- **Location:** The location (that is, the Azure region) of a resource.
- **Tags:** The tags associated with a resource.
- **Property aliases:** Resource-specific properties. These properties are different for different resources.

In the next section, you will learn more about safekeeping resources in production environments.

Azure locks

Locks are mechanisms for stopping certain activities on resources. RBAC provides rights to users, groups, and applications within a certain scope. There are out-of-the-box RBAC roles, such as owner, contributor, and reader. With the contributor role, it is possible to delete or modify a resource. How can such activities be prevented despite the user having a contributor role? Enter Azure locks.

Azure locks can help in two ways:

- They can lock resources such that they cannot be deleted, even if you have owner access.
- They can lock resources in such a way that they can neither be deleted nor have their configuration modified.

Locks are typically very helpful for resources in production environments that should not be modified or deleted accidentally.

Locks can be applied at the levels of subscription, resource group, management group, and individual resource. Locks can be inherited between subscriptions, resource groups, and resources. Applying a lock at the parent level will ensure that those resources at the child level will also inherit it. Resources that are added later in the sub-scope also inherit the lock configuration by default. Applying a lock at the resource level will also prevent the deletion of the resource group containing the resource.

Locks are applied only to operations that help in managing a resource, rather than operations that are within a resource. Users need either **Microsoft.Authorization/*** or **Microsoft.Authorization/locks/*** RBAC permissions to create and modify locks.

Locks can be created and applied through the Azure portal, Azure PowerShell, the Azure CLI, Azure Resource Manager templates, and REST APIs.

Creating a lock using an Azure Resource Manager template is done as follows:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/  
deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "lockedResource": {  
            "type": "string"  
        }  
    },  
    "resources": [  
        {  
            "name": "[concat(parameters('lockedResource'), '/Microsoft.  
Authorization/myLock')]",  
            "type": "Microsoft.Storage/storageAccounts/providers/locks",  
            "apiVersion": "2019-06-01",  
            "properties": {  
                "level": "CannotDelete"  
            }  
        }  
    ]  
}
```

The **resources** section of the Azure Resource Manager template code consists of a list of all the resources to be provisioned or updated within Azure. There is a storage account resource, and the storage account has a lock resource. A name for the lock is provided using dynamic string concatenation, and the lock that's applied is of the **CannotDelete** type, which means that the storage account is locked for deletion. The storage account can only be deleted after the removal of the lock.

Creating and applying a lock to a resource using PowerShell is done as follows:

```
New-AzResourceLock -LockLevel CanNotDelete -LockName LockSite '  
-ResourceName examplesite -ResourceType Microsoft.Web/sites '  
-ResourceGroupName exempleresourcegroup
```

Creating and applying a lock to a resource group using PowerShell is done as follows:

```
New-AzResourceLock -LockName LockGroup -LockLevel CanNotDelete '  
-ResourceGroupName exempleresourcegroup
```

Creating and applying a lock to a resource using the Azure CLI is done as follows:

```
az lock create --name LockSite --lock-type CanNotDelete \  
--resource-group exempleresourcegroup --resource-name examplesite \  
--resource-type Microsoft.Web/sites
```

Creating and applying a lock to a resource group using the Azure CLI is done as follows:

```
az lock create --name LockGroup --lock-type CanNotDelete \  
--resource-group exempleresourcegroup
```

To create or delete resource locks, the user should have access to the **Microsoft.Authorization/*** or **Microsoft.Authorization/locks/*** actions. You can further give granular permissions as well. Owners and user access administrators will have access to creating or deleting locks by default.

If you are wondering what the **Microsoft.Authorization/*** and **Microsoft.Authorization/locks/*** keywords are, you will get to know more about them in the next section.

Let's now look at Azure RBAC.

Azure RBAC

Azure provides authentication using Azure Active Directory for its resources. Once an identity has been authenticated, the resources the identity will be allowed to access should be decided. This is known as authorization. Authorization evaluates the permissions that have been afforded to an identity. Anybody with access to an Azure subscription should be given just enough permissions so that their specific job can be performed, and nothing more.

Authorization is popularly also known as RBAC. RBAC in Azure refers to the assigning of permissions to identities within a scope. The scope could be a management group, a subscription, a resource group, or individual resources.

RBAC helps in the creation and assignment of different permissions to different identities. This helps in segregating duties within teams, rather than everyone having all permissions. RBAC helps in making people responsible for their job only, because others might not even have the necessary access to perform it. It should be noted that providing permissions at a greater scope automatically ensures that child resources inherit those permissions. For example, providing an identity with read access to a resource group means that the identity will have read access to all the resources within that group, too.

Azure provides three general-purpose, built-in roles. They are as follows:

- The owner role, which has full access to all resources
- The contributor role, which has access to read/write resources
- The reader role, which has read-only permissions to resources

There are more roles provided by Azure, but they are resource-specific, such as the network contributor and security manager roles.

To get all roles provided by Azure for all resources, execute the **Get-AzRoleDefinition** command in the PowerShell console.

Each role definition has certain allowed and disallowed actions. For example, the owner role has all actions permitted; no action is prohibited:

```
PS C:\Users\riskaria> Get-AzRoleDefinition -Name "Owner"

Name          : Owner
Id            : 8e3af657-a8ff-443c-a75c-2fe8c4bcb635
IsCustom      : False
Description   : Lets you manage everything, including access to resources.
Actions       : {*}
NotActions    : {}
DataActions   : {}
NotDataActions: {}
AssignableScopes : {/}
```

Each role comprises multiple permissions. Each resource provides a list of operations. The operations supported by a resource can be obtained using the **Get-AzProviderOperation** cmdlet. This cmdlet takes the name of the provider and resource to retrieve the operations:

```
PS C:\Users\riskaria> Get-AzProviderOperation -OperationSearchString "Microsoft.Insights/*" | select Operation
```

This will result in the following output:

```
PS C:\Users\riskaria> Get-AzProviderOperation -OperationSearchString "Microsoft.Insights/*" | select Operation
```

Operation

The output shown here provides all the actions available within the **Microsoft.Insights** resource provider across its associated resources. The resources include **Metrics**, **Register**, and others, while the actions include **Read**, **Write**, and others.

Let's now look at custom roles.

Custom roles

Azure provides numerous out-of-the-box generic roles, such as owner, contributor, and reader, as well as specialized resource-specific roles, such as virtual machine contributor. Having a reader role assigned to a user/group or service principal will mean reader permissions being assigned to the scope. The scope could be a resource, resource group, or a subscription. Similarly, a contributor would be able to read as well as modify the assigned scope. A virtual machine contributor would be able to modify virtual machine settings and not any other resource settings. There are, however, times when existing roles might not suit our requirements. In such cases, Azure allows the creation of custom roles. They can be assigned to users, groups, and service principals and are applicable to resources, resource groups, and subscriptions.

Custom roles are created by combining multiple permissions. For example, a custom role can consist of operations from multiple resources. In the next code block, a new role definition is being created, but instead of setting all properties manually, one of the existing "**Virtual Machine Contributor**" roles is retrieved because it almost matches with the configuration of the new custom role. Avoid using the same name as built-in roles, as that would create conflict. Then, the ID property is nullified and a new name and description is provided. The code also clears all the actions, adds some actions, adds a new scope after clearing the existing scope, and finally creates a new custom role:

```
$role = Get-AzRoleDefinition "Virtual Machine Contributor"  
$role.Id = $null  
$role.Name = "Virtual Machine Operator"  
$role.Description = "Can monitor and restart virtual machines."  
$role.Actions.Clear()  
$role.Actions.Add("Microsoft.Storage/*/read")  
$role.Actions.Add("Microsoft.Network/*/read")  
$role.Actions.Add("Microsoft.Compute/*/read")  
$role.Actions.Add("Microsoft.Compute/virtualMachines/start/action")  
$role.Actions.Add("Microsoft.Compute/virtualMachines/restart/action")  
$role.Actions.Add("Microsoft.Authorization/*/read")  
$role.Actions.Add("Microsoft.Resources/subscriptions/resourceGroups/read")  
$role.Actions.Add("Microsoft.Insights/alertRules/*")  
$role.Actions.Add("Microsoft.Support/*")  
$role.AssivableScopes.Clear()  
$role.AssivableScopes.Add("/subscriptions/548f7d26-b5b1-468e-ad45-
```

```
6ee12accf7e7")
```

```
New-AzRoleDefinition -Role $role
```

There is a preview feature available in the Azure portal that you can use to create custom RBAC roles from the Azure portal itself. You have the option to create roles from scratch, clone an existing role, or start writing the JSON manifest. Figure 5.3 shows the **Create a custom role** blade, which is available at **IAM > +Add** section:

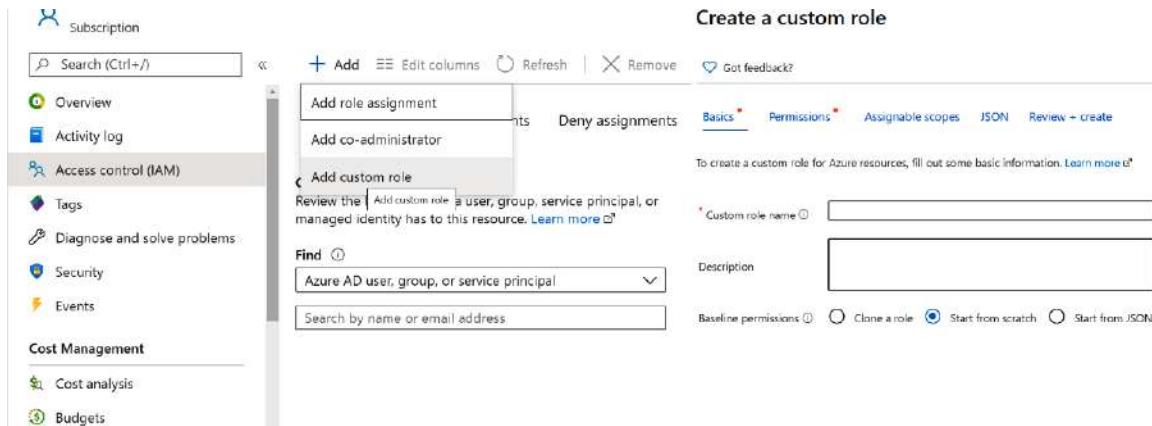


Figure 5.3: Creating custom roles from the Azure portal

This makes the process of custom role creation hassle-free.

How are locks different from RBAC?

Locks are not the same as RBAC. RBAC helps in allowing or denying permissions for resources. These permissions relate to performing operations, such as read, write, and update operations on resources. Locks, on the other hand, relate to disallowing permissions to configure or delete resources.

In the next section, we will be discussing Azure Blueprints, which helps us with the orchestration of artifacts, such as role assignments, policy assignments, and more, that we have discussed so far.

Azure Blueprints

You will be familiar with the word blueprint, which refers to the plan or drawing that is used by an architect to architect a solution. Similarly, in Azure, cloud architects can leverage Azure Blueprints to define a set of repeatable Azure resources that adheres to an organization's standards, processes, and patterns.

Blueprints allows us to orchestrate the deployment of various resources and other artifacts, such as:

- Role assignments
- Policy assignments
- Azure Resource Manager templates
- Resource groups

Azure blueprint objects are replicated to multiple regions and are backed by Azure Cosmos DB. The replication helps in providing consistent access to resources and maintaining the organization's standards irrespective of which region you are deploying to.

Azure Blueprints comprises various artifacts, and you can find the list of supported artifacts here: <https://docs.microsoft.com/azure/governance/blueprints/overview#blueprint-definition>.

Blueprints can be created from the Azure portal, Azure PowerShell, the Azure CLI, REST APIs, or ARM templates.

In the next section, we will look at an example of implementing Azure governance features. Services and features such as RBAC, Azure Policy, and Azure resource locks will be used in the example.

An example of implementing Azure governance features

In this section, we will go through a sample architecture implementation for a fictitious organization that wants to implement Azure governance and cost management features.

Background

Company Inc is a worldwide company that is implementing a social media solution on an Azure IaaS platform. They use web servers and application servers deployed on Azure virtual machines and networks. Azure SQL Server acts as the backend database.

RBAC for Company Inc

The first task is to ensure that the appropriate teams and application owners can access their resources. It is recognized that each team has different requirements. For the sake of clarity, Azure SQL is deployed in a separate resource group to the Azure IaaS artifacts.

The administrator assigns the following roles for the subscription:

Role	Assigned to	Description
Owner	Administrator	Manages all resource groups and the subscription
Security Manager	Security administrators	This role allows users to look at Azure Security Center and the status of the resources.
Contributor	Infrastructure management	Managing virtual machines and other resources.
Reader	Developers	Can view resources but cannot modify them. Developers are expected to work in their development/testing environments.

Table 5.1: Different roles with access details

Azure Policy

The company should implement Azure Policy to ensure that its users always provision resources according to the company guidelines.

The policies in Azure govern various aspects related to the deployment of resources. The policies will also govern updates after the initial deployment. Some of the policies that should be implemented are given in the following section.

Deployments to certain locations

Azure resources and deployments can only be executed for certain chosen locations. It would not be possible to deploy resources in regions outside of the policy. For example, the regions that are allowed are West Europe and East US. It should not be possible to deploy resources in any other region.

Tags of resources and resource groups

Every resource in Azure, including the resource groups, will mandatorily have tags assigned to it. The tags will include, as a minimum, details about the department, environment, creation date, and project name.

Diagnostic logs and Application Insights for all resources

Every resource deployed on Azure should have diagnostic logs and application logs enabled wherever possible.

Azure locks

A company should implement Azure locks to ensure that crucial resources are not deleted accidentally. Every resource that is crucial for the functioning of a solution needs to be locked down. This means that even the administrators of the services running on Azure do not have the capability to delete these resources; the only way to delete a resource is to remove the lock first.

You should also note that:

All production and pre-production environments, apart from the development and testing environments, would be locked for deletion.

All development and testing environments that have single instances would also be locked for deletion.

All resources related to the web application would be locked for deletion for all production environments.

All shared resources would be locked for deletion irrespective of the environment.

Summary

In this chapter, you learned that governance and cost management are among the top priorities for companies moving to the cloud. Having an Azure subscription with a pay-as-you-go scheme can harm the company budget, because anyone with access to the subscription can provision as many resources as they like. Some resources are free, but others are expensive.

You also learned that it is important for organizations to remain in control of their cloud costs. Tags help in generating billing reports. These reports could be based on departments, projects, owners, or any other criteria. While cost is important, governance is equally important. Azure provides locks, policies, and RBAC to implement proper governance. Policies ensure that resource operations can be denied or audited, locks ensure that resources cannot be modified or deleted, and RBAC ensures that employees have the right permissions to perform their jobs. With these features, companies can have sound governance and cost control for their Azure deployments.

In the next chapter, we will be discussing cost management in Azure. We will go through different optimization methods, cost management, and billing APIs.

6

Cost management for Azure solutions

In the previous chapter, we discussed tags, policies, and locks and how they can be leveraged from a compliance standpoint. Tags allow us to add metadata to our resources; they also help us in the logical management of resources. In the Azure portal, we can filter resources based on tags. If we assume there is a large number of resources, which is quite common in enterprises, filtering will help us to easily manage our resources. Another benefit of tags is that they can be used to filter our billing reports or usage reports in terms of tags. In this chapter, we are going to explore cost management for Azure solutions.

The primary reason why corporations are moving to the cloud is to save on cost. There is no upfront cost for having an Azure subscription. Azure provides a pay-as-you-go subscription, where billing is based on consumption. Azure measures resource usage and provides monthly invoices. There is no upper limit for Azure consumption. As we are on a public cloud, Azure (like any other service provider) has some hard and soft limits on the number of resources that can be deployed. Soft limits can be increased by working with Azure Support. There are some resources that have a hard limit. The service limits can be found at <https://docs.microsoft.com/azure/azure-resource-manager/management/azure-subscription-service-limits>, and the default limit varies based on the type of subscription you have.

It is important for companies to keep a close watch on Azure consumption and usage. Although they can create policies to set organizational standards and conventions, there is also a need to keep track of billing and consumption data. Moreover, they should employ best practices for consuming Azure resources so that the return is maximized. For this, architects need to know about Azure resources and features, their corresponding costs, and the cost/benefit analysis of features and solutions.

In this chapter, we will cover the following topics:

- Azure offer details
- Billing
- Invoicing
- Usage and quotas
- Usage and Billing APIs
- Azure pricing calculator
- Best practices for cost optimization

Let's go ahead and discuss each of these points.

Azure offer details

Azure has different offers that you can purchase. So far, we have discussed pay-as-you-go, but there are others, such as **Enterprise Agreements (EAs)**, Azure Sponsorship, and Azure in CSP. We will cover each of these as they are very important for billing:

- **Pay-as-you-go:** This is a commonly known offering, where customers pay based on the consumption and the rates are available in the public-facing documentation of Azure. Customers receive an invoice every month for usage from Microsoft and they can pay via credit card or an invoice payment method.
- **EAs:** EAs entail a monetary commitment with Microsoft, which means that organizations sign an agreement with Microsoft and promise that they will use x amount of Azure resources. If the usage goes above the agreed amount, the customer will receive an overage invoice. Customers can create multiple accounts under an EA and have multiple subscriptions inside these accounts. There are two types of EA: Direct EA and Indirect EA. Direct EA customers have a direct billing relationship with Microsoft; on the other hand, with Indirect EA, the billing is managed by a partner. EA customers will get better offers and discounts because of the commitment they make with Microsoft. The EA is managed via a portal called the EA portal (<https://ea.azure.com>), and you need to have enrolment privileges to access this portal.

- **Azure in CSP:** Azure in CSP is where a customer reaches out to a **Cloud Solution Provider (CSP)** partner, and this partner provisions a subscription for the customer. The billing will be completely managed by the partner; the customer will not have a direct billing relationship with Microsoft. Microsoft invoices the partner and the partner invoices the customer, adding their margin.
- **Azure Sponsorship:** Sponsorship is given by Microsoft to start-ups, NGOs, and other non-profit organizations to use Azure. The sponsorship is for a fixed term and a fixed amount of credit. If the term expires or credit gets exhausted, the subscription will be converted to a pay-as-you-go subscription. If organizations want to renew their sponsorship entitlement, they have to work with Microsoft.

We have just outlined a few of Azure's offerings. The complete list is available at <https://azure.microsoft.com/support/legal/offer-details>, which includes other offerings, such as Azure for Students, Azure Pass, and Dev/Test subscriptions.

Next, let's discuss billing in Azure.

Understanding billing

Azure is a service utility that has the following features:

- No upfront costs
- No termination fees
- Billing per second, per minute, or per hour depending on the type of resource
- Payment based on consumption-on-the-go

In such circumstances, it is very difficult to estimate the upfront cost of consuming Azure resources. Every resource in Azure has its own cost model and charge based on storage, usage, and time span. It is very important for management, administration, and finance departments to keep track of usage and costs. Azure provides usage and billing reporting capabilities to help upper management and administrators generate cost and usage reports based on multiple criteria.

The Azure portal provides detailed billing and usage information through the **Cost Management + Billing** feature, which can be accessed from the master navigation blade, as shown in *Figure 6.1*:

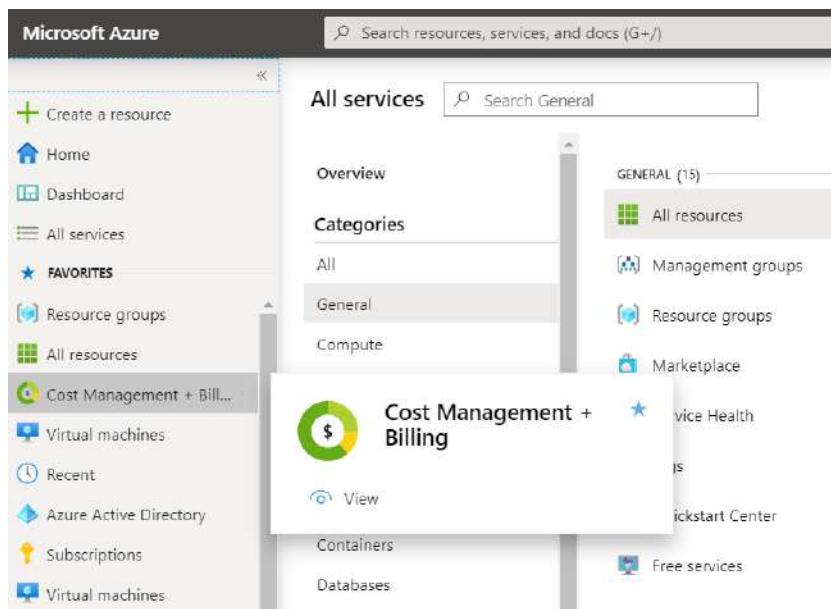


Figure 6.1: Cost Management + Billing service in the Azure portal

Please note that if your billing is managed by a CSP, you will not have access to this feature. CSP customers can view their costs in the pay-as-you-go scheme if they transition their CSP legacy subscriptions to the Azure plan. We will discuss the Azure plan and the Modern Commerce platform later in the chapter.

Cost Management + Billing shows all the subscriptions and the billing scopes you have access to, as shown in Figure 6.2:

Subscription name	Subscription ID	Status	Last billed amount	Due date	Current Cost
Pay-As-You-Go Dev/Test		Active	Not available	Not available	₹0.00
Pay-As-You-Go		Active	Not available	Not available	₹0.00
Pay-As-You-Go		Active	Not available	Not available	₹0.00
[PROD] ritihin.net		Active	Not available	Not available	₹231.45
Project Hawk Eye 360		Active	Not available	Not available	₹947.10
Pay-As-You-Go Dev/Test		Active	Not available	Not available	₹0.00

Figure 6.2: Billing overview of user subscriptions

The **Cost Management** section has several blades, such as:

- **Cost analysis** for analyzing the usage of a scope.
- **Budgets** for setting budgets.
- **Cost alerts** for notifying administrators when the usage exceeds a certain threshold.
- **Advisor recommendations** for getting advice on how to make potential savings. We will discuss Azure Advisor in the last section of this chapter.
- **Exports** for automating usage exports to Azure Storage.
- **Cloudyn**, which is a tool used by CSP partners to analyze costs, as they don't have access to Cost Management.
- **Connectors for AWS** for connecting your AWS consumption data to Azure Cost Management.

The different options available in Azure Cost Management are shown in *Figure 6.3*:

The screenshot shows the 'Cost Management: Pay-As-You-Go Dev/Test | Overview' page. The left sidebar includes links for Overview, Go to subscription, Access control, Diagnose and solve problems, Cost Management (Cost analysis, Cost alerts, Budgets, Advisor recommendations, Cloudyn), Settings (Exports, Connectors for AWS (Preview)), and a search bar. The main content area features three cards:

- Analyze cloud costs**: Break down and analyze costs to identify anomalies and drive a deeper understanding of cost and usage.
- Monitor with budgets**: Create a budget to control costs and configure alerts to warn teams about impending budget.
- Optimize with recommendations**: View Advisor recommendations to identify unused or underutilized resources. Take action to reduce costs.

The top right of the page displays the message: "Analyze and optimize cloud costs. Visualize and monitor your cloud costs and trends, improve your organizational accountability, and optimize your cloud efficiency. Learn more".

Figure 6.3: Cost Management overview

Clicking on the **Cost analysis** menu on this blade provides a comprehensive interactive dashboard, using which cost can be analyzed with different dimensions and measures:

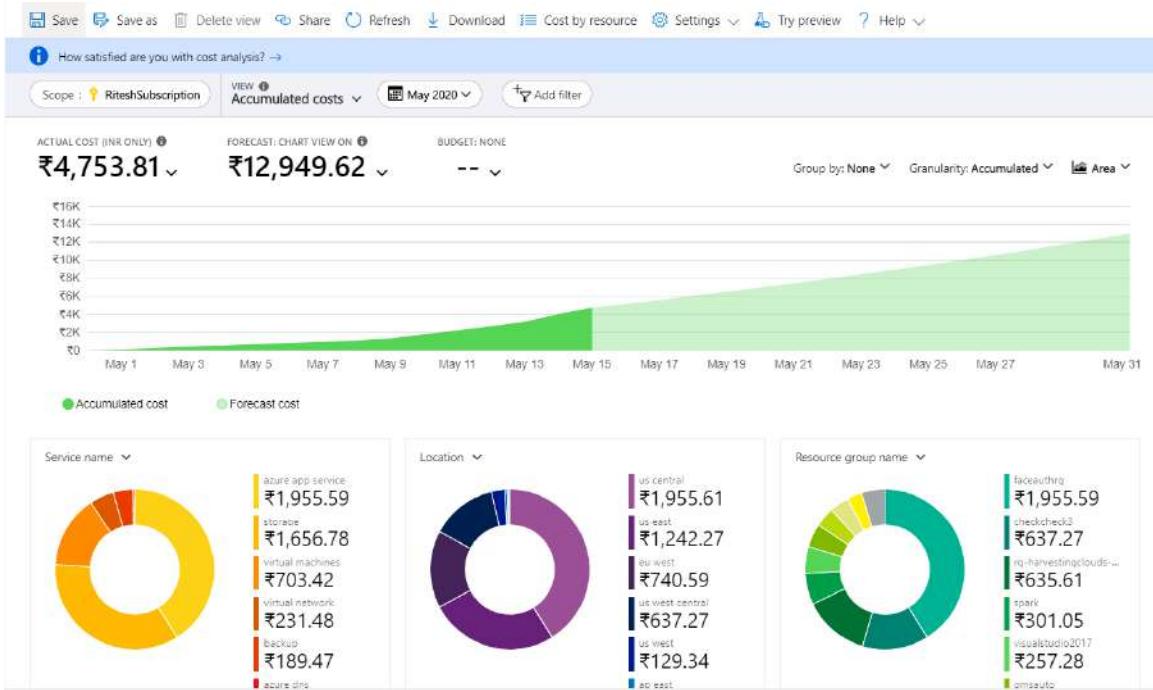


Figure 6.4: Analyzing subscription costs through the Cost analysis option

The dashboard not only shows the current cost but also forecasts the cost and breaks it down based on multiple dimensions. **Service name**, **Location**, and **Resource group name** are provided by default, but they can be changed to other dimensions as well. There will be always a scope associated with every view. Some of the available scopes are billing account, management group, subscription, and resource group. You can switch the scope depending on the level you want to analyze.

The **Budget** menu on the left allows us to set up the budget for better cost management and provides alerting features in case the actual cost is going to breach the budget estimates:

The screenshot shows the Azure Cost Management + Billing dashboard. The left sidebar has a 'Subscription' icon and a search bar. Below it are several links: Overview, Go to subscription, Access control, Diagnose and solve problems, Cost analysis, Cost alerts, **Budgets** (which is selected and highlighted in grey), Advisor recommendations, and Cloudyn. The main content area is titled 'Cost Management: [PROD] rithin.net | Create budget'. It features a 'Create a budget' button and a 'Set alerts' link. A descriptive text says: 'Create a budget and set alerts to help you monitor your costs.' Below this is a 'Budget scoping' section with a note: 'The budget you create will be assigned to the selected scope. Use additional filters like resource groups to have your budget monitor with more granularity as needed.' It shows the current scope as '[PROD] rithin.net' with a 'Change scope' link. There is also an 'Add filter' button. The 'Budget Details' section asks for a unique name ('Enter a unique name'), a reset period ('Monthly'), a creation date ('2020 May 1'), and an expiration date ('2022 April 30').

Figure 6.5: Creating a budget

Cost Management also allows us to fetch cost data from other clouds, such as AWS, within the current dashboards, thereby managing costs for multiple clouds from a single blade and dashboard. However, this feature is in preview at the time of writing. This connector will become chargeable after August 25, 2020.

You need to fill in your AWS role details and other details to pull the cost information, as shown in *Figure 6.5*. If you are unsure how to create the policy and role in AWS, refer to <https://docs.microsoft.com/azure/cost-management-billing/costs/aws-integration-set-up-configure#create-a-role-and-policy-in-aws>:

The screenshot shows the Azure Cost Management + Billing interface on the left and a 'Create connector' dialog on the right.

Left Side (Cost Management + Billing):

- Header: Cost Management: [PROD] rithin.net | Connectors for AWS (Preview)
- Submenu: Overview, Go to subscription, Access control, Diagnose and solve problems.
- Cost Management: Cost analysis, Cost alerts, Budgets, Advisor recommendations, Cloudyn.
- Settings: Exports, Connectors for AWS (Preview).
- Support + troubleshooting: New support request.

Right Side (Create connector dialog):

Title: Create connector

Instructions: Create a connector to bring AWS cost and usage report data into Azure Cost Management. After you create a connector, you'll have a single location to view all your cloud costs.

Review Tab: It might take a few hours for the new AWS scopes and their cost data to appear. Learn more.

Basics Tab: This connector will become chargeable on Aug 26, 2020. Turn auto-renew on anytime before the date to continue using the service. Learn more.

Basics	
Display name	aws-cost
Management Group	mgmt
Billing	Trial
Auto-Renew	Off
Subscription	[PROD] rithin.net

AWS properties	
Role ARN	arn:aws:iam::487610537256:role/acm-manager
External ID	*****
Report name	432263259397

Buttons: Previous, Next, Create.

Figure 6.6: Creating an AWS connector in Cost Management

The cost reports can also be exported to a storage account on a scheduled basis.

Some of the cost analysis is also available in the **Subscriptions** blade. In the **Overview** section, you can see resources and their cost. Also, there is another graph, where you can see your current spend, forecast, and balance credit (if you are using a credit-based subscription).

Figure 6.7 shows cost information:

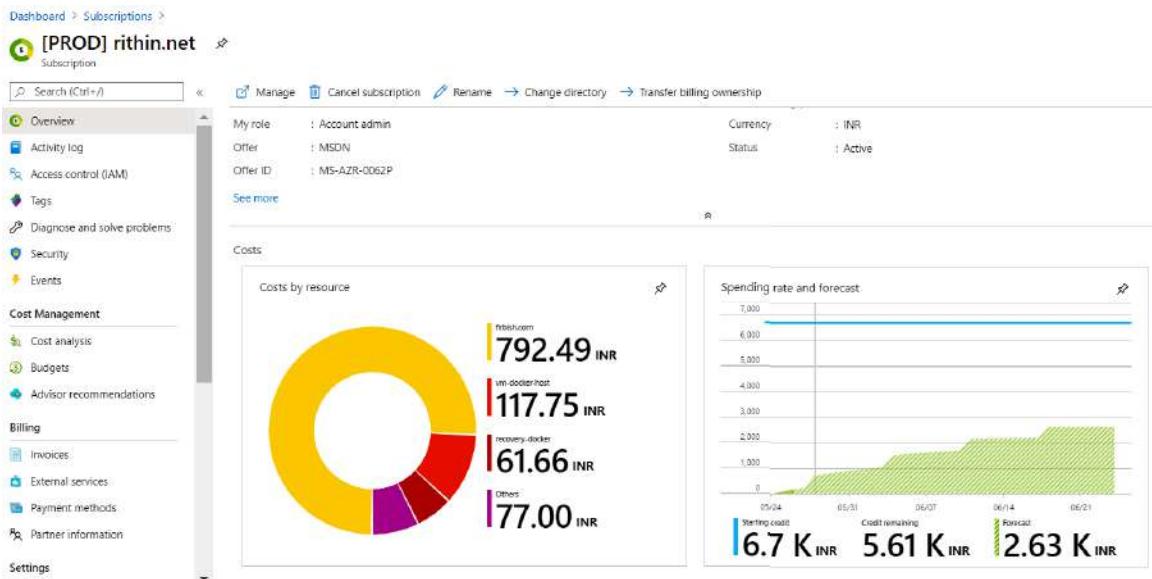


Figure 6.7: Cost analysis for the subscription

Clicking on any of the costs in Figure 6.7 will redirect you to the **Cost Management – Cost Analysis** section. There are a lot of dimensions in Cost Management with which you can group the data for analysis. The available dimensions will vary based on the scope you have selected. Some of the commonly used dimensions are as follows:

- Resource types
- Resource group
- Tag
- Resource location
- Resource ID
- Meter category
- Meter subcategory
- Service

At the beginning of the chapter, we said that tags can be used for cost management. For example, let's say you have a tag called Department with values of IT, HR, and Finance. Tagging the resources appropriately will help you understand the cost incurred by each department. You can also download the cost report as a CSV, Excel, or PNG file using the **Download** button.

Additionally, Cost Management supports multiple views. You can create your own dashboard and save it. EA customers get the added benefit of the Cost Management connector or Power BI. With the connector, users can pull the usage statistics to Power BI and create visualizations.

Up to this point, we have been discussing how we can keep track of our usage using Cost Management. In the next section, we will explore how invoicing works for the services we have used.

Invoicing

Azure's billing system also provides information about invoices that are generated monthly.

Depending on the offer type, the method of invoicing may vary. For pay-as-you-go users, the invoices will be sent monthly to the account administrator. However, for EA customers, the invoice will be sent to the contact on the enrollment.

Clicking on the **Invoices** menu brings up a list of all the invoices generated, and clicking on any of the invoices provides details about that invoice. *Figure 6.8* shows how the invoices are shown in the Azure portal:

Invoice ID	Billing Period	Invoice date	Amount	Status	Type	Download
G001290052	4/1/2020-4/30/2020	5/9/2020	INR 0.00	Paid	Azure Marketplace and Reserv...	Download
G001124632	3/1/2020-3/31/2020	4/9/2020	INR 0.00	Paid	Azure Marketplace and Reserv...	Download
G001009558	2/1/2020-2/29/2020	3/9/2020	INR 0.00	Paid	Azure Marketplace and Reserv...	Download

Figure 6.8: List of invoices and their details

There are two types of invoices: one is for Azure services such as SQL, Virtual Machines, and Networking. Another type is for Azure Marketplace and Reservations. Azure Marketplace provides partner services from different vendors for customers. We will be talking about Azure Reservations later on.

By default, for a pay-as-you-go subscription, the account admin has access to the invoices. If they want, they can delegate access to other users, such as the organization's finance team, by choosing the **Access invoice** option in Figure 6.8. Additionally, the account admin can opt for email addresses where they want to send out the copies of the invoices.

The **Email Invoice** option is not available for Support Plan now. Alternatively, you can visit the Accounts portal and download the invoice. Microsoft is slowly moving away from this portal, and most of the features are getting deprecated as they are integrated into the Azure portal.

So far, we have discussed subscriptions and how invoicing is done. Something new that has been introduced by Microsoft is Modern Commerce. With this new commerce experience, the purchase process and experience has been simplified. Let's take a closer look at Modern Commerce and learn how it is different from the legacy platform that we have discussed so far.

The Modern Commerce experience

If your organization is already working with Microsoft, you will know that there are multiple agreements involved for each offer, such as Web Direct, EAs, CSP, **Microsoft Service and Product Agreement (MSPA)**, **Server Cloud Enrollments (SCE)**, and so on. Along with this, each of them has its own portal; for example, EAs have the EA portal, CSP has the Partner Center portal, and Volume Licensing has its own portal too.

Each offer comes with a different set of terms and conditions, and the customers need to go through them every time they make a purchase. The transition from one offer to another is not very easy as each offer has a different set of terms and conditions. Let's imagine that you already have an EA subscription and would like to convert it to a CSP subscription; you may have to delete some of the partner services as they are not supported in CSP. For each product, each offer will have different rules. From a customer standpoint, it's very hard to understand what supports what and how rules differ.

Addressing this issue, Microsoft has recently issued a new agreement called **Microsoft Customer Agreement (MCA)**. This will act as the basic terms and conditions. You can make amendments to it whenever required when you sign up for a new program.

For Azure, there will be three **Go-To-Market (GTM)** programs:

- **Field Led:** Customers will interact directly with the Microsoft Accounts Team and the billing will be directly managed by Microsoft. Eventually, this will replace EAs.
- **Partner Led:** This is equivalent to the Azure-in-CSP program, where a partner manages your billing. There are different partners across the world. A quick web search will help you find the partners around you. This program will replace the Azure-in-CSP program. As the first step to Modern Commerce, a partner will sign a **Microsoft Partner Agreement (MPA)** with Microsoft and transition their existing customers by making them sign the MCA. At the time of writing this book, many partners have transitioned their customers to Modern Commerce, and the new commerce experience is available in 139 countries.
- **Self Service:** This will be a replacement for Web Direct. It doesn't require any involvement from the partner or the Microsoft Accounts Team. Customers can directly purchase from microsoft.com and they will sign the MCA during the purchase.

In Azure, the billing will be done on the Azure Plan, and the billing will be always aligned with the calendar month. Buying an Azure Plan is very similar to buying any other subscription. The difference is that the MCA will be signed during the process.

Azure Plan can host multiple subscriptions, and it will act as a root-level container. All the usage is tied back to a single Azure Plan. All the subscriptions inside the Azure Plan will act as containers to host services, such as Virtual Machines, SQL Database, and Networking.

Some of the changes and advancements we could observe after the introduction of Modern Commerce are as follows:

- Eventually, the portals will be deprecated. For example, earlier EA customers were only able to download the enrollment usage information from the EA portal. Now Microsoft has integrated it into Azure Cost Management with a richer experience than the EA portal.
- Pricing will be done in USD and billed in the local currency. If your currency is not USD, then the **foreign exchange (FX)** rate will be applied and is available in your invoice. Microsoft uses FX rates from Thomson Reuters, and these rates will be assigned on the first of every month. This value will be consistent throughout the month, irrespective of what the market rate is.
- CSP customers who transition to the new Azure Plan will be able to use Cost Management. Access to Cost Management opens a new world of cost tracking, as it provides access to all native Cost Management features.

All the subscriptions that we have discussed so far will eventually be moved to an Azure Plan, which is the future of Azure. Now that you understand the basics of Modern Commerce, let's discuss another topic that has a very important role when we are architecting solutions. Most services have limits by default; some of these limits can be increased while some are hard limits. When we are architecting a solution, we need to make sure that there is ample quota. Capacity planning is a vital part of architectural design. In the next section, you will learn more about limits on subscriptions.

Usage and quotas

As mentioned in the previous section, capacity planning needs to be one of the first steps when we architect a solution. We need to verify whether the subscription has enough quota to accommodate the new resources we are architecting. If not, during the deployment, we may face issues.

Each subscription has a limited quota for each resource type. For example, there could be a maximum of 10 public IP addresses provisioned with an MSDN Microsoft account. Similarly, all resources have a maximum default limit for each resource type. These resource type numbers for a subscription can be increased by contacting Azure Support or clicking on the **Request Increase** button in the **Usage + Quota** blade on the **Subscription** page.

Considering the number of resources in each region, it'll be a challenge to go through the list. The portal provides options to filter the dataset and look for what we want. In *Figure 6.9*, you can see that if we filter the location to **Central US** and set the resource provider to **Microsoft.Storage**, we can confirm which quotas are available for storage accounts:

The screenshot shows the Azure portal's 'Usage + Quota' blade for a subscription named '[PROD] rithin.net'. The left sidebar navigation includes 'Subscription', 'Search (Ctrl+)', 'Customer services', 'Payment methods', 'Partner information', 'Programmatic deployment', 'Resource groups', 'Resources', 'Usage + quotas' (which is selected), 'Policies', 'Management certificates', 'My permissions', 'Resource providers', 'Deployments', 'Properties', and 'Resource locks'. 'Support + troubleshooting' is at the bottom. The main content area has a header with 'Refresh' and a note: 'You can use each Microsoft Azure resource up to its quota. Each subscription has separate quotas and usage is tracked per subscription. If you reach a quota cap, you can request an increase via Help + Support. Learn more.' It features a 'Request Increase' button. Below is a table with columns: Quota, Provider, Location, and Usage. One row is shown: 'Storage Accounts' under 'Provider' (Microsoft.Storage), 'Central US' under 'Location', and '0 %' / '0 of 250' under 'Usage'. There is also a 'Filter items...' input field.

Figure 6.9: Usage and quota for a given location and resource provider

You can clearly see in *Figure 6.9* that we haven't created any storage accounts in Central US, and that leaves us with a quota of 250 accounts. If the solution we are architecting requires more than 250 accounts, we need to click on **Request Increase**, which would contact Azure Support.

This blade gives us the freedom to perform capacity planning prior to deployment.

When filtering the report, we used the term **resource provider** and selected **Microsoft Storage**. In the next section, will take a closer look at what this term means.

Resource providers and resource types

Whether you are interacting with the Azure portal, filtering services, or filtering the billing usage report, you might need to work with resource providers and resource types. For example, when you are creating a virtual machine, you are interacting with the **Microsoft.Compute** resource provider and the **virtualMachines** resource type. The **create** button that you click on to create the virtual machines communicates with the resource provider via an API to get your deployment done. This is always denoted in the format **{resource-provider}/{resource-type}**. So, the resource type for the virtual machine is **Microsoft.Compute/virtualMachines**. In short, resource providers help to create resource types.

Resource providers need to be registered with an Azure subscription. Resource types will not be available in a subscription if resource providers are not registered. By default, most providers are automatically registered; having said that, there will be scenarios where we must manually register.

To get a list of providers that are available, the ones that are registered and the ones that are not registered, and to register non-registered providers or vice versa, the dashboard shown in *Figure 6.10* can be used. For this operation, you need to have the necessary roles assigned—Owner or Contributor roles will suffice. *Figure 6.10* shows what the dashboard looks like:

Dashboard >

[PROD] rithin.net | Resource providers

Subscription

Search (Ctrl+ /) Register Unregister Refresh

Filter by name...

Provider	Status
Microsoft.RecoveryServices	Registered
Microsoft.AlertsManagement	Registered
Microsoft.Authorization	Registered
Microsoft.Automation	Registered
Microsoft.DomainRegistration	Registered
Microsoft.ResourceHealth	Registered
Microsoft.Web	Registered
Microsoft.Storage	Registered
microsoft.insights	Registered
Microsoft.Network	Registered
Microsoft.Compute	Registered
Microsoft.AAD	Registered
Microsoft.Advisor	Registered
Microsoft.Security	Registered

External services
Payment methods
Partner information
Settings
Programmatic deployment
Resource groups
Resources
Usage + quotas
Policies
Management certificates
My permissions
Resource providers
Deployments
Properties
Resource locks
Support + troubleshooting
New support request

Figure 6.10: List of registered and non-registered resource providers

In the previous section, we discussed how to download invoices and usage information. If you need to download the data programmatically and save it, then you can use APIs. The next section is all about Azure Billing APIs.

Usage and Billing APIs

Although the portal is a great way to find usage, billing, and invoice information manually, Azure also provides the following APIs to programmatically retrieve details and create customized dashboards and reports. The APIs vary depending on the kind of subscription you are using. As there are many APIs, we will be sharing the Microsoft documentation with each API so that you can explore them all.

Azure Enterprise Billing APIs

EA customers have a dedicated set of APIs available for them to work with billing data. The following APIs use the API key from the EA portal for authentication; tokens from Azure Active Directory will not work with them:

- **Balance and Summary API:** As we discussed earlier, EA works with a monetary commitment, and it is very important to track the balance, overage, credit adjustments, and Azure Marketplace charges. Using this API, customers can pull the balance and summary for a billing period.
- **Usage Details API:** The Usage Details API will help you to get daily usage information about the enrollment with granularity down to the instance level. The response of this API will be like the usage report that can be downloaded from the EA portal.
- **Marketplace Store Charge API:** This is a dedicated API for extracting the charges for Marketplace purchases.
- **Price Sheet API:** Each enrollment will have a special price sheet, and discounts vary from customers to customers. The Price Sheet API can pull the price list.
- **Reserved Instance Details API:** We haven't discussed Azure Reservations so far, but it will be discussed by the end of this chapter. Using this API, you can get usage information about the reservations and a list of the reservations in the enrollment.

Here is the link to the documentation for the EA APIs: <https://docs.microsoft.com/azure/cost-management-billing/manage/enterprise-api>.

Let's take a look at the Azure Consumption APIs now.

Azure Consumption APIs

Azure Consumption APIs can be used with EA as well as Web Direct (with some exceptions) subscriptions. This requires a token, which needs to be generated by authenticating against Azure Active Directory. Since these APIs also support EA, don't confuse this token with the EA API key that we mentioned in the previous section. Here are the key APIs that are self-explanatory:

- Usage Details API
- Marketplace Charges API
- Reservation Recommendations API
- Reservation Details and Summary API

EA customers have additional support for the following APIs:

- Price sheet
- Budgets
- Balances

Documentation is available here: <https://docs.microsoft.com/azure/cost-management-billing/manage/consumption-api-overview>.

Additionally, there is another set of APIs that can be only used by Web Direct customers:

- **Azure Resource Usage API:** This API can be used with an EA or pay-as-you-go subscription to download the usage data.
- **Azure Resource RateCard API:** This is only applicable to Web Direct; EAs are not supported. Web Direct customers can use this to download price sheets.
- **Azure Invoice Download API:** This is only applicable to Web Direct customers. It's used to download invoices programmatically.

The names may look familiar, and the difference is only the endpoint that we are calling. For Azure Enterprise Billing APIs, the URL will start with <https://consumption.azure.com>, and for Azure Consumption APIs, the URL starts with <https://management.azure.com>. This is how you can differentiate them. In the next section, you will be seeing a new set of APIs that are specifically used by Cost Management.

Azure Cost Management APIs

With the introduction of Azure Cost Management, a new set of APIs are available for the customer to use. These APIs are the backbone of Cost Management, which we used in the Azure portal earlier. The key APIs are as follows:

- **Query Usage API:** This is the same API used by Cost Analysis in the Azure portal. We can customize the response with what we need using a payload. It's very useful when we want a customized report. The date range cannot exceed 365 days.
- **Budgets API:** Budgets is another feature of Azure Cost Management, and this API lets us interact with the budgets programmatically.
- **Forecast API:** This can be used to get a forecast of a scope. The Forecast API is only available for EA customers now.
- **Dimensions API:** Earlier, when we were discussing Cost Management, we said that Cost Management supports multiple dimensions based on the scope. If you would like to get the list of dimensions supported based on a specific scope, you can use this API.
- **Exports API:** Another feature of Cost Management is that we can automate the report export to a storage account. The Exports API can be used to interact with the export configuration, such as the name of the storage account, customization, frequency, and so on.

Check out the official documentation at <https://docs.microsoft.com/rest/api/cost-management>.

Since Modern Commerce is expanding MCA, there's a whole new set of APIs that can be explored here: <https://docs.microsoft.com/rest/api/billing>.

You may have noticed that we haven't mentioned CSP in any of these scenarios. In CSP, the customer doesn't have access to the billing as it's managed by a partner, hence the APIs are not exposed. However, a transition to an Azure Plan would let the CSP customer use the Azure Cost Management APIs to see the retail rates.

Any programming or scripting language can be used to use these APIs and mix them together to create complete comprehensive billing solutions. In the next section, we will be focusing on the Azure pricing calculator, which will help the customer or architect to understand the cost of a deployment.

Azure pricing calculator

Azure provides a cost calculator for users and customers to estimate their cost and usage. This calculator is available at <https://azure.microsoft.com/pricing/calculator>:

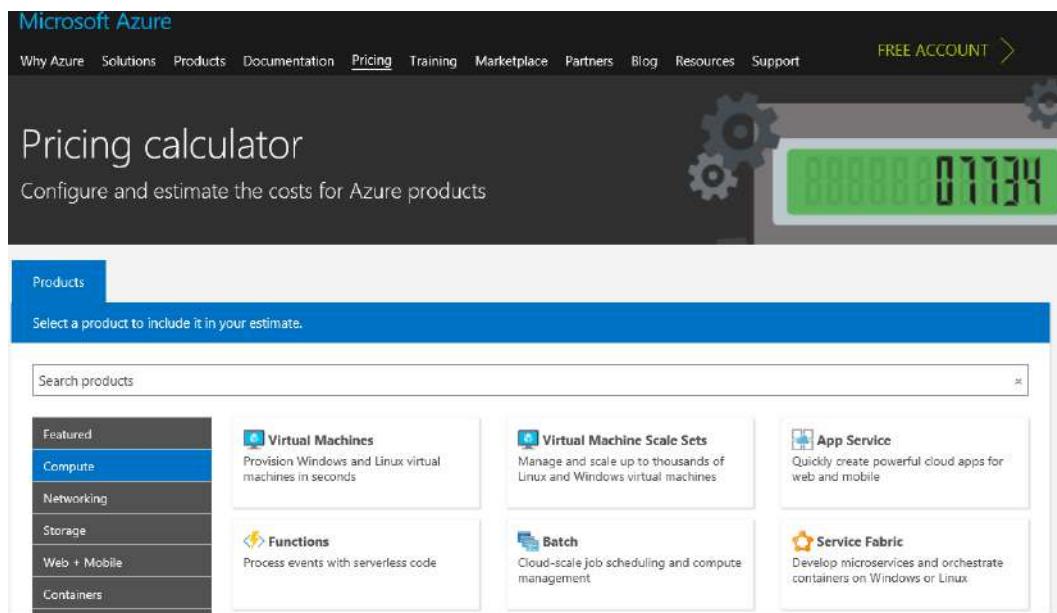


Figure 6.11: Azure pricing calculator

Users can select multiple resources from the left menu, and they will be added to the calculator. In the following example, a virtual machine is added. Further configuration with regard to virtual machine region, operating system, type, tier, instance size, number of hours, and count should be provided to calculate costs:

Your Estimate

Virtual Machines 1: D1: 1 cores, 3.5 GB RAM, 50 GB disk

Virtual Machines		
REGION:	OPERATING SYSTEM:	TYPE:
West US	Windows	(OS Only)
TIER:		
Standard	<input type="checkbox"/> ADD MANAGED DISKS	
INSTANCE:	D1: 1 Core(s), 3.5 GB RAM, 50 GB Disk, \$ 0.140/hour	
1	744	\$ 104.16
Virtual machines	Hours	

Figure 6.12: Providing configuration details to calculate resource costs

Similarly, the cost for Azure Functions, which relates to virtual machine memory size, execution time, and executions per seconds, is shown in *Figure 6.13*:

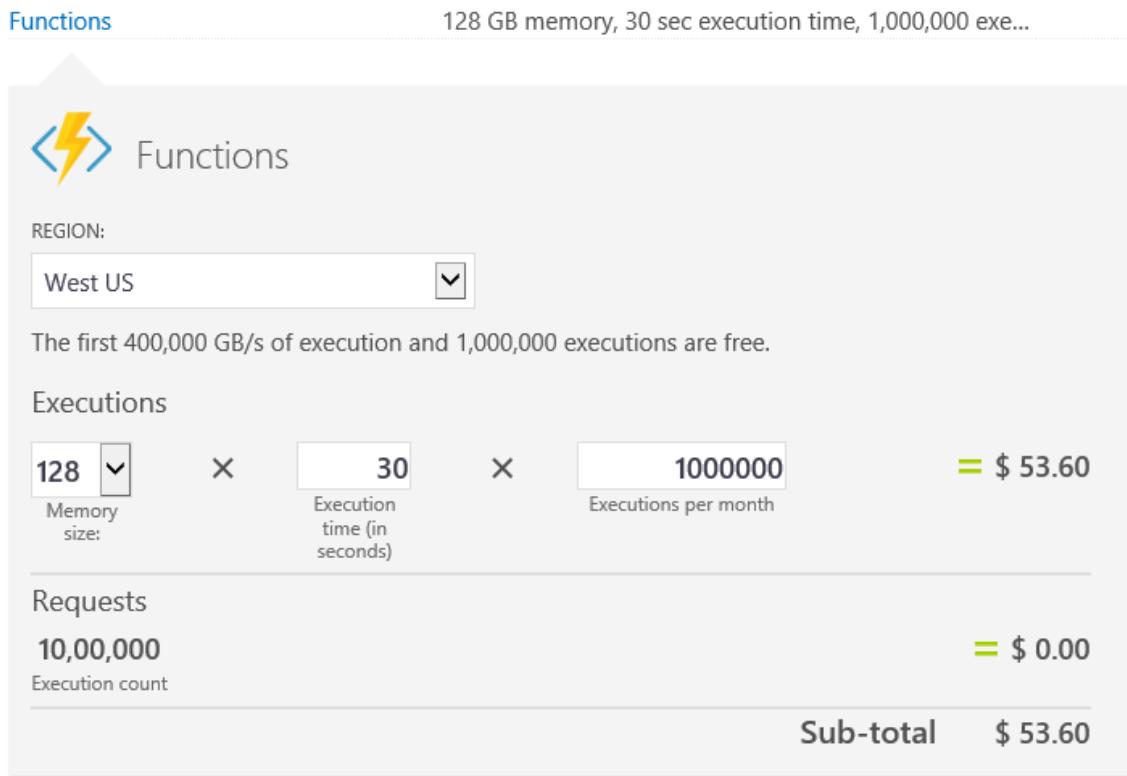


Figure 6.13: Calculating costs for Azure Functions

Azure provides different levels and support plans:

- Default support: free
- Developer support: \$29 per month
- Standard support: \$300 per month
- Profession direct: \$1,000 per month

To see a complete comparison of the support plans, please refer to <https://azure.microsoft.com/support/plans>.

You can select the support that you want depending on your requirements. Finally, the overall estimated cost is displayed:

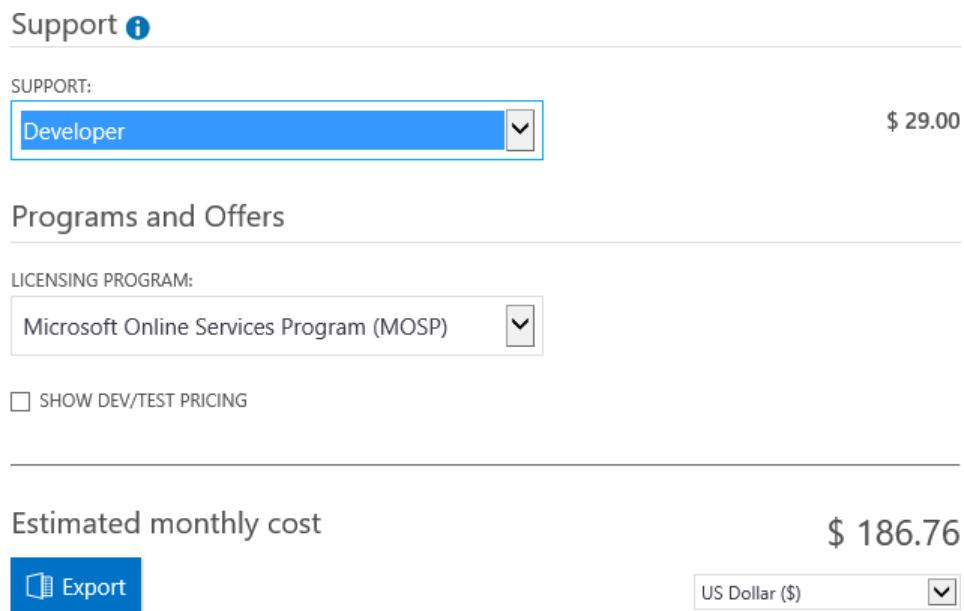


Figure 6.14: Cost estimation for selected support plan

It is important that architects understand every Azure feature used in their architecture and solution. The success of the Azure calculator depends on the resources that are selected and how they are configured. Any misrepresentation would lead to biased and incorrect estimates and would be different than the actual billing.

We have reached the last section of this chapter. We have covered the basics of billing, and now it's time to learn the best practices. Following the best practices will help you achieve cost optimization.

Best practices

Architects need to understand their architecture and the Azure components that are utilized. Based on the active monitoring, audits, and usage, they should determine the best offering from Microsoft in terms of SKU, size, and features. This section will detail some of the best practices to be adopted from a cost optimization perspective.

Azure Governance

Azure Governance can be defined as a set of processes or mechanisms that can be leveraged to maintain complete control over the resources deployed in Azure. Some of the key points are as follows:

- Set up a naming convention for all resource types and resource groups. Ensure that the naming convention is followed consistently and comprehensively across all resources and resource groups. This can be done by establishing Azure policies.

- Set up a logical organization and multiple taxonomies for resources, resource groups, and subscriptions by applying tags to them. Tags categorize resources and can also help evaluate costs from different perspectives. This can be done by establishing Azure policies, and multiple policies can be combined into initiatives. These initiatives can be applied, which in turn will apply all policies for compliance checks and reporting.
- Use Azure Blueprints instead of ARM templates directly. This will ensure that the deployment of new environments, resources, and resource groups can be standardized according to corporate standards, including naming conventions and the use of tags.

Compute best practices

Compute refers to services that help in the execution of services. Some of the best compute practices that Azure architects should follow for optimal utilization of resources and cost efficiency are as follows:

- Leverage Azure Advisor to see the available options to save costs on your virtual machines and to find out whether a virtual machine is underutilized. Advisor uses machine learning patterns and artificial intelligence to analyze your usage and provide recommendations. These recommendations play an important role in cost optimization.
- Use Azure **Reserved Instances (RI)**. RIs can get you potential savings on compute costs by paying the cost of the virtual machine upfront or monthly. The term of the RI can be a year or three years. If you buy an RI, that will cut down the compute cost and you will only be seeing the disk, network, and license (if any) charges for a virtual machine. If you have five virtual machines, you can opt for five RIs to suppress the cost of compute completely. RIs automatically look for virtual machines with the matching SKU and attaches to it. Potential savings may vary from 20% to 40% depending on the size of the virtual machine.
- Using **Azure Hybrid Benefit (AHUB)**, you can use your own Windows Server or SQL licenses to reduce the license cost. Combining RIs and AHUB can give you immense savings.
- Choose the best location for your compute services, such as virtual machines. Choose a location where all the Azure features are together in the same region. This will avoid egress traffic.
- Choose the optimal size for virtual machines. A large virtual machine costs more than a small one, and a large virtual machine might not be required at all.

- Resize virtual machines during times of demand and reduce their size when demand subsides. Azure releases new SKUs quite frequently. If a new size suits your needs better, then it must be used.
- Shut down compute services during off-hours or when they are not needed. This is for non-production environments.
- Deallocate virtual machines instead of shutting them down. This will release all their resources, and the meter for their consumption will stop.
- Use dev/test labs for development and testing purposes. They provide policies, auto-shutdown, and auto-start features.
- With virtual machine scale sets, start with a small number of virtual machines and scale out when demand increases.
- Choose the correct size (small, medium, or large) for application gateways. They are backed up by virtual machines and can help reduce costs.
- Choose a Basic tier application gateway if the web application firewall is not needed.
- Choose the correct tiers for VPN gateways (Basic VPN, Standard, High performance, and Ultra performance).
- Reduce network traffic between Azure regions.
- Use a load balancer with a public IP to access multiple virtual machines rather than assigning a public IP to each virtual machine.
- Monitor virtual machines and calculate performance and usage metrics. Based on those calculations, determine whether you want to upscale or downscale your virtual machines. This could result in downsizing and reducing the number of virtual machines.

Considering these best practices while architecting will inevitably lead to cost savings. Now that we have covered compute, let's take a similar approach to storage.

Storage best practices

Since we are hosting our applications in the cloud, Azure Storage will be used to store the data related to these applications. If we don't follow the right practices, things may go wrong. Some of the best practices for storage are as follows:

- Choose an appropriate storage redundancy type (GRS, LRS, RA-GRS). GRS is costlier than LRS.

- Archive storage data to the cool or archive access tier. Keep data that is frequently accessed in the hot tier.
- Remove blobs that are not required.
- Delete virtual machine operating system disks explicitly after deleting virtual machines if they are not needed.
- Storage accounts should be metered based on their size, write, read, list, and container operations.
- Prefer standard disks over premium disks; use premium disks only if your business requires it.
- Using CDN and caching for static files instead of fetching them from storage every time.
- Azure offers reserved capacity to save costs on blob data.

Keeping these best practices handy will help you to architect cost-effective storage solutions. In the next section, we are going to discuss the best practices involved in the deployment of PaaS services.

PaaS best practices

There are many PaaS services offered by Azure and if they are misconfigured, the chances are that you might end up with unexpected charges in the invoice. To avoid this scenario, you can leverage the following best practices:

- Choose an appropriate Azure SQL tier (Basic, Standard, Premium RS, or Premium) and appropriate performance levels.
- Choose appropriately between a single database and an elastic database. If there are a lot of databases, it is more cost efficient to use elastic databases than single databases.
- Re-architect your solution to use PaaS (serverless or microservices with containers) solutions instead of IaaS solutions. These PaaS solutions take away maintenance costs and are available on a per-minute consumption basis. If you do not consume these services, there is no cost, despite your code and services being available around the clock.

There are resource-specific cost optimizations, and it is not possible to cover all of them in a single chapter. You are advised to read the documentation related to each feature's cost and usage.

General best practices

So far, we have looked at service-specific best practices, and we will wind up this section with some general guidelines:

- The cost of resources is different across regions. Try an alternate region, provided it's not creating any performance or latency issues.
- EAs offer better discounts than other offers. You can speak to the Microsoft Account Team and see what benefits you can get if you sign up to an EA.
- If Azure costs can be pre-paid, then you get discounts for all kinds of subscriptions.
- Delete or remove unused resources. Figure out resources that are underutilized and reduce their SKU or size. If they aren't needed, delete them.
- Use Azure Advisor and take its recommendations seriously.

As mentioned earlier, these are some generic guidelines and as you architect more solutions, you'll be able to create a set of best practices for yourself. But to begin with, you can consider these ones. Nevertheless, each component in Azure has its own best practices, and referring to the documentation while you are architecting will help you to create a cost-effective solution.

Summary

In this chapter, we learned the importance of cost management and administration when working in a cloud environment. We also covered the various Azure pricing options and various price optimization capabilities that Azure offers. Managing a project's cost is paramount primarily because the monthly expense could be very low but could rise if the resources are not monitored periodically. Cloud architects should design their applications in a cost-effective manner. They should use appropriate Azure resources, appropriate SKUs, tiers, and sizes, and know when to start, stop, scale up, scale out, scale down, scale in, transfer data, and more. Proper cost management will ensure that the actual expense meets the budgetary expenses.

In the next chapter, we will look at various Azure features related to data services, such as Azure SQL, Cosmos DB, and sharding.

7

Azure OLTP solutions

Azure provides both **Infrastructure as a Service (IaaS)** and **Platform as a Service (PaaS)** services. These types of services provide organizations with different levels and controls over storage, compute, and networks. Storage is the resource used when working with the storage and transmission of data. Azure provides lots of options for storing data, such as Azure Blob storage, Table storage, Cosmos DB, Azure SQL Database, Azure Data Lake Storage, and more. While some of these options are meant for big data storage, analytics, and presentation, there are others that are meant for applications that process transactions. Azure SQL is the primary resource in Azure that works with transactional data.

This chapter will focus on various aspects of using transactional data stores, such as Azure SQL Database and other open-source databases that are typically used in **Online Transaction Processing (OLTP)** systems, and will cover the following topics:

- OLTP applications
- Relational databases
- Deployment models
- Azure SQL Database
- Single Instance
- Elastic pools
- Managed Instance
- Cosmos DB

We will start this chapter by looking at what OLTP applications are and listing the OLTP services of Azure and their use cases.

OLTP applications

As mentioned earlier, OLTP applications are applications that help in the processing and management of transactions. Some of the most prevalent OLTP implementations can be found in retail sales, financial transaction systems, and order entry. These applications perform data capture, data processing, data retrieval, data modification, and data storage. However, it does not stop here. OLTP applications treat these data tasks as transactions. Transactions have a few important properties and OLTP applications account for these properties. These properties are grouped under the acronym **ACID**. Let's discuss these properties in detail:

- **Atomicity:** This property states that a transaction must consist of statements and either all statements should complete successfully or no statement should be executed. If multiple statements are grouped together, these statements form a transaction. Atomicity means each transaction is treated as the lowest single unit of execution that either completes successfully or fails.
- **Consistency:** This property focuses on the state of data in a database. It dictates that any change in state should be complete and based on the rules and constraints of the database, and that partial updates should not be allowed.

- **Isolation:** This property states that there can be multiple concurrent transactions executed on a system and each transaction should be treated in isolation. One transaction should not know about or interfere with any other transaction. If the transactions were to be executed in sequence, by the end, the state of data should be the same as before.
- **Durability:** This property states that the data should be persisted and available, even after failure, once it is committed to the database. A committed transaction becomes a fact.

Now that you know what OLTP applications are, let's discuss the role of relational databases in OLTP applications.

Relational databases

OLTP applications have generally relied on relational databases for their transaction management and processing. Relational databases typically come in a tabular format consisting of rows and columns. The data model is converted into multiple tables where each table is connected to another table (based on rules) using relationships. This process is also known as normalization.

There are multiple services in Azure that support OLTP applications and the deployment of relational databases. In the next section, we will take a look at the services in Azure that are related to OLTP applications.

Azure cloud services

A search for **sql** in the Azure portal provides multiple results. I have marked some of them to show the resources that can be used directly for OLTP applications:

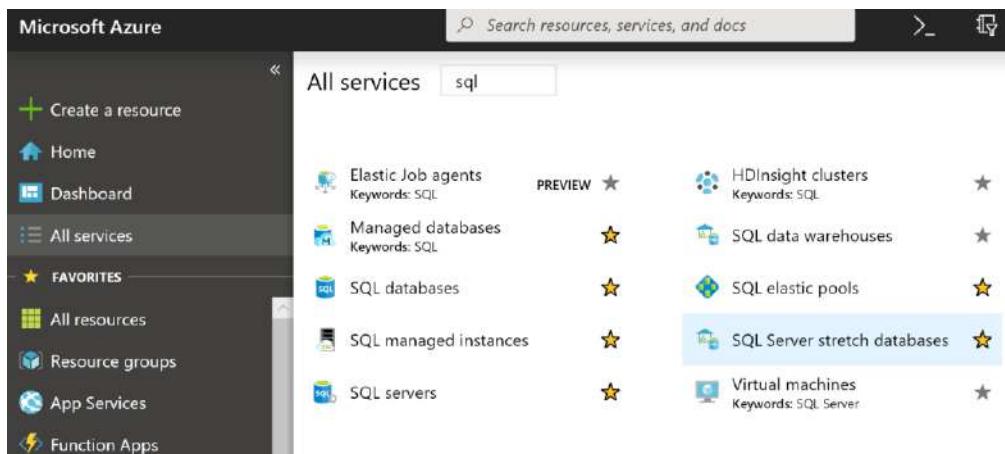


Figure 7.1: List of Azure SQL services

Figure 7.1 shows the varied features and options available for creating SQL Server-based databases on Azure.

Again, a quick search for **database** in the Azure portal provides multiple resources, and the marked ones in Figure 7.2 can be used for OLTP applications:

Figure 7.2: List of Azure services used for OLTP applications

Figure 7.2 shows resources provided by Azure that can host data in a variety of databases, including the following:

- MySQL databases
- MariaDB databases
- PostgreSQL databases
- Cosmos DB

Next, let's discuss deployment models.

Deployment models

Deployment models in Azure are classified based on the level of management or control. It's up to the user to select which level of management or control they prefer; either they can go for complete control by using services such as Virtual Machines, or they can use managed services where things will be managed by Azure for them.

There are two deployment models for deploying databases on Azure:

- Databases on Azure Virtual Machines (IaaS)
- Databases hosted as managed services (PaaS)

We will now try to understand the difference between deployment on Azure Virtual Machines and managed instances. Let's start with Virtual Machines.

Databases on Azure Virtual Machines

Azure provides multiple **stock keeping units (SKUs)** for virtual machines. There are high-compute, high-throughput (IOPS) machines that are also available along with general-use virtual machines. Instead of hosting a SQL Server, MySQL, or any other database on on-premises servers, it is possible to deploy these databases on these virtual machines. The deployment and configuration of these databases are no different than that of on-premises deployments. The only difference is that the database is hosted on the cloud instead of using on-premises servers. Administrators must perform the same activities and steps that they normally would for an on-premises deployment. Although this option is great when customers want full control over their deployment, there are models that can be more cost-effective, scalable, and highly available compared to this option, which will be discussed later in this chapter.

The steps to deploy any database on Azure Virtual Machines are as follows:

1. Create a virtual machine with a size that caters to the performance requirements of the application.
2. Deploy the database on top of it.
3. Configure the virtual machine and database configuration.

This option does not provide any out-of-the-box high availability unless multiple servers are provisioned. It also does not provide any features for automatic scaling unless custom automation supports it.

Disaster recovery is also the responsibility of the customer. Servers should be deployed on multiple regions connected using services like global peering, VPN gateways, ExpressRoute, or Virtual WAN. It is possible for these virtual machines to be connected to an on-premises datacenter through site-to-site VPNs or ExpressRoute without having any exposure to the outside world.

These databases are also known as **unmanaged databases**. On the other hand, databases hosted with Azure, other than virtual machines, are managed by Azure and are known as **managed services**. In the next section, we will cover these in detail.

Databases hosted as managed services

Managed services mean that Azure provides management services for the databases. These managed services include the hosting of the database, ensuring that the host is highly available, ensuring that the data is replicated internally for availability during disaster recovery, ensuring scalability within the constraint of a chosen SKU, monitoring the hosts and databases and generating alerts for notifications or executing actions, providing log and auditing services for troubleshooting, and taking care of performance management and security alerts.

In short, there are a lot of services that customers get out of the box when using managed services from Azure, and they do not need to perform active management on these databases. In this chapter, we will look at Azure SQL Database in depth and provide information on other databases, such as MySQL and Postgres. Also, we will cover non-relational databases such as Cosmos DB, which is a NoSQL database.

Azure SQL Database

Azure SQL Server provides a relational database hosted as a PaaS. Customers can provision this service, bring their own database schema and data, and connect their applications to it. It provides all the features of SQL Server when deployed on a virtual machine. These services do not provide a user interface to create tables and its schema, nor do they provide any querying capabilities directly. SQL Server Management Studio and the SQL CLI tools should be used to connect to these services and directly work with them.

Azure SQL Database comes with three distinct deployment models:

- **Single Instance:** In this deployment model, a single database is deployed on a logical server. This involves the creation of two resources on Azure: a SQL logical server and a SQL database.
- **Elastic pool:** In this deployment mode, multiple databases are deployed on a logical server. Again, this involves the creation of two resources on Azure: a SQL logical server and a SQL elastic database pool—this holds all the databases.
- **Managed Instance:** This is a relatively new deployment model from the Azure SQL team. This deployment reflects a collection of databases on a logical server, providing complete control over the resources in terms of system databases. Generally, system databases are not visible in other deployment models, but they are available in the model. This model comes very close to the deployment of SQL Server on-premises:

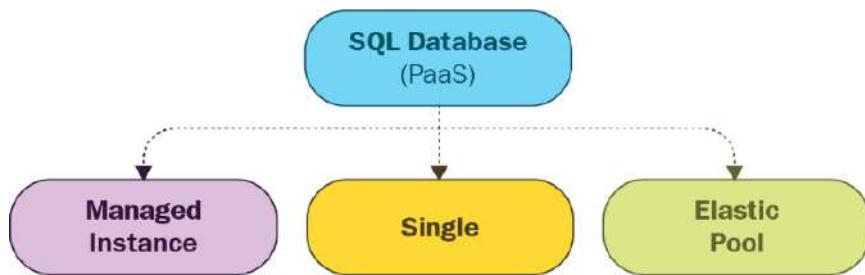


Figure 7.3: Azure SQL Database deployment models

If you are wondering when to use what, you should look at a feature comparison between SQL Database and SQL Managed Instance. A complete feature comparison is available at <https://docs.microsoft.com/azure/azure-sql/database/features-comparison>.

Next, we will cover some of the features of SQL Database. Let's start with application features.

Application features

Azure SQL Database provides multiple application-specific features that cater to the different requirements of OLTP systems:

- **Columnar store:** This feature allows the storage of data in a columnar format rather than in a row format.
- **In-memory OLTP:** Generally, data is stored in back-end files in SQL, and data is pulled from them whenever it is needed by the application. In contrast to this, in-memory OLTP puts all data in memory and there is no latency in reading the storage for data. Storing in-memory OLTP data on SSD provides the best possible performance for Azure SQL.
- All features of on-premises SQL Server.

The next feature we are going to discuss is high availability.

High availability

Azure SQL, by default, is 99.99% highly available. It has two different architectures for maintaining high availability based on SKUs. For the Basic, Standard, and General SKUs, the entire architecture is broken down into the following two layers.

- Compute layer
- Storage layer

There is redundancy built in for both of these layers to provide high availability:

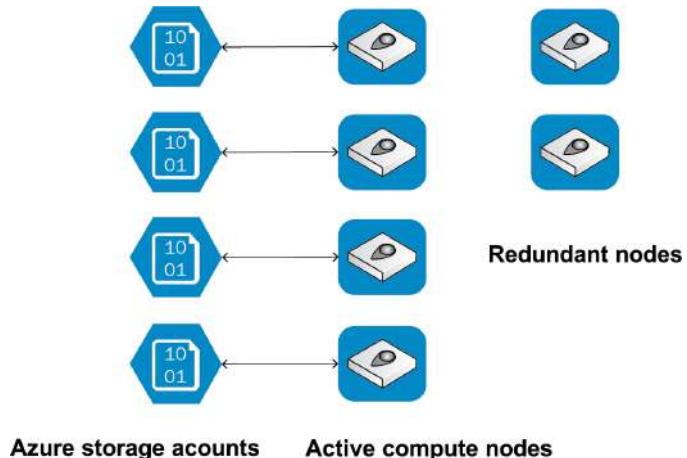


Figure 7.4: Compute and storage layers in standard SKUs

For the Premium and business-critical SKUs, both compute and storage are on the same layer. High availability is achieved by the replication of compute and storage deployed in a four-node cluster, using technology similar to SQL Server Always On availability groups:

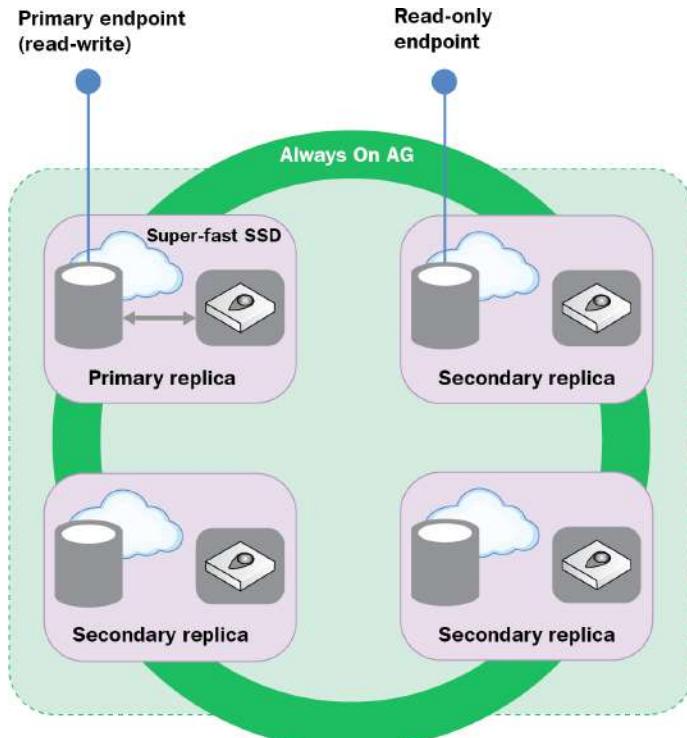


Figure 7.5: Four-node cluster deployment

Now that you know how high availability is handled, let's jump to the next feature: backups.

Backups

Azure SQL Database also provides features to automatically back up databases and store them on storage accounts. This feature is important especially in cases where a database becomes corrupt or a user accidentally deletes a table. This feature is available at the server level, as shown in *Figure 7.6*:

The screenshot shows the Azure portal interface for managing backups. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, and Manage Backups (which is selected). The main area shows a table for configuring backup policies. One row is selected for the database 'mycommerce' with a retention period of '35 days'. A modal window titled 'Configure policies' is overlaid on the page, specifically for the 'Point In Time Restore Configuration' tab. It allows setting 'Configure PiTR backup retention' to 'Days' (set to 35). Below this, there are sections for 'Long-term Retention Configurations' with options for 'Weekly LTR Backups' (set to 0 weeks), 'Monthly LTR Backups' (set to 0 weeks), and 'Yearly LTR Backups' (set to Week 1). At the bottom of the modal are 'Apply' and 'Cancel' buttons.

Figure 7.6: Backing up databases in Azure

Architects should prepare a backup strategy so that backups can be used in times of need. While configuring backups, ensure that your backups occur neither too infrequently nor too frequently. Based on the business needs, a weekly backup or even a daily backup should be configured, or even more frequently than that, if required. These backups can be used for restoration purposes.

Backups will help in business continuity and data recovery. You can also go for geo-replication to recover the data during a region failure. In the next section, we will cover geo-replication.

Geo-replication

Azure SQL Database also provides the benefit of being able to replicate a database to a different region, also known as a secondary region; this is completely based on the plan that you are choosing. The database at the secondary region can be read by applications. Azure SQL Database allows readable secondary databases. This is a great business continuity solution as a readable database is available at any point in time. With geo-replication, it is possible to have up to four secondaries of a database in different regions or the same region. With geo-replication, it is also possible to fail over to a secondary database in the event of a disaster. Geo-replication is configured at the database level, as shown in Figure 7.7:

The screenshot shows the Azure portal interface for managing a database named 'myecommerce'. The left sidebar contains navigation links for Overview, Activity log, Tags, Diagnose and solve problems, Quick start, Query editor (preview), Settings (Configure, Geo-Replication selected), Connection strings, Sync to other databases, Add Azure Search, Properties, Locks, Automation script, Security (Advanced Threat Protection, Auditing), and a search bar.

The main content area is titled 'myecommerce - Geo-Replication' and displays a world map with green circles indicating replication targets. A message box states: 'Select a region on the map or from the Target Regions list to create a secondary database.' Below the map, an information box says: 'You can now automatically manage replication, connectivity and failover of this database by adding it to failover group.'

SERVER/DATABASE	FAILOVER POLICY	STATUS
West Europe	shardserver1/myecommerce	None
Primary		Online

Figure 7.7: Geo-replication in Azure

If you scroll down on this screen, the regions that can act as secondaries are listed, as shown in Figure 7.8:

	SERVER/DATABASE	FAILOVER POLICY	STATUS
PRIMARY	West Europe shardserver1/myecommerce	None	Online
SECONDARIES	<i>Geo-Replication is not configured</i>		
TARGET REGIONS	<input type="checkbox"/> West US <input type="checkbox"/> West US 2 <input type="checkbox"/> Central US <input type="checkbox"/> West Central US <input type="checkbox"/> South Central US <input type="checkbox"/> North Central US <input type="checkbox"/> Canada Central <input type="checkbox"/> East US <input type="checkbox"/> Canada East		

Figure 7.8: List of available secondaries for geo-replication

Before architecting solutions that involve geo-replication, we need to validate the data residency and compliance regulations. If customer data is not allowed to be stored outside a region due to compliance reasons, we shouldn't be replicating it to other regions.

In the next section, we will explore scalability options.

Scalability

Azure SQL Database provides vertical scalability by adding more resources (such as compute, memory, and IOPS). This can be done by increasing the number of **Database Throughput Units (DTUs)** or compute and storage resources in the case of the **vCore** model:

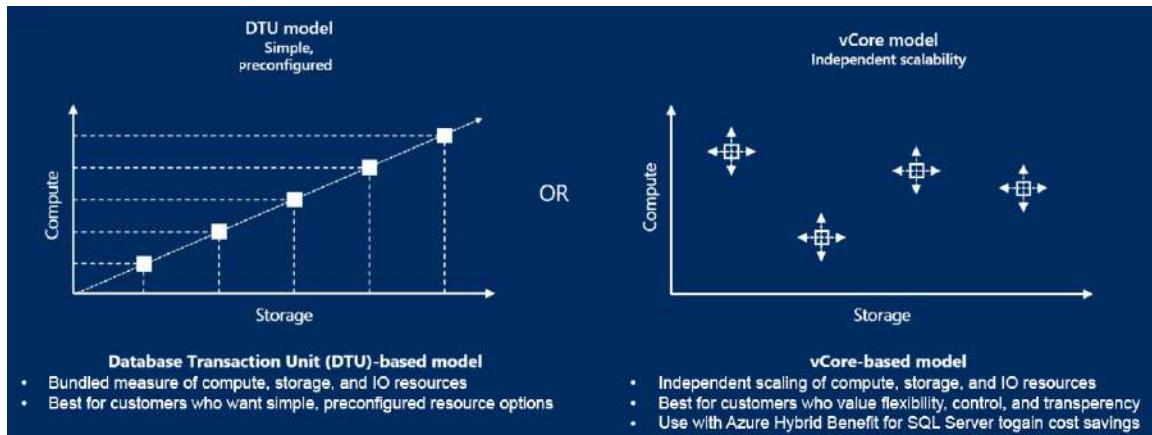


Figure 7.9: Scalability in Azure SQL Database

We have covered the differences between DTU-based model and the vCore-based model later in this chapter.

In the next section, we will cover security, which will help you understand how to build secure data solutions in Azure.

Security

Security is an important factor for any database solution and service. Azure SQL provides enterprise-grade security for Azure SQL, and this section will list some of the important security features in Azure SQL.

Firewall

Azure SQL Database, by default, does not provide access to any requests. Source IP addresses should be explicitly accepted for access to SQL Server. There is an option to allow all Azure-based services access to a SQL database as well. This option includes virtual machines hosted on Azure.

The firewall can be configured at the server level instead of the database level. The Allow access to Azure services option allows all services, including virtual machines, to access a database hosted on a logical server.

By default, this will be turned off due to security reasons; enabling this would allow access from all Azure services:

The screenshot shows the 'Firewall settings' page for the 'sharedserver1' SQL server. At the top, there's a navigation bar with 'Home > sharedserver1 (sharedserver1/sharedserver1) > Firewall settings'. Below it, the title 'Firewall settings' is displayed next to a shield icon, with 'sharedserver1 (SQL server)' underneath. There are three buttons: 'Save' (blue), 'Discard' (red), and '+ Add client IP'.

Under the 'Deny public network access' section, there are two options: 'Yes' (disabled) and 'No' (selected). A tooltip explains that setting it to Yes allows connections via approved private endpoint only and disables any existing firewall rules. It also provides a link to 'Learn more'.

The 'Minimal TLS Version' is set to '> 1.0' (selected), with other options like '> 1.1' and '> 1.2' available.

In the 'Connection Policy' section, 'Default' is selected, with 'Proxy' and 'Redirect' as other options.

The 'Allow Azure services and resources to access this server' section has 'Yes' selected. A tooltip states that connections from specified IPs provide access to all databases in sharedserver1.

The 'Client IP address' is listed as 117.210.191.108.

Rule name	Start IP	End IP	...		
ClientIPAddress_2020-6-19...	117.210.192.205	117.210.192.205	...		
ClientIPAddress_2020-6...✓	117.210.191.109	✓	117.210.191.109	✓	...

Figure 7.10: Configuring a firewall at the server level in Azure

Azure SQL Server on dedicated networks

Although access to SQL Server is generally available through the internet, it is possible for access to SQL Server to be limited to requests coming from virtual networks. This is a relatively new feature in Azure. This helps in accessing data within SQL Server from an application on another server of the virtual network without the request going through the internet.

For this, a service endpoint of the **Microsoft.Sql** type should be added within the virtual network, and the virtual network should be in the same region as that of Azure SQL Database:

The screenshot shows the Azure portal interface for managing a virtual network named "azureclitest-vnet". The left sidebar lists various management options like Firewall, Security, DNS servers, Peering, Service endpoints (which is selected and highlighted in grey), Private endpoints, Properties, Locks, and Export template. Below this is a "Monitoring" section with Diagnostic settings. The main content area is titled "Service endpoints" and shows a table with two columns: "Service" and "Subnet". A search bar at the top right says "Filter service endpoints". A large "Add" button is located at the top right of the main table area. To the right of the main table, a modal window titled "Add service endpoints" is open. It has a "Service *" dropdown menu where "Microsoft.Sql" is selected. Below the dropdown is a "Filter services" input field. A list of service names is displayed, with "Microsoft.Sql" being highlighted. At the bottom of the modal is a blue "Add" button.

Figure 7.11: Adding a Microsoft.Sql service endpoint

An appropriate subnet within the virtual network should be chosen:

The screenshot shows the Azure portal interface for managing service endpoints in a virtual network named "azureclitest-vnet". The left sidebar lists various settings like Address space, Connected devices, Subnets, DDoS protection, Firewall, DNS servers, Peerings, and Service endpoints. The "Service endpoints" option is currently selected. The main pane displays a table with columns "SERVICE" and "SUBNET". A single row is present with the value "Microsoft.Sql" under SERVICE and "default" under SUBNET. A tooltip on the right provides information about selecting the "default" subnet, stating: "rules using Azure public IP addresses will stop working with this switch. Please ensure IP firewall rules allow for this switch before setting up service endpoints. You may also experience temporary interruption to service traffic from this subnet while configuring service endpoints."

Figure 7.12: Choosing a subnet for the Microsoft.Sql service

Finally, from the Azure SQL Server configuration blade, an existing virtual network should be added that has a **Microsoft.Sql** service endpoint enabled:

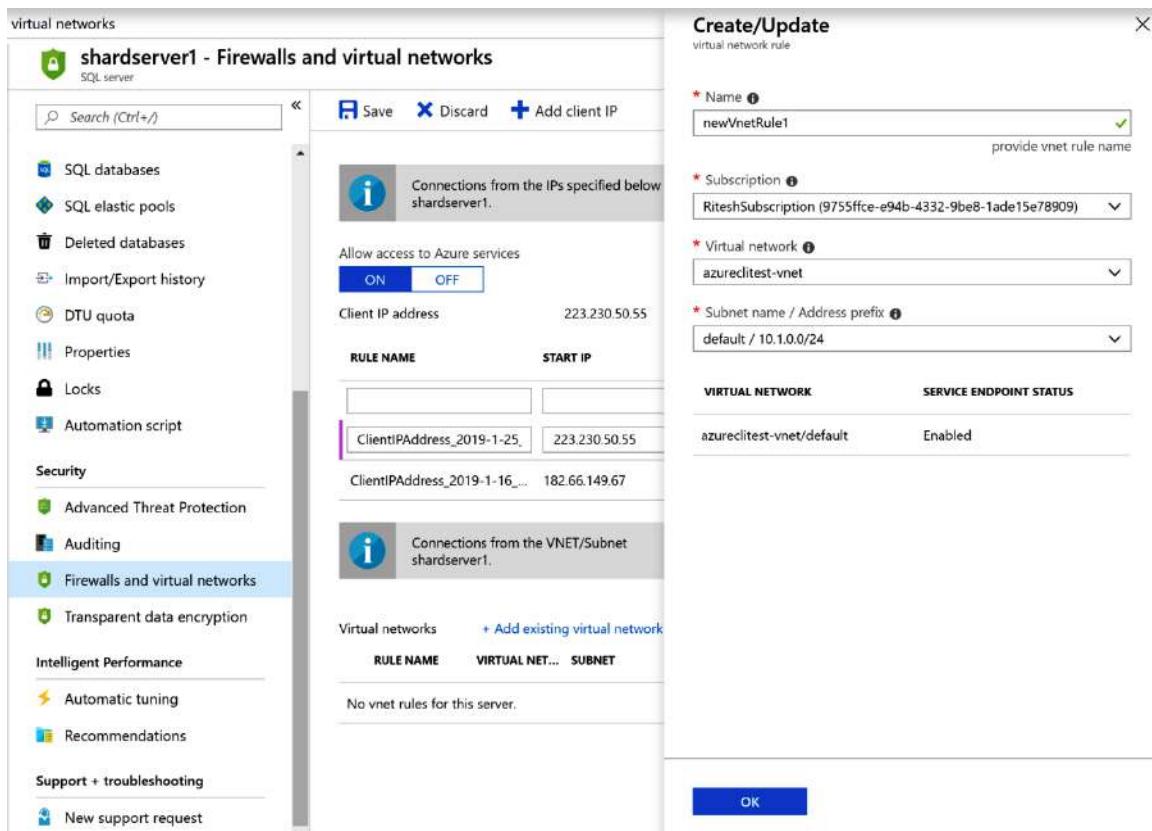


Figure 7.13: Adding a virtual network with the Microsoft.Sql service endpoint

Encrypted databases at rest

The databases should be in an encrypted form when at rest. **At rest** here means that the data is at the storage location of the database. Although you might not have access to SQL Server and its database, it is preferable to encrypt the database storage.

Databases on a filesystem can be encrypted using keys. These keys must be stored in Azure Key Vault and the vault must be available in the same region as that of Azure SQL Server. The filesystem can be encrypted by using the **Transparent data encryption** menu item of the SQL Server configuration blade and by selecting **Yes** for **Use your own key**.

The key is an RSA 2048 key and must exist within the vault. SQL Server will decrypt the data at the page level when it wants to read it and send it to the caller; then, it will encrypt it after writing to the database. No changes to the applications are required, and it is completely transparent to them:

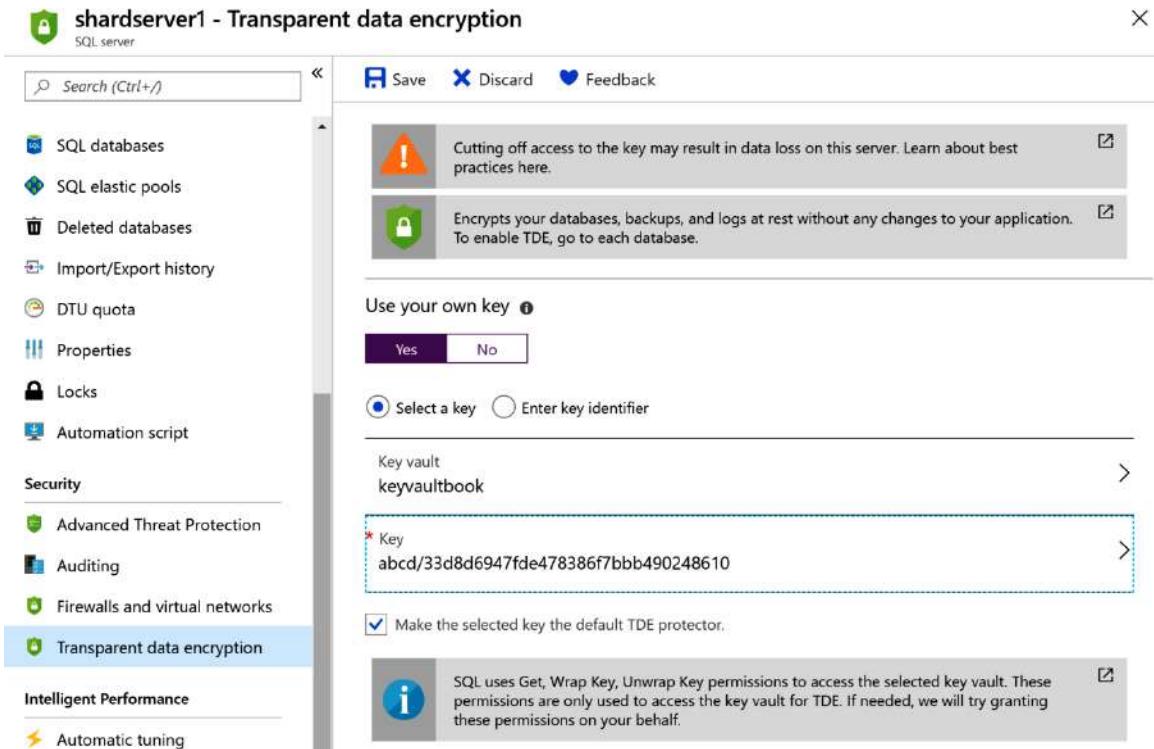


Figure 7.14: Transparent data encryption in SQL Server

Dynamic data masking

SQL Server also provides a feature that masks individual columns that contain sensitive data, so that no one apart from privileged users can view actual data by querying it in SQL Server Management Studio. Data will remain masked and will only be unmasked when an authorized application or user queries the table. Architects should ensure that sensitive data, such as credit card details, social security numbers, phone numbers, email addresses, and other financial details, is masked.

Masking rules may be defined on a column in a table. There are four main types of masks—you can check them out here: <https://docs.microsoft.com/sql/relational-databases/security/dynamic-data-masking?view=sql-server-ver15#define-a-dynamic-data-mask>.

Figure 7.15 shows how data masking is added:

MASK NAME	MASK FUNCTION
dbo_Customers_FirstName	Default value (0, xxxx, 01-01-1900)

SCHEMA	TABLE	COLUMN	
dbo	Customers	LastName	Add mask
dbo	Salary	SalaryID	Add mask

Figure 7.15: Dynamic data masking in SQL Database

Azure Active Directory integration

Another important security feature of Azure SQL is that it can be integrated with Azure **Active Directory (AD)** for authentication purposes. Without integrating with Azure AD, the only authentication mechanism available to SQL Server is via username and password authentication—that is, SQL authentication. It is not possible to use integrated Windows authentication. The connection string for SQL authentication consists of both the username and password in plaintext, which is not secure. Integrating with Azure AD enables the authentication of applications with Windows authentication, a service principal name, or token-based authentication. It is a good practice to use Azure SQL Database integrated with Azure AD.

There are other security features, such as advanced threat protection, auditing of the environment, and monitoring, that should be enabled on any enterprise-level Azure SQL Database deployments.

With that, we've concluded our look at the features of Azure SQL Database and can now move on to the types of SQL databases.

Single Instance

Single Instance databases are hosted as a single database on a single logical server. These databases do not have access to the complete features provided by SQL Server. Each database is isolated and portable. Single instances support the vCPU-based and DTU-based purchasing models that we discussed earlier.

Another added advantage of a single database is cost-efficiency. If you are in a vCore-based model, you can opt for lower compute and storage resources to optimize costs. If you need more compute or storage power, you can always scale up. Dynamic scalability is a prominent feature of single instances that helps to scale resources dynamically based on business requirements. Single instances allow existing SQL Server customers to lift and shift their on-premises applications to the cloud.

Other features include availability, monitoring, and security.

When we started our section on Azure SQL Database, we mentioned elastic pools as well. You can also transition a single database to an elastic pool for resource sharing. If you are wondering what resource sharing and what elastic pools are, in the next section, we will cover this.

Elastic pools

An elastic pool is a logical container that can host multiple databases on a single logical server. Elastic pools are available in the vCore-based and DTU-based purchasing models. The vCPU-based purchasing model is the default and recommended method of deployment, where you'll get the freedom to choose your compute and storage resources based on your business workloads. As shown in *Figure 7.16*, you can select how many cores and how much storage is required for your database:

Cost summary	
Gen5 - General Purpose	
Cost per vCore (in INR)	8907.57
vCores selected	x 8
Cost per GB (in INR)	9.12
Max storage selected (in GB)	x 130
ESTIMATED COST / MONTH	72446.34 INR

Figure 7.16: Setting up elastic pools in the vCore-based model

Also, at the top of the preceding figure, you can see there is an option that says **Looking for basic, standard, premium?** If you select this, the model will be switched to the DTU model.

The SKUs available for elastic pools in the DTU-based model are as follows:

- Basic
- Standard
- Premium

Figure 7.17 shows the maximum amounts of DTUs that can be provisioned for each SKU:

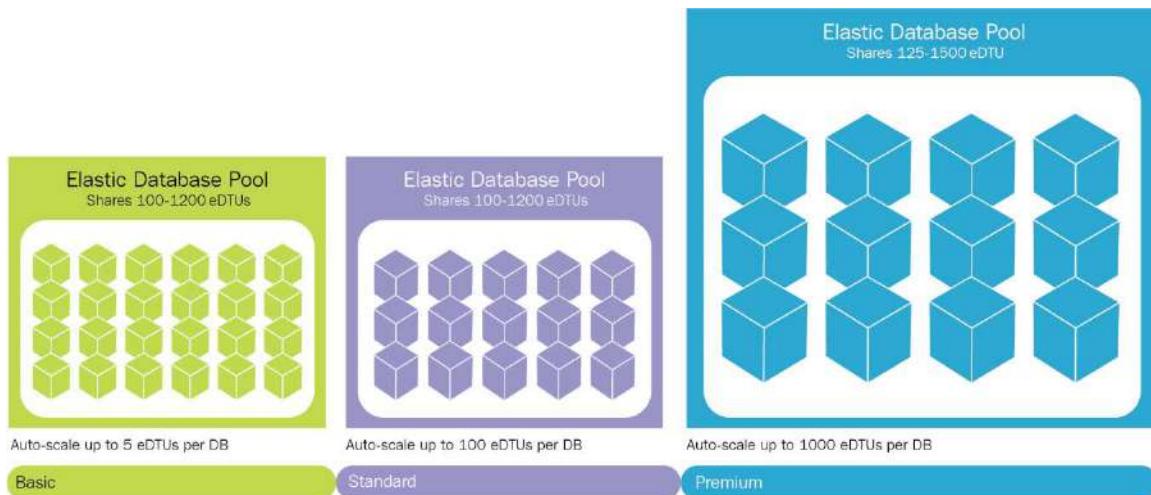


Figure 7.17: Amount of DTUs per SKU in an elastic pool

All the features discussed for Azure SQL single instances are available to elastic pools as well; however, horizontal scalability is an additional feature that enables sharding. Sharding refers to the vertical or horizontal partitioning of data and the storage of that data in separate databases. It is also possible to have autoscaling of individual databases in an elastic pool by consuming more DTUs than are actually allocated to that database.

Elastic pools also provide another advantage in terms of cost. You will see in a later section that Azure SQL Database is priced using DTUs, and DTUs are provisioned as soon as the SQL Server service is provisioned. DTUs are charged for irrespective of whether those DTUs are consumed. If there are multiple databases, then it is possible to put these databases into elastic pools and for them to share the DTUs among them.

All information for implementing sharding with Azure SQL elastic pools has been provided at <https://docs.microsoft.com/azure/sql-database/sql-database-elastic-scale-introduction>.

Next, we will discuss the Managed Instance deployment option, which is a scalable, intelligent, cloud-based, fully managed database.

Managed Instance

Managed Instance is a unique service that provides a managed SQL server similar to what's available on on-premises servers. Users have access to master, model, and other system databases. Managed Instance is ideal when there are multiple databases and customers migrating their instances to Azure. Managed Instance consists of multiple databases.

Azure SQL Database provides a new deployment model known as Azure SQL Database Managed Instance that provides almost 100% compatibility with the SQL Server Enterprise Edition Database Engine. This model provides a native virtual network implementation that addresses the usual security issues and is a highly recommended business model for on-premises SQL Server customers. Managed Instance allows existing SQL Server customers to lift and shift their on-premises applications to the cloud with minimal application and database changes while preserving all PaaS capabilities at the same time. These PaaS capabilities drastically reduce the management overhead and total cost of ownership, as shown in Figure 7.18:

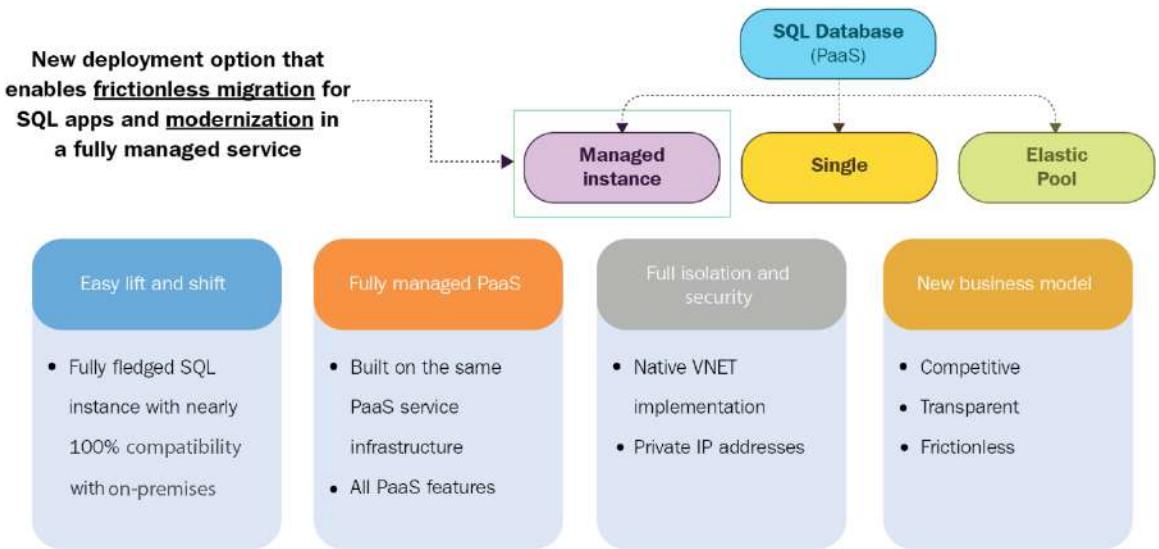


Figure 7.18: Azure SQL Database Managed Instance

The complete comparison between Azure SQL Database, Azure SQL Managed Instance, and SQL Server on an Azure virtual machine is available here: <https://docs.microsoft.com/azure/azure-sql/azure-sql-iaas-vs-paas-what-is-overview#comparison-table>.

The key features of Managed Instance are shown in the *Figure 7.19*:

Feature	Description
SQL Server version / build	SQL Server Database Engine (latest stable)
Managed automated backups	Yes
Built-in instance and database monitoring and metrics	Yes
Automatic software patching	Yes
The latest Database Engine features	Yes
Number of data files (ROWS) per the database	Multiple
Number of log files (LOG) per database	1
VNet - Azure Resource Manager deployment	Yes
VNet - Classic deployment model	No
Portal support	Yes
Built-in Integration Service (SSIS)	No - SSIS is a part of Azure Data Factory PaaS
Built-in Analysis Service (SSAS)	No - SSAS is separate PaaS
Built-in Reporting Service (SSRS)	No - use Power BI or SSRS IaaS

Figure 7.19: SQL Database Managed Instance features

We have mentioned the terms vCPU-based pricing model and DTU-based pricing model at several points throughout the chapter. It's time that we took a closer look at these pricing models.

SQL database pricing

Azure SQL previously had just one pricing model—a model based on DTUs—but an alternative pricing model based on vCPUs has also been launched. The pricing model is selected based on the customer's requirements. The DTU-based model is selected when the customer wants simple and preconfigured resource options. On the other hand, the vCore-based model offers the flexibility to choose compute and storage resources. It also provides control and transparency.

Let's take a closer look at each of these models.

DTU-based pricing

The DTU is the smallest unit of performance measure for Azure SQL Database. Each DTU corresponds to a certain amount of resources. These resources include storage, CPU cycles, IOPS, and network bandwidth. For example, a single DTU might provide three IOPS, a few CPU cycles, and IO latencies of 5 ms for read operations and 10 ms for write operations.

Azure SQL Database provides multiple SKUs for creating databases, and each of these SKUs has defined constraints for the maximum amount of DTUs. For example, the Basic SKU provides just **5** DTUs with a maximum **2 GB** of data, as shown in *Figure 7.20*:

The screenshot shows the Azure SQL Database Pricing calculator interface. At the top, there are 'Configure' and close ('X') buttons. Below is a feedback section with a heart icon and 'Feedback'.

Basic	Standard	Premium
For less demanding workloads	For workloads with typical performance requirements	For IO-intensive workloads
Starting at 329.89 INR / month	Starting at 991.50 INR / month	Starting at 30734.76 INR / month

vCore-based purchasing options: Click here to customize your performance using vCores >

DTUs What is a DTU?

5 (Basic)

Max data size

100 MB 2 GB

Cost Summary

Cost per DTU (in INR)	65.98
DTUs selected	x 5
EST. COST PER MONTH	329.89 INR

Figure 7.20: DTUs for different SKUs

On the other hand, the standard SKU provides anything between **10 DTUs** and **300 DTUs** with a maximum of **250 GB** of data. As you can see here, each DTU costs around 991 rupees, or around \$1.40:

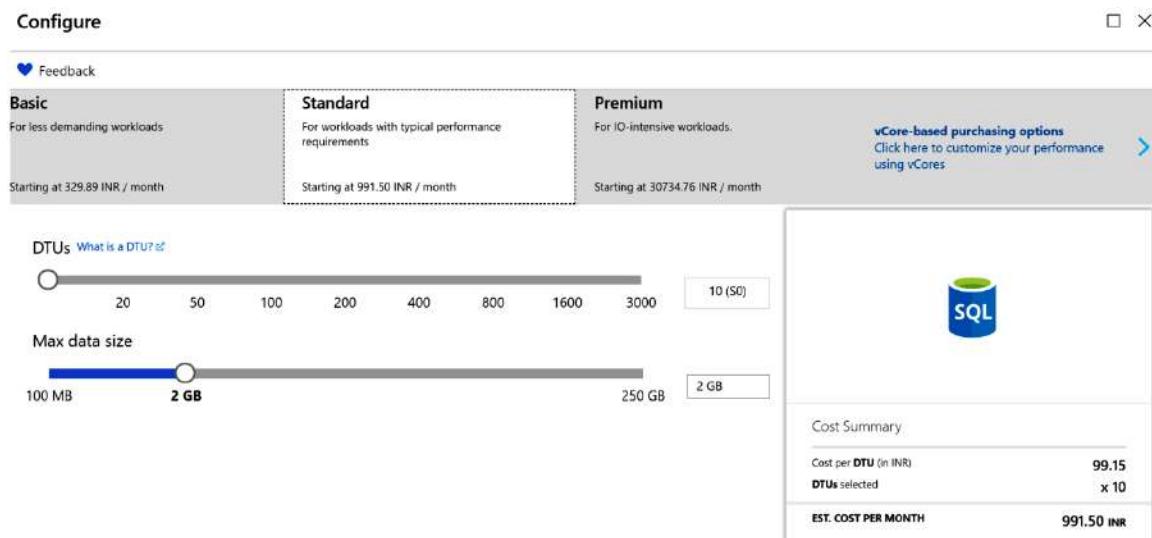


Figure 7.21: Cost summary for the selected number of DTUs in the Standard SKU

A comparison of these SKUs in terms of performance and resources is provided by Microsoft and is shown in the Figure 7.22:

	Basic	Standard	Premium
Target workload	Development and production	Development and production	Development and production
Uptime SLA	99.99%	99.99%	99.99%
Backup retention	7 days	35 days	35 days
CPU	Low	Low, Medium, High	Medium, High
IO throughput (approximate)	2.5 IOPS per DTU	2.5 IOPS per DTU	48 IOPS per DTU
IO latency (approximate)	5 ms (read), 10 ms (write)	5 ms (read), 10 ms (write)	2 ms (read/write)
Columnstore indexing	N/A	S3 and above	Supported
In-memory OLTP	N/A	N/A	Supported

Figure 7.22: SKU comparison in Azure

Once you provision a certain number of DTUs, the back-end resources (CPU, IOPS, and memory) are allocated and are charged for whether they are consumed or not. If more DTUs are procured than are actually needed, it leads to waste, while there would be performance bottlenecks if insufficient DTUs were provisioned.

Azure provides elastic pools for this reason as well. As you know, there are multiple databases in an elastic pool and DTUs are assigned to elastic pools instead of individual databases. It is possible for all databases within a pool to share the DTUs. This means that if a database has low utilization and is consuming only five DTUs, there will be another database consuming 25 DTUs in order to compensate.

It is important to note that, collectively, DTU consumption cannot exceed the amount of DTUs provisioned for the elastic pool. Moreover, there is a minimum amount of DTUs that should be assigned to each database within the elastic pool, and this minimum DTU count is preallocated for the database.

An elastic pool comes with its own SKUs:

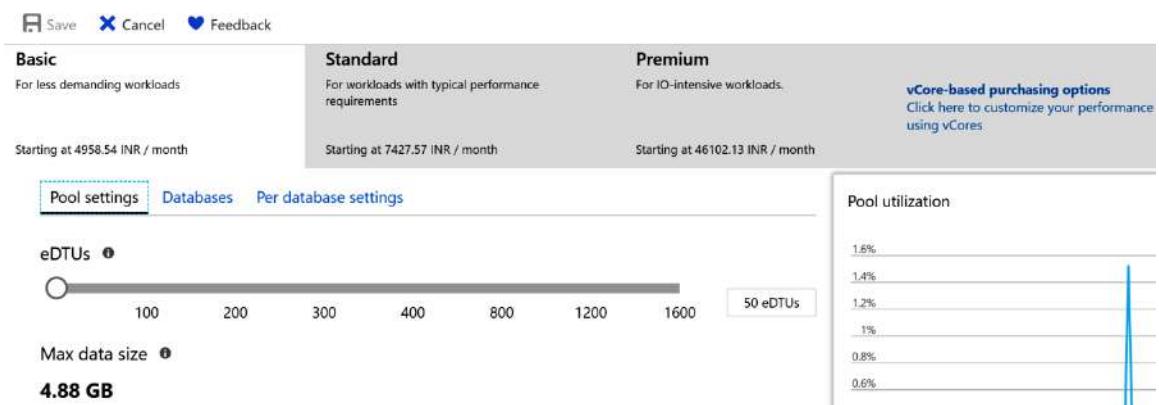


Figure 7.23: SKUs in an elastic pool

Also, there is a limit on the maximum number of databases that can be created within a single elastic pool. The complete limits can be reviewed here: <https://docs.microsoft.com/azure/azure-sql/database/resource-limits-dtu-elastic-pools>.

vCPU-based pricing

This is the new pricing model for Azure SQL. This pricing model provides options to procure the number of **virtual CPUs (vCPUs)** allocated to the server instead of setting the amount of DTUs required for an application. A vCPU is a logical CPU with attached hardware, such as storage, memory, and CPU cores.

In this model, there are three SKUs: **General Purpose**, **Hyperscale**, and **Business Critical**, with a varied number of vCPUs and resources available. This pricing is available for all SQL deployment models:

SKU	Description	Cost per vCore (in INR)	Cost per Month (in INR)
General Purpose	Scalable compute and storage options Up to 7,000 IOPS 5-10 ms latency Starting at 8292.71 INR / month	8233.91	16844.11 INR
Hyperscale	On-demand scalable storage Data up to 200,000 IOPS, 1-2 ms latency Log up to 7,000 IOPS, 5-10ms latency Starting at 9880.66 INR / month		
Business Critical	High transaction rate and high resiliency Up to 200,000 IOPS 1-2 ms latency Starting at 16595.69 INR / month		

Compute Generation

- Gen4**: up to 24 vCores, up to 168 GB memory
- Gen5** (selected): up to 80 vCores, up to 408 GB memory

vCores How do vCores compare with DTUs? 4 6 8 10 12 14 16 18 20 24 32 40 80 2 vCores

Max data size 32 GB 1 TB 32 GB ALLOCATED LOG STORAGE 9.6 GB

Cost Summary

Category	Value
Gen5 - General Purpose (GP_Gen5_2)	8233.91
Cost per vCore (in INR)	
vCores selected	x 2
Cost per GB (in INR)	9.05
Max storage selected (in GB)	x 41.6
EST. COST PER MONTH	16844.11 INR

Figure 7.24: vCPU pricing for the General Purpose SKU

How to choose the appropriate pricing model

Architects should be able to choose an appropriate pricing model for Azure SQL Database. DTUs are a great mechanism for pricing where there is a usage pattern applicable and available for the database. Since resource availability in the DTU scheme of things is linear, as shown in the next diagram, it is quite possible for usage to be more memory-intensive than CPU-intensive. In such cases, it is possible to choose different levels of CPU, memory, and storage for a database.

In DTUs, resources come packaged, and it is not possible to configure these resources at a granular level. With a vCPU model, it is possible to choose different levels of memory and CPU for different databases. If the usage pattern for an application is known, using the vCPU pricing model could be a better option compared to the DTU model. In fact, the vCPU model also provides the benefit of hybrid licenses if an organization already has on-premises SQL Server licenses. There is a discount of up to 30% provided to these SQL Server instances.

In Figure 7.25, you can see from the left-hand graph that as the amount of DTUs increases, resource availability also grows linearly; however, with vCPU pricing (in the right-hand graph), it is possible to choose independent configurations for each database:

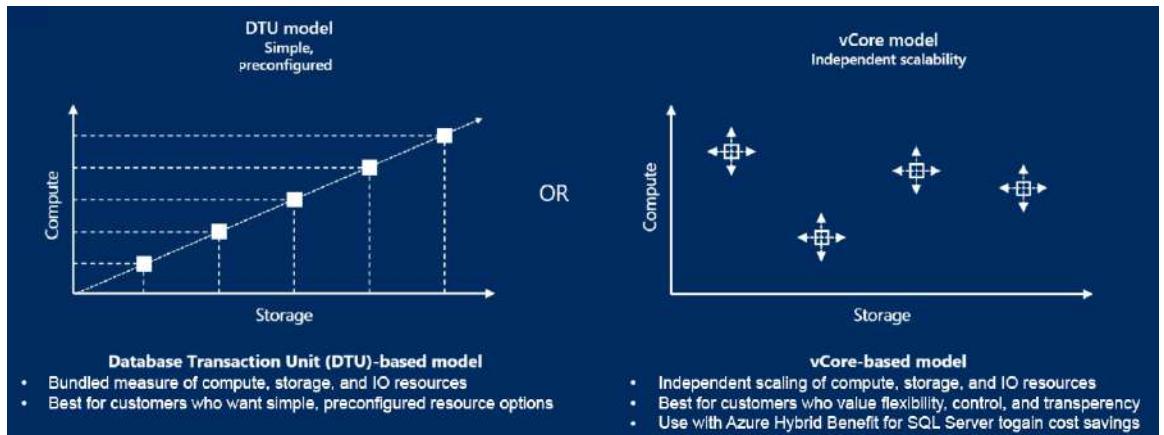


Figure 7.25: Storage-compute graph for the DTU and vCore models

With that, we can conclude our coverage of Azure SQL Database. We discussed different deployment methods, features, pricing, and plans related to Azure SQL Database. In the next section, we will be covering Cosmos DB, which is a NoSQL database service.

Azure Cosmos DB

Cosmos DB is Azure's truly cross-region, highly available, distributed, multi-model database service. Cosmos DB is for you if you would like your solution to be highly responsive and always available. As this is a cross-region multi-model database, we can deploy applications closer to the user's location and achieve low latency and high availability.

With the click of a button, throughput and storage can be scaled across any number of Azure regions. There are a few different database models to cover almost all non-relational database requirements, including:

1. SQL (documents)
2. MongoDB
3. Cassandra
4. Table
5. Gremlin Graph

The hierarchy of objects within Cosmos DB starts with the Cosmos DB account. An account can have multiple databases, and each database can have multiple containers. Depending on the type of database, the container might consist of documents, as in the case of SQL; semi-structured key-value data within Table storage; or entities and relationships among those entities, if using Gremlin and Cassandra to store NoSQL data.

Cosmos DB can be used to store OLTP data. It accounts for ACID with regard to transaction data, with a few caveats.

Cosmos DB provides for ACID requirements at the single document level. This means data within a document, when updated, deleted, or inserted, will have its atomicity, consistency, isolation, and durability maintained. However, beyond documents, consistency and atomicity have to be managed by the developer themselves.

Pricing for Cosmos DB can be found here: <https://azure.microsoft.com/pricing/details/cosmos-db/>.

Figure 7.26 shows some features of Azure Cosmos DB:

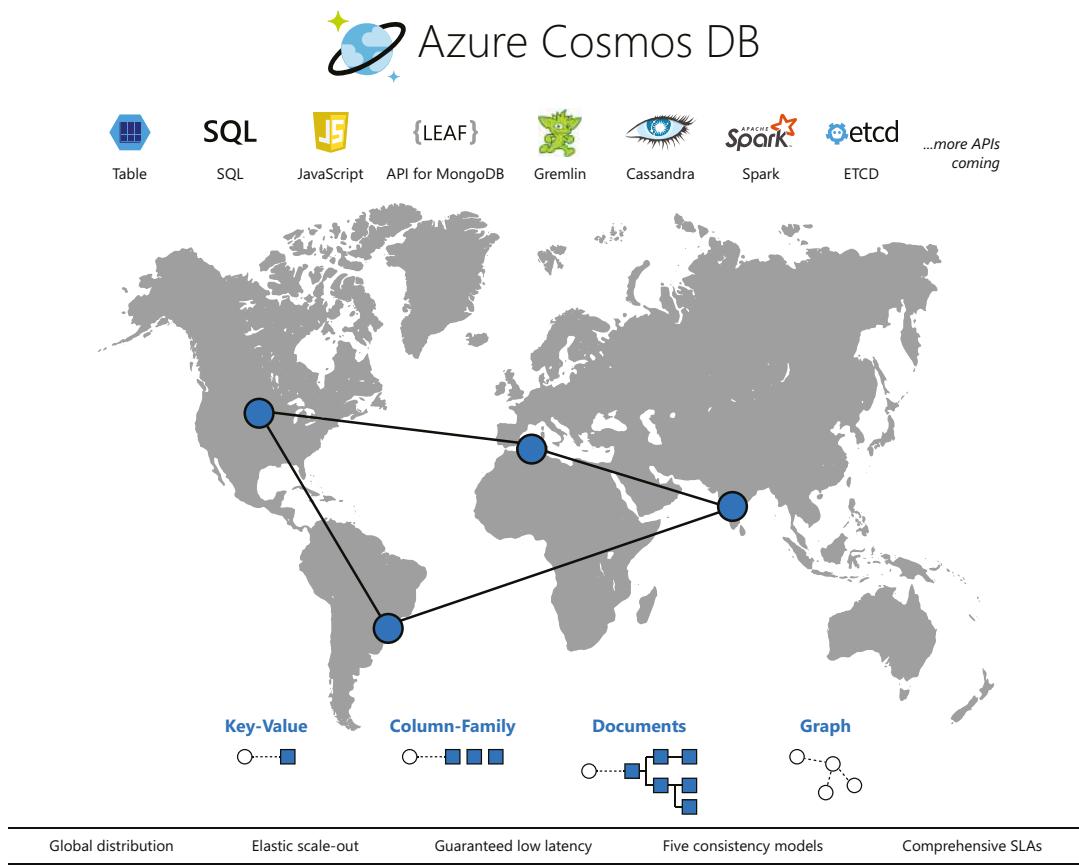


Figure 7.26: An overview of Azure Cosmos DB

In the next section, we will cover some key features of Azure Cosmos DB.

Features

Some of the key benefits of Azure Cosmos DB are:

- **Global distribution:** Highly responsive and highly available applications can be built worldwide using Azure Cosmos DB. With the help of replication, replicas of data can be stored in Azure regions that are close to users, hence providing less latency and global distribution.
- **Replication:** You can opt in to or opt out of replication to a region any time you like. Let's say you have a replica of your data available in the East US region, and your organization is planning to shut down their processes in East US and migrate to UK South. With just a few clicks, East US can be removed, and UK South can be added to the account for replication.
- **Always On:** Cosmos DB provides 99.999% of high availability for both read and write. The regional failover of a Cosmos DB account to another region can be invoked via the Azure portal or programmatically. This ensures business continuity and disaster recovery planning for your application during a region failure.
- **Scalability:** Cosmos DB offers unmatched elastic scalability for writes and reads all around the globe. The scalability response is massive, meaning that you can scale from thousands to hundreds of millions of requests/second with a single API call. The interesting thing is that this is done around the globe, but you need to pay only for throughput and storage. This level of scalability is ideal for handling unexpected spikes.
- **Low latency:** As mentioned earlier, replicating copies of data to locations nearer to users drastically reduces latency; it means that users can access their data in milliseconds. Cosmos DB guarantees less than 10 ms of latency for both reads and writes all around the world.
- **TCO Savings:** As Cosmos DB is a fully managed service, the level of management required from the customer is low. Also, the customer doesn't have to set up datacenters across the globe to accommodate users from other regions.
- **SLA:** It offers an SLA of 99.999% high availability.
- **Support for Open-Source Software (OSS) APIs:** The support for OSS APIs is another added advantage of Cosmos DB. Cosmos DB implements APIs for Cassandra, Mongo DB, Gremlin, and Azure Table storage.

Use case scenarios

If your application involves high levels of data reads and writes at a global scale, then Cosmos DB is the ideal choice. The common types of applications that have such requirements include web, mobile, gaming, and Internet of Things applications. These applications would benefit from the high availability, low latency, and global presence of Cosmos DB.

Also, the response time provided by Cosmos DB is near real time. The Cosmos DB SDKs can be leveraged to develop iOS and Android applications using the Xamarin framework.

A couple of the popular games that use Cosmos DB are **The Walking Dead: No Man's Land**, by Next Games, and **Halo 5: Guardians**.

A complete list of use case scenarios and examples can be found here: <https://docs.microsoft.com/azure/cosmos-db/use-cases>.

Cosmos DB is the go-to service in Azure for storing semi-structured data as part of OLTP applications. I could write a whole book on the features and capabilities of Cosmos DB alone; the intention of this section was to give you an introduction to Cosmos DB and the role it plays in handling OLTP applications.

Summary

In this chapter, you learned that Azure SQL Database is one of the flagship services of Azure. A plethora of customers are using this service today and it provides all the enterprise capabilities that are needed for a mission-critical database management system.

You discovered that there are multiple deployment types for Azure SQL Database, such as Single Instance, Managed Instance, and elastic pools. Architects should perform a complete assessment of their requirements and choose the appropriate deployment model. After they choose a deployment model, they should choose a pricing strategy between DTUs and vCPUs. They should also configure all the security, availability, disaster recovery, monitoring, performance, and scalability requirements in Azure SQL Database regarding data.

In the next chapter, we will be discussing how to build secure applications in Azure. We will cover the security practices and features of most services.

8

Architecting secure applications on Azure

In the previous chapter, we discussed Azure data services. As we are dealing with sensitive data, security is a big concern. Security is, undoubtedly, the most important non-functional requirement for architects to implement. Enterprises put lots of emphasis on having their security strategy implemented correctly. In fact, security is one of the top concerns for almost every stakeholder in an application's development, deployment, and management. It becomes all the more important when an application is built for deployment to the cloud.

In order for you to understand how you can secure your applications on Azure depending upon the nature of the deployment, the following topics will be covered in this chapter:

- Understanding security in Azure
- Security at the infrastructure level
- Security at the application level
- Authentication and authorization in Azure applications
- Working with OAuth, Azure Active Directory, and other authentication methods using federated identity, including third-party identity providers such as Facebook
- Understanding managed identities and using them to access resources

Security

As mentioned before, security is an important element for any software or service. Adequate security should be implemented so that an application can only be used by people who are allowed to access it, and users should not be able to perform operations that they are not allowed to execute. Similarly, the entire request-response mechanism should be built using methods that ensure that only intended parties can understand messages, and to make sure that it is easy to detect whether messages have been tampered with or not.

For the following reasons, security in Azure is even more important. Firstly, the organizations deploying their applications are not in full control of the underlying hardware and networks. Secondly, security has to be built into every layer, including hardware, networks, operating systems, platforms, and applications. Any omissions or misconfigurations can render an application vulnerable to intruders. For example, you might have heard of the recent vulnerability that affected Zoom meetings that let hackers record meetings even when the meeting host had disabled recording for attendees. Sources claim that millions of Zoom accounts have been sold on the dark web. The company has taken the necessary action to address this vulnerability.

Security is a big concern nowadays, especially when hosting applications in the cloud, and can lead to dire consequences if mishandled. Hence, it's necessary to understand the best practices involved in securing your workloads. We are progressing in the area of DevOps, where development and operations teams collaborate effectively with the help of tools and practices, and security has been a big concern there as well.

To accommodate security principles and practices as a vital part of DevOps without affecting the overall productivity and efficiency of the process, a new culture known as **DevSecOps** has been introduced. DevSecOps helps us to identify security issues early in the development stage rather than mitigating them after shipping. In a development process that has security as a key principle of every stage, DevSecOps reduces the cost of hiring security professionals at a later stage to find security flaws with software.

Securing an application means that unknown and unauthorized entities are unable to access it. This also means that communication with the application is secure and not tampered with. This includes the following security measures:

- **Authentication:** Authentication checks the identity of a user and ensures that the given identity can access the application or service. Authentication is performed in Azure using OpenID Connect, which is an authentication protocol built on OAuth 2.0.
- **Authorization:** Authorization allows and establishes permissions that an identity can perform within the application or service. Authorization is performed in Azure using OAuth.
- **Confidentiality:** Confidentiality ensures that communication between the user and the application remains secure. The payload exchange between entities is encrypted so that it will make sense only to the sender and the receiver, but not to others. The confidentiality of messages is ensured using symmetric and asymmetric encryption. Certificates are used to implement cryptography—that is, the encryption and decryption of messages.

Symmetric encryption uses a single key, which is shared with the sender and the receiver, while asymmetric encryption uses a pair of private and public keys for encryption, which is more secure. SSH key pairs in Linux, which are used for authentication, is a very good example of asymmetric encryption.

- **Integrity:** Integrity ensures that the payload and message exchange between the sender and the receiver is not tampered with. The receiver receives the same message that was sent by the sender. Digital signatures and hashes are the implementation mechanisms to check the integrity of incoming messages.

Security is a partnership between the service provider and the service consumer. Both parties have different levels of control over deployment stacks, and each should implement security best practices to ensure that all threats are identified and mitigated. We already know from *Chapter 1, Getting started with Azure*, that the cloud broadly provides three paradigms—IaaS, PaaS, and SaaS—and each of these has different levels of collaborative control over the deployment stack. Each party should implement security practices for the components under its control and within its scope. Failure to implement security at any layer in the stack or by any party would make the entire deployment and application vulnerable to attack. Every organization needs to have a life cycle model for security, just as for any other process. This ensures that security practices are continuously improved to avoid any security flaws. In the next section, we'll be discussing the security life cycle and how it can be used.

Security life cycle

Security is often regarded as a non-functional requirement for a solution. However, with the growing number of cyberattacks at the moment, nowadays it is considered a functional requirement of every solution.

Every organization follows some sort of application life cycle management for their applications. When security is treated as a functional requirement, it should follow the same process of application development. Security should not be an afterthought; it should be part of the application from the beginning. Within the overall planning phase for an application, security should also be planned. Depending on the nature of the application, different kinds and categories of threats should be identified, and, based on these identifications, they should be documented in terms of scope and approach to mitigate them. A threat modeling exercise should be undertaken to illustrate the threat each component could be subject to. This will lead to designing security standards and policies for the application. This is typically the security design phase. The next phase is called the threat mitigation or build phase. In this phase, the implementation of security in terms of code and configuration is executed to mitigate security threats and risks.

A system cannot be secure until it is tested. Appropriate penetration tests and other security tests should be performed to identify potential threat mitigation that has not been implemented or has been overlooked. The bugs from testing are remediated and the cycle continues throughout the life of the application. This process of application life cycle management, shown in *Figure 8.1*, should be followed for security:

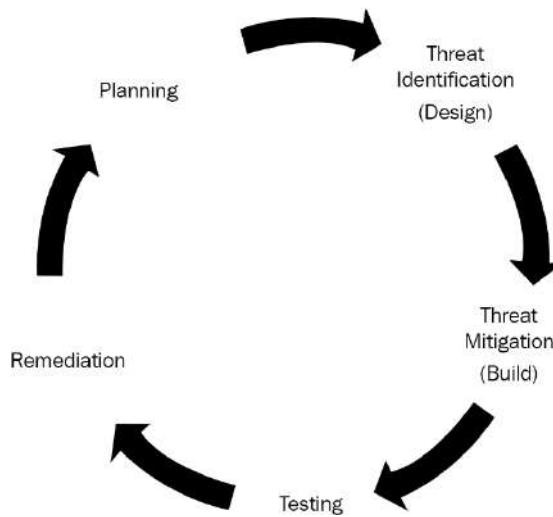


Figure 8.1: Security life cycle

Planning, threat modeling, identification, mitigation, testing, and remediation are iterative processes that continue even when an application or service is operational. There should be active monitoring of entire environments and applications to proactively identify threats and mitigate them. Monitoring should also enable alerts and audit logs to help in reactive diagnosis, troubleshooting, and the elimination of threats and vulnerabilities.

The security life cycle of any application starts with the planning phase, which eventually leads to the design phase. In the design phase, the application's architecture is decomposed into granular components with discrete communication and hosting boundaries. Threats are identified based on their interaction with other components within and across hosting boundaries. Within the overall architecture, threats are mitigated by implementing appropriate security features, and once the mitigation is in place, further testing is done to verify whether the threat still exists. After the application is deployed to production and becomes operational, it is monitored for any security breaches and vulnerabilities, and either proactive or reactive remediation is conducted.

As mentioned earlier, different organizations have different processes and methods to implement the security life cycle; likewise, Microsoft provides complete guidance and information about the security life cycle, which is available at <https://www.microsoft.com/securityengineering/sdl/practices>. Using the practices that Microsoft has shared, every organization can focus on building more secure solutions. As we are progressing in the era of cloud computing and migrating our corporate and customer data to the cloud, learning how to secure that data is vital. In the next section, we will explore Azure security and the different levels of security, which will help us to build secure solutions in Azure.

Azure security

Azure provides all its services through datacenters in multiple Azure regions. These datacenters are interconnected within regions, as well as across regions. Azure understands that it hosts mission-critical applications, services, and data for its customers. It must ensure that security is of the utmost importance for its datacenters and regions.

Customers deploy applications to the cloud based on their belief that Azure will protect their applications and data from vulnerabilities and breaches. Customers will not move to the cloud if this trust is broken, and so Azure implements security at all layers, as seen in Figure 8.2, from the physical perimeter of datacenters to logical software components. Each layer is protected, and even the Azure datacenter team does not have access to them:

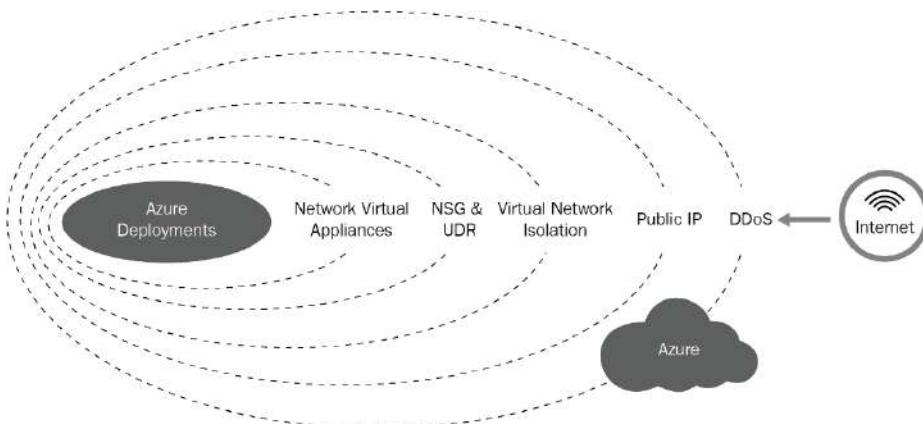


Figure 8.2: Security features at different layers in Azure datacenters

Security is of paramount importance to both Microsoft and Azure. Microsoft ensures that trust is built with its customers, and it does so by ensuring that its customers' deployments, solutions, and data are completely secure, both physically and virtually. People will not use a cloud platform if it is not physically and digitally secure.

To ensure that customers have trust in Azure, each activity in the development of Azure is planned, documented, audited, and monitored from a security perspective. The physical Azure datacenters are protected from intrusion and unauthorized access. In fact, even Microsoft personnel and operations teams do not have access to customer solutions and data. Some of the out-of-the-box security features provided by Azure are listed here:

- **Secure user access:** A customer's deployment, solution, and data can only be accessed by the customer. Even Azure datacenter personnel do not have access to customer artifacts. Customers can allow access to other people; however, that is at the discretion of the customer.

- **Encryption at rest:** Azure encrypts all its management data, which includes a variety of enterprise-grade storage solutions to accommodate different needs. Microsoft also provides encryption to managed services such as Azure SQL Database, Azure Cosmos DB, and Azure Data Lake Storage as well. Since the data is encrypted at rest, it cannot be read by anyone. It also provides this functionality to its customers, as well as those who can encrypt their data at rest.
- **Encryption at transit:** Azure encrypts all data that flows from its network. It also ensures that its network backbone is protected from any unauthorized access.
- **Active monitoring and auditing:** Azure monitors all its datacenters actively on an ongoing basis. It actively identifies any breach, threat, or risk, and mitigates them.

Azure meets country-specific, local, international, and industry-specific compliance standards. You can explore the complete list of Microsoft compliance offerings at <https://www.microsoft.com/trustcenter/compliance/complianceofferings>. Keep this as a reference while deploying compliant solutions in Azure. Now that we know the key security features in Azure, let's go ahead and take a deep dive into IaaS security. In the next section, we will explore how customers can leverage the security features available for IaaS in Azure.

IaaS security

Azure is a mature platform for deploying IaaS solutions. There are lots of users of Azure who want complete control over their deployments, and they typically use IaaS for their solutions. It is important that these deployments and solutions are secure, by default and by design. Azure provides rich security features to secure IaaS solutions. In this section, some of the main features will be covered.

Network security groups

The bare minimum of IaaS deployment consists of virtual machines and virtual networks. A virtual machine might be exposed to the internet by applying a public IP to its network interface, or it might only be available to internal resources. Some of those internal resources might, in turn, be exposed to the internet. In any case, virtual machines should be secured so that unauthorized requests should not even reach them. Virtual machines should be secured using facilities that can filter requests on the network itself, rather than the requests reaching a virtual machine and it having to take action on them.

Ring-fencing is a mechanism that virtual machines use as one of their security mechanisms. This fence can allow or deny requests depending on their protocol, origin IP, destination IP, originating port, and destination port. This feature is deployed using the Azure **network security groups (NSGs)** resource. NSGs are composed of rules that are evaluated for both incoming and outgoing requests. Depending on the execution and evaluation of these rules, it is determined whether the requests should be allowed or denied access.

NSGs are flexible and can be applied to a virtual network subnet or individual network interfaces. When applied to a subnet, the security rules are applied to all virtual machines hosted on the subnet. On the other hand, applying to a network interface affects requests to only a particular virtual machine associated with that network interface. It is also possible to apply NSGs to both network subnets and network interfaces simultaneously. Typically, this design should be used to apply common security rules at the network subnet level, and unique security rules at the network interface level. It helps to design modular security rules.

The flow for evaluating NSGs is shown in *Figure 8.3*:

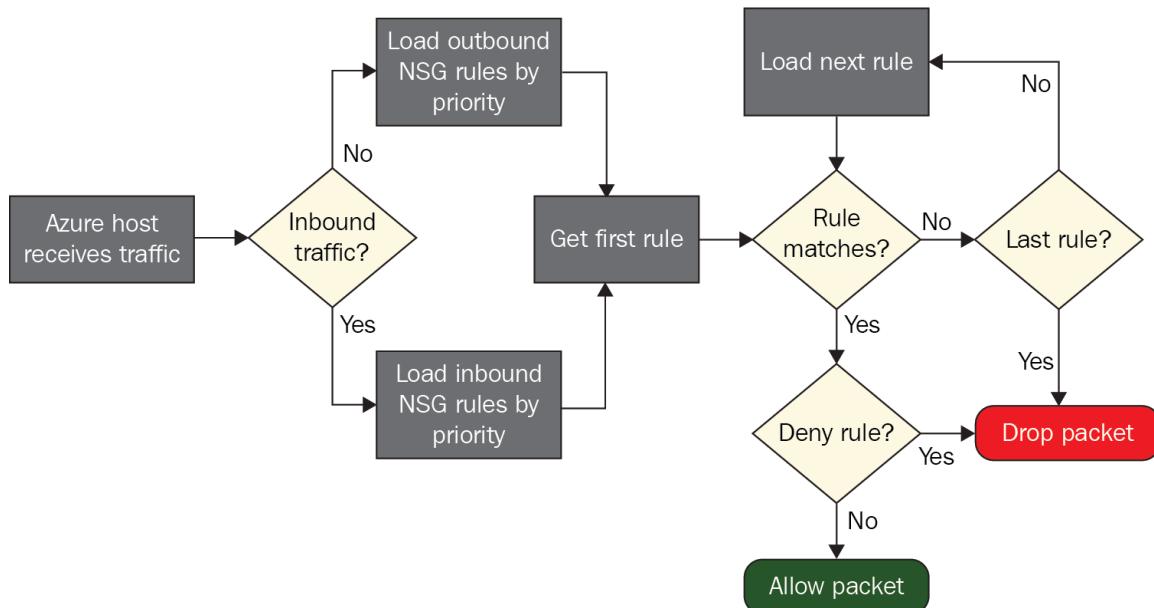


Figure 8.3: A flow diagram representing the evaluation of NSGs

When a request reaches an Azure host, depending on whether it's an inbound or outbound request, appropriate rules are loaded and executed against the request/response. If the rule matches the request/response, either the request/response is allowed or denied. The rule matching consists of important request/response information, such as the source IP address, destination IP address, source port, destination port, and protocol used. Additionally, NSGs support service tags. A service tag denotes a group of IP address prefixes from a given Azure service. Microsoft manages the address prefixes and automatically updates them. This eliminates the hassle of updating the security rules every time there is an address prefix change.

The set of service tags available for use is available at <https://docs.microsoft.com/azure/virtual-network/service-tags-overview#available-service-tags>. Service tags can be used with NSGs as well as with Azure Firewall. Now that you have learned about how NSGs work, let's take a look at the NSG design, which will help you determine the primary points you should consider while creating NSG rules to improve security.

NSG design

The first step in designing an NSG is to ascertain the security requirements of the resource. The following should be determined or considered:

- Is the resource accessible from the internet only?
- Is the resource accessible from both internal resources and the internet?
- Is the resource accessible from internal resources only?
- Based on the architecture of the solution being deployed, determine the dependent resources, load balancers, gateways, and virtual machines used.
- Configure a virtual network and its subnet.

Using the results of these investigations, an adequate NSG design should be created. Ideally, there should be multiple network subnets for each workload and type of resource. It is not recommended to deploy both load balancers and virtual machines on the same subnet.

Taking project requirements into account, rules should be determined that are common for different virtual machine workloads and subnets. For example, for a SharePoint deployment, the front-end application and SQL servers are deployed on separate subnets, so rules for each subnet should be determined.

After common subnet-level rules are identified, rules for individual resources should be identified, and these should be applied at the network interface level. It is important to understand that if a rule allows an incoming request on a port, that port can also be used for outgoing requests without any configuration.

If resources are accessible from the internet, rules should be created with specific IP ranges and ports wherever possible, instead of allowing traffic from all the IP ranges (usually represented as 0.0.0.0/0). Careful functional and security testing should be executed to ensure that adequate and optimal NSG rules are opened and closed.

Firewalls

NSGs provide external security perimeters for requests. However, this does not mean that virtual machines should not implement additional security measures. It is always better to implement security both internally and externally. Virtual machines, whether in Linux or Windows, provide a mechanism to filter requests at the operating system level. This is known as a firewall in both Windows and Linux.

It is advisable to implement firewalls for operating systems. They help build a virtual security wall that allows only those requests that are considered trusted. Any untrusted requests are denied access. There are even physical firewall devices, but on the cloud, operating system firewalls are used. Figure 8.4 shows firewall configuration for a Windows operating system:



Figure 8.4: Firewall configuration

Firewalls filter network packets and identify incoming ports and IP addresses. Using the information from these packets, the firewall evaluates the rules and decides whether it should allow or deny access.

When it comes to Linux, there are different firewall solutions available. Some of the firewall offerings are very specific to the distribution that is being used; for example, SUSE uses SuSEfirewall2 and Ubuntu uses ufw. The most widely used implementations are firewalld and iptables, which are available on every distribution.

Firewall design

As a best practice, firewalls should be evaluated for individual operating systems. Each virtual machine has a distinct responsibility within the overall deployment and solution. Rules for these individual responsibilities should be identified and firewalls should be opened and closed accordingly.

While evaluating firewall rules, it is important to take NSG rules at both the subnet and individual network interface level into consideration. If this is not done properly, it is possible that rules are denied at the NSG level, but left open at the firewall level, and vice versa. If a request is allowed at the NSG level and denied at the firewall level, the application will not work as intended, while security risks increase if a request is denied at the NSG level and allowed at the firewall level.

A firewall helps you build multiple networks isolated by its security rules. Careful functional and security testing should be executed to ensure that adequate and optimal firewall rules are opened and closed.

It makes the most sense to use Azure Firewall, which is a cloud-based network service on top of NSGs. It is very easy to set up, provides central management for administration, and requires zero maintenance. Azure Firewall and NSGs combined can provide security between virtual machines, virtual networks, and even different Azure subscriptions. Having said that, if a solution requires that extra level of security, we can consider implementing an operating system-level firewall. We'll be discussing Azure Firewall in more depth in one of the upcoming sections, *Azure Firewall*.

Application security groups

NSGs are applied at the virtual network subnet level or directly to individual network interfaces. While it is sufficient to apply NSGs at the subnet level, there are times when this is not enough. There are different types of workloads available within a single subnet and each of them requires a different security group. It is possible to assign security groups to individual **network interface cards (NICs)** of the virtual machines, but it can easily become a maintenance nightmare if there is a large number of virtual machines.

Azure has a relatively new feature known as application security groups. We can create application security groups and assign them directly to multiple NICs, even when those NICs belong to virtual machines in different subnets and resource groups. The functionality of application security groups is similar to NSGs, except that they provide an alternate way of assigning groups to network resources, providing additional flexibility in assigning them across resource groups and subnets. Application security groups can simplify NSGs; however, there is one main limitation. We can have one application security group in the source and destination of a security rule, but having multiple application security groups in a source or destination is not supported right now.

One of the best practices for creating rules is to always minimize the number of security rules that you need, to avoid maintenance of explicit rules. In the previous section, we discussed the usage of service tags with NSGs to eliminate the hassle of maintaining the individual IP address prefixes of each service. Likewise, when using application security groups, we can reduce the complexity of explicit IP addresses and multiple rules. This practice is recommended wherever possible. If your solution demands an explicit rule with an individual IP address or range of IP addresses, only then should you opt for it.

Azure Firewall

In the previous section, we discussed using Azure Firewall within a Windows/Linux operating system to allow or disallow requests and responses through particular ports and services. While operating system firewalls play an important role from a security point of view and must be implemented for any enterprise deployment, Azure provides a security resource known as Azure Firewall that has a similar functionality of filtering requests based on rules and determining whether a request should be allowed or rejected.

The advantage of using Azure Firewall is that it evaluates a request before it reaches an operating system. Azure Firewall is a network resource and is a standalone service protecting resources at the virtual network level. Any resources, including virtual machines and load balancers, that are directly associated with a virtual network can be protected using Azure Firewall.

Azure Firewall is a highly available and scalable service that can protect not only HTTP-based requests but any kind of request coming into and going out from a virtual network, including FTP, SSH, and RDP. Azure Firewall can also span multiple Availability Zones during deployment to provide increased availability.

It is highly recommended that Azure Firewall is deployed for mission-critical workloads on Azure, alongside other security measures. It is also important to note that Azure Firewall should be used even if other services, such as Azure Application Gateway and Azure Front Door, are used, since all these tools have different scopes and features. Additionally, Azure Firewall provides support for service tags and threat intelligence. In the previous section, we discussed the advantages of using service tags. Threat intelligence can be used to generate alerts when traffic comes from or goes to known malicious IP addresses and domains, which are recorded in the Microsoft Threat Intelligence feed.

Reducing the attack surface area

NSGs and firewalls help with managing authorized requests to the environment. However, the environment should not be overly exposed to attacks. The surface area of the system should be optimally enabled to achieve its functionality, but disabled enough that attackers cannot find loopholes and access areas that are open without any intended use, or open but not adequately secured. Security should be adequately hardened, making it difficult for any attacker to break into the system.

Some of the configurations that should be done include the following:

- Remove all unnecessary users and groups from the operating system.
- Identify group membership for all users.
- Implement group policies using directory services.
- Block script execution unless it is signed by trusted authorities.
- Log and audit all activities.
- Install malware and anti-virus software, schedule scans, and update definitions frequently.
- Disable or shut down services that are not required.
- Lock down the filesystem so only authorized access is allowed.
- Lock down changes to the registry.
- A firewall must be configured according to the requirements.
- PowerShell script execution should be set to **Restricted** or **RemoteSigned**. This can be done using the `Set-ExecutionPolicy -ExecutionPolicy Restricted` or `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned` PowerShell commands.
- Enable enhanced protection through Internet Explorer.
- Restrict the ability to create new users and groups.

- Remove internet access and implement jump servers for RDP.
- Prohibit logging into servers using RDP through the internet. Instead, use site-to-site VPN, point-to-site VPN, or express routes to RDP into remote machines from within the network.
- Regularly deploy all security updates.
- Run the security compliance manager tool on the environment and implement all of its recommendations.
- Actively monitor the environment using Security Center and Operations Management Suite.
- Deploy virtual network appliances to route traffic to internal proxies and reverse proxies.
- All sensitive data, such as configuration, connection strings, and credentials, should be encrypted.

The aforementioned are some of the key points that should be considered from a security standpoint. The list will keep on growing, and we need to constantly improve security to prevent any kind of security breach.

Implementing jump servers

It is a good idea to remove internet access from virtual machines. It is also a good practice to limit remote desktop services' accessibility from the internet, but then how do you access the virtual machines at all? One good way is to only allow internal resources to RDP into virtual machines using Azure VPN options. However, there is also another way—using **jump servers**.

Jump servers are servers that are deployed in the **demilitarized zone (DMZ)**. This means it is not on the network hosting the core solutions and applications. Instead, it is on a separate network or subnet. The primary purpose of the jump server is to accept RDP requests from users and help them log in to it. From this jump server, users can further navigate to other virtual machines using RDP. It has access to two or more networks: one that has connectivity to the outside world, and another that's internal to the solution. The jump server implements all the security restrictions and provides a secure client to connect to other servers. Normally, access to emails and the internet is disabled on jump servers.

An example of deploying a jump server with **virtual machine scale sets (VMSSes)**, using Azure Resource Manager templates is available at <https://azure.microsoft.com/resources/templates/201-vmss-windows-jumpbox>.

Azure Bastion

In the previous section, we discussed implementing jump servers. Azure Bastion is a fully managed service that can be provisioned in a virtual network to provide RDP/SSH access to your virtual machines directly in the Azure portal over TLS. The Bastion host will act as a jump server and eliminate the need for public IP addresses for your virtual machines. The concept of using Bastion is the same as implementing a jump server; however, since this is a managed service, it's completely managed by Azure.

Since Bastion is a fully managed service from Azure and is hardened internally, we don't need to apply additional NSGs on the Bastion subnet. Also, since we are not attaching any public IPs to our virtual machines, they are protected against port scanning.

Application security

Web applications can be hosted within IaaS-based solutions on top of virtual machines, and they can be hosted within Azure-provided managed services, such as App Service. App Service is part of the PaaS deployment paradigm, and we will look into it in the next section. In this section, we will look at application-level security.

SSL/TLS

Secure Socket layer (SSL) is now deprecated and has been replaced by **Transport Layer security (TLS)**. TLS provides end-to-end security by means of cryptography. It provides two types of cryptography:

- Symmetric: The same key is available to both the sender of the message and the receiver of the message, and it is used for both the encryption and decryption of the message.
- Asymmetric: Every stakeholder has two keys—a private key and a public key. The private key remains on the server or with the user and remains a secret, while the public key is distributed freely to everyone. Holders of the public key use it to encrypt the message, which can only be decrypted by the corresponding private key. Since the private key stays with the owner, only they can decrypt the message. **Rivest-Shamir-Adleman (RSA)** is one of the algorithms used to generate these pairs of public-private keys.
- The keys are also available in certificates popularly known as X.509 certificates, although certificates have more details apart from just the keys and are generally issued by trusted certificate authorities.

TLS should be used by web applications to ensure that message exchange between users and the server is secure and confidential and that identities are being protected. These certificates should be purchased from a trusted certificate authority instead of being self-signed certificates.

Managed identities

Before we take a look at managed identities, it is important to know how applications were built without them.

The traditional way of application development is to use secrets, such as a username, a password, or SQL connection strings, in configuration files. Putting these secrets into configuration files makes application changes to these secrets easy and flexible without modifying code. It helps us stick to the "open for extension, closed for modification" principle. However, this approach has a downside from a security point of view. The secrets can be viewed by anyone who has access to configuration files since generally these secrets are listed there in plain text. There are a few hacks to encrypt them, but they aren't foolproof.

A better way to use secrets and credentials within an application is to store them in a secrets repository such as Azure Key Vault. Azure Key Vault provides full security using the **hardware security module (HSM)**, and the secrets are stored in an encrypted fashion with on-demand decryption using keys stored in separate hardware. Secrets can be stored in Key Vault, with each secret having a display name and key. The key is in the form of a URI that can be used to refer to the secret from applications, as shown in *Figure 8.5*:

The screenshot shows the Azure Key Vault interface for managing secrets. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events (preview), Settings (with Keys and Secrets selected), and Help & support. The main content area displays a table of secrets:

Name	Type	Status
storageKey		✓ Enabled
storageKey1		✓ Enabled
storageKey2		✓ Enabled

Figure 8.5: Storing secrets inside a key vault

Within application configuration files, we can refer to the secret using the name or the key. However, there is another challenge now. How does the application connect to and authenticate with the key vault?

Key vaults have access policies that define permissions to a user or group with regard to access to secrets and credentials within the key vault. The users, groups, or service applications that can be provided access are provisioned and hosted within **Azure Active Directory (Azure AD)**. Although individual user accounts can be provided access using Key Vault access policies, it is a better practice to use a service principal to access the key vault. A service principal has an identifier, also known as an application ID or client ID, along with a password. The client ID, along with its password, can be used to authenticate with Azure Key Vault. This service principal can be allowed to access the secrets. The access policies for Azure Key Vault are granted in the **Access policies** pane of your key vault. In Figure 8.6, you can see that the service principal—<https://keyvaultbook.com>—has given access to the key vault called **keyvaultbook**:

Name	Email	Key Permissions	Secret Permissions	Certificate Permissions	Action
APPLICATION					
https://keyvaultbook.com		3 selected	3 selected	4 selected	Delete
http://automationoncrtcred2		3 selected	3 selected	4 selected	Delete
USER					
Ritesh Modi	callritz_hotmail.com#...	9 selected	7 selected	15 selected	Delete

Figure 8.6: Granted access for a service principal to access a key vault

This brings us to another challenge: to access the key vault, we need to use the client ID and secret in our configuration files to connect to the key vault, get hold of the secret, and retrieve its value. This is almost equivalent to using a username, password, and SQL connection string within configuration files.

This is where managed identities can help. Azure launched managed service identities and later renamed them managed identities. Managed identities are identities managed by Azure. In the background, managed identities also create a service principal along with a password. With managed identities, there is no need to put credentials in configuration files.

Managed identities can only be used to authenticate with services that support Azure AD as an identity provider. Managed identities are meant only for authentication. If the target service does not provide **role-based access control (RBAC)** permission to the identity, the identity might not be able to perform its intended activity on the target service.

Managed identities come in two flavors:

- System-assigned managed identities
- User-assigned managed identities

System-assigned identities are generated by the service itself. For example, if an app service wants to connect to Azure SQL Database, it can generate the system-assigned managed identity as part of its configuration options. These managed identities also get deleted when the parent resource or service is deleted. As shown in Figure 8.7, a system-assigned identity can be used by App Service to connect to Azure SQL Database:

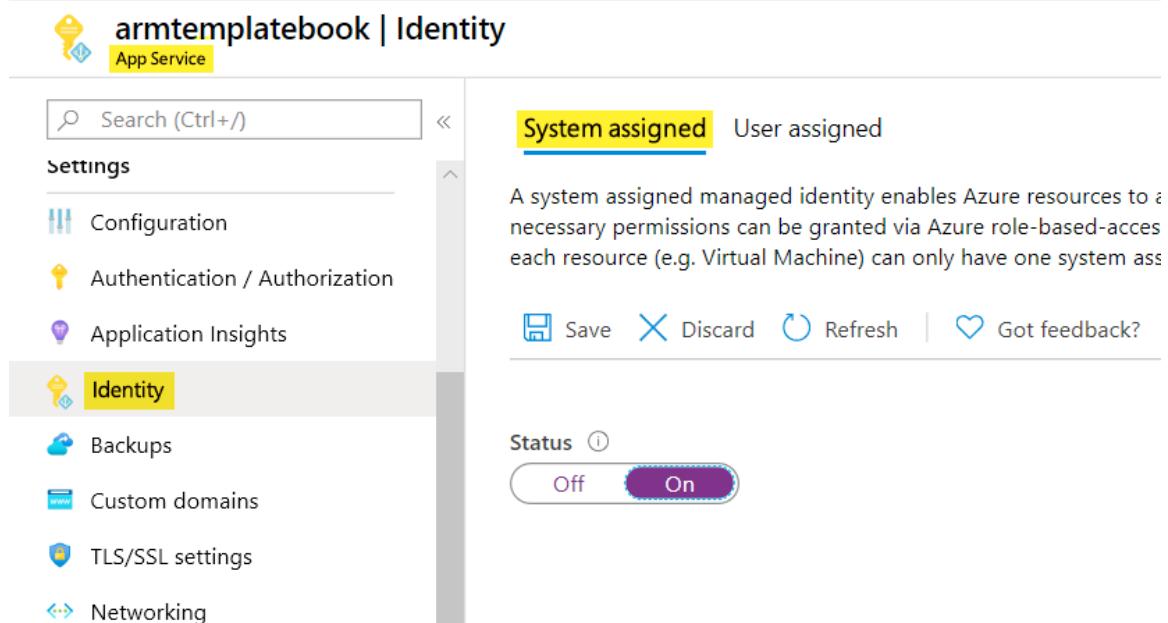


Figure 8.7: Enabling a system-assigned managed identity for App Service

User-assigned managed identities are created as standalone separate identities and later assigned to Azure services. They can be applied and reused with multiple Azure services since their life cycles do not depend on the resource they are assigned to.

Once a managed identity is created and RBAC or access permissions are given to it on the target resource, it can be used within applications to access the target resources and services.

Azure provides an SDK as well a REST API to talk to Azure AD and get an access token for managed identities, and then use the token to access and consume the target resources.

The SDK comes as part of the **Microsoft.Azure.Services.AppAuthentication** NuGet package for C#. Once the access token is available, it can be used to consume the target resource.

The code needed to get the access token is as follows:

```
var tokenProvider = new AzureServiceTokenProvider();  
  
string token = await tokenProvider.GetAccessTokenAsync("https://vault.azure.net");
```

Alternatively, use this:

```
string token = await tokenProvider.GetAccessTokenAsync("https://database.windows.net");
```

It should be noted that the application code needs to run in the context of App Service or a function app because the identity is attached to them and is only available in code when it's run from within them.

The preceding code has two different use cases. The code to access the key vault and Azure SQL Database is shown together.

It is important to note that applications do not provide any information related to managed identities in code and is completely managed using configuration. The developers, individual application administrators, and operators will not come across any credentials related to managed identities, and, moreover, there is no mention of them in code either. Credential rotation is completely regulated by the resource provider that hosts the Azure service. The default rotation occurs every 46 days. It's up to the resource provider to call for new credentials if required, so the provider could wait for more than 46 days.

In the next section, we will be discussing a cloud-native **security information and event manager (SIEM)**: Azure Sentinel.

Azure Sentinel

Azure provides an SIEM and **security orchestration automated response (SOAR)** as a standalone service that can be integrated with any custom deployment on Azure. Figure 8.8 shows some of the key features of Azure Sentinel:

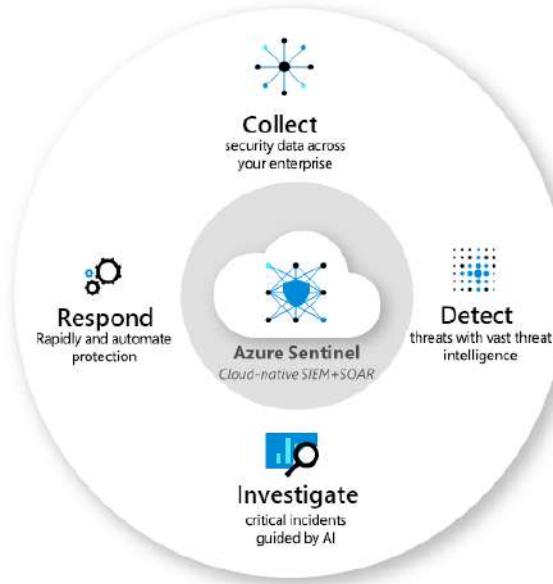


Figure 8.8: Key features of Azure Sentinel

Azure Sentinel collects information logs from deployments and resources and performs analytics to find patterns and trends related to various security issues that are pulled from data sources.

There should be active monitoring of the environment, logs should be collected, and information should be culled from these logs as a separate activity from code implementation. This is where the SIEM service comes into the picture. There are numerous connectors that can be used with Azure Sentinel; each of these connectors will be used to add data sources to Azure Sentinel. Azure Sentinel provides connectors for Microsoft services such as Office 365, Azure AD, and Azure Threat Protection. The collected data will be fed to a Log Analytics workspace, and you can write queries to search these logs.

SIEM tools such as Azure Sentinel can be enabled on Azure to get all the logs from log analytics and Azure Security Center, which in turn can get them from multiple sources, deployments, and services. SIEM can then run its intelligence on top of this collected data and generate insights. It can generate reports and dashboards based on discovered intelligence for consumption, but it can also investigate suspicious activities and threats, and take action on them.

While Azure Sentinel may sound very similar in functionality to Azure Security Center, Azure Sentinel can do much more than Azure Security Center. Its ability to collect logs from other avenues using connectors makes it different from Azure Security Center.

PaaS security

Azure provides numerous PaaS services, each with its own security features. In general, PaaS services can be accessed using credentials, certificates, and tokens. PaaS services allow the generation of short-lived security access tokens. Client applications can send these security access tokens to represent trusted users. In this section, we will cover some of the most important PaaS services that are used in almost every solution.

Azure Private Link

Azure Private Link provides access to Azure PaaS services as well as Azure-hosted customer-owned/partner-shared services over a private endpoint in your virtual network. While using Azure Private Link, we don't have to expose our services to the public internet, and all traffic between our service and the virtual network goes via Microsoft's backbone network.

Azure Private Endpoint is the network interface that helps to privately and securely connect to a service that is powered by Azure Private Link. Since the private endpoint is mapped to the instance of the PaaS service, not to the entire service, users can only connect to the resource. Connections to any other service are denied, and this protects against data leakage. Private Endpoint also lets you access securely from on-premises via ExpressRoute or VPN Tunnels. This eliminates the need to set up public peering or to pass through the public internet to reach the service.

Azure Application Gateway

Azure provides a Level 7 load balancer known as Azure Application Gateway that can not only load balance but also help in routing using values in URL. It also has a feature known as Web Application Firewall. Azure Application Gateway supports TLS termination at the gateway, so the back-end servers will get the traffic unencrypted. This has several advantages, such as better performance, better utilization of the back-end servers, and intelligent routing of packets. In the previous section, we discussed Azure Firewall and how it protects resources at the network level. Web Application Firewall, on the other hand, protects the deployment at the application level.

Any deployed application that is exposed to the internet faces numerous security challenges. Some of the important security threats are as follows:

- Cross-site scripting
- Remote code execution
- SQL injection
- **Denial of Service (DoS)** attacks
- **Distributed Denial of Service (DDoS)** attacks

There are many more, though.

A large number of these attacks can be addressed by developers by writing defensive code and following best practices; however, it is not just the code that should be responsible for identifying these issues on a live site. Web Application Firewall configures rules that can identify such issues, as mentioned before, and deny requests.

It is advised to use Application Gateway Web Application Firewall features to protect applications from live security threats. Web Application Firewall will either allow the request to pass through it or stop it, depending on how it's configured.

Azure Front Door

Azure has launched a relatively new service known as Azure Front Door. The role of Azure Front Door is quite similar to that of Azure Application Gateway; however, there is a difference in scope. While Application Gateway works within a single region, Azure Front Door works at the global level across regions and datacenters. It has a web application firewall as well that can be configured to protect applications deployed in multiple regions from various security threats, such as SQL injection, remote code execution, and cross-site scripting.

Application Gateway can be deployed behind Front Door to address connection draining. Also, deploying Application Gateway behind Front Door will help with the load balancing requirement, as Front Door can only perform path-based load balancing at the global level. The addition of Application Gateway to the architecture will provide further load balancing to the back-end servers in the virtual network.

Azure App Service Environment

Azure App Service is deployed on shared networks behind the scenes. All SKUs of App Service use a virtual network, which can potentially be used by other tenants as well. In order to have more control and a secure App Service deployment on Azure, services can be hosted on dedicated virtual networks. This can be accomplished by using **Azure App Service Environment (ASE)**, which provides complete isolation to run your App Service at a high scale. This also provides additional security by allowing you to deploy Azure Firewall, Application Security Groups, NSGs, Application Gateway, Web Application Firewall, and Azure Front Door. All App Service plans created in App Service Environment will be in an isolated pricing tier, and we cannot choose any other tier.

All the logs from this virtual network and compute can then be collated in Azure Log Analytics and Security Center, and finally with Azure Sentinel.

Azure Sentinel can then provide insights and execute workbooks and runbooks to respond to security threats in an automated way. Security playbooks can be run in Azure Sentinel in response to alerts. Every security playbook comprises measures that need to be taken in the event of an alert. The playbooks are based on Azure Logic Apps, and this will give you the freedom to use and customize the built-in templates available with Logic Apps.

Log Analytics

Log Analytics is a new analytics platform for managing cloud deployments, on-premises datacenters, and hybrid solutions.

It provides multiple modular solutions—a specific functionality that helps to implement a feature. For example, security and audit solutions help to ascertain a complete view of security for an organization's deployment. Similarly, there are many more solutions, such as automation and change tracking, that should be implemented from a security perspective. Log Analytics security and audit services provide information in the following five categories:

- **Security domains:** These provide the ability to view security records, malware assessments, update assessments, network security, identity and access information, and computers with security events. Access is also provided to the Azure Security Center dashboard.
- **Anti-malware assessment:** This helps to identify servers that are not protected against malware and have security issues. It provides information about exposure to potential security problems and assesses their criticality of any risk. Users can take proactive actions based on these recommendations. Azure Security Center sub-categories provide information collected by Azure Security Center.

- **Notable issues:** This quickly identifies active issues and grades their severity.
- **Detections:** This category is in preview mode. It enables the identification of attack patterns by visualizing security alerts.
- **Threat intelligence:** This helps to identify attack patterns by visualizing the total number of servers with outbound malicious IP traffic, the malicious threat type, and a map that shows where these IPs come from.

The preceding details, when viewed from the portal, are shown in *Figure 8.9*:

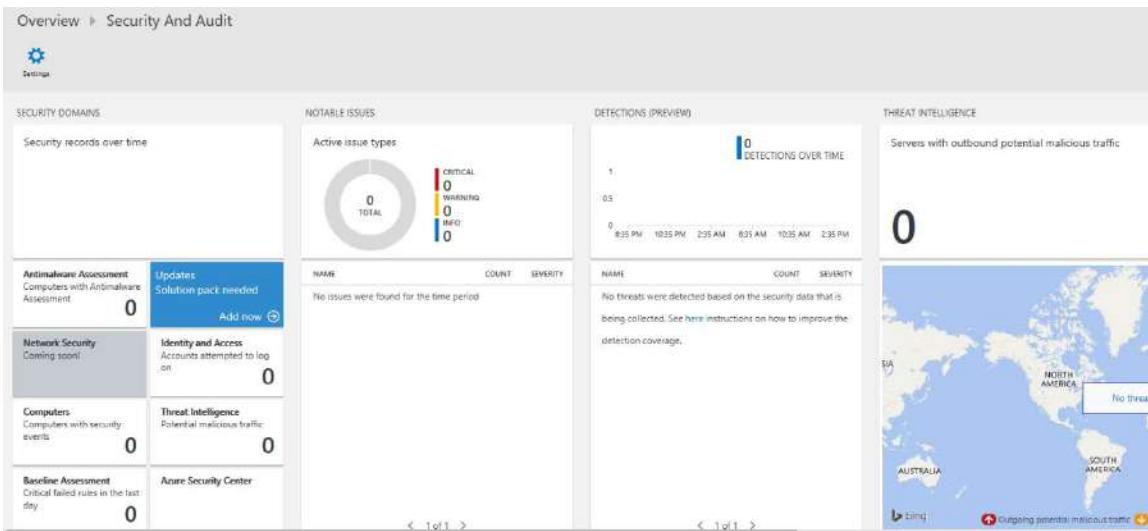


Figure 8.9: Information being displayed in the Security And Audit pane of Log Analytics

Now that you have learned about security for PaaS services, let's explore how to secure data stored in Azure Storage.

Azure Storage

Storage accounts play an important part in the overall solution architecture. Storage accounts can store important information, such as user **personal identifiable information (PII)** data, business transactions, and other sensitive and confidential data. It is of the utmost importance that storage accounts are secure and only allow access to authorized users. The stored data is encrypted and transmitted using secure channels. Storage, as well as the users and client applications consuming the storage account and its data, plays a crucial role in the overall security of data. Data should be kept encrypted at all times. This also includes credentials and connection strings connecting to data stores.

Azure provides RBAC to govern who can manage Azure storage accounts. These RBAC permissions are given to users and groups in Azure AD. However, when an application to be deployed on Azure is created, it will have users and customers that are not available in Azure AD. To allow users to access the storage account, Azure Storage provides storage access keys. There are two types of access keys at the storage account level—primary and secondary. Users possessing these keys can connect to the storage account. These storage access keys are used in the authentication step when accessing the storage account. Applications can access storage accounts using either primary or secondary keys. Two keys are provided so that if the primary key is compromised, applications can be updated to use the secondary key while the primary key is regenerated. This helps minimize application downtime. Moreover, it enhances security by removing the compromised key without impacting applications. The storage key details, as seen on the Azure portal, are shown in *Figure 8.10*:

The screenshot shows the 'Access keys' section of the Azure Storage account settings. It includes a note about using access keys for authentication and regenerating them regularly. The 'Storage account name' field is filled with a placeholder. The 'Default keys' table lists two entries:

NAME	KEY	CONNECTION STRING
key1	[REDACTED]	[REDACTED] DefaultEndpointsProtocol=https;AccountName= [REDACTED] [Download] [Regenerate]
key2	[REDACTED]	[REDACTED] DefaultEndpointsProtocol=https;AccountName= [REDACTED] [Download] [Regenerate]

Figure 8.10: Access keys for a storage account

Azure Storage provides four services—blob, files, queues, and tables—in an account. Each of these services also provides infrastructure for their own security using secure access tokens.

A **shared access signature (SAS)** is a URI that grants restricted access rights to Azure Storage services: blobs, files, queues, and tables. These SAS tokens can be shared with clients who should not be trusted with the entire storage account key to restrict access to certain storage account resources. By distributing an SAS URI to these clients, access to resources is granted for a specified period.

SAS tokens exist at both the storage account and the individual blob, file, table, and queue levels. A storage account-level signature is more powerful and has the right to allow and deny permissions at the individual service level. It can also be used instead of individual resource service levels.

SAS tokens provide granular access to resources and can be combined as well. These tokens include read, write, delete, list, add, create, update, and process. Moreover, even access to resources can be determined while generating SAS tokens. It could be for blobs, tables, queues, and files individually, or a combination of them. Storage account keys are for the entire account and cannot be constrained for individual services—neither can they be constrained from the permissions perspective. It is much easier to create and revoke SAS tokens than it is for storage account access keys. SAS tokens can be created for use for a certain period of time, after which they automatically become invalid.

It is to be noted that if storage account keys are regenerated, then the SAS token based on them will become invalid and a new SAS token should be created and shared with clients. In *Figure 8.11*, you can see an option to select the scope, permissions, start date, end date, allowed IP address, allowed protocols, and signing key to create an SAS token:

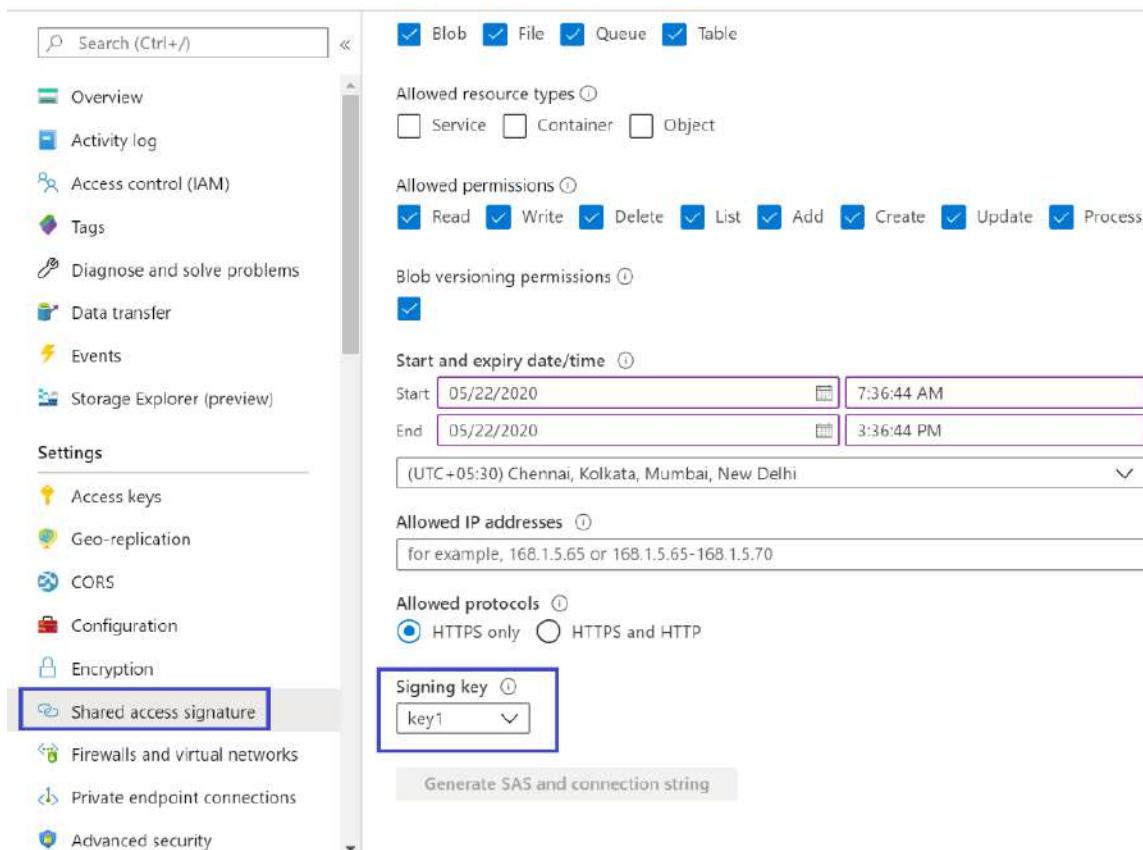


Figure 8.11: Creating an SAS token

If we are regenerating **key1**, which we used to sign the SAS token in the earlier example, then we need to create a new SAS token with **key2** or the new **key1**.

Cookie stealing, script injection, and DoS attacks are common means used by attackers to disrupt an environment and steal data. Browsers and the HTTP protocol implement a built-in mechanism that ensures that these malicious activities cannot be performed. Generally, anything that is cross-domain is not allowed by either HTTP or browsers. A script running in one domain cannot ask for resources from another domain. However, there are valid use cases where such requests should be allowed. The HTTP protocol implements **cross-origin resource sharing (CORS)**. With the help of CORS, it is possible to access resources across domains and make them work. Azure Storage configures CORS rules for blob, file, queue, and table resources. Azure Storage allows the creation of rules that are evaluated for each authenticated request. If the rules are satisfied, the request is allowed to access the resource. In Figure 8.12, you can see how to add CORS rules to each of the storage services:

The screenshot shows the Azure Storage account settings page. The left sidebar lists various settings: Search, Settings, Access keys, Geo-replication, **CORS** (which is selected), Configuration, Encryption, Shared access signature, Firewalls and virtual networks, Private endpoint connections, and Advanced security. The main pane is titled 'Save' and 'Discard'. It contains a note about setting CORS rules for storage services. Below this is a 'Learn more' link. The 'Blob service' tab is selected under the 'Service' section. The 'Allowed origins' table has one row: 'http://www.contoso...'. The 'Allowed methods' column shows 'GET'. The 'Allowed headers' column shows 'x-ms-meta-data*' and 'x-ms-meta-*'. The 'Exposed headers' column shows 'x-ms-meta-*'. The 'Max age' column shows '200'. There are edit and delete icons for each row.

Figure 8.12: Creating CORS rules for a storage account

Data must not only be protected while in transit; it should also be protected while at rest. If data at rest is not encrypted, anybody who has access to the physical drive in the datacenter can read the data. Although the possibility of a data breach is negligible, customers should still encrypt their data. Storage service encryption also helps protect data at rest. This service works transparently and injects itself without users knowing about it. It encrypts data when the data is saved in a storage account and decrypts it automatically when it is read. This entire process happens without users performing any additional activity.

Azure account keys must be rotated periodically. This will ensure that an attacker is not able to gain access to storage accounts.

It is also a good idea to regenerate the keys; however, this must be evaluated with regard to its usage in existing applications. If it breaks the existing application, these applications should be prioritized for change management, and changes should be applied gradually.

It is always recommended to have individual service-level SAS tokens with limited timeframes. This token should only be provided to users who should access the resources. Always follow the principle of least privilege and provide only the necessary permissions.

SAS keys and storage account keys should be stored in Azure Key Vault. This provides secure storage and access to them. These keys can be read at runtime by applications from the key vault, instead of storing them in configuration files.

Additionally, you can also use Azure AD to authorize the requests to the blob and queue storage. We'll be using RBAC to give necessary permissions to a service principal, and once we authenticate the service principal using Azure AD, an OAuth 2.0 token is generated. This token can be added to the authorization header of your API calls to authorize a request against blob or queue storage. Microsoft recommends the use of Azure AD authorization while working with blob and queue applications due to the superior security provided by Azure AD and its simplicity compared to SAS tokens.

In the next section, we are going to assess the security options available for Azure SQL Database.

Azure SQL

SQL Server stores relational data on Azure, which is a managed relational database service. It is also known as a **Database as a Service (DBaaS)** that provides a highly available, scalable, performance-centric, and secure platform for storing data. It is accessible from anywhere, with any programming language and platform. Clients need a connection string comprising the server, database, and security information to connect to it.

SQL Server provides firewall settings that prevent access to anyone by default. IP addresses and ranges should be whitelisted to access SQL Server. Architects should only allow IP addresses that they are confident about and that belong to customers/partners. There are deployments in Azure for which either there are a lot of IP addresses or the IP addresses are not known, such as applications deployed in Azure Functions or Logic Apps. For such applications to access Azure SQL, Azure SQL allows whitelisting of all IP addresses to Azure services across subscriptions.

It is to be noted that firewall configuration is at the server level and not the database level. This means that any changes here affect all databases within a server. In Figure 8.13, you can see how to add clients IPs to the firewall to grant access to the server:

The screenshot shows the 'Firewall settings' page for the database 'dbemployeerecords-prod'. At the top, there are buttons for 'Save' and 'Discard', and a link to 'Add client IP'. Below this, a section titled 'Deny public network access' has a 'Yes' button highlighted. A tooltip explains that setting it to Yes allows connections via approved private endpoint only and disables any existing firewall rules. Under 'Minimal TLS Version', the value is set to >1.2. A tooltip states that this setting applies to all SQL Database and SQL Data Warehouse databases associated with the server, rejecting login attempts from clients using TLS versions less than the specified value. In the 'Connection Policy' section, 'Default' is selected. Below it, 'Allow Azure services and resources to access this server' is set to No. A tooltip indicates that this setting provides access to all databases in the specified VNET/Subnet. The 'Client IP address' is listed as 117.210.181.243. The 'Client IP address rules' table lists two entries:

Rule name	Start IP	End IP	...
allow-vendors	56.17.113.55	56.17.113.55	...
allow-internal	172.17.1.0	172.17.1.250	...

A tooltip for the rules table states that connections from the specified VNET/Subnet provide access to all databases in the database.

Figure 8.13: Configuring firewall rules

Azure SQL also provides enhanced security by encrypting data at rest. This ensures that nobody, including the Azure datacenter administrators, can view the data stored in SQL Server. The technology used by SQL Server for encrypting data at rest is known as **Transparent Data Encryption (TDE)**. There are no changes required at the application level to implement TDE. SQL Server encrypts and decrypts data transparently when the user saves and reads data. This feature is available at the database level. We can also integrate TDE with Azure Key Vault to have **Bring Your Own Key (BYOK)**. Using BYOK, we can enable TDE using a customer-managed key in Azure Key Vault.

SQL Server also provides **dynamic data masking (DDM)**, which is especially useful for masking certain types of data, such as credit card details or user PII data. Masking is not the same as encryption. Masking does not encrypt data, but only masks it, which ensures that data is not in a human-readable format. Users should mask and encrypt sensitive data in Azure SQL Server.

SQL Server also provides an auditing and threat detection service for all servers. There are advanced data collection and intelligence services running on top of these databases to discover threats and vulnerabilities and alert users to them. Audit logs are maintained by Azure in storage accounts and can be viewed by administrators to be actioned. Threats such as SQL injection and anonymous client logins can generate alerts that administrators can be informed about over email. In Figure 8.14, you can see how to enable Threat Detection:

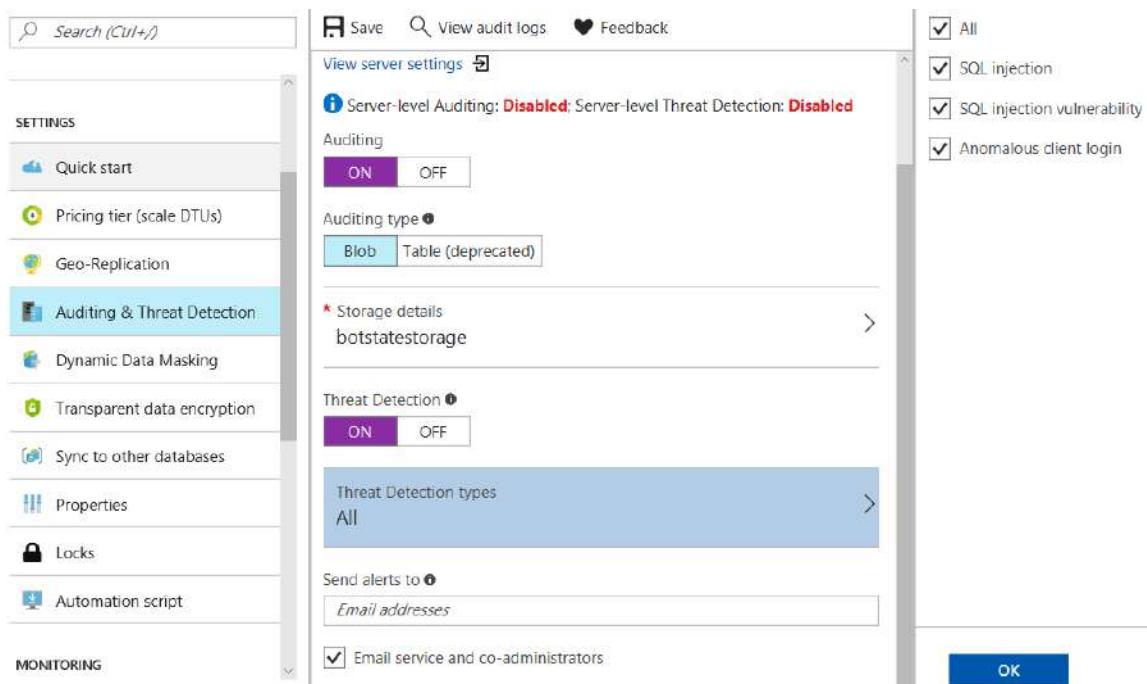


Figure 8.14: Enabling Threat Protection and selecting the types of threat to detect

Data can be masked in Azure SQL. This helps us store data in a format that cannot be read by humans:

The screenshot shows the Azure portal interface for configuring masking rules. At the top, there are buttons for Save, Discard, Add mask (which is highlighted in blue), and Feedback. On the left, under 'Masking rules', it says 'You haven't created any masking rules.' Below that, a note states 'SQL users excluded from masking (administrators are always excluded)'. Under 'Recommended fields to mask', it says 'There are no recommended fields to mask'. On the right, there is a sidebar for 'Mask name' (with a placeholder 'Mask name'), 'Select what to mask' (Schema dropdown), 'Table' dropdown, 'Column' dropdown, and 'Select how to mask' (Masking field format dropdown set to 'Default value (0, xxxx, 01-01-1900)').

Figure 8.15: Configuring the settings to mask data

Azure SQL also provides TDE to encrypt data at rest, as shown in Figure 8.16:

The screenshot shows the Azure portal interface for enabling TDE. On the left, a sidebar lists 'SETTINGS' with options: Quick start, Pricing tier (scale DTUs), Geo-Replication, Auditing & Threat Detection, Dynamic Data Masking, and Transparent data encryption (which is highlighted in blue). On the right, there is a summary card for TDE, featuring a shield icon, a note that says 'Encrypts your databases, backups enable TDE, go to each database.', a 'Learn more' link, a 'Data encryption' toggle switch set to 'ON', an 'Encryption status' section showing 'Encrypted' with a green checkmark, and a 'Transparent data encryption' section.

Figure 8.16: Enabling TDE

To conduct a vulnerability assessment on SQL Server, you can leverage SQL Vulnerability Assessment, which is a part of the unified package for advanced SQL security capabilities known as Advanced Data Security. SQL Vulnerability Assessment can be used by customers proactively to improve the security of the database by discovering, tracking, and helping you to remediate potential database vulnerabilities.

We have mentioned Azure Key Vault a few times in the previous sections, when we discussed managed identities, SQL Database, and so on. You know the purpose of Azure Key Vault now, and in the next section, we will be exploring some methods that can help secure the contents of your key vault.

Azure Key Vault

Securing resources using passwords, keys, credentials, certificates, and unique identifiers is an important element of any environment and application from the security perspective. They need to be protected, and ensuring that these resources remain secure and do not get compromised is an important pillar of security architecture. Management and operations that keep the secrets and keys secure, while making them available when needed, are important aspects that cannot be ignored. Typically, these secrets are used all over the place—within the source code, inside configuration files, on pieces of paper, and in other digital formats. To overcome these challenges and store all secrets uniformly in a centralized secure storage, Azure Key Vault should be used.

Azure Key Vault is well integrated with other Azure services. For example, it would be easy to use a certificate stored in Azure Key Vault and deploy it to an Azure virtual machine's certificate store. All kinds of keys, including storage keys, IoT and event keys, and connection strings, can be stored as secrets in Azure Key Vault. They can be retrieved and used transparently without anyone viewing them or storing them temporarily anywhere. Credentials for SQL Server and other services can also be stored in Azure Key Vault.

Azure Key Vault works on a per-region basis. What this means is that an Azure Key Vault resource should be provisioned in the same region where the application and service are deployed. If a deployment consists of more than one region and needs services from Azure Key Vault, multiple Azure Key Vault instances should be provisioned.

An important feature of Azure Key Vault is that the secrets, keys, and certificates are not stored in general storage. This sensitive data is backed up by the HSM. This means that this data is stored in separate hardware on Azure that can only be unlocked by keys owned by users. To provide added security, you can also implement virtual network service endpoints for Azure Key Vault. This will restrict access to the key vault to specific virtual networks. You can also restrict access to an IPv4 address range.

In the Azure Storage section, we discussed using Azure AD to authorize requests to blobs and queues. It was mentioned that we use an OAuth token, which is obtained from Azure AD, to authenticate API calls. In the next section, you will learn how authentication and authorization are done using OAuth. Once you have completed the next section, you will be able to relate it to what we discussed in the Azure Storage section.

Authentication and authorization using OAuth

Azure AD is an identity provider that can authenticate users based on already available users and service principals available within the tenant. Azure AD implements the OAuth protocol and supports authorization on the internet. It implements an authorization server and services to enable the OAuth authorization flow, implicit as well as client credential flows. These are different well-documented OAuth interaction flows between client applications, authorization endpoints, users, and protected resources.

Azure AD also supports **single sign-on (SSO)**, which adds security and ease when signing in to applications that are registered with Azure AD. You can use OpenID Connect, OAuth, SAML, password-based, or the linked or disabled SSO method when developing new applications. If you are unsure of which to use, refer to the flowchart from Microsoft here: <https://docs.microsoft.com/azure/active-directory/manage-apps/what-is-single-sign-on#choosing-a-single-sign-on-method>.

Web applications, JavaScript-based applications, and native client applications (such as mobile and desktop applications) can use Azure AD for both authentication and authorization. There are social media platforms, such as Facebook, Twitter, and so on, that support the OAuth protocol for authorization.

One of the easiest ways to enable authentication for web applications using Facebook is shown next. There are other methods that use security binaries, but that is outside the scope of this book.

In this walkthrough, an Azure App Service will be provisioned along with an App Service Plan to host a custom web application. A valid Facebook account will be needed as a prerequisite in order to redirect users to it for authentication and authorization.

A new resource group can be created using the Azure portal, as shown in *Figure 8.17*:

Create a resource group

Basics Tags Review + create

Resource group - A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. [Learn more](#)

Project details

Subscription * ⓘ RiteshSubscription

Resource group * ⓘ FaceauthRG

Resource details

Region * ⓘ (US) East US

Figure 8.17: Creating a new resource group

After the resource group has been created, a new app service can be created using the portal, as shown in *Figure 8.18*:

Web App

Project Details

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

Subscription * ⓘ RiteshSubscription

Resource Group * ⓘ FaceauthRG [Create new](#)

Instance Details

Name * faceauthenticationdemo.azurewebsites.net

Publish * [Code](#) Docker Container

Runtime stack * .NET Core 2.1 (LTS)

Operating System * Linux Windows

Region * Central US

Not finding your App Service Plan? Try a different region.

App Service Plan

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

Windows Plan (Central US) * ⓘ (New) ASP-FaceauthRG-87df [Create new](#)

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

Figure 8.18: Creating a new application

It is important to note the URL of the web application because it will be needed later when configuring the Facebook application.

Once the web application is provisioned in Azure, the next step is to create a new application in Facebook. This is needed to represent your web application within Facebook and to generate appropriate client credentials for the web application. This is the way Facebook knows about the web application.

Navigate to developers.facebook.com and log in using the appropriate credentials. Create a new application by selecting the **Create App** option under **My Apps** in the top-right corner, as shown in Figure 8.19:



Figure 8.19: Creating a new application from the Facebook developer portal

The web page will prompt you to provide a name for the web application to create a new application within Facebook:

Create a New App ID

Get started integrating Facebook into your app or website

Display Name

AzureforArchitectsoauth

Contact Email

callrites@yahoo.com

This email address is used to contact you about potential policy violations, app restrictions or steps to recover the app if it's been deleted or compromised.

By proceeding, you agree to the Facebook Platform Policies

[Cancel](#)

[Create App ID](#)

Figure 8.20: Adding a new application

Add a new **Facebook Login** product and click on **Set Up** to configure login for the custom web application to be hosted on Azure App Service:

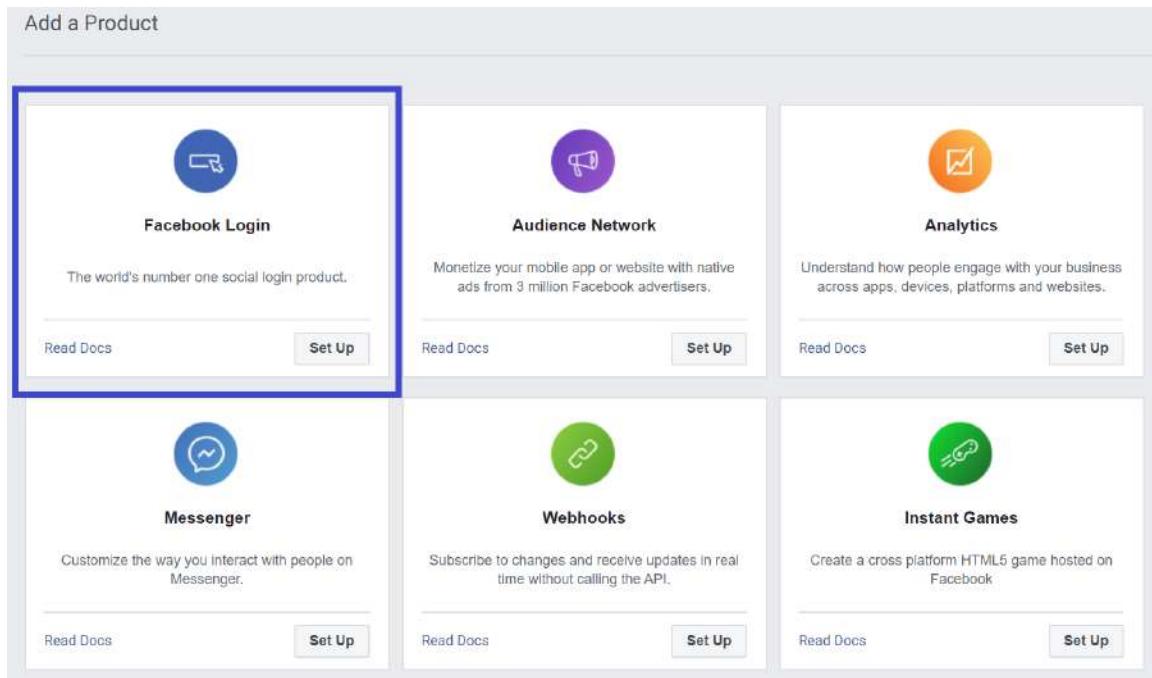


Figure 8.21: Adding Facebook login to the application

The **Set Up** button provides a few options, as shown in *Figure 8.22*, and these options configure the OAuth flow, such as authorization flow, implicit flow, or client credential flow. Select the **Web** option because that is what needs Facebook authorization:

Use the Quickstart to add Facebook Login to your app. To get started, select the platform for this app.



Figure 8.22: Selecting the platform

Provide the URL of the web application that we noted earlier after provisioning the web application on Azure:

1. Tell Us about Your Website

Tell us what the URL of your site is.

Site URL

`https://faceauthenticationdemo.azurewebsites.net`

Save

Figure 8.23: Providing the site URL to the application

Click on the **Settings** item in the menu on the left and provide the OAuth redirect URL for the application. Azure already has well-defined callback URLs for each of the popular social media platforms, and the one used for Facebook is `domain name/.auth/login/facebook/callback`:

The screenshot shows the Facebook for Developers interface. On the left, there's a sidebar with options like Dashboard, Settings, Roles, Alerts, App Review, and a Facebook Login section which is currently selected. The main content area is titled "Client OAuth Settings". It contains several configuration options with checkboxes:

- Client OAuth Login**: Yes (selected). Description: Enables the standard OAuth client token flow. Secure your application and prevent abuse by locking down which token redirect URIs are allowed with the options below. Disable globally if not used. [?]
- Web OAuth Login**: Yes (selected). Description: Enables web-based Client OAuth Login. [?]
- Enforce HTTPS**: Yes (selected). Description: Enforce the use of HTTPS for Redirect URIs and the JavaScript SDK. Strongly recommended. [?]
- Force Web OAuth Reauthentication**: No (selected). Description: When on, prompts people to enter their Facebook password in order to log in on the web. [?]
- Embedded Browser OAuth Login**: No (selected). Description: Enable webview Redirect URIs for Client OAuth Login. [?]
- Use Strict Mode for Redirect URIs**: Yes (selected). Description: Only allow redirects that use the Facebook SDK or that exactly match the Valid OAuth Redirect URIs. Strongly recommended. [?]

 Below these settings, there's a section for "Valid OAuth Redirect URIs" with a text input field containing the URL `https://faceauthenticationdemo.azurewebsites.net/.auth/login/facebook/callback`. There's also a "Login from Devices" checkbox set to No.

Figure 8.24: Adding OAuth redirect URIs

Go to the **Basic** settings from the menu on the left and note the values for **App ID** and **App Secret**. These are needed to configure the Azure App Services authentication/authorization:

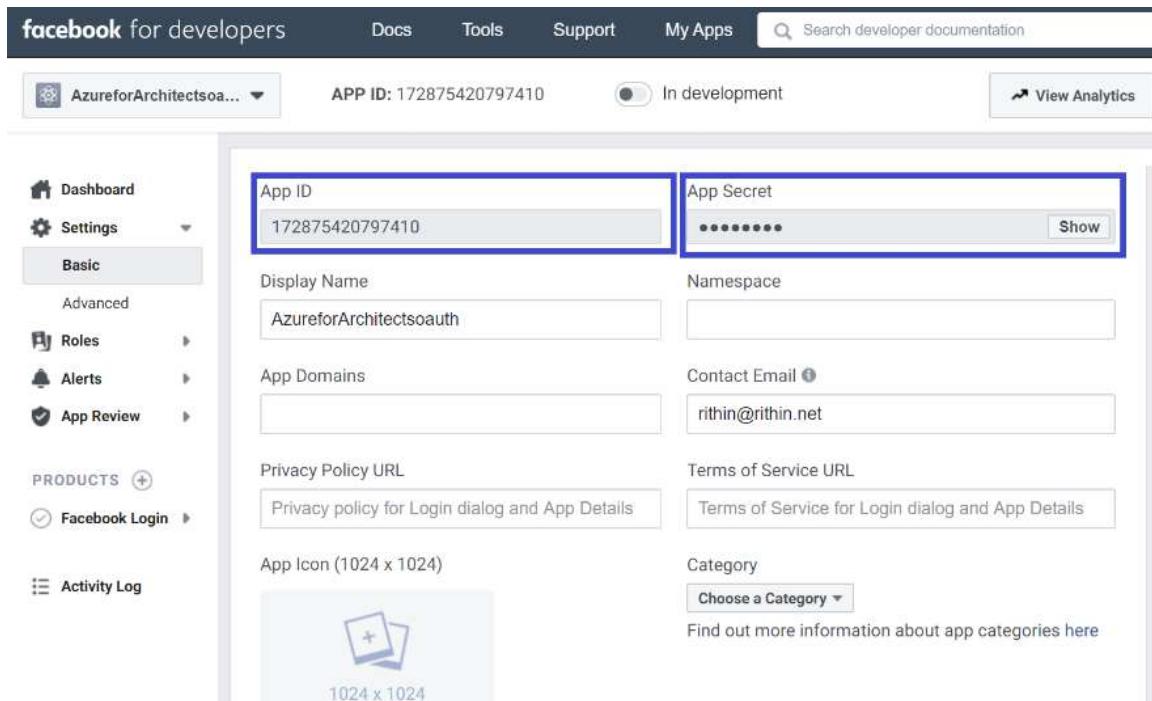


Figure 8.25: Finding the App ID and App Secret

In the Azure portal, navigate back to the Azure App Service created in the first few steps of this section and navigate to the authentication/authorization blade. Switch on **App Services Authentication**, select **Log in with Facebook** for authentication, and click on the **Facebook** item from the list:

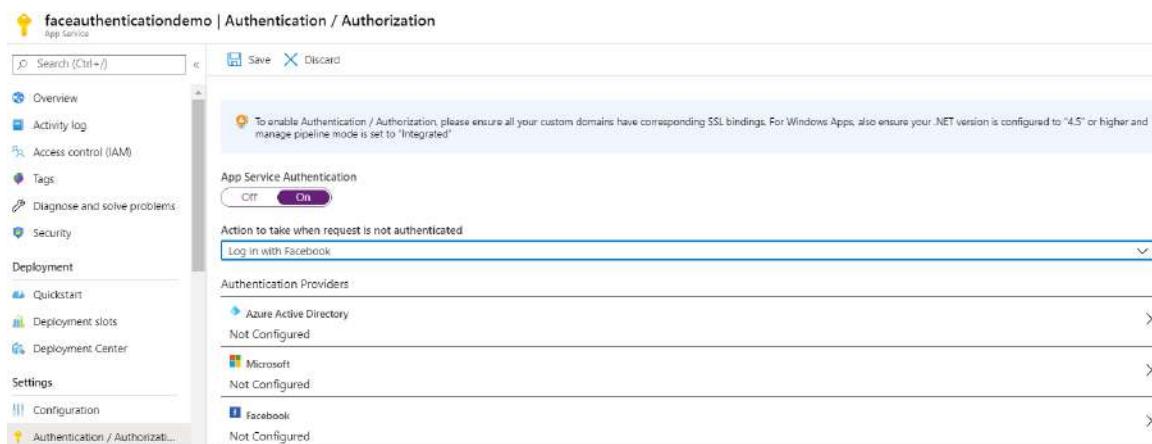


Figure 8.26: Enabling Facebook authentication in App Service

On the resultant page, provide the already noted app ID and app secret, and also select the scope. The scope decides the information shared by Facebook with the web application:

 These settings allow users to sign in with Facebook. Click here to learn more.

App ID	537218586990886
App Secret	f218c2a096454a9618fc0eda6c1718f0
<input type="checkbox"/> Scope	
<input checked="" type="checkbox"/> public_profile	Provides access to a subset of items that are part of a person's public profile.
<input type="checkbox"/> user_friends	Provides access the list of friends that also use your app.
<input type="checkbox"/> email	Provides access to the person's primary email address via the email property on the user object.
<input type="checkbox"/> user_actions.books	Provides access to all common books actions published by any app the person has used.
<input type="checkbox"/> user_actions.fitness	Provides access to all common Open Graph fitness actions published by any app the person has used.
<input type="checkbox"/> user_actions.music	Provides access to all common Open Graph music actions published by any app the person has used.
<input type="checkbox"/> user_actions.news	Provides access to all common Open Graph news actions published by any app the person has used which publishes these actions.
<input type="checkbox"/> user_actions.video	Provides access to all common Open Graph video actions published by any app the person has used which publishes these actions.
<input type="checkbox"/> user_birthday	Access the date and month of a person's birthday.
<input type="checkbox"/> user_education_history	Provides access to a person's education history through the education field on the User object.
<input type="checkbox"/> user_events	Provides read-only access to the Events a person is hosting or has RSVP'd to.
<input type="checkbox"/> user_games_activity	Provides access to read a person's game activity (scores, achievements) in any game the person has played.
<input type="checkbox"/> user_hometown	Provides access to a person's hometown location through the hometown field on the User object.

OK

Figure 8.27: Selecting the scope

Click **OK** and click the **Save** button to save the authentication/authorization settings.

Now, if a new incognito browser session is initiated and you go to the custom web application, the request should get redirected to Facebook. As you might have seen on other websites, when you use **Log in with Facebook**, you will be asked to give your credentials:

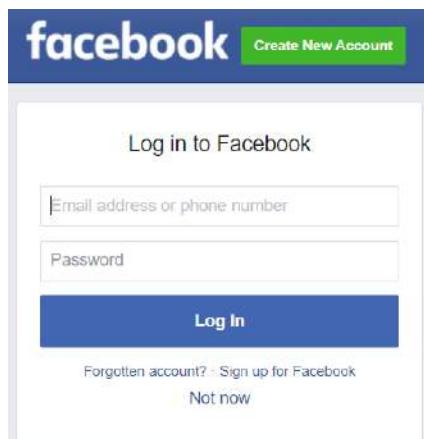


Figure 8.28: Logging in to the website using Facebook

Once you have entered your credentials, a user consent dialog box will ask for permission for data from Facebook to be shared with the web application:

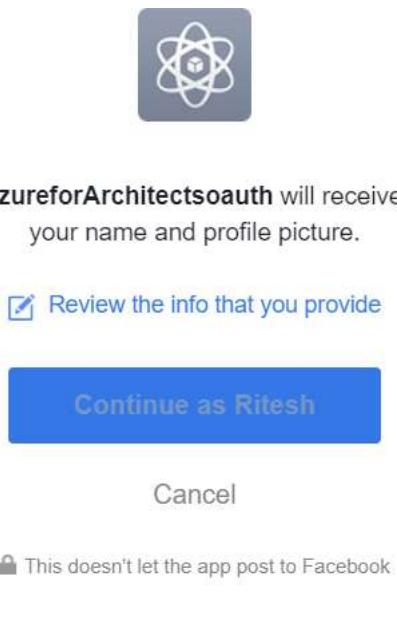


Figure 8.29: User consent to share your information with the application

If consent is provided, the web page from the web application should appear:

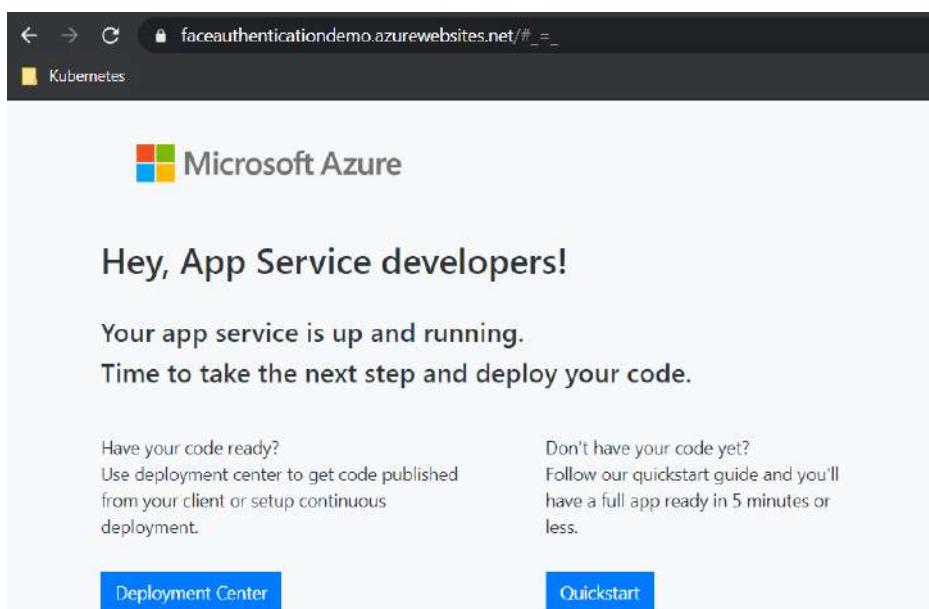


Figure 8.30: Accessing the landing page

A similar approach can be used to protect your web application using Azure AD, Twitter, Microsoft, and Google. You can also integrate your own identity provider if required.

The approach shown here illustrates just one of the ways to protect a website using credentials stored elsewhere and the authorization of external applications to access protected resources. Azure also provides JavaScript libraries and .NET assemblies to use the imperative programming method to consume the OAuth endpoints provided by Azure AD and other social media platforms. You are recommended to use this approach for greater control and flexibility for authentication and authorization within your applications.

So far, we have discussed security features and how they can be implemented. It is also relevant to have monitoring and auditing in place. Implementing an auditing solution will help your security team to audit the logs and take precautionary measures. In the next section, we will be discussing the security monitoring and auditing solutions in Azure.

Security monitoring and auditing

Every activity in your environment, from emails to changing a firewall, can be categorized as a security event. From a security standpoint, it's necessary to have a central logging system to monitor and track the changes made. During an audit, if you find suspicious activity, you can discover what the flaw in the architecture is and how it can be remediated. Also, if you had a data breach, the logs will help security professionals to understand the pattern of an attack and how it was executed. Also, necessary preventive measures can be taken to avoid similar incidents in the future. Azure provides the following two important security resources to manage all security aspects of the Azure subscription, resource groups, and resources:

- Azure Monitor
- Azure Security Center

Of these two security resources, we will first explore Azure Monitor.

Azure Monitor

Azure Monitor is a one-stop shop for monitoring Azure resources. It provides information about Azure resources and their state. It also offers a rich query interface, using information that can be sliced and diced using data at the levels of subscription, resource group, individual resource, and resource type. Azure Monitor collects data from numerous data sources, including metrics and logs from Azure, customer applications, and the agents running in virtual machines. Other services, such as Azure Security Center and Network Watcher, also ingest data to the Log Analytics workspace, which can be analyzed from Azure Monitor. You can use REST APIs to send custom data to Azure Monitor.

Azure Monitor can be used through the Azure portal, PowerShell, the CLI, and REST APIs:

The screenshot shows the Azure Activity log interface. At the top, there are navigation links: 'Dashboard' > 'Activity log'. Below that is a title 'Activity log' with a subtitle 'Logs'. A toolbar includes 'Edit columns', 'Refresh', 'Diagnostics settings', 'Download as CSV', 'Logs' (with a dropdown), 'Pin current filters', and 'Reset filters'. There is also a search bar and a 'Quick Insights' button. Filter options include 'Management Group : None', 'Subscription : 2 selected', 'Timespan : Last 6 hours', 'Event severity : All', and a 'Add Filter' button. The main area displays a table with 13 items, each representing a management operation. The columns are 'Operation name', 'Status', 'Time', 'Time stamp', and 'Subscription'. The operations listed are: Delete resource group, Delete SQL server, Update SQL database, Update SQL database, Update SQL server, Validate Deployment, Registers the Microsoft SQL Database Resource Provider, Check Server Name Availability, Check Server Name Availability, Registers the Microsoft SQL Database Resource Provider, Delete Disk, List Storage Account Keys, and Incident.

Operation name	Status	Time	Time stamp	Subscription
> i Delete resource group	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Delete SQL server	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Update SQL database	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Update SQL database	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Update SQL server	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Validate Deployment	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Registers the Microsoft SQL Database Resource Provider	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Check Server Name Availability	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Check Server Name Availability	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Registers the Microsoft SQL Database Resource Provider	Succeeded	3 hours ago	Fri May 22 ...	Project Hawk Eye 360
> i Delete Disk	Failed	3 hours ago	Fri May 22 ...	[PROD] rithin.net
> i List Storage Account Keys	Succeeded	4 hours ago	Fri May 22 ...	Project Hawk Eye 360
> ! Incident	Resolved	6 hours ago	Fri May 22 ...	Project Hawk Eye 360

Figure 8.31: Exploring activity logs

The following logs are those provided by Azure Monitor:

- **Activity log:** This shows all management-level operations performed on resources. It provides details about the creation time, the creator, the resource type, and the status of resources.
- **Operation log (classic):** This provides details of all operations performed on resources within a resource group and subscription.
- **Metrics:** This gets performance information for individual resources and sets alerts on them.
- **Diagnostic settings:** This helps us to configure the effects logs by setting up Azure Storage for storing logs, streaming logs in real time to Azure Event Hubs, and sending them to Log Analytics.
- **Log search:** This helps integrate Log Analytics with Azure Monitor.

Azure Monitor can identify security-related incidents and take appropriate action. It is important that only authorized individuals should be allowed to access Azure Monitor, since it might contain sensitive information.

Azure Security Center

Azure Security Center, as the name suggests, is a one-stop shop for all security needs. There are generally two activities related to security—implementing security and monitoring for any threats and breaches. Security Center has been built primarily to help with both these activities. Azure Security Center enables users to define their security policies and get them implemented on Azure resources. Based on the current state of Azure resources, Azure Security Center provides security recommendations to harden the solution and individual Azure resources. The recommendations include almost all Azure security best practices, including the encryption of data and disks, network protection, endpoint protection, access control lists, the whitelisting of incoming requests, and the blocking of unauthorized requests. The resources range from infrastructure components, such as load balancers, network security groups, and virtual networks, to PaaS resources, such as Azure SQL and Storage. Here is an excerpt from the **Overview** pane of Azure Security Center, which shows the overall secure score of the subscription, resource security hygiene, and more:

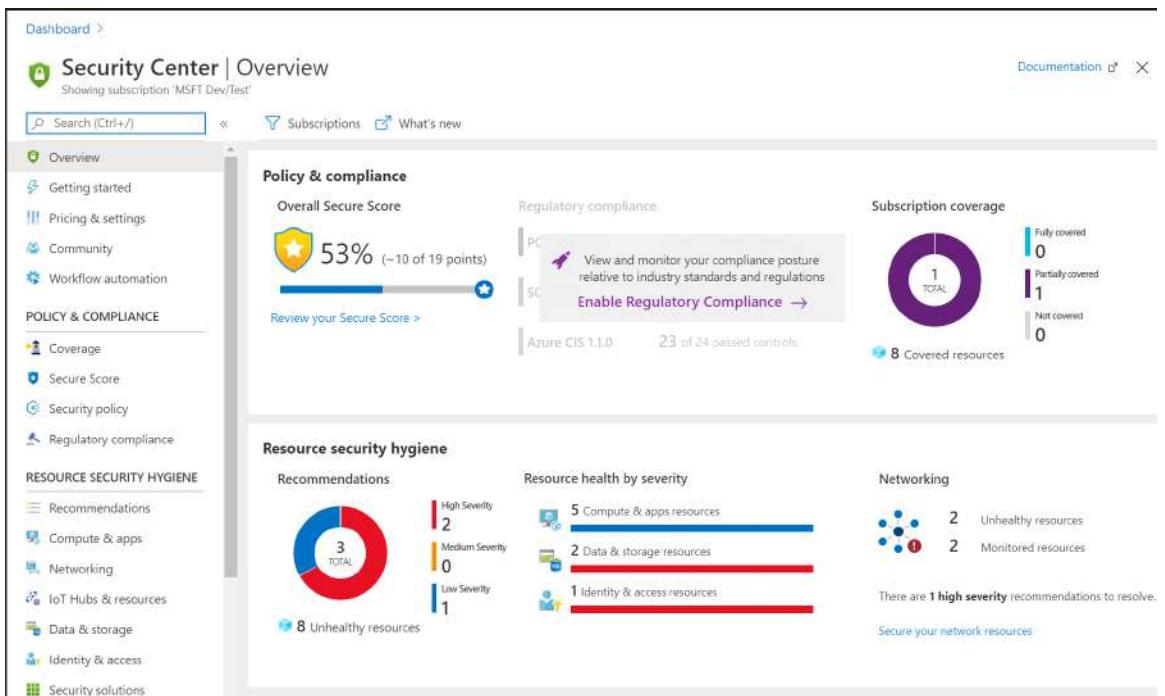


Figure 8.32: Azure Security Center overview

Azure Security Center is a rich platform that provides recommendations for multiple services, as shown in Figure 8.33. Also, these recommendations can be exported to CSV files for reference:

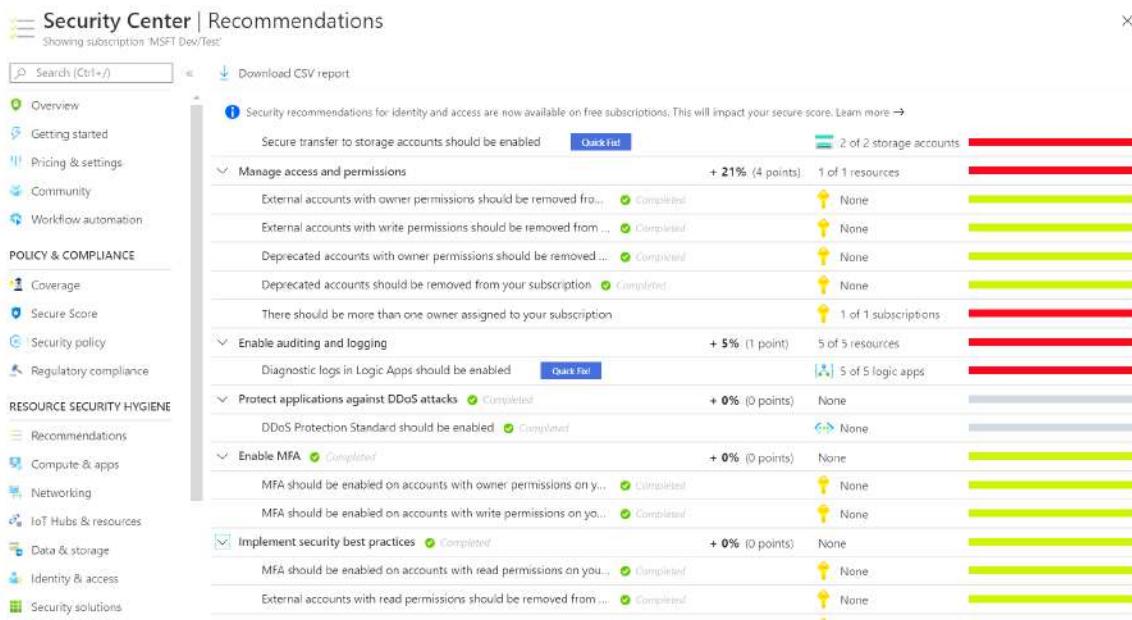


Figure 8.33: Azure Security Center recommendations

As was mentioned at the beginning of this section, monitoring and auditing are crucial in an enterprise environment. Azure Monitor can have multiple data sources and can be used to audit logs from these sources. Azure Security Center gives continuous assessments and prioritized security recommendations along with the overall secure score.

Summary

Security is always an important aspect of any deployment or solution. It has become much more important and relevant because of deployment to the cloud. Moreover, there is an increasing threat of cyberattacks. In these circumstances, security has become a focal point for organizations. No matter the type of deployment or solution, whether it's IaaS, PaaS, or SaaS, security is needed across all of them. Azure datacenters are completely secure, and they have a dozen international security certifications. They are secure by default. They provide IaaS security resources, such as NSGs, network address translation, secure endpoints, certificates, key vaults, storage, virtual machine encryption, and PaaS security features for individual PaaS resources. Security has a complete life cycle of its own and it should be properly planned, designed, implemented, and tested, just like any other application functionality.

We discussed operating system firewalls and Azure Firewall and how they can be leveraged to increase the overall security landscape of your solution. We also explored new Azure services, such as Azure Bastion, Azure Front Door, and Azure Private Link.

Application security was another key area, and we discussed performing authentication and authorization using OAuth. We did a quick demo of how to create an app service and integrate Facebook login. Facebook was just an example; you could use Google, Twitter, Microsoft, Azure AD, or any custom identity provider.

We also explored the security options offered by Azure SQL, which is a managed database service provided by Azure. We discussed the implementation of security features, and in the final section, we concluded with monitoring and auditing with Azure Monitor and Azure Security Center. Security plays a vital role in your environment. An architect should always design and architect solutions with security as one of the main pillars of the architecture; Azure provides many options to achieve this.

Now that you know how to secure your data in Azure, in the next chapter, we will focus on big data solutions from Hadoop, followed by Data Lake Storage, Data Lake Analytics, and Data Factory.

9

Azure Big Data solutions

In the previous chapter, you learned about the various security strategies that can be implemented on Azure. With a secure application, we manage vast amounts of data. Big data has been gaining significant traction over the last few years. Specialized tools, software, and storage are required to handle it. Interestingly, these tools, platforms, and storage options were not available as services a few years back. However, with new cloud technology, Azure provides numerous tools, platforms, and resources to create big data solutions easily. This chapter will detail the complete architecture for ingesting, cleaning, filtering, and visualizing data in a meaningful way.

The following topics will be covered in this chapter:

- Big data overview
- Data integration
- **Extract-Transform-Load (ETL)**
- Data Factory
- Data Lake Storage
- Tools ecosystems such as Spark, Databricks, and Hadoop
- Databricks

Big data

With the influx of cheap devices—such as Internet of Things devices and hand-held devices—the amount of data that is being generated and captured has increased exponentially. Almost every organization has a great deal of data and they are ready to purchase more if needed. When large quantities of data arrive in multiple different formats and on an ever-increasing basis, then we can say we are dealing with big data. In short, there are three key characteristics of big data:

- **Volume:** By volume, we mean the quantity of data both in terms of size (in GB, TB, and PB, for instance) and in terms of the number of records (as in a million rows in a hierarchical data store, 100,000 images, half a billion JSON documents, and so on).
- **Velocity:** Velocity refers to the speed at which data arrives or is ingested. If data does not change frequently or new data does not arrive frequently, the velocity of data is said to be low, while if there are frequent updates and a lot of new data arrives on an ongoing basis frequently, it is said to have high velocity.
- **Variety:** Variety refers to different kinds and formats of data. Data can come from different sources in different formats. Data can arrive as structured data (as in comma-separated files, JSON files, or hierarchical data), as semi-structured databases (as in schema-less NoSQL documents), or as unstructured data (such as binary large objects (blobs), images, PDFs, and so on). With so many variants, it's important to have a defined process for processing ingested data.

In the next section, we will check out the general big data process.

Process for big data

When data comes from multiple sources in different formats and at different speeds, it is important to set out a process of storing, assimilating, filtering, and cleaning data in a way that helps us to work with that data more easily and make the data useful for other processes. There needs to be a well-defined process for managing data. The general process for big data that should be followed is shown in *Figure 9.1*:



Figure 9.1: Big data process

There are four main stages of big data processing. Let's explore them in detail:

- **Ingest:** This is the process of bringing and ingesting data into the big data environment. Data can come from multiple sources, and connectors should be used to ingest that data within the big data platform.
- **Store:** After ingestion, data should be stored in the data pool for long-term storage. The storage should be there for both historical as well as live data and must be capable of storing structured, semi-structured, and non-structured data. There should be connectors to read the data from data sources, or the data sources should be able to push data to storage.
- **Analysis:** After data is read from storage, it should be analyzed, a process that requires filtering, grouping, joining, and transforming data to gather insights.
- **Visualize:** The analysis can be sent as reports using multiple notification platforms or used to generate dashboards with graphs and charts.

Previously, the tools needed to capture, ingest, store, and analyze big data were not readily available for organizations due to the involvement of expensive hardware and large investments. Also, no platform was available to process them. With the advent of the cloud, it has become easier for organizations to capture, ingest, store, and perform big data analytics using their preferred choice of tools and frameworks. They can pay the cloud provider to use their infrastructure and avoid any capital expenditure. Moreover, the cost of the cloud is very cheap compared to any on-premises solution.

Big data demands an immense amount of compute, storage, and network resources. Generally, the amount of resources required is not practical to have on a single machine or server. Even if, somehow, enough resources are made available on a single server, the time it takes to process an entire big data pool is considerably large, since each job is done in sequence and each step has a dependency upon the prior step. There is a need for specialized frameworks and tools that can distribute work across multiple servers and eventually bring back the results from them and present to the user after appropriately combining the results from all the servers. These tools are specialized big data tools that help in achieving availability, scalability, and distribution out of the box to ensure that a big data solution can be optimized to run quickly with built-in robustness and stability.

The two prominent Azure big data services are HD Insights and Databricks. Let's go ahead and explore the various tools available in the big data landscape.

Big data tools

There are many tools and services in the big data space, and we are going to cover some of them in this chapter.

Azure Data Factory

Azure Data Factory is the flagship ETL service in Azure. It defines incoming data (in terms of its format and schema), transforms data according to business rules and filters, augments existing data, and finally transfers data to a destination store that is readily consumable by other downstream services. It is able to run pipelines (containing ETL logic) on Azure, as well as custom infrastructure, and can also run SQL Server Integration Services packages.

Azure Data Lake Storage

Azure Data Lake Storage is enterprise-level big data storage that is resilient, highly available, and secure out of the box. It is compatible with Hadoop and can scale to petabytes of data storage. It is built on top of Azure storage accounts and hence gets all of the benefits of storage account directly. The current version is called Gen2, after the capabilities of both Azure Storage and Data Lake Storage Gen1 were combined.

Hadoop

Hadoop was created by the Apache software foundation and is a distributed, scalable, and reliable framework for processing big data that breaks big data down into smaller chunks of data and distributes them within a cluster. A Hadoop cluster comprises two types of servers—masters and slaves. The master server contains the administrative components of Hadoop, while the slaves are the ones where the data processing happens. Hadoop is responsible for the logical partition data between slaves; slaves perform all transformation on data, gather insights, and pass them back to master nodes who will collate them to generate the final output. Hadoop can scale to thousands of servers, with each server providing compute and storage for the jobs. Hadoop is available as a service using the **HDInsight** service in Azure.

There are three main components that make up the Hadoop core system:

HDFS: Hadoop Distributed File System is a file system for the storage of big data. It is a distributed framework that helps by breaking down large big data files into smaller chunks and placing them on different slaves in a cluster. HDFS is a fault-tolerant file system. This means that although different chunks of data are made available to different slaves in the cluster, there is also the replication of data between the slaves to ensure that in the event of any slave's failure, that data will also be available on another server. It also provides fast and efficient access to data to the requestor.

MapReduce: MapReduce is another important framework that enables Hadoop to process data in parallel. This framework is responsible for processing data stored within HDFS slaves and mapping them to the slaves. After the slaves are done processing, the "reduce" part brings information from each slave and collates them together as the final output. Generally, both HDFS and MapReduce are available on the same node, such that the data does not need to travel between slaves and higher efficiency can be achieved when processing them.

YARN: Yet Another Resource Negotiator (YARN) is an important Hadoop architectural component that helps in scheduling jobs related to applications and resource management within a cluster. YARN was released as part of Hadoop 2.0, with many casting it as the successor to MapReduce as it is more efficient in terms of batch processing and resource allocation.

Apache Spark

Apache Spark is a distributed, reliable analytics platform for large-scale data processing. It provides a cluster that is capable of running transformation and machine learning jobs on large quantities of data in parallel and bringing a consolidated result back to the client. It comprises master and worker nodes, where the master nodes are responsible for dividing and distributing the actions within jobs and data between worker nodes, as well as consolidating the results from all worker nodes and returning the results to the client. An important thing to remember while using Spark is that the logic or calculations should be easily parallelized, and the amount of data is too large to fit on one machine. Spark is available in Azure as a service from HDInsight and Databricks.

Databricks

Databricks is built on top of Apache Spark. It is a Platform as a Service where a managed Spark cluster is made available to users. It provides lots of added features, such as a complete portal to manage Spark cluster and its nodes, as well as helping to create notebooks, schedule and run jobs, and provide security and support for multiple users.

Now, it's time to learn how to integrate data from multiple sources and work with them together using the tools we've been talking about.

Data integration

We are well aware of how integration patterns are used for applications; applications that are composed of multiple services are integrated together using a variety of patterns. However, there is another paradigm that is a key requirement for many organizations, which is known as data integration. The surge in data integration has primarily happened during the last decade, when the generation and availability of data has become incredibly high. The velocity, variety, and volume of data being generated has increased drastically, and there is data almost everywhere.

Every organization has many different types of applications, and they all generate data in their own proprietary format. Often, data is also purchased from the marketplace. Even during mergers and amalgamations of organizations, data needs to be migrated and combined.

Data integration refers to the process of bringing data from multiple sources and generating a new output that has more meaning and usability.

There is a definite need for data integration in the following scenarios:

- Migrating data from a source or group of sources to a target destination. This is needed to make data available in different formats to different stakeholders and consumers.
- Getting insights from data. With the rapidly increasing availability of data, organizations want to derive insights from it. They want to create solutions that provide insights; data from multiple sources should be merged, cleaned, augmented, and stored in a data warehouse.
- Generating real-time dashboards and reports.
- Creating analytics solutions.

Application integration has a runtime behavior when users are consuming the application—for example, in the case of credit card validation and integration. On the other hand, data integration happens as a back-end exercise and is not directly linked to user activity.

Let's move on to understanding the ETL process with Azure Data Factory.

ETL

A very popular process known as ETL helps in building a target data source to house data that is consumable by applications. Generally, the data is in a raw format, and to make it consumable, the data should go through the following three distinct phases:

- **Extract:** During this phase, data is extracted from multiple places. For instance, there could be multiple sources and they all need to be connected together in order to retrieve the data. Extract phases typically use data connectors consisting of connection information related to the target data source. They might also have temporary storage to bring the data from the data source and store it for faster retrieval. This phase is responsible for the ingestion of data.
- **Transform:** The data that is available after the extract phase might not be directly consumable by applications. This could be for a variety of reasons; for example, the data might have irregularities, there might be missing data, or there might be erroneous data. Or, there might even be data that is not needed at all. Alternatively, the format of the data might not be conducive to consumption by the target applications. In all of these cases, transformation has to be applied to the data in such a way that it can be efficiently consumed by applications.

- **Load:** After transformation, data should be loaded to the target data source in a format and schema that enables faster, easier, and performance-centric availability for applications. Again, this typically consists of data connectors for destination data sources and loading data into them.

Next, let's cover how Azure Data Factory relates to the ETL process.

A primer on Azure Data Factory

Azure Data Factory is a fully managed, highly available, highly scalable, and easy-to-use tool for creating integration solutions and implementing ETL phases. Data Factory helps you to create new pipelines in a drag and drop fashion using a user interface, without writing any code; however, it still provides features to allow you to write code in your preferred language.

There are a few important concepts to learn about before using the Data Factory service, which we will be exploring in more detail in the following sections:

- **Activities:** Activities are individual tasks that enable the running and processing of logic within a Data Factory pipeline. There are multiple types of activities. There are activities related to data movement, data transformation, and control activities. Each activity has a policy through which it can decide the retry mechanism and retry interval.
- **Pipelines:** Pipelines in Data Factory are composed of groups of activities and are responsible for bringing activities together. Pipelines are the workflows and orchestrators that enable the running of the ETL phases. Pipelines allow the weaving together of activities and allow the declaration of dependencies between them. By using dependencies, it is possible to run some tasks in parallel and other tasks in sequence.
- **Datasets:** Datasets are the sources and destinations of data. These could be Azure storage accounts, Data Lake Storage, or a host of other sources.
- **Linked services:** These are services that contain the connection and connectivity information for datasets and are utilized by individual tasks for connecting to them.
- **Integration runtime:** The main engine that is responsible for the running of Data Factory is called the integration runtime. The integration runtime is available on the following three configurations:
- **Azure:** In this configuration, Data Factory runs on the compute resources that are provided by Azure.

- **Self-hosted:** Data Factory, in this configuration, runs when you bring your own compute resources. This could be through on-premises or cloud-based virtual machine servers.
- **Azure SQL Server Integration Services (SSIS):** This configuration allows the running of traditional SSIS packages written using SQL Server.
- **Versions:** Data Factory comes in two different versions. It is important to understand that all new developments will happen on V2, and that V1 will stay as it is, or fade out at some point. V2 is preferred for the following reasons:
 - It provides the capability to run SQL Server integration packages.
 - It has enhanced functionalities compared to V1.
 - It comes with enhanced monitoring, which is missing in V1.

Now that you have a fair understanding of Data Factory, let's get into the various storage options available on Azure.

A primer on Azure Data Lake Storage

Azure Data Lake Storage provides storage for big data solutions. It is specially designed for storing the large amounts of data that are typically needed in big data solutions. It is an Azure-provided managed service. Customers need to bring their data and store it in a data lake.

There are two versions of Azure Data Lake Storage: version 1 (Gen1) and the current version, version 2 (Gen2). Gen2 has all the functionality of Gen1, but one particular difference is that it is built on top of Azure Blob storage.

As Azure Blob storage is highly available, can be replicated multiple times, is disaster-ready, and is low in cost, these benefits are transferred to Data Lake Storage Gen2. Data Lake Storage Gen2 can store any kind of data, including relational, non-relational, file system-based, and hierarchical data.

Creating a Data Lake Storage Gen2 instance is as simple as creating a new storage account. The only change that needs to be done is enabling the hierarchical namespace from the **Advanced** tab of your storage account. It is important to note that there is no direct migration or conversion from a general storage account to Azure Data Lake Storage or vice versa. Also, general storage accounts are for storing files, while Data Lake Storage is optimized for reading and ingesting large quantities of data.

Next, we will look into the process and main phases while working with big data. These are distinct phases and each is responsible for different activities on data.

Migrating data from Azure Storage to Data Lake Storage Gen2

In this section, we will be migrating data from Azure Blob storage to another Azure container of the same Azure Blob storage instance, and we will also migrate data to an Azure Data Lake Storage Gen2 instance using an Azure Data Factory pipeline. The following sections outline the steps that need to be taken to create such an end-to-end solution.

Preparing the source storage account

Before we can create Azure Data Factory pipelines and use them for migration, we need to create a new storage account, consisting of a number of containers, and upload the data files. In the real world, these files and the storage connection would already be prepared. The first step for creating a new Azure storage account is to create a new resource group or choose an existing resource group within an Azure subscription.

Provisioning a new resource group

Every resource in Azure is associated with a resource group. Before we provision an Azure storage account, we need to create a resource group that will host the storage account. The steps for creating a resource group are given here. It is to be noted that a new resource group can be created while provisioning an Azure storage account or an existing resource group can be used:

1. Navigate to the Azure portal, log in, and click on **+ Create a resource**; then, search for **Resource group**.
2. Select **Resource group** from the search results and create a new resource group. Provide a name and choose an appropriate location. Note that all the resources should be hosted in the same resource group and location so that it is easy to delete them.

After provisioning the resource group, we will provision a storage account within it.

Provisioning a storage account

In this section, we will go through the steps of creating a new Azure storage account. This storage account will fetch the data source from which data will be migrated. Perform the following steps to create a storage account:

1. Click on **+ Create a resource** and search for **Storage Account**. Select **Storage Account** from the search results and then create a new storage account.
2. Provide a name and location, and then select a subscription based on the resource group that was created earlier.
3. Select **StorageV2 (general purpose v2)** for **Account kind**, **Standard** for **Performance**, and **Locally-redundant storage (LRS)** for **Replication**, as demonstrated in *Figure 9.2*:

Create storage account

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. [Learn more](#)

PROJECT DETAILS

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

* Subscription	<input type="text" value="RiteshSubscription"/>
	<input type="button" value="Create new"/>
* Resource group	<input type="text" value="BigDataSolutions"/>
	<input type="button" value="Create new"/>

INSTANCE DETAILS

The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model instead. [Choose classic deployment model](#)

* Storage account name <small>?</small>	<input type="text" value="adfsamplesourcedata"/> <small>✓</small>
* Location	<input type="text" value="East US"/> <small>▼</small>
Performance <small>?</small>	<input checked="" type="radio"/> Standard <input type="radio"/> Premium
Account kind <small>?</small>	<input type="text" value="StorageV2 (general purpose v2)"/> <small>▼</small>
Replication <small>?</small>	<input type="text" value="Locally-redundant storage (LRS)"/> <small>▼</small>
Access tier (default) <small>?</small>	<input type="radio"/> Cool <input checked="" type="radio"/> Hot

Figure 9.2: Configuring the storage account

- Now create a couple of containers within the storage account. The **rawdata** container contains the files that will be extracted by the Data Factory pipeline and will act as the source dataset, while **finaldata** will contain files that the Data Factory pipelines will write data to and will act as the destination dataset:

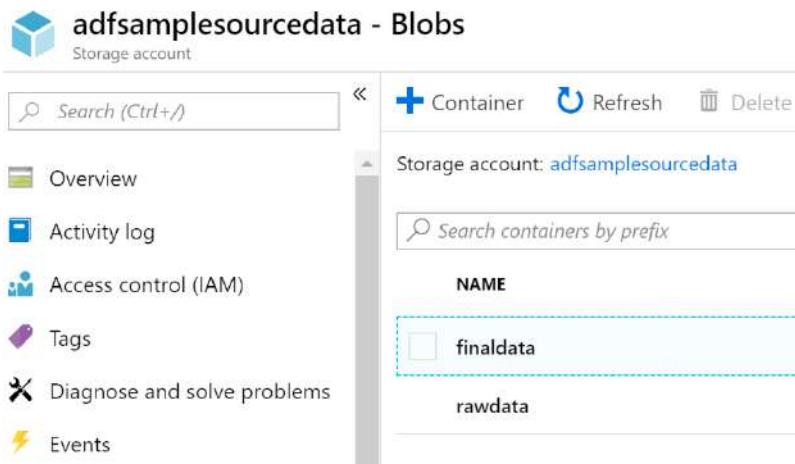


Figure 9.3: Creating containers

- Upload a data file (this file is available with the source code) to the **rawdata** container, as shown in *Figure 9.4*:

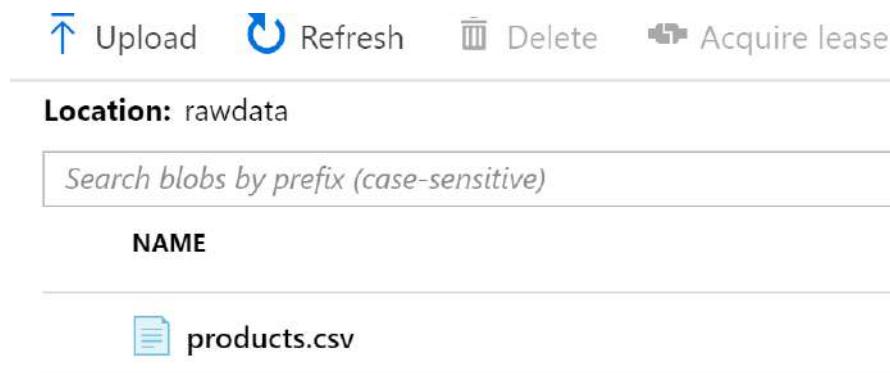


Figure 9.4: Uploading a data file

After completing these steps, the source data preparation activities are complete. Now we can focus on creating a Data Lake Storage instance.

Provisioning the Data Lake Storage Gen2 service

As we already know, the Data Lake Storage Gen2 service is built on top of the Azure storage account. Because of this, we will be creating a new storage account in the same way that we did earlier—with the only difference being the selection of **Enabled** for **Hierarchical namespace** in the **Advanced** tab of the new Azure storage account. This will create the new Data Lake Storage Gen2 service:

Create storage account

Azure Files

Large file shares ⓘ Disabled Enabled

i The current combination of storage account kind, performance, replication and location does not support large file shares.

Data protection

Blob soft delete ⓘ Disabled Enabled

i Data protection and hierarchical namespace cannot be enabled simultaneously.

File share soft delete ⓘ Disabled Enabled

i Data protection and hierarchical namespace cannot be enabled simultaneously.

Versioning ⓘ Disabled Enabled

i The current combination of subscription, storage account kind, performance, replication and location does not support versioning.

Data Lake Storage Gen2

Hierarchical namespace ⓘ Disabled Enabled

NFS v3 ⓘ Disabled Enabled

i Sign up is currently required to utilize the NFS v3 feature on a per-subscription basis. [Sign up for NFS v3](#)

Figure 9.5: Creating a new storage account

After the creation of the data lake, we will focus on creating a new Data Factory pipeline.

Provisioning Azure Data Factory

Now that we have provisioned both the resource group and Azure storage account, it's time to create a new Data Factory resource:

1. Create a new Data Factory pipeline by selecting **V2** and by providing a name and location, along with a resource group and subscription selection.

Data Factory has three different versions, as shown in *Figure 9.6*. We've already discussed **V1** and **V2**:

New data factory

Name *

Version ⓘ

V2

Subscription *

Microsoft Azure Sponsorship

Resource Group *

akscluster

Create new

Location * ⓘ

West Central US

Enable GIT ⓘ

GIT URL * ⓘ

Repo name * ⓘ

testrepo

Branch Name * ⓘ

master

Root folder * ⓘ

Pipelines

Figure 9.6: Selecting the version of Data Factory

- Once the Data Factory resource is created, click on the **Author & Monitor** link from the central pane.

This will open another window, consisting of the Data Factory designer for the pipelines.

The code of the pipelines can be stored in version control repositories such that it can be tracked for code changes and promote collaboration between developers. If you missed setting up the repository settings in these steps, that can be done later.

The next section will focus on configuration related to version control repository settings if your Data Factory resource was created without any repository settings being configured.

Repository settings

Before creating any Data Factory artifacts, such as datasets and pipelines, it is a good idea to set up the code repository for hosting files related to Data Factory:

- From the **Authoring** page, click on the **Manage** button and then **Git Configuration** in the left menu. This will open another pane; click on the **Set up code repository** button in this pane:

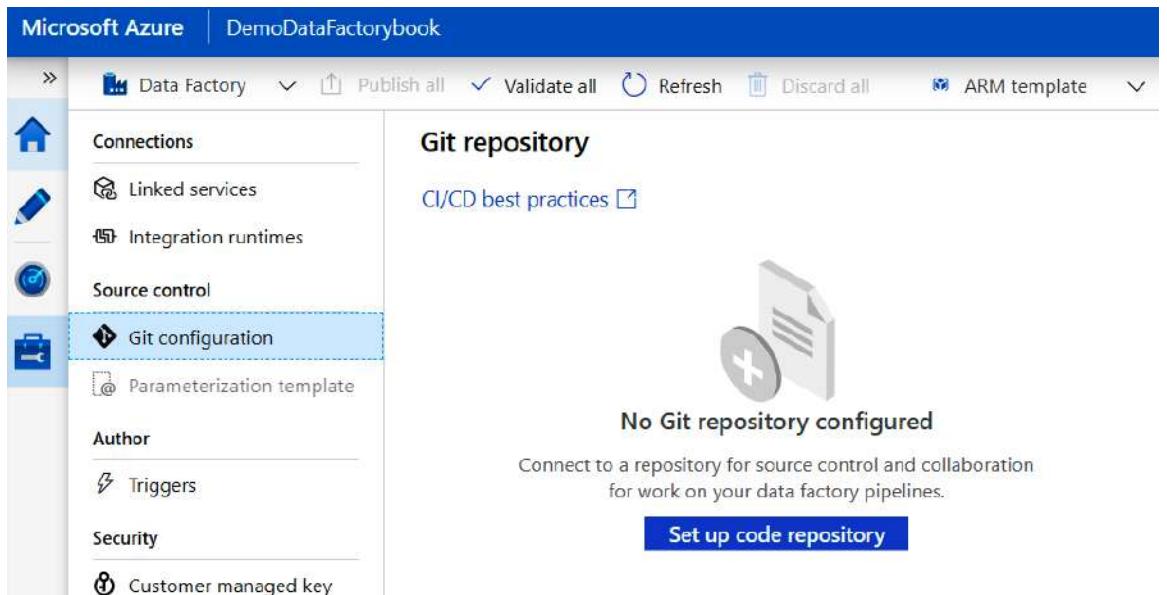


Figure 9.7: Setting up a Git repository

- From the resultant blade, select any one of the types of repositories that you want to store Data Factory code files in. In this case, let's select **Azure DevOps Git**:

Repository settings

Repository type * Azure DevOps Git

Select a different directory

Azure Active Directory * [REDACTED]

Azure DevOps Account * [REDACTED]

Project name * [REDACTED]

Git repository name * Create new Use existing [REDACTED]

Collaboration branch * [REDACTED]

Root folder * [/]

Import existing Data Factory resources to repository

Branch to import resources into * Use Collaboration Create new Use existing

Figure 9.8: Selecting the appropriate Git repository type

- Create a new repository or reuse an existing repository from Azure DevOps. You should already have an account in Azure DevOps. If not, visit, <https://dev.azure.com> and use the same account used for the Azure portal to login and create a new organization and project within it. Refer to *Chapter 13, Integrating Azure DevOps*, to learn more about creating organizations and projects in Azure DevOps.

Now, we can move back to the Data Factory authoring window and start creating artifacts for our new pipeline.

In the next section, we will prepare the datasets that will be used within our Data Factory pipelines.

Data Factory datasets

Now we can go back to the Data Factory pipeline. First, create a new dataset that will act as the source dataset. It will be the first storage account that we create and upload the sample **product.csv** file to:

1. Click on **+ Datasets** -> **New DataSet** from the left menu and select **Azure Blob Storage as data store** and **delimitedText** as the format for the source file. Create a new linked service by providing a name and selecting an Azure subscription and storage account. By default, **AutoResolveIntegrationRuntime** is used for the runtime environment, which means Azure will provide the runtime environment on Azure-managed compute. Linked services provide multiple authentication methods, and we are using the **shared access signature (SAS) uniform resource locator (URI)** method. It is also possible to use an account key, service principal, and managed identity as authentication methods:

The screenshot shows the 'New linked service (Azure Blob Storage)' configuration page. The form fields are as follows:

- Name ***: AzureBlobStorage1
- Description**: (empty)
- Connect via integration runtime ***: AutoResolveIntegrationRuntime
- Authentication method**: Account key
- Connection string** (selected tab): Azure Key Vault
- Account selection method**:
 - From Azure subscription
 - Enter manually
- Azure subscription**: Select all
- Storage account name ***: (empty)
- Additional connection properties**:
 -
- Test connection**:
 - To linked service
 - To file path

Figure 9.9: Implementing the authentication method

2. Then, on the resultant lower pane in the **General** tab, click on the **Open properties** link and provide a name for the dataset:

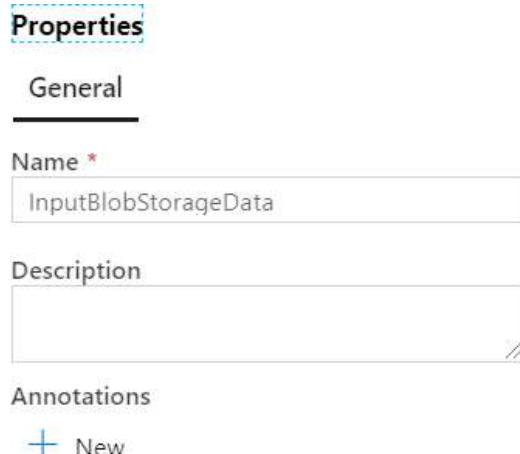


Figure 9.10: Naming the dataset

3. From the **Connection** tab, provide details about the container, the blob file name in the storage account, the row delimiter, the column delimiter, and other information that will help Data Factory to read the source data appropriately.

The **Connection** tab, after configuration, should look similar to *Figure 9.11*. Notice that the path includes the name of the container and the name of the file:

Setting	Value
Linked service *	AzureBlobStorage1
File path *	rawdata / Directory / products.csv
Compression type	none
Column delimiter	Comma (,) <input type="button" value="Edit"/>
Row delimiter	Auto detect (\r\n, \n, or \r\n\r\n) <input type="button" value="Edit"/>
Encoding	Default(UTF-8)
Escape character	Backslash (\) <input type="button" value="Edit"/>
Quote character	Double quote (")

Figure 9.11: Configuring the connection

- At this point, if you click on the **Preview data** button, it shows preview data from the **product.csv** file. On the **Schema** tab, add two columns and name them **ProductID** and **ProductPrice**. The schema helps in providing an identifier to the columns and also mapping the source columns in the source dataset to the target columns in the target dataset, when the names are not the same.

Now that the first dataset is created, let's create the second one.

Creating the second dataset

Create a new dataset and linked service for the destination blob storage account in the same way that you did before. Note that the storage account is the same as the source but the container is different. Ensure that the incoming data has schema information associated with it as well, as shown in Figure 9.12:

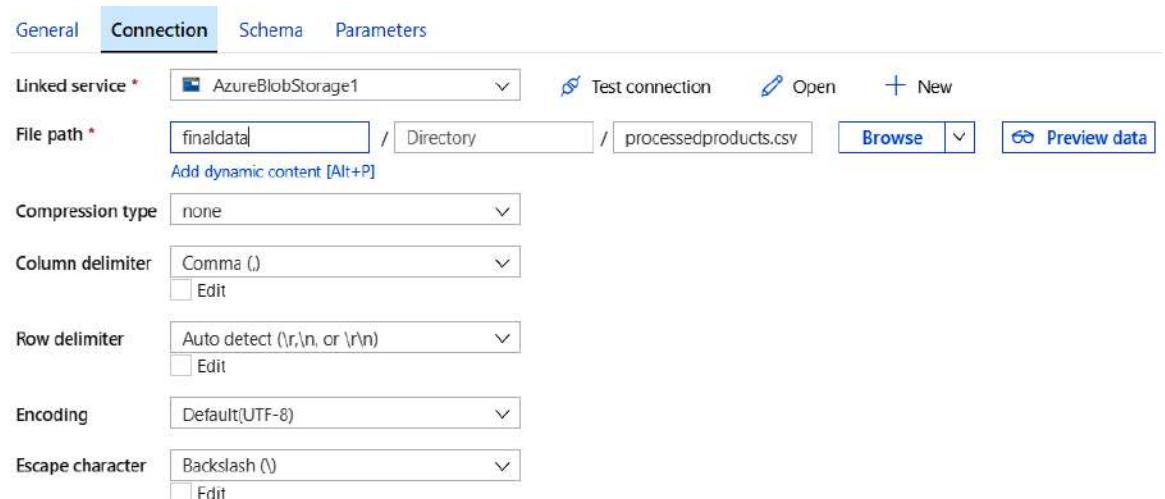


Figure 9.12: Creating the second dataset

Next, we will create a third dataset.

Creating a third dataset

Create a new dataset for the Data Lake Storage Gen2 instance as the target dataset. To do this, select the new dataset and then select **Azure Data Lake Storage Gen2**.

Give the new dataset a name and create a new linked service in the **Connection** tab. Choose **Use account key** as the authentication method and the rest of the configuration will be auto-filled after selecting the storage account name. Then, test the connection by clicking on the **Test connection** button. Keep the default configuration for the rest of the tabs, as shown in Figure 9.13:

New Linked Service (Azure Data Lake Storage Gen2 ... X

Name *

Description

Connect via integration runtime *

AutoResolveIntegrationRuntime

Authentication method

Use account key

Account selection method

From Azure subscription Enter manually

Azure subscription

Select all

Storage account name *

datalakegen2productstore

Sign up to the public preview of Azure Data Lake Storage Gen2.

Annotations

+ New

► Advanced ⓘ

Cancel **Test connection** **Finish**

Figure 9.13: Configuration in Connection tabs

Now that we have the connection to source data and also connections to both the source and destination data stores, it's time to create the pipelines that will contain the logic of the data transformation.

Creating a pipeline

After all the datasets are created, we can create a pipeline that will consume those datasets. The steps for creating a pipeline are given next:

1. Click on the **+ Pipelines => New Pipeline** menu from the left menu to create a new pipeline. Then, drag and drop the **Copy Data** activity from the **Move & Transform** menu, as demonstrated in Figure 9.14:

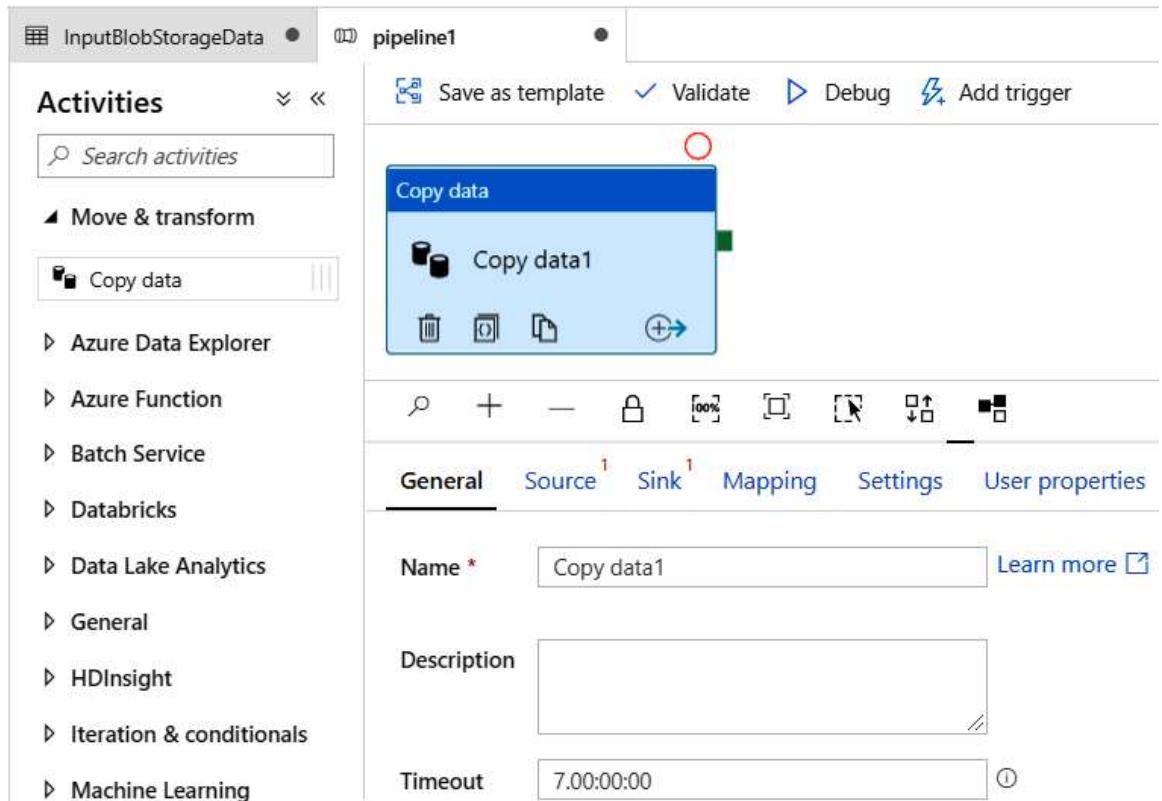


Figure 9.14: Pipeline menu

2. The resultant **General** tab can be left as it is, but the **Source** tab should be configured to use the source dataset that we configured earlier:

Source Dataset * Edit + New

Copy file recursively

Figure 9.15: Source tab

3. The **Sink** tab is used to configure the destination data store and dataset, and it should be configured to use the target dataset that we configured earlier:

Sink Dataset * Edit + New

Copy behavior

Figure 9.16: Sink tab

4. On the **Mapping** tab, map the columns from the source to the destination dataset columns, as shown in Figure 9.17:

Field / Type	Field	Type	Include
ProductID(String)	ProductId	String	<input checked="" type="checkbox"/>
ProductPrice(Decimal)	ProductPrice	Decimal	<input checked="" type="checkbox"/>

Figure 9.17: Mapping tab

Adding one more Copy Data activity

Within our pipeline, we can add multiple activities, each responsible for a particular transformation task. The task looked at in this section is responsible for copying data from the Azure storage account to Azure Data Lake Storage:

1. Add another **Copy Data** activity from the left activity menu to migrate data to Data Lake Storage; both of the copy tasks will run in parallel:

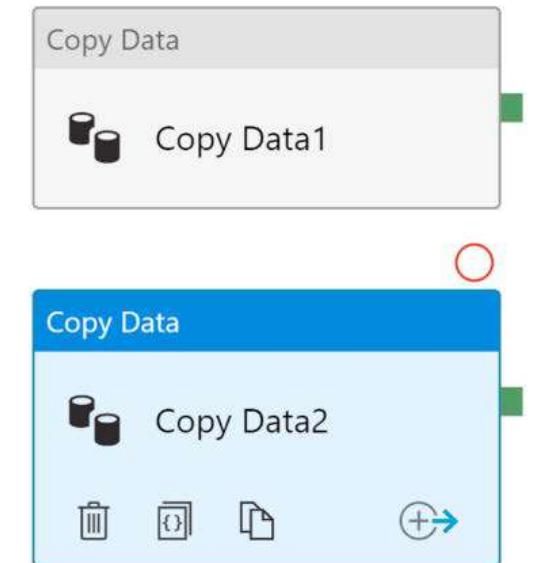


Figure 9.18: Copy Data activities

The configuration for the source is the Azure Blob storage account that contains the **product.csv** file.

The sink configuration will target the Data Lake Storage Gen2 account.

2. The rest of the configuration can be left in the default settings for the second Copy Data activity.

After the authoring of the pipeline is complete, it can be published to a version control repository such as GitHub.

Next, we will look into creating a solution using Databricks and Spark.

Creating a solution using Databricks

Databricks is a platform for using Spark as a service. We do not need to provision master and worker nodes on virtual machines. Instead, Databricks provides us with a managed environment consisting of master and worker nodes and also manages them. We need to provide the steps and logic for the processing of data, and the rest is taken care of by the Databricks platform.

In this section, we will go through the steps of creating a solution using Databricks. We will be downloading sample data to analyze.

The sample CSV has been downloaded from <https://ourworldindata.org/coronavirus-source-data>, although it is also provided with the code of this book. The URL mentioned before will have more up-to-date data; however, the format might have changed, and so it is recommended to use the file available with the code samples of this book:

1. The first step in creating a Databricks solution is to provision it from the Azure portal. There is a 14-day evaluation SKU available along with two other SKUs—standard and premium. The premium SKU has Azure Role-Based Access Control at the level of notebooks, clusters, jobs, and tables:

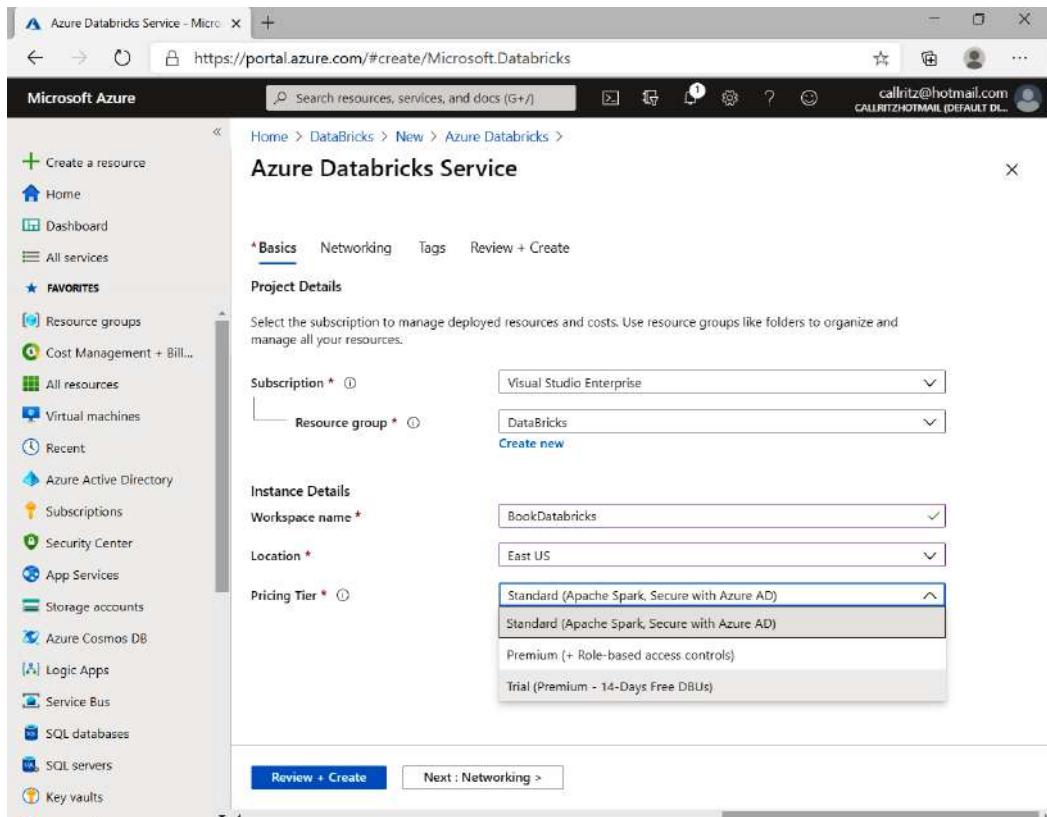


Figure 9.19: Azure portal—Databricks service

2. After the Data bricks workspace is provisioned, click on the **Launch workspace** button from the **Overview** pane. This will open a new browser window and will eventually log you in to the Databricks portal.
3. From the Databricks portal, select **Clusters** from the left menu and create a new cluster, as shown in *Figure 9.20*:

Create Cluster

New Cluster Cancel **Create Cluster** 2-8 Workers: 28.0-112.0 GB Memory, 8-32 Cores, 1.5-6 DBU
1 Driver: 14.0 GB Memory, 4 Cores, 0.75 DBU

Cluster Name

Cluster Mode

Pool

Databricks Runtime Version Learn more

New This Runtime version supports only Python 3.

Autopilot Options
 Enable autoscaling
 Terminate after minutes of inactivity

Worker Type <small>Standard_DS3_v2</small>	Min Workers <input type="text" value="2"/>	Max Workers <input type="text" value="8"/>
Driver Type <small>Same as worker</small>		

▶ Advanced Options

Figure 9.20: Creating a new cluster

4. Provide the name, the Databricks runtime version, the number of worker types, the virtual machine size configuration, and the driver type server configuration.
5. The creation of the cluster might take a few minutes. After the creation of the cluster, click on **Home**, select a user from its context menu, and create a new notebook:

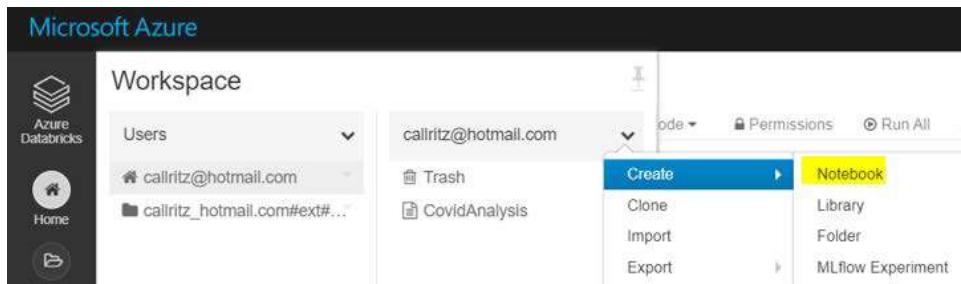


Figure 9.21: Selecting a new notebook

- Provide a name to the notebook, as shown next:

Create Notebook

The dialog box has three input fields: 'Name' containing 'CovidAnalysis', 'Default Language' set to 'Python', and 'Cluster' set to 'BookBigDataCluster'. At the bottom are 'Cancel' and 'Create' buttons, with 'Create' being highlighted in blue.

Figure 9.22: Creating a notebook

- Create a new storage account, as shown next. This will act as storage for the raw COVID data in CSV format:

Create storage account

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

The form includes sections for 'Subscription' (set to 'Visual Studio Enterprise'), 'Resource group' (set to 'DataBricks' with a 'Create new' link), 'Instance details' (deployment model is Resource Manager), and storage configuration. The storage section includes fields for 'Storage account name' ('coronadatastorage'), 'Location' ('East US'), 'Performance' (radio buttons for 'Standard' and 'Premium' with 'Standard' selected), 'Account kind' ('StorageV2 (general purpose v2)'), 'Replication' ('Locally-redundant storage (LRS)'), and 'Access tier (default)' ('Cool' and 'Hot' with 'Hot' selected).

Figure 9.23: Creating a new storage account

8. Create a container for storing the CSV file, as shown next:

The screenshot shows the Azure Storage Container creation interface. At the top, there are buttons for '+ Container', 'Change access level', 'Refresh', and 'Delete'. Below that is a search bar labeled 'Search containers by prefix'. A table follows, with columns 'Name', 'Last modified', and 'Public acc'. A message at the bottom says 'You don't have any containers yet. Click '+ Container' to get started.' On the right, a form is displayed with 'Name *' set to 'coviddata' and a green checkmark. Under 'Public access level', 'Private (no anonymous access)' is selected.

Figure 9.24: Creating a container

9. Upload the `owid-covid-data.csv` file to this container.

Once you have completed the preceding steps, the next task is to load the data.

Loading data

The second major step is to load the COVID data within the Databricks workspace. This can be done in two main ways:

- Mount the Azure storage container in Databricks and then load the files available within the mount.
- Load the data directly from the storage account. This approach has been used in the following example.

The following steps should be performed to load and analyze data using Databricks:

1. The first step is to connect and access the storage account. The key for the storage account is needed, which is stored within the Spark configuration. Note that the key here is `"fs.azure.account.key.coronadatastorage.blob.core.windows.net"` and the value is the associated key:

```
spark.conf.set("fs.azure.account.key.coronadatastorage.blob.core.windows.net", "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx=")
```

2. The key for the Azure storage account can be retrieved by navigating to the settings and the **Access Keys** property of the storage account in the portal.

The next step is to load the file and read the data within the CSV file. The schema should be inferred from the file itself instead of being provided explicitly. There is also a header row, which is represented using the option in the next command.

The file is referred to using the following format: `wasbs://{{container}}@{{storage account name}}.blob.core.windows.net/{{filename}}`.

3. The **read** method of the **SparkSession** object provides methods to read files. To read CSV files, the **csv** method should be used along with its required parameters, such as the path to the CSV file. There are additional optional parameters that can be supplied to customize the reading process of the data files. There are multiple types of file formats, such as JSON, **Optimized Row Columnar (ORC)**, and Parquet, and relational databases such as SQL Server and MySQL, NoSQL data stores such as Cassandra and MongoDB, and big data platforms such as Apache Hive that can all be used within Spark. Let's take a look at the following command to understand the implementation of Spark DataFrames:

```
coviddata = spark.read.format("csv").option("inferSchema", "true").
option("header", "true").load("wasbs://coviddata@coronadatastorage.blob.
core.windows.net/owid-covid-data.csv")
```

Using this command creates a new object of the DataFrame type in Spark. Spark provides **Resilient Distributed Dataset (RDD)** objects to manipulate and work with data. RDDs are low-level objects and any code written to work with them might not be optimized. DataFrames are higher-level constructs over RDDs and provide optimization to access and work with them RDDs. It is better to work with DataFrames than RDDs.

DataFrames provide data in row-column format, which makes it easier to visualize and work with data. Spark DataFrames are similar to pandas DataFrames, with the difference being that they are different implementations.

4. The following command shows the data in a DataFrame. It shows all the rows and columns available within the DataFrame:

```
coviddata.show()
```

You should get a similar output to what you can see in Figure 9.25:

iso_code	location	date	total_cases	new_cases	total_deaths	new_deaths	total_cases_per_million	new_cases_per_million	total_deaths_per_million	new_deaths_per_million	median_age	aged_65_older	aged_70_older	gdp_per_capita	extreme_poverty	cvd_death_rate	diabetes_prevalence	female_smokers	male_smokers	handwashing_facilities	hospital_beds_per_100k
null	null	2020-03-13 00:00:00	2	2	0	0	18.733	18.733	18.733	18.733	0.0	106766.0	584.8	0.0	0.0	0.0	0.0	0.0	0.0		
41.2	ABW	Aruba 2020-03-13 00:00:00	13.085	7.452	35973.781	0	11.62	11.62	11.62	11.62	0.0	106766.0	584.8	0.0	0.0	0.0	0.0	0.0	0.0		
null	ABW	Aruba 2020-03-28 00:00:00	13.085	7.452	35973.781	4	2	0	37.465	37.465	18.733	18.733	0.0	106766.0	584.8	0.0	0.0	0.0	0.0	0.0	
41.2							11.62	11.62	11.62	11.62											

Figure 9.25: The raw data in a DataFrame

5. The schema of the loaded data is inferred by Spark and can be checked using the following command:

```
coviddata.printSchema()
```

This should give you a similar output to this:

```

root
|-- iso_code: string (nullable = true)
|-- location: string (nullable = true)
|-- date: timestamp (nullable = true)
|-- total_cases: integer (nullable = true)
|-- new_cases: integer (nullable = true)
|-- total_deaths: integer (nullable = true)
|-- new_deaths: integer (nullable = true)
|-- total_cases_per_million: double (nullable = true)
|-- new_cases_per_million: double (nullable = true)
|-- total_deaths_per_million: double (nullable = true)
|-- new_deaths_per_million: double (nullable = true)

```

Figure 9.26: Getting the schema of the DataFrame for each column

- To count the number of rows within the CSV file, the following command can be used, and its output shows that there are 19,288 rows in the file:

```
coviddata.count()
```

▶ (1) Spark Jobs

Out[7]: 19288

Figure 9.27: Finding the count of records in a DataFrame

- The original DataFrame has more than 30 columns. We can also select a subset of the available columns and work with them directly, as shown next:

```
CovidDataSmallSet = coviddata.select("location", "date", "new_cases", "new_deaths")
CovidDataSmallSet.show()
```

The output of the code will be as shown in Figure 9.28:

location	date	new_cases	new_deaths
Aruba	2020-03-13 00:00:00	2	0
Aruba	2020-03-20 00:00:00	2	0
Aruba	2020-03-24 00:00:00	8	0
Aruba	2020-03-25 00:00:00	5	0
Aruba	2020-03-26 00:00:00	2	0
Aruba	2020-03-27 00:00:00	9	0
Aruba	2020-03-28 00:00:00	0	0
Aruba	2020-03-29 00:00:00	0	0
Aruba	2020-03-30 00:00:00	22	0
Aruba	2020-04-01 00:00:00	5	0
Aruba	2020-04-02 00:00:00	0	0

Figure 9.28: Selecting a few columns from the overall columns

8. It is also possible to filter data using the **filter** method, as shown next:

```
CovidDataSmallSet.filter(" location == 'United States' ").show()
```

9. It is also possible to add multiple conditions together using the AND (**&**) or OR (**|**) operators:

```
CovidDataSmallSet.filter((CovidDataSmallSet.location == 'United States') |  
(CovidDataSmallSet.location == 'Aruba')).show()
```

10. To find out the number of rows and other statistical details, such as the mean, maximum, minimum, and standard deviation, the **describe** method can be used:

```
CovidDataSmallSet.describe().show()
```

Upon using the preceding command, you'll get a similar output to this:

summary	location	new_cases	new_deaths
count	19288	19288	19288
mean	null	536.6524263790958	35.05174201576109
stddev	null	4828.611755359697	335.3488977234115
min	Afghanistan	-2461	0
max	Zimbabwe	107909	10520

Figure 9.29: Showing each column's statistics using the **describe** method

11. It is also possible to find out the percentage of null or empty data within specified columns. A couple of examples are shown next:

```
from pyspark.sql.functions import col  
(coviddata.where(col("diabetes_prevalence").isNull()).count() * 100)/  
coviddata.count()
```

The output shows **5.998548320199087**, which means 95% of the data is null. We should remove such columns from data analysis. Similarly, running the same command on the **total_tests_per_thousand** column returns **73.62090418913314**, which is much better than the previous column.

12. To drop some of the columns from the DataFrame, the next command can be used:

```
coviddatanew=coviddata.drop("iso_code").drop("total_tests").drop("total_  
tests").drop("new_tests").drop("total_tests_per_thousand").drop("new_  
tests_per_thousand").drop("new_tests_smoothed").drop("new_tests_smoothed_  
per_thousand ")
```

13. At times, you will need to have an aggregation of data. In such scenarios, you can perform the grouping of data, as shown here:

```
coviddatanew = coviddata.groupBy('location').agg({'date': 'max'})
```

This will display the data from the **groupBy** statement:

location	max(date)
Chad	2020-05-23 00:00:00
Anguilla	2020-05-23 00:00:00
Paraguay	2020-05-23 00:00:00
Russia	2020-05-23 00:00:00
International	2020-03-10 00:00:00
Yemen	2020-05-23 00:00:00
World	2020-05-23 00:00:00
Senegal	2020-05-23 00:00:00
Sweden	2020-05-23 00:00:00
Guyana	2020-05-23 00:00:00
Eritrea	2020-05-23 00:00:00
Jersey	2020-05-23 00:00:00
Philippines	2020-05-23 00:00:00

Figure 9.30: Data from the groupby statement

14. As you can see in the **max (date)** column, the dates are mostly the same for all the countries, we can use this value to filter the records and get a single row for each country representing the maximum date:

```
coviddatauniquecountry = coviddata.filter("date='2020-05-23 00:00:00'")  
coviddatauniquecountry.show()
```

15. If we take a count of records for the new DataFrame, we get **209**.

We can save the new DataFrame into another CSV file, which may be needed by other data processors:

```
coviddatauniquecountry.rdd.saveAsTextFile("dbfs:/mnt/coronadatastorage/  
uniquecountry.csv")
```

We can check the newly created file with the following command:

```
%fs ls /mnt/coronadatastorage/
```

The mounted path will be displayed as shown in Figure 9.31:



Figure 9.31: The mounted path within the Spark nodes

16. It is also possible to add the data into the Databricks catalog using the `createTempView` or `createOrReplaceTempView` method within the Databricks catalog. Putting data into the catalog makes it available in a given context. To add data into the catalog, the `createTempView` or `createOrReplaceTempView` method of the DataFrame can be used, providing a new view for the table within the catalog:

```
coviddatauniquecountry.createOrReplaceTempView("corona")
```

17. Once the table is in the catalog, it is accessible from your SQL session, as shown next:

```
spark.sql("select * from corona").show()
```

The data from the SQL statement will appear as shown in Figure 9.32:

location	date	total_cases
Aruba	2020-05-23 00:00:00	101
Afghanistan	2020-05-23 00:00:00	9216
Angola	2020-05-23 00:00:00	60
Anguilla	2020-05-23 00:00:00	3
Albania	2020-05-23 00:00:00	981
Andorra	2020-05-23 00:00:00	762
United Arab Emirates	2020-05-23 00:00:00	27892
Argentina	2020-05-23 00:00:00	10636
Armenia	2020-05-23 00:00:00	5928
Antigua and Barbuda	2020-05-23 00:00:00	25
Australia	2020-05-23 00:00:00	7095
Austria	2020-05-23 00:00:00	16361
Azerbaijan	2020-05-23 00:00:00	3855
Burundi	2020-05-23 00:00:00	42
Belgium	2020-05-23 00:00:00	56511
Benin	2020-05-23 00:00:00	135
Bonaire Sint Eust...	2020-05-23 00:00:00	6
Burkina Faso	2020-05-23 00:00:00	814
Bangladesh	2020-05-23 00:00:00	30205
Bulgaria	2020-05-23 00:00:00	2408

Figure 9.32: Data from the SQL statement

18. It is possible to perform an additional SQL query against the table, as shown next:

```
spark.sql("select * from corona where location in ('India','Angola') order by location").show()
```

That was a small glimpse of the possibilities with Databricks. There are many more features and services within it that could not be covered within a single chapter. Read more about it at <https://azure.microsoft.com/services/databricks>.

Summary

This chapter dealt with the Azure Data Factory service, which is responsible for providing ETL services in Azure. Since it is a platform as a service, it provides unlimited scalability, high availability, and easy-to-configure pipelines. Its integration with Azure DevOps and GitHub is also seamless. We also explored the features and benefits of using Azure Data Lake Storage Gen2 to store any kind of big data. It is a cost-effective, highly scalable, hierarchical data store for handling big data, and is compatible with Azure HDInsight, Databricks, and the Hadoop ecosystem.

By no means did we have a complete deep dive into all the topics mentioned in this chapter. It was more about the possibilities in Azure, especially with Databricks and Spark. There are multiple technologies in Azure related to big data, including HDInsight, Hadoop, Spark and its related ecosystem, and Databricks, which is a Platform as a Service environment for Spark with added functionality. In the next chapter, you will learn about the serverless computing capabilities in Azure.

10

Serverless in Azure – Working with Azure Functions

In the previous chapter, you learned about various big data solutions available on Azure. In this chapter, you will learn how serverless technology can help you deal with a large amount of data.

Serverless is one of the hottest buzzwords in technology these days, and everyone wants to ride this bandwagon. Serverless brings a lot of advantages in overall computing, software development processes, infrastructure, and technical implementation. There is a lot going on in the industry: at one end of the spectrum is **Infrastructure as a Service (IaaS)**, and at the other is serverless. In between the two are **Platform as a Service (PaaS)** and containers. I have met many developers and it seems to me that there is some confusion among them about IaaS, PaaS, containers, and serverless computing. Also, there is much confusion about use cases, applicability, architecture, and implementation for the serverless paradigm. Serverless is a new paradigm that is changing not only technology but also the culture and processes within organizations.

We will begin this chapter by introducing serverless, and will cover the following topics as we progress:

- Functions as a Service
- Azure Functions
- Azure Durable Functions
- Azure Event Grid

Serverless

Serverless refers to a deployment model in which users are responsible for only their application code and configuration. In serverless computing, customers do not have to bother about bringing their own underlying platform and infrastructure and, instead, can concentrate on solving their business problems.

Serverless does not mean that there are no servers. Code and configuration will always need compute, storage, and networks to run. However, from the customer's perspective, there is no visibility of such compute, storage, and networks. They do not care about the underlying platform and infrastructure. They do not need to manage or monitor infrastructure and the platform. Serverless provides an environment that can scale up and down, in and out, automatically, without the customer even knowing about it. All operations related to platforms and infrastructures happen behind the scenes and are executed by the cloud provider. Customers are provided with performance-related **service-level agreements (SLAs)** and Azure ensures that it meets those SLAs irrespective of the total demand.

Customers are required to only bring in their code; it is the responsibility of the cloud provider to provide the infrastructure and platform needed to run the code. Let's go ahead and dive into the various advantages of Azure Functions.

The advantages of Azure Functions

Serverless computing is a relatively new paradigm that helps organizations convert large functionalities into smaller, discrete, on-demand functions that can be invoked and executed through automated triggers and scheduled jobs. They are also known as **Functions as a Service (FaaS)**, in which organizations can focus on their domain challenges instead of the underlying infrastructure and platform. FaaS also helps in devolving solution architectures into smaller, reusable functions, thereby increasing return on investment.

There is a plethora of serverless compute platforms available. Some of the important ones are listed here:

- Azure Functions
- AWS Lambda
- IBM OpenWhisk
- Iron.io
- Google Cloud Functions

In fact, every few days it feels like there is a new platform/framework being introduced, and it is becoming increasingly difficult for enterprises to decide on the framework that works best for them. Azure provides a rich serverless environment known as Azure Functions, and what follows are some of the features that it supports:

- Numerous ways to invoke a function—manually, on a schedule, or based on an event.
- Numerous types of binding support.
- The ability to run functions synchronously as well as asynchronously.
- The ability to execute functions based on multiple types of triggers.
- The ability to run both long- and short-duration functions. However, large and long-running functions are not recommended as they may lead to unexpected timeouts.
- The ability to use proxy features for different function architectures.
- Multiple usage models including consumption, as well as the App Service model.
- The ability to author functions using multiple languages, such as JavaScript, Python, and C#.
- Authorization based on OAuth.
- The Durable Functions extension helps in writing stateful functions.
- Multiple authentication options, including Azure AD, Facebook, Twitter, and other identity providers.
- The ability to easily configure inbound and outbound parameters.
- Visual Studio integration for authoring Azure functions.
- Massive parallelism.

Let's take a look at FaaS and what roles it plays in serverless architecture.

FaaS

Azure provides FaaS. These are serverless implementations from Azure. With Azure Functions, code can be written in any language the user is comfortable with and Azure Functions will provide a runtime to execute it. Based on the language chosen, an appropriate platform is provided for users to bring their own code. Functions are a unit of deployment and can automatically be scaled out and in. When dealing with functions, users cannot view the underlying virtual machines and platform, but Azure Functions provides a small window to view them via the **Kudu Console**.

There are two main components of Azure Functions:

- The Azure Functions runtime
- Azure Functions binding and triggers

Let's learn about these components in detail.

The Azure Functions runtime

The core of Azure Functions is its runtime. The precursor to Azure Functions was Azure WebJobs. The code for Azure WebJobs also forms the core for Azure Functions. There are additional features and extensions added to Azure WebJobs to create Azure Functions. The Azure Functions runtime is the magic that makes functions work. Azure Functions is hosted within Azure App Service. Azure App Service loads the Azure runtime and either waits for an external event or a manual activity to occur. On arrival of a request or the occurrence of a trigger, App Service loads the incoming payload, reads the function's **function.json** file to find the function's bindings and trigger, maps the incoming data to incoming parameters, and invokes the function with parameter values. Once the function completes its execution, the value is again passed back to the Azure Functions runtime by way of an outgoing parameter defined as a binding in the **function.json** file. The function runtime returns the values to the caller. The Azure Functions runtime acts as the glue that enables the entire performance of functions.

The current Azure runtime version is ~3. It is based on the .NET Core 3.1 framework. Prior to this, version ~2 was based on the .NET Core 2.2 framework. The first version, ~1, was based on the .NET 4.7 framework.

There were substantial changes from version 1 to 2 because of changes in the underlying framework itself. However, there are very few breaking changes from version 2 to 3 and most functions written in version 2 would continue to run on version 3 as well. However, it is recommended that adequate testing is done after migrating from version 2 to 3. There were also breaking changes from version 1 to 2 with regard to triggers and bindings. Triggers and bindings are now available as extensions, with each one in a different assembly in versions 2 and 3.

Azure Functions bindings and triggers

If the Azure Functions runtime is the brain of Azure Functions, then Azure Functions bindings and triggers are its heart. Azure Functions promote loose coupling and high cohesion between services using triggers and bindings. Applications written targeting non-serverless environments implement code using imperative syntax for incoming and outgoing parameters and return values. Azure Functions uses a declarative mechanism to invoke functions using triggers and configures the flow of data using bindings.

Binding refers to the process of creating a connection between the incoming data and the Azure function along with mapping the data types. The connection could be in a single direction from the runtime to Azure Functions and vice versa or could be multi-directional—the binding can transmit data between the Azure runtime and Azure Functions in both directions. Azure Functions defines bindings declaratively.

Triggers are a special type of binding through which functions can be invoked based on external events. Apart from invoking a function, triggers also pass the incoming data, payload, and metadata to the function.

Bindings are defined in the **function.json** file as follows:

```
{
  "bindings": [
    {
      "name": "checkOut",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "checkout-items",
      "connection": "AzureWebJobsDashboard"
    },
    {
      "name": "Orders",
      "type": "table",
      "direction": "out",
      "tableName": "OrderDetails",
      "connection": "<>Connection to table storage account>>"
    }
  ],
  "disabled": false
}
```

In this example, a trigger is declared that invokes the function whenever there is a new item in the storage queue. The type is **queueTrigger**, the direction is inbound, **queueName** is **checkout-items**, and details about the target storage account connection and table name are also shown. All these values are important for the functioning of this binding. The **checkOut** name can be used within the function's code as a variable.

Similarly, a binding for the return value is declared. Here, the return value is named **Orders** and the data is the output from Azure Functions. The binding writes the return data into Azure Table Storage using the connection string provided.

Both bindings and triggers can be modified and authored using the **Integrate** tab in Azure Functions. In the backend, the **function.json** file is updated. The **checkOut** trigger is declared as shown here:

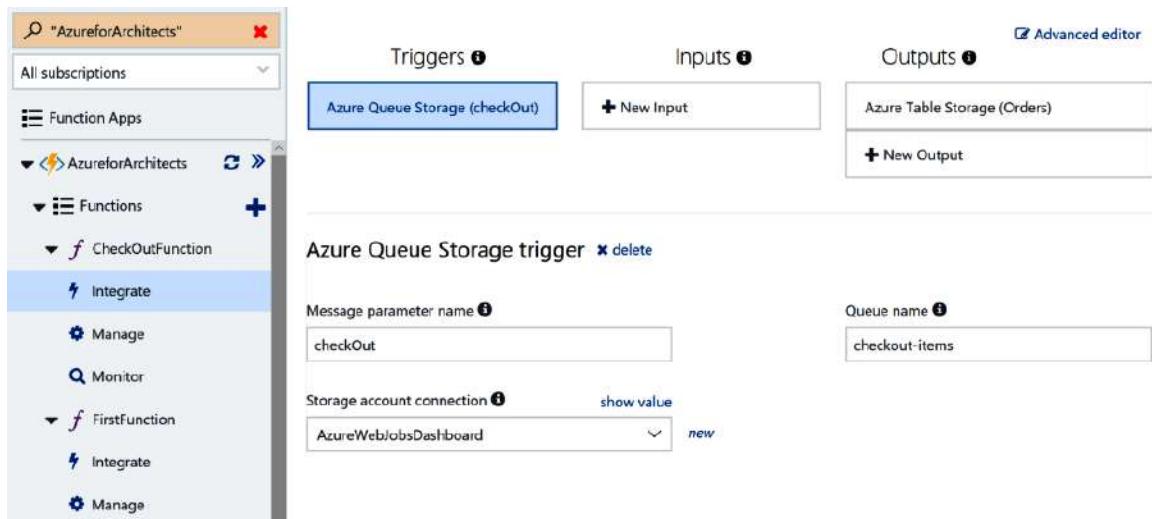


Figure 10.1: The Triggers section of the Integrate tab

The **Orders** output is shown next:

Figure 10.2: Adding output details for the storage account

The authors of Azure functions do not need to write any plumbing code to get data from multiple sources. They just decide the type of data expected from the Azure runtime. This is shown in the next code segment. Notice that the checkout is available as a string to the function. Multiple data types can be used as binding for functions. For example, a queue binding can provide the following:

- A plain old CLR (Common Language Runtime) object (POCO)
- A string
- A byte
- **CloudQueueMessage**

The author of the function can use any one of these data types, and the Azure Functions runtime will ensure that a proper object is sent to the function as a parameter. The following is a code snippet for accepting string data and the Functions runtime will encapsulate incoming data into a **string** data type before invoking the function. If the runtime is unable to cast the incoming data to a string, it will generate an exception:

```
using System;
public static void Run(string checkOut, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: { checkOut }");
}
```

It is also important to know that, in *Figure 10.2*, the storage account names are **AzureWebJobsStorage** and **AzureWebJobsDashboard**. Both of these are keys defined in the **appSettings** section and contain storage account connection strings. These storage accounts are used internally by Azure Functions to maintain its state and the status of function execution.

For more information on Azure bindings and triggers, refer to <https://docs.microsoft.com/azure/azure-functions/functions-bindings-storage-queue>.

Now that we have a fair understanding of Azure bindings and triggers, let's check out the various configuration options offered by Azure Functions.

Azure Functions configuration

Azure Functions provides configuration options at multiple levels. It provides configuration for the following:

- The platform itself
- Functions App Services

These settings affect every function contained by them. More information about these settings are available at <https://docs.microsoft.com/azure/azure-functions/functions-how-to-use-azure-function-app-settings>.

Platform configuration

Azure functions are hosted within Azure App Service, so they get all of its features. Diagnostic and monitoring logs can be configured easily using platform features. Furthermore, App Service provides options for assigning SSL certificates, using a custom domain, authentication, and authorization as part of its networking features.

Although customers are not concerned about the infrastructure, operating system, file system, or platform on which functions actually execute, Azure Functions provides the necessary tooling to peek within the underlying system and make changes. The in-portal console and the Kudu Console are the tools used for this purpose. They provide a rich editor to author Azure functions and edit their configuration.

Azure Functions, just like App Service, lets you store the configuration information within the **web.config** application settings section, which can be read on demand. Some of the platform features of function apps are shown in Figure 10.3:

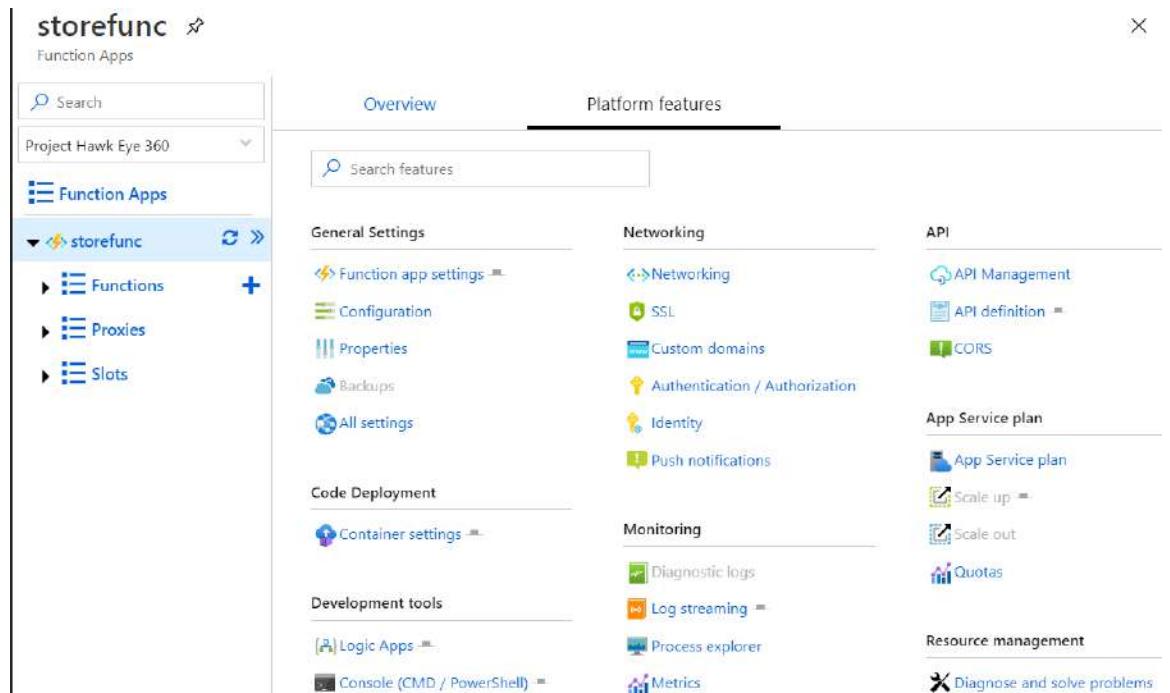


Figure 10.3: Platform features of a function app

These platform features can be used to configure authentication, custom domains, SSL, and so on. Also, the **Platform Features** tab provides an overview of the development tools that can be used with the function app. In the next section, we will take a look at the function app settings that are available in the platform features.

App Service function settings

These settings affect all functions. Application settings can be managed here. Proxies in Azure Functions can be enabled and disabled. We will discuss proxies later in this chapter. They also help in changing the edit mode of a function application and the deployment to slots:

The screenshot shows the 'Function app settings' tab in the Azure portal. At the top, there is a message: 'Application Insights is not configured. Configure Application Insights to capture function logs.' Below this, there are tabs for 'Overview', 'Platform features', and 'Function app settings'. The 'Function app settings' tab is active.

Daily Usage Quota (GB-Sec): An input field labeled 'Enter value in GB-sec' and a button labeled 'Set quota'.

Configuration: A link to 'Manage application settings'.

Runtime version: Shows 'Runtime version: 3.0.13107 (~3)'. Below it, a warning message states: 'Cannot Upgrade with Existing Functions' with the note: 'Major version upgrades can introduce breaking changes to languages and bindings. When upgrading major versions of the runtime, consider creating a new function app and migrate your functions to this new app.' There are three slot selection buttons: '~1', '~2', and the currently selected one, '~3'.

Function app edit mode: A link to 'Change the edit mode of your function app'. It shows two options: 'Read/Write' and the currently selected 'Read Only'.

Host Keys (All functions): A table listing host keys with columns: NAME, VALUE, and ACTIONS (Copy).

NAME	VALUE	ACTIONS
_master	Click to show	<input type="button" value="Copy"/>
default	Click to show	<input type="button" value="Copy"/>

Add new host key: A blue button at the bottom of the host keys section.

Figure 10.4: Function app settings

Budget is a very important aspect of the success of any project. Let's explore the various plans offered for Azure Functions.

Azure Functions cost plans

Azure Functions is based on the Azure App Service and provides a pocket-friendly model for users. There are three cost models.

A consumption plan

This is based on the per-second consumption and execution of functions. This plan calculates the cost based on the compute usage during the actual consumption and execution of the function. If a function is not executed, there is no cost associated with it. However, it does not mean that performance is compromised in this plan. Azure functions will automatically scale out and in based on demand, to ensure basic minimum performance levels are maintained. A function execution is allowed 10 minutes for completion.

One of the major drawbacks of this plan is that if there is no consumption of functions for a few seconds, the function might get cold and the next request that comes up might face a short delay in getting a response as the function is idle. This phenomenon is called a cold start. However, there are workarounds that can keep functions warm even when there are no legitimate requests. This can be done by writing a scheduled function that keeps invoking the target function to keep it warm.

A premium plan

This is a relatively new plan and provides lots of benefits compared to both App Service and a consumption plan. In this plan, there are no cold starts for Azure functions. Functions can be associated with a private network and customers can choose their own virtual machine sizes for executing functions. It provides numerous out-of-the-box facilities that were not possible previously with the other two types of plans.

An App Service plan

This plan provides functions with completely dedicated virtual machines in the backend, and so the cost is directly proportional to the cost of the virtual machine and its size. There is a fixed cost associated with this plan, even if functions are not invoked. Function code can run for as long as necessary. Although there is no time restriction, the default limit is set to 30 minutes. This can be changed by changing the value in the `hosts.json` file. Within the App Service plan, the function runtime goes idle if not used for a few minutes and can be activated only using an HTTP trigger. There is an **Always On** setting that can be used to prevent the function runtime from going idle. Scaling is either manual or based on autoscale settings.

Along with the flexible pricing option, Azure also offers various hosting options for architecture deployment.

Azure Functions destination hosts

The Azure Functions runtime can be hosted on Windows as well as on Linux hosts. PowerShell Core, Node.js, Java, Python, and .NET Core-based functions can run on both Windows as well as Linux operating systems. It is important to know which type of underlying operating system is required for the functions because this configuration setting is tied to the function app and in turn to all functions that are contained in it.

Also, it is possible to run functions within Docker containers. This is because Azure provides Docker images that have a pre-built function runtime installed in them and functions can be hosted using such images. Now, Docker images can be used to create containers within Kubernetes Pods and hosted on Azure Kubernetes Service, Azure Container Instances, or on unmanaged Kubernetes clusters. These images can be stored within Docker Hub, Azure Container Registry, or any other global as well as private image repositories.

To have a clearer understanding, let's look into some of the most prominent use cases for Azure Functions.

Azure Functions use cases

Azure Functions has many implementations. Let's have a look at some of these use cases.

Implementing microservices

Azure Functions helps in breaking down large applications into smaller, discrete functional code units. Each unit is treated independently of others and evolves in its own life cycle. Each such code unit has its own compute, hardware, and monitoring requirements. Each function can be connected to all other functions. These units are woven together by orchestrators to build complete functionality. For example, in an e-commerce application, there can be individual functions (code units), each responsible for listing catalogs, recommendations, categories, subcategories, shopping carts, checkouts, payment types, payment gateways, shipping addresses, billing addresses, taxes, shipping charges, cancellations, returns, emails, SMS, and so on. Some of these functions are brought together to create use cases for e-commerce applications, such as product browsing and checkout flow.

Integration between multiple endpoints

Azure Functions can build overall application functionality by integrating multiple functions. The integration can be based on the triggering of events or it could be on a push basis. This helps in decomposing large monolithic applications into small components.

Data processing

Azure Functions can be used for processing incoming data in batches. It can help in processing data in multiple formats, such as XML, CSV, JSON, and TXT. It can also run conversion, enrichment, cleaning, and filtering algorithms. In fact, multiple functions can be used, each doing either conversion or enrichment, cleaning or filtering. Azure Functions can also be used to incorporate advanced cognitive services, such as **optical character recognition (OCR)**, computer vision, and image manipulation and conversion. This is ideal if you want to process API responses and convert them.

Integrating legacy applications

Azure Functions can help in integrating legacy applications with newer protocols and modern applications. Legacy applications might not be using industry-standard protocols and formats. Azure Functions can act as a proxy for these legacy applications, accepting requests from users or other applications, converting the data into a format understood by a legacy application, and talking to it on protocols it understands. This opens a world of opportunity for integrating and bringing old and legacy applications into the mainstream portfolio.

Scheduled jobs

Azure Functions can be used to execute continuously or periodically for certain application functions. These application functions can perform tasks such as periodically taking backups, restoring, running batch jobs, exporting and importing data, and bulk emailing.

Communication gateways

Azure Functions can be used in communication gateways when using notification hubs, SMS, email, and so on. For example, you can use Azure Functions to send a push notification to Android and iOS devices using Azure Notification Hubs.

Azure functions are available in different types, which must be selected based on their relationship to optimizing workloads. Let's have a closer look at them.

Types of Azure functions

Azure functions can be categorized into three different types:

- **On-demand functions:** These are functions that are executed when they are explicitly called or invoked. Examples of such functions include HTTP-based functions and webhooks.
- **Scheduled functions:** These functions are like timer jobs and execute functions on fixed intervals.
- **Event-based functions:** These functions are executed based on external events. For example, uploading a new file to Azure Blob storage generates an event that could start the execution of Azure functions.

In the following section, you will learn how to create an event-driven function that will be connected to an Azure Storage account.

Creating an event-driven function

In this example, an Azure function will be authored and connected to an Azure Storage account. The Storage account has a container for holding all Blob files. The name of the Storage account is **incomingfiles** and the container is **orders**, as shown in *Figure 10.5*:

The screenshot shows the Azure Blob service interface. At the top, it displays the storage account name "incomingfiles". Below the header, there are two buttons: "Container" with a plus sign icon and "Refresh" with a circular arrow icon. The main content area provides detailed information about the storage account:

- Storage account: [incomingfiles](#)
- Status: Primary: Available
- Location: West Central US
- Subscription ([change](#)): RiteshSubscription
- Subscription ID: 9755ffce-e94b-4332-9be8-1ade15e78909

At the bottom, there is a search bar labeled "Search containers by prefix" and a table with a single row containing the word "orders".

Figure 10.5: Storage account details

Perform the following steps to create a new Azure function from the Azure portal:

1. Click on the + button beside the **Functions** menu on the left.
2. Select **In-Portal** from the resultant screen and click on the **Continue** button.
3. Select **Azure Blob Storage trigger**, as shown in Figure 10.6:

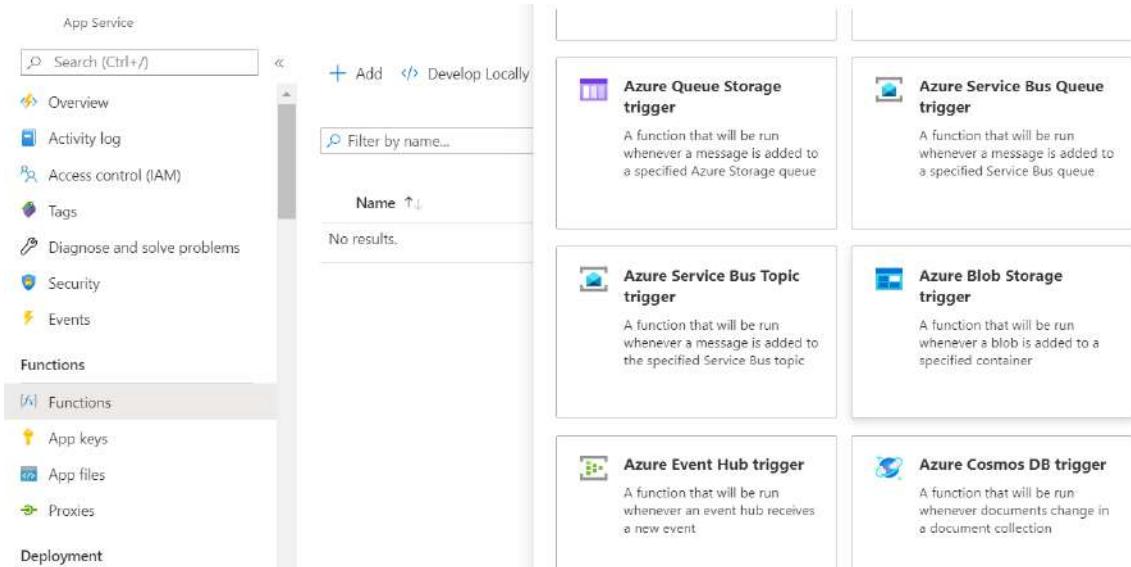


Figure 10.6: Selecting Azure Blob Storage trigger

Right now, this Azure function does not have connectivity to the Storage account. Azure functions need connection information for the Storage account, and that is available from the **Access keys** tab in the Storage account. The same information can be obtained using the Azure Functions editor environment. In fact, that environment allows the creation of a new Storage account from the same editor environment.

The Azure Blob Storage trigger can be added using the **New** button beside the **Storage account connection** input type. It allows the selection of an existing Storage account or the creation of a new Storage account. Since I already have a couple of Storage accounts, I am reusing them, but you should create a separate Azure Storage account. Selecting a Storage account will update the settings in the **appSettings** section with the connection string added to it.

Ensure that a container already exists within the Blob service of the target Azure Storage account. The path input refers to the path to the container. In this case, the **orders** container already exists within the Storage account. The **Create** button shown here will provision the new function monitoring the Storage account container:

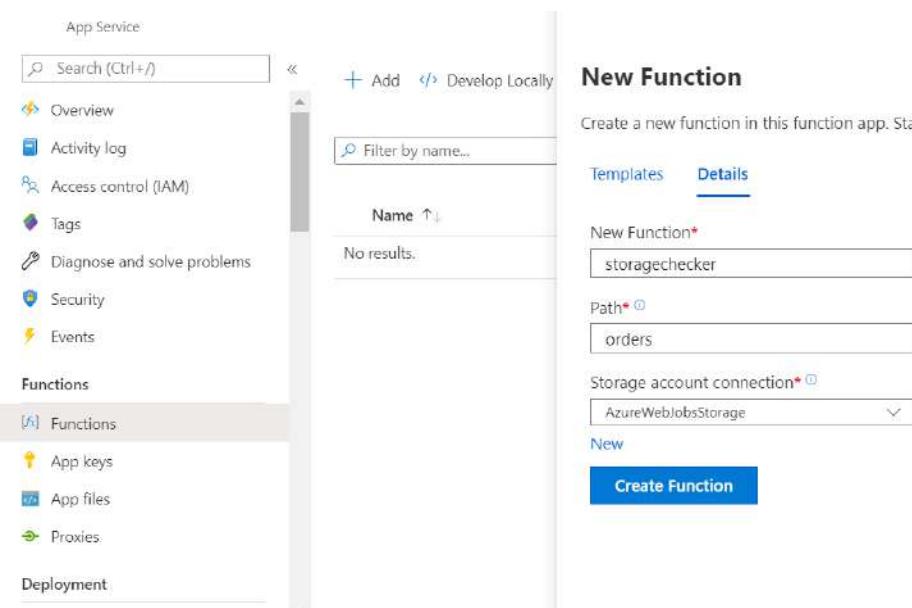


Figure 10.7: Creating a function that monitors the Storage account container

The code for the **storagerelatedfunctions** function is as follows:

```
public static void Run(Stream myBlob, TraceWriter log)
{
    log.Info($"C# Blob trigger function Processed blob\n \n Size {myBlob.Length} Bytes");
}
```

The bindings are shown here:

```
{
  "bindings": [
    {
      "name": "myBlob",
      "type": "blobTrigger",
      "direction": "in",
      "path": "orders",
      "connection": "azureforarchitead2b_STORAGE"
    }
  ],
  "disabled": false
}
```

Now, uploading any blob file to the **orders** container should trigger the function:

```
run.csx Save Run
1 public static void Run(Stream myBlob, TraceWriter log)
2 {
3     log.Info($"C# Blob trigger function Processed blob\n \n Size: {myBlob.Length}");
4 }
5
```

Logs

2017-09-20T10:24:12.504 Function started (Id=ac68a8f8-712f-4df8-975e-238b0ecf5b05)
2017-09-20T10:24:12.536 C# Blob trigger function Processed blob
Size: 2480471 Bytes
2017-09-20T10:24:12.536 Function completed (Success, Id=ac68a8f8-712f-4df8-975e-238b0ecf5b05, Dur...

Figure 10.8: C# Blob trigger function processed blob

In the next section, we will dive into Azure Function Proxies, which will help you to efficiently handle the requests and responses of your APIs.

Function Proxies

Azure Function Proxies is a relatively new addition to Azure Functions. Function Proxies helps in hiding the details of Azure functions and exposing completely different endpoints to customers. Function Proxies can receive requests on endpoints, modify the content, body, headers, and URL of the request by changing the values, and augment them with additional data and pass it internally to Azure functions. Once they get a response from these functions, they can again convert, override, and augment the response and send it back to the client.

It also helps in invoking different functions for CRUD (create, read, delete, and update) operations using different headers, thereby breaking large functions into smaller ones. It provides a level of security by not exposing the original function endpoint and also helps in changing the internal function implementation and endpoints without impacting its caller. Function Proxies helps by providing clients with a single function URL and then invoking multiple Azure functions in the backend to complete workflows. More information about Azure Function Proxies can be found at <https://docs.microsoft.com/azure/azure-functions/functions-proxies>.

In the next section, we will cover Durable Functions in detail.

Durable Functions

Durable Functions is one of the latest additions to Azure Functions. It allows architects to write stateful workflows in an Orchestrator function, which is a new function type. As a developer, you can choose to code it or use any form of IDE. Some advantages of using Durable Functions are:

- Function output can be saved to local variables and you can call other functions synchronously and asynchronously.
- The state is preserved for you.

The following is the basic mechanism for invoking Durable Functions:

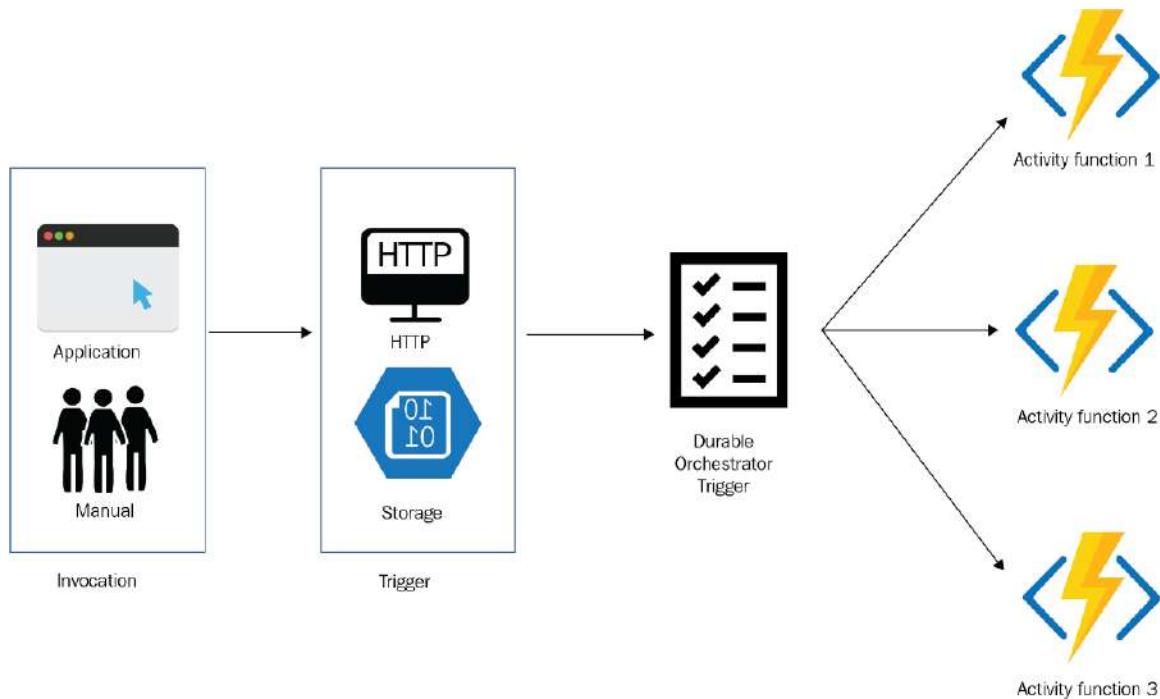


Figure 10.9: Mechanism for invoking Durable Functions

Azure Durable Functions can be invoked by any trigger provided by Azure Functions. These triggers include HTTP, Blob storage, Table Storage, Service Bus queues, and more. They can be triggered manually by someone with access to them, or by an application. Figure 10.9 shows a couple of triggers as an example. These are also known as starter Durable Functions. The starter durable functions invoke the **durable orchestrator trigger**, which contains the main logic for orchestration, and orchestrates the invocation of activity functions.

The code written within the durable orchestrator must be deterministic. This means that no matter the number of times the code is executed, the values returned by it should remain the same. The Orchestrator function is a long-running function by nature. This means it can be hydrated, state-serialized, and it goes to sleep after it calls a durable activity function. This is because it does not know when the durable activity function will complete and does not want to wait for it. When the durable activity function finishes its execution, the Orchestrator function is executed again. The function execution starts from the top and executes until it either calls another durable activity function or finishes the execution of the function. It has to re-execute the lines of code that it already executed earlier and should get the same results that it got earlier. Note that the code written within the durable orchestrator must be deterministic. This means that no matter the number of times the code is executed, the values returned by it should remain the same.

Let me explain this with the help of an example. If we use a general .NET Core datetime class and return the current date time, it will result in a new value every time we execute the function. The Durable Functions context object provides **CurrentUtcDateTime**, which will return the same datetime value during re-execution that it returned the first time.

These orchestration functions can also wait for external events and enable scenarios related to human hand-off. This concept will be explained later in this section.

These activity functions can be called with or without a retry mechanism. Durable Functions can help to solve many challenges and provides features to write functions that can do the following:

- Execute long-running functions
- Maintain state
- Execute child functions in parallel or sequence
- Recover from failure easily
- Orchestrate the execution of functions in a workflow

Now that you have a fair understanding of the inner workings of a durable function, let's explore how to create a durable function in Visual Studio.

Steps for creating a durable function using Visual Studio

The following are the steps to create a durable function:

1. Navigate to the Azure portal and click on **Resource groups** in the left menu.
2. Click on the **+Add** button in the top menu to create a new resource group.
3. Provide the resource group information on the resultant form and click on the **Create** button, as shown here:

Create a resource group

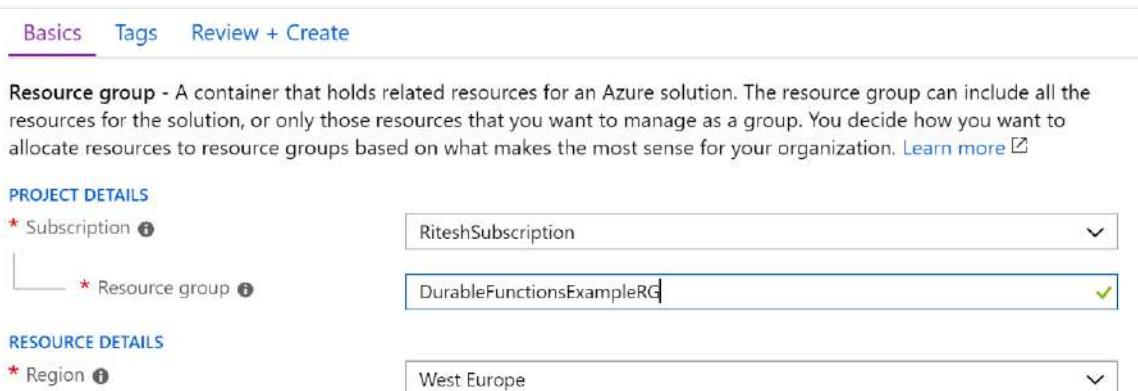


Figure 10.10: Creating a resource group

4. Navigate to the newly created resource group and add a new function app by clicking on the **+Add** button in the top menu and search for **function app** in the resultant search box.
5. Select **Function App** and click on the **Create** button. Fill in the resultant function app form and click on the **Create** button. You can also reuse the function app we created earlier.
6. Once the function app is created, we will get into our local development environment with visual studio 2019 installed on it. We will get started with Visual Studio and create a new project of type **Azure functions**, provide it with a name, and select **Azure Functions v3 (.NET core)** for **Function runtime**.
7. After the project is created, we need to add the **DurableTask** NuGet package to the project for working with Durable Functions. The version used at the time of writing this chapter is 2.2.2:



Figure 10.11: Adding a DurableTask NuGet package

8. Now, we can code our durable functions within Visual Studio. Add a new function, provide it with a name, and select the **Durable Functions Orchestration** trigger type:

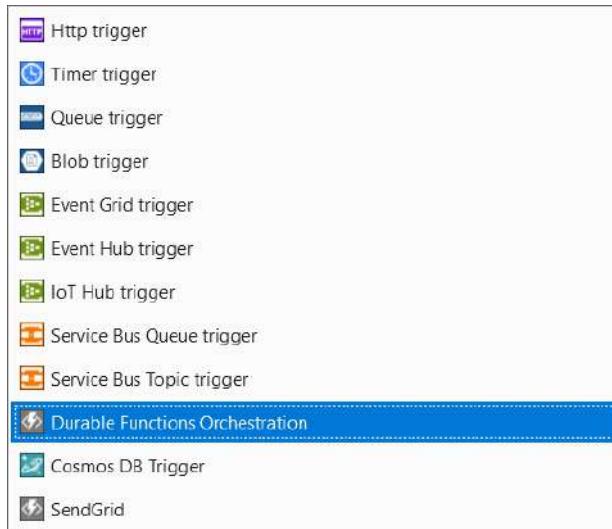


Figure 10.12: Selecting a Durable Functions Orchestration trigger

9. Visual Studio generates the boilerplate code for Durable Functions, and we are going to use it to learn about Durable Functions. Durable Functions activities are functions that are invoked by the main Orchestrator function. There is generally one main Orchestrator function and multiple Durable Functions activities. Once the extension is installed, provide a name for the function and write code that does something useful, such as sending an email or an SMS, connecting to external systems and executing logic, or executing services using their endpoints, such as cognitive services.

Visual Studio generates three sets of functions in a single line of code:

- **HttpStart:** This is the starter function. This means that it is responsible for starting the durable function orchestration. The code generated consists of an HTTP trigger starter function; however, it could be any trigger-based function, such as **BlobTrigger**, a **ServiceBus** queue, or a trigger-based function.
- **RunOrchestrator:** This is the main durable orchestration function. It is responsible for accepting parameters from the starter function and in turn, invokes multiple durable task functions. Each durable task function is responsible for a functionality and these durable tasks can be invoked either in parallel or in sequence depending on the need.
- **SayHello:** This is the durable task function that is invoked from the durable function orchestrator to do a particular job.

10. The code for the starter function (**HttpStart**) is shown next. This function has a trigger of type HTTP and it accepts an additional binding of type **DurableClient**. This **DurableClient** object helps in invoking the Orchestrator function:

```
[FunctionName("Function1_HttpStart")]
0 references
public static async Task<HttpResponseMessage> HttpStart(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    // Function input comes from the request content.
    string instanceId = await starter.StartNewAsync("Function1", null);

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

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

Figure 10.13: Code for the starter function

11. The code for the Orchestrator function (**RunOrchestrator**) is shown next. This function has a trigger of type **OrchestrationTrigger** and accepts a parameter of type **IDurableOrchestrationContext**. This context object helps in invoking durable tasks:

```
[FunctionName("Orchestrator")]
0 references
public static async Task<List<string>> RunOrchestrator(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var outputs = new List<string>();

    // Replace "hello" with the name of your Durable Activity Function.
    outputs.Add(await context.CallActivityAsync<string>("Function1_Hello", "Tokyo"));
    outputs.Add(await context.CallActivityAsync<string>("Function1_Hello", "Seattle"));
    outputs.Add(await context.CallActivityAsync<string>("Function1_Hello", "London"));

    // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
    return outputs;
}
```

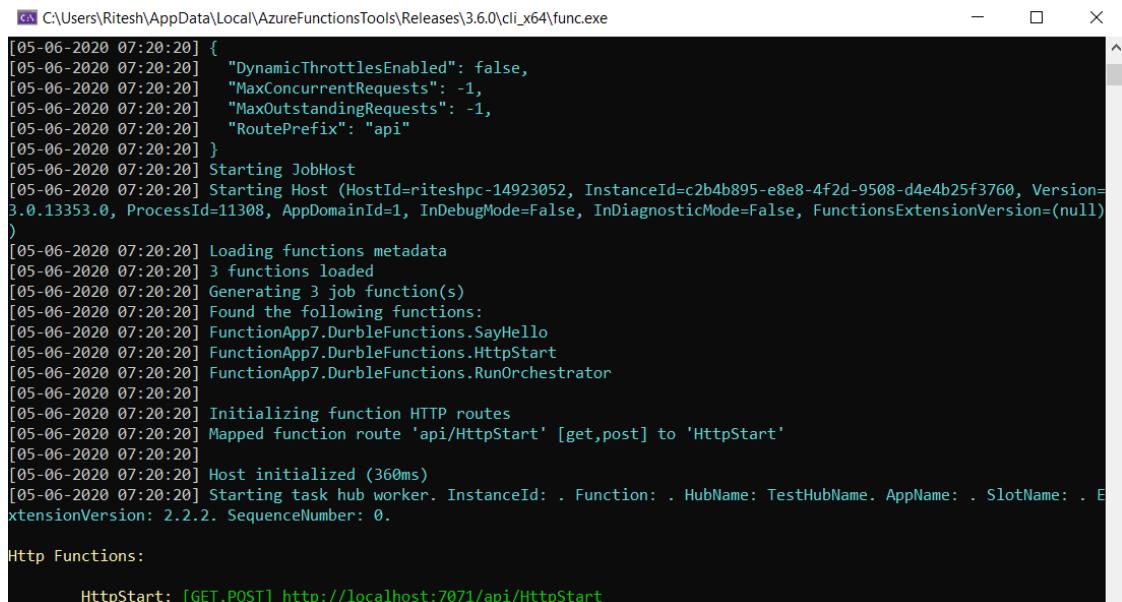
Figure 10.14: Code for orchestrator trigger function

12. The code for the durable task function (`HelloFunction`) is shown next. This function has a trigger of type `ActivityTrigger` and accepts a parameter that can be any type needed for it to execute its functionality. It has a return value of type `string` and the function is responsible for returning a string value to the orchestration function:

```
[FunctionName("HelloFunction")]
0 references
public static string SayHello([ActivityTrigger] string name, ILogger log)
{
    log.LogInformation($"Saying hello to {name}.");
    return $"Hello {name}!";
}
```

Figure 10.15: Code for the durable task function

Next, we can execute the function locally, which will start a storage emulator if one's not already started, and will provide a URL for the HTTP trigger function:



```
C:\Users\Ritesh\AppData\Local\AzureFunctionsTools\Releases\3.6.0\cli_x64\func.exe
[05-06-2020 07:20:20] {
[05-06-2020 07:20:20]   "DynamicThrottlesEnabled": false,
[05-06-2020 07:20:20]   "MaxConcurrentRequests": -1,
[05-06-2020 07:20:20]   "MaxOutstandingRequests": -1,
[05-06-2020 07:20:20]   "RoutePrefix": "api"
[05-06-2020 07:20:20] }
[05-06-2020 07:20:20] Starting JobHost
[05-06-2020 07:20:20] Starting Host (HostId=riteshpc-14923052, InstanceId=c2b4b895-e8e8-4f2d-9508-d4e4b25f3760, Version=3.0.13353.0, ProcessId=11308, AppDomainId=1, InDebugMode=False, InDiagnosticMode=False, FunctionsExtensionVersion=(null))
[05-06-2020 07:20:20] Loading functions metadata
[05-06-2020 07:20:20] 3 functions loaded
[05-06-2020 07:20:20] Generating 3 job function(s)
[05-06-2020 07:20:20] Found the following functions:
[05-06-2020 07:20:20] FunctionApp7.DurbleFunctions.SayHello
[05-06-2020 07:20:20] FunctionApp7.DurbleFunctions.HttpStart
[05-06-2020 07:20:20] FunctionApp7.DurbleFunctions.RunOrchestrator
[05-06-2020 07:20:20]
[05-06-2020 07:20:20] Initializing function HTTP routes
[05-06-2020 07:20:20] Mapped function route 'api/HttpStart' [get,post] to 'HttpStart'
[05-06-2020 07:20:20]
[05-06-2020 07:20:20] Host initialized (360ms)
[05-06-2020 07:20:20] Starting task hub worker. InstanceId: . Function: . HubName: TestHubName. AppName: . SlotName: . ExtensionVersion: 2.2.2. SequenceNumber: 0.

Http Functions:

HttpStart: [GET,POST] http://localhost:7071/api/HttpStart
```

Figure 10.16: Starting the storage emulator

We are going to invoke this URL using a tool known as **Postman** (this can be downloaded from <https://www.getpostman.com/>). We just need to copy the URL and execute it in Postman. This activity is shown in Figure 10.17:

The screenshot shows the Postman interface with a GET request to `http://localhost:7071/api/HttpStart`. The 'Params' tab is selected, showing a single parameter `Key` with value `Value`. The 'Body' tab shows a JSON response with five URLs generated by the orchestrator:

```

1 "id": "5d9943bba7c943f8acd28d1c29df9617",
2 "statusQueryGetUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/5d9943bba7c943f8acd28d1c29df9617?taskHub=TestHubName&connection=Storage&code=SopC0YBjTYCMkXhrKFJ2ChmanflyHtx8Edhqve7KElrRwa50U2mtGu==",
3 "sendEventPostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/5d9943bba7c943f8acd28d1c29df9617/raiseEvent/{eventName}?taskHub=TestHubName&connection=Storage&code=SopC0YBjTYCMkXhrKFJ2ChmanflyHtx8Edhqve7KElrRwa50U2mtGu==",
4 "terminatePostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/5d9943bba7c943f8acd28d1c29df9617/terminate?reason={text}&taskHub=TestHubName&connection=Storage&code=SopC0YBjTYCMkXhrKFJ2ChmanflyHtx8Edhqve7KElrRwa50U2mtGu==",
5 "purgeHistoryDeleteUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/5d9943bba7c943f8acd28d1c29df9617?taskHub=TestHubName&connection=Storage&code=SopC0YBjTYCMkXhrKFJ2ChmanflyHtx8Edhqve7KElrRwa50U2mtGu=="
    
```

Figure 10.17: Invoking URLs using Postman

Notice that five URLs are generated when you start the orchestrator:

- The **statusQueryGetUri** URL is used to find the current status of the orchestrator. Clicking this URL on Postman opens a new tab, and if we execute this request, it shows the status of the workflow:

The screenshot shows the Postman interface with a JSON response for the `statusQueryGetUri` URL. The response contains the following data:

```

{
  "name": "Orchestrator",
  "instanceId": "5e3af8a1f19e4a31967a1ae08fea47ba",
  "runtimeStatus": "Completed",
  "input": null,
  "customStatus": null,
  "output": [
    "Hello Tokyo!",
    "Hello Seattle!",
    "Hello London!"
  ],
  "createdTime": "2020-06-05T07:24:12Z",
  "lastUpdatedTime": "2020-06-05T07:24:42Z"
}
    
```

Figure 10.18: Current status of the orchestrator

- The **terminatePostUri** URL is used for stopping an already running Orchestrator function.
- The **sendEventPostUri** URL is used to post an event to a suspended durable function. Durable functions can be suspended if they are waiting for an external event. This URL is used in those cases.

- The **purgeHistoryDeleteUri** URL is used to delete the history maintained by Durable Functions for a particular invocation from its Table Storage account.

Now that you know how to work with Durable Functions using Visual Studio, let's cover another aspect of Azure functions: chaining them together.

Creating a connected architecture with functions

A connected architecture with functions refers to creating multiple functions, whereby the output of one function triggers another function and provides data for the next function to execute its logic. In this section, we will continue with the previous scenario of the Storage account. In this case, the output of the function being triggered using Azure Storage Blob files will write the size of the file to Azure Cosmos DB.

The configuration of Cosmos DB is shown next. By default, there are no collections created in Cosmos DB.

A collection will automatically be created when creating a function that will be triggered when Cosmos DB gets any data:

Create Azure Cosmos DB Account

The screenshot shows the 'Create Azure Cosmos DB Account' wizard. The current step is 'Project Details'. It includes a purple banner at the top with a lightning bolt icon and text about a 33% discount offer. Below the banner, there are sections for 'Subscription *', 'Resource Group *', 'Instance Details' (with fields for 'Account Name *', 'API *', 'Location *', 'Geo-Redundancy', and 'Multi-region Writes'), and an 'Apache Spark' section. At the bottom, there is a note about the discount offer and buttons for 'Review + create' and 'Next: Networking'.

Create a new Azure Cosmos DB account with multi-region writes in any region by February 29, 2020 and receive up to 33% off for the life of your account.

Project Details

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

Subscription *: Microsoft Azure Sponsorship

Resource Group *: AzureForArchitectsThirdEdition
Create new

Instance Details

Account Name *: azureforarchitectthirdedition

API *: Core (SQL)

Apache Spark: Notebooks (preview) Notebooks with Apache Spark (preview) None
Sign up for Apache Spark preview

Location *: (US) West US

Geo-Redundancy: Enable Disable

Multi-region Writes: Enable Disable

*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. Offer limited to accounts with both account locations and geo-redundancy, and applies only to multi-region writes in those same regions. Both Geo-Redundancy and Multi-region Writes must be enabled in account settings. Actual discount will vary based on number of qualifying regions selected.

Review + create Previous Next: Networking

Figure 10.19: Creating an Azure Cosmos DB account

Let's follow the below steps to retrieve data for the next function from the output of one function.

1. Create a new database, **testdb**, within Cosmos DB, and create a new collection named **testcollection** within it. You need both the database and the collection name when configuring Azure functions:

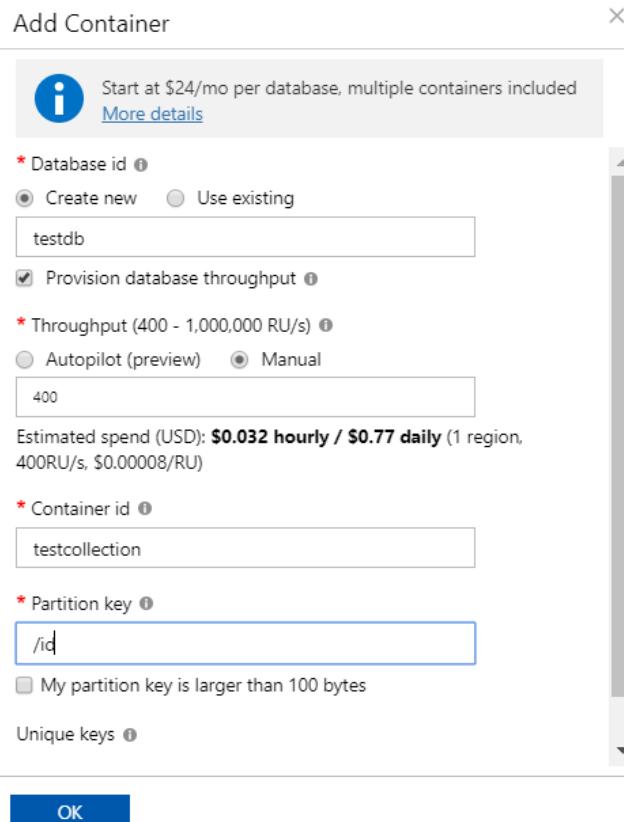


Figure 10.20: Adding a container

2. Create a new function that will have a Blob Storage trigger and output CosmosDB binding. The value returned from the function will be the size of the data for the uploaded file. This returned value will be written to Cosmos DB. The output binding will write to the Cosmos DB collection. Navigate to the **Integrate** tab and click on the **New Output** button below the **Outputs** label and select **Azure Cosmos DB**:

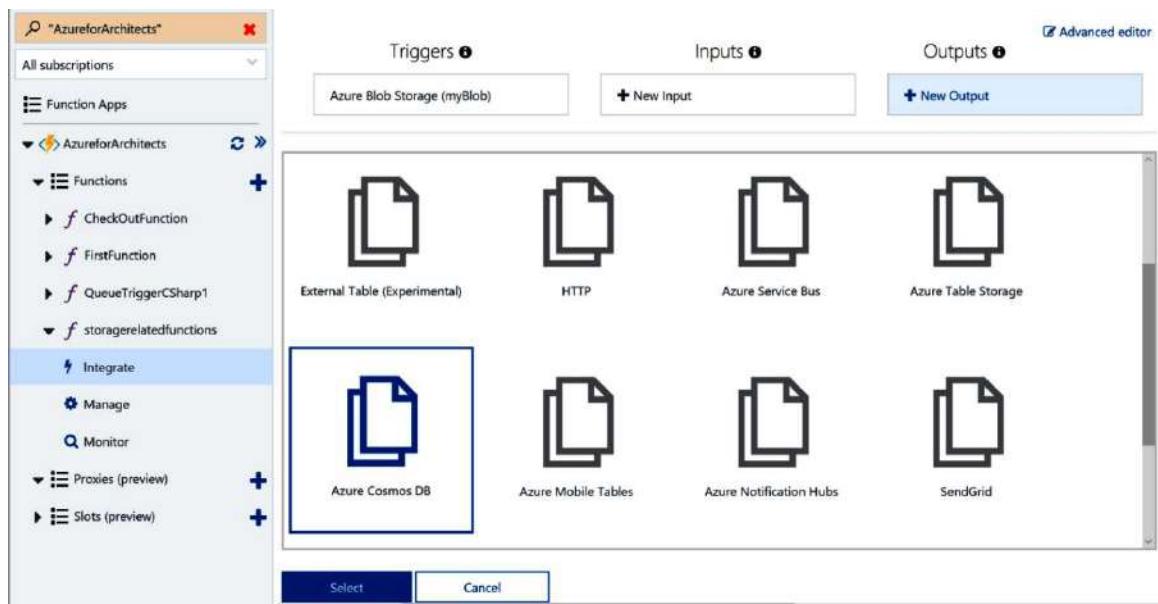


Figure 10.21: Binding output to Azure Cosmos DB

- Provide the appropriate names for the database and collection (check the checkbox to create the collection if it does not exist), click on the **New** button to select your newly created Azure Cosmos DB, and leave the parameter name as **outputDocument**:

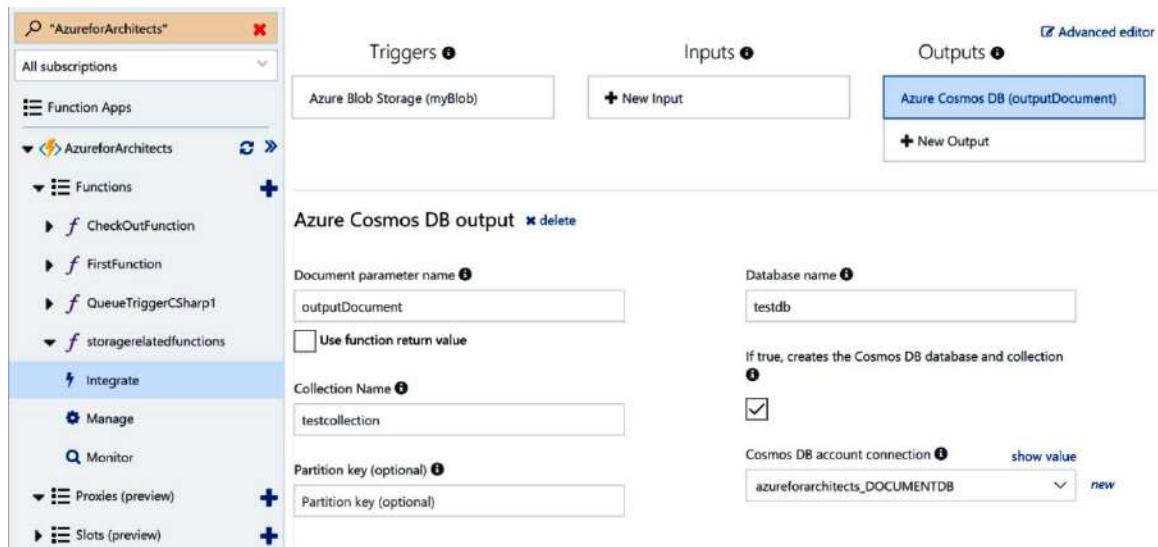
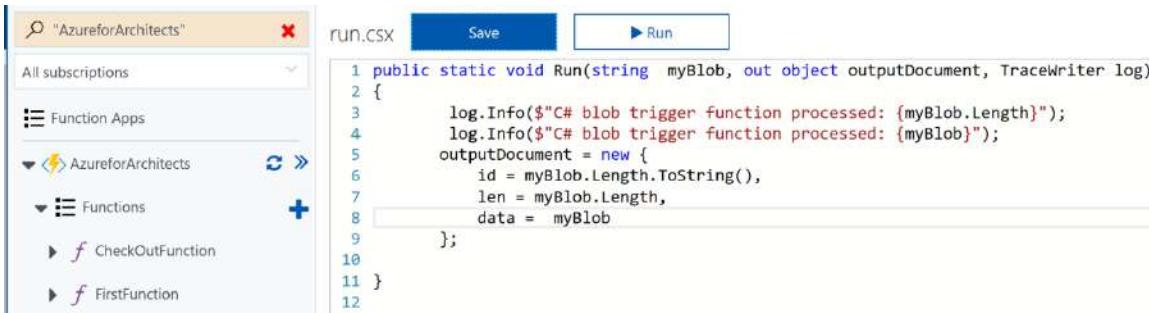


Figure 10.22: Newly created Azure Cosmos DB

4. Modify the function as shown in Figure 10.23:



```

    "AzureforArchitects" ✘
    All subscriptions
    Function Apps
    AzureforArchitects
    Functions
    CheckOutFunction
    FirstFunction
  
```

run.csx Save Run

```

1 public static void Run(string myBlob, out object outputDocument, TraceWriter log)
2 {
3     log.Info($"C# blob trigger function processed: {myBlob.Length}");
4     log.Info($"C# blob trigger function processed: {myBlob}");
5     outputDocument = new {
6         id = myBlob.Length.ToString(),
7         len = myBlob.Length,
8         data = myBlob
9     };
10
11 }
12
  
```

Figure 10.23: Modifying the function

5. Now, uploading a new file to the orders collection in the Azure Storage account will execute a function that will write to the Azure Cosmos DB collection. Another function can be written with the newly created Azure Cosmos DB account as a trigger binding. It will provide the size of files and the function can act on it. This is shown here:



Create Upload ... More Save Discard Delete Refresh Properties

testcollection

Documents

Search by Document Id or Partition Key

ID

137
1580
603

```

1 [
2   "id": "137",
3   "len": 137,
4   "data": "<?xml version='1.0' encoding='utf-8'?>`n<configuration>`n   <Ritesh>
5   id='1'>aaaa</Ritesh>`r`n   <Ritesh id='2'>bbbb</Ritesh>`r`n</configuration>"`n
  ]
  
```

Figure 10.24: Writing a trigger binding function

This section covered how the output of one function can be used to retrieve data for the next function. In the next section, you will learn about how to enable serverless eventing by understanding about Azure Event Grid.

Azure Event Grid

Azure Event Grid is a relatively new service. It has also been referred to as a serverless eventing platform. It helps with the creation of applications based on events (also known as **event-driven design**). It is important to understand what events are and how we dealt with them prior to Event Grid. An event is something that happened – that is, an activity that changed the state of a subject. When a subject undergoes a change in its state, it generally raises an event.

Events typically follow the publish/subscribe pattern (also popularly known as the **pub/sub pattern**), in which a subject raises an event due to its state change, and that event can then be subscribed to by multiple interested parties, also known as **subscribers**. The job of the event is to notify the subscribers of such changes and also provide them with data as part of its context. The subscribers can take whatever action they deem necessary, which varies from subscriber to subscriber.

Prior to Event Grid, there was no service that could be described as a real-time event platform. There were separate services, and each provided its own mechanism for handling events.

For example, Log Analytics, also known as **Operations Management Suite (OMS)**, provides an infrastructure for capturing environment logs and telemetry on which alerts can be generated. These alerts can be used to execute a runbook, a webhook, or a function. This is near to real time, but they are not completely real time. Moreover, it was quite cumbersome to trap individual logs and act on them. Similarly, there is Application Insights, which provides similar features to Log Analytics but for applications instead.

There are other logs, such as activity logs and diagnostic logs, but again, they rely on similar principles as other log-related features. Solutions are deployed on multiple resource groups in multiple regions, and events raised from any of these should be available to the resources that are deployed elsewhere.

Event Grid removes all barriers, and as a result, events can be generated by most resources (they are increasingly becoming available), and even custom events can be generated. These events can then be subscribed to by any resource, in any region, and in any resource group within the subscription.

Event Grid is already laid down as part of the Azure infrastructure, along with data centers and networks. Events raised in one region can easily be subscribed to by resources in other regions, and since these networks are connected, it is extremely efficient for the delivery of events to subscribers.

Event Grid

Event Grid lets you create applications with event-based architecture. There are publishers of events and there are consumers of events; however, there can be multiple subscribers for the same event.

The publisher of an event can be an Azure resource, such as Blob storage, **Internet of Things (IoT) hubs**, and many others. These publishers are also known as event sources. These publishers use out-of-the-box Azure topics to send their events to Event Grid. There is no need to configure either the resource or the topic. The events raised by Azure resources are already using topics internally to send their events to Event Grid. Once the event reaches the grid, it can be consumed by the subscribers.

The subscribers, or consumers, are resources who are interested in events and want to execute an action based on these events. These subscribers provide an event handler when they subscribe to the topic. The event handlers can be Azure functions, custom webhooks, logic apps, or other resources. Both the event sources and subscribers that execute event handlers are shown in Figure 10.25:

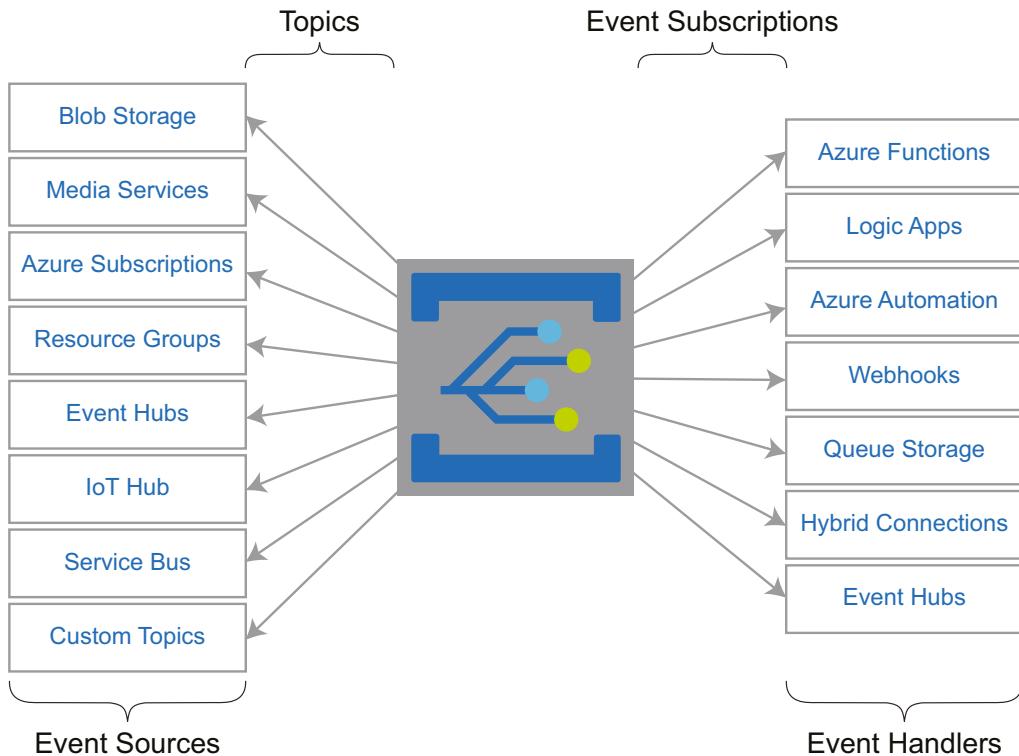


Figure 10.25: The Event Grid architecture

When an event reaches a topic, multiple event handlers can be executed simultaneously, each taking its own action.

It is also possible to raise a custom event and send a custom topic to Event Grid. Event Grid provides features for creating custom topics, and these topics are automatically attached to Event Grid. These topics know the storage for Event Grid and automatically send their messages to it. Custom topics have two important properties, as follows:

- **An endpoint:** This is the endpoint of the topic. Publishers and event sources use this endpoint to send and publish their events to Event Grid. In other words, topics are recognized using their endpoints.
- **Keys:** Custom topics provide a couple of keys. These keys enable security for the consumption of the endpoint. Only publishers with these keys can send and publish their messages to Event Grid.

Each event has an event type and it is recognized by it. For example, Blob storage provides event types, such as **blobAdded** and **blobDeleted**. Custom topics can be used to send a custom-defined event, such as a custom event of the **KeyVaultSecretExpired** type.

On the other hand, subscribers have the ability to accept all messages or only get events based on filters. These filters can be based on the event type or other properties within the event payload.

Each event has at least the following five properties:

- **id**: This is the unique identifier for the event.
- **eventType**: This is the event type.
- **eventTime**: This is the date and time when the event was raised.
- **subject**: This is a short description of the event.
- **data**: This is a dictionary object and contains either resource-specific data or any custom data (for custom topics).

Currently, Event Grid's functionalities are not available with all resources; however, Azure is continually adding more and more resources with Event Grid functionality.

To find out more about the resources that can raise events related to Event Grid and handlers that can handle these events, please go to <https://docs.microsoft.com/azure/event-grid/overview>.

Resource events

In this section, the following steps are provided to create a solution in which events that are raised by Blob storage are published to Event Grid and ultimately routed to an Azure function:

1. Log in to the Azure portal using the appropriate credentials and create a new Storage account in an existing or a new resource group. The Storage account should be either **StorageV2** or **Blob storage**. As demonstrated in Figure 10.26, Event Grid will not work with **StorageV1**:

Create storage account

Basics Advanced Tags Review + create

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. [Learn more](#)

PROJECT DETAILS

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

* Subscription	<input type="text" value="RiteshSubscription"/>	▼
* Resource group	<input type="text" value="azureclitest"/>	▼
	Create new	

INSTANCE DETAILS

The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model instead. [Choose classic deployment model](#)

* Storage account name ?	<input type="text" value="storageforeventgrid"/>	✓
* Location	<input type="text" value="West Europe"/>	▼
Performance ?	<input checked="" type="radio"/> Standard <input type="radio"/> Premium	
Account kind ?	<input type="text" value="StorageV2 (general purpose v2)"/>	▼
Replication ?	<input type="text" value="Read-access geo-redundant storage (RA-GRS)"/>	▼
Access tier (default) ?	<input type="radio"/> Cool <input checked="" type="radio"/> Hot	

[Review + create](#)

[Previous](#)

[Next : Advanced >](#)

Figure 10.26: Creating a new storage account

2. Create a new function app or reuse an existing function app to create an Azure function. The Azure function will be hosted within the function app.
3. Create a new function using the **Azure Event Grid trigger** template. Install the **Microsoft.Azure.WebJobs.Extensions.EventGrid** extension if it's not already installed, as shown in Figure 10.27:

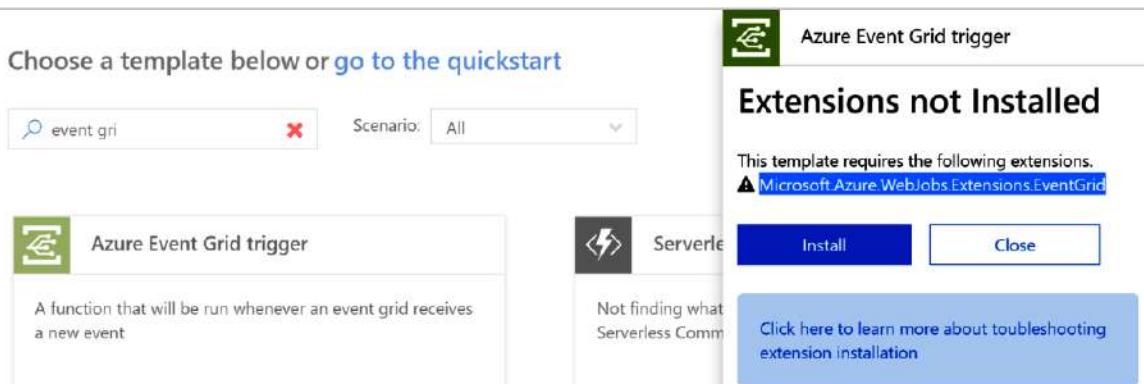


Figure 10.27: Installing extensions for an Azure Event Grid trigger

4. Name the **StorageEventHandler** function and create it. The following default generated code will be used as the event handler:

The screenshot shows the 'run.csx' code editor with the following C# code:

```

1 #r "Microsoft.Azure.EventGrid"
2 using Microsoft.Azure.EventGrid.Models;
3
4 public static void Run(EventGridEvent eventGridEvent, ILogger log)
5 {
6     log.LogInformation(eventGridEvent.Data.ToString());
7 }
8

```

Buttons for 'Save', 'Run', and 'Add Event Grid subscription' are visible above the code editor.

Figure 10.28: Event handler code

The subscription to Storage events can be configured either from the Azure Functions **user interface (UI)** by clicking on **Add Event Grid subscription**, or from the storage account itself.

5. Click on the **Add Event Grid subscription** link in the Azure Functions UI to add a subscription to the events raised by the storage account created in the previous step. Provide a meaningful name for the subscription, and then choose **Event Schema** followed by **Event Grid Schema**. Set **Topic Types** as **Storage Accounts**, set an appropriate **Subscription**, and the resource group containing the storage account:

The screenshot shows the 'Create Event Subscription' page. At the top, there's a breadcrumb navigation: Home > DurableFunctionDemoApp - StorageEventHandler > Create Event Subscription. Below that is a section titled 'Create Event Subscription' with a 'Event Grid' icon. There are three tabs: 'Basic' (selected), 'Filters', and 'Additional Features'. A note below says: 'Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource.' with a 'Learn more' link.

EVENT SUBSCRIPTION DETAILS

Name	StorageEventSubscription	<input checked="" type="checkbox"/>
Event Schema	Event Grid Schema	<input type="button" value="▼"/>

TOPIC DETAILS

Pick a topic resource for which events should be generated and pushed. [Learn more](#)

Topic Types	Storage Accounts	<input type="button" value="▼"/>
Subscription	RiteshSubscription	<input type="button" value="▼"/>
Resource Group	azureclitest	<input type="button" value="▼"/>
Resource	someoddstoreacc1 storageforeventgrid someoddstoreacc	<input type="button" value="^"/>

EVENT TYPES

Pick which event types get pushed to your d

<input checked="" type="checkbox"/> Subscribe to all event types
--

Figure 10.29: Creating an Event Grid subscription

Ensure that the **Subscribe to all event types** checkbox is checked and click on the **Create** button (it should be enabled as soon as a storage account is selected).

6. If we now navigate to the storage account in the Azure portal and click on the **Events** link in the left-hand menu, the subscription for the storage account should be visible:

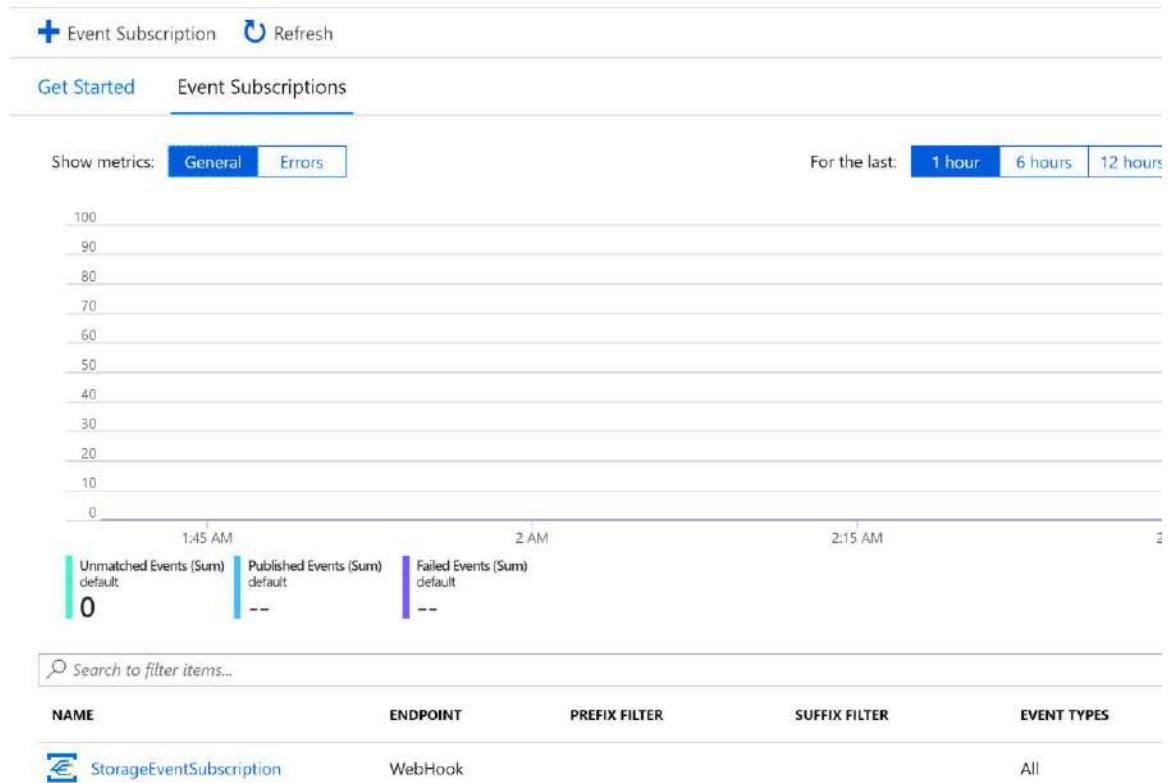


Figure 10.30: Event subscription list

7. Upload a file to the Blob storage after creating a container, and the Azure function should be executed. The upload action will trigger a new event of the **blobAdded** type and send it to the Event Grid topic for storage accounts. As shown in *Figure 10.31*, the subscription is already set to get all the events from this topic, and the function gets executed as part of the event handler:

The screenshot shows the Azure Functions developer tools interface. At the top, there's a navigation bar with 'run.csx', 'Save', 'Run', and 'Add Event Grid subscription'. Below the navigation bar is the function code in run.csx:

```
1 #r "Microsoft.Azure.EventGrid"
2 using Microsoft.Azure.EventGrid.Models;
3
4 public static void Run(EventGridEvent eventGridEvent, ILogger log)
5 {
6     log.LogInformation(eventGridEvent.Data.ToString());
7 }
8
```

Below the code editor are two tabs: 'Logs' (selected) and 'Console'. The 'Logs' tab displays the following execution logs:

```
2019-01-10T07:48:40 welcome, you are now connected to log-streaming service.
2019-01-10T07:48:51.428 [Information] Executing 'Functions.StorageEventHandler' (Reason='EventGrid trigger fired at 2019-01-10T07:48:51.4287500+00:00', Id=5c4e2121-f903-40bb-a45a-e49e674348ec)
2019-01-10T07:48:51.429 [Information]
    "api": "PutBlocklist",
    "clientRequestId": "010e48eb-a88e-4a48-becl-44da8e8b503c",
    "requestId": "a0217c1f-f01e-0032-53b8-a83345000000",
    "eTag": "0x80676000FA07895",
    "contentType": "application/pdf",
    "contentLength": 126822,
    "blobType": "BlockBlob",
    "url": "https://storageforeventgrid.blob.core.windows.net/filecontainer/sup",
    "sequencer": "00000000000000000000000000000000042d800000000000002d532",
    "leaseName": null
```

Figure 10.31: Triggering a new event

In this section, you learned how events raised by Blob storage can be routed to an Azure function. In the next section, you will learn how to leverage custom events.

Custom events

In this example, instead of using out-of-box resources to generate events, custom events will be used. We will use PowerShell to create this solution and reuse the same Azure function that was created in the last exercise as the handler:

1. Log in and connect to your Azure subscription of choice using **Login-AzAccount** and **Set-AzContext** cmdlet.
 2. The next step is to create a new Event Grid topic in Azure in a resource group. The **New-AzEventGridTopic** cmdlet is used to create a new topic:

```
New-AzEventGridTopic -ResourceGroupName CustomEventGridDemo -Name "KeyVaultAssetsExpiry" -Location "West Europe"
```

3. Once the topic is created, its endpoint URL and key should be retrieved as they are needed to send and publish the event to it. The **Get-AzEventGridTopic** and **Get-AzEventGridTopicKey** cmdlets are used to retrieve these values. Note that **Key1** is retrieved to connect to the endpoint:

```
$topicEndpoint = (Get-AzEventGridTopic -ResourceGroupName containers -Name KeyVaultAssetsExpiry).Endpoint
```

```
$keys = (Get-AzEventGridTopicKey -ResourceGroupName containers -Name KeyVaultAssetsExpiry).Key1
```

4. A new hash table is created with all five important Event Grid event properties. A new **id** property is generated for the ID, the **subject** property is set to **Key vault Asset Expiry**, **eventType** is set to **Certificate Expiry**, **eventTime** is set to the current time, and **data** contains information regarding the certificate:

```
$eventgridDataMessage = @{
    id = [System.guid]::NewGuid()
    subject = "Key Vault Asset Expiry"
    eventType = "Certificate Expiry"
    eventTime = [System.DateTime]::UtcNow
    data = @{
        CertificateThumbprint = "sdfervdserwetsgfhgdg"
        ExpiryDate = "1/1/2019"
        Createdon = "1/1/2018"
    }
}
```

5. Since Event Grid data should be published in the form of a JSON array, the payload is converted in the JSON array. The "[,]" square brackets represent a JSON array:

```
$finalBody = "[" + $(ConvertTo-Json $eventgridDataMessage) + "]"
```

6. The event will be published using the HTTP protocol, and the appropriate header information has to be added to the request. The request is sent using the application/JSON content type and the key belonging to the topic is assigned to the **aeg-sas-key** header. It is mandatory to name the header and key set to **aeg-sas-key**:

```
$header = @{
    "contentType" = "application/json"
    "aeg-sas-key" = $keys}
```

- A new subscription is created to the custom topic with a name, the resource group containing the topic, the topic name, the webhook endpoint, and the actual endpoint that acts as the event handler. The event handler in this case is the Azure function:

```
New-AzEventGridSubscription -TopicName KeyVaultAssetsExpiry
-EventSubscriptionName "customtopicsubscriptionautocar" -ResourceGroupName
CustomEventGridDemo -EndpointType webhook '
-Endpoint "https://durablefunctiondemoapp.
azurewebsites.net/runtime/webhooks/
EventGrid?functionName=StorageEventHandler&code=0aSw6sxvtFmafXHvt7i0w/
Dsb8o1M9RKKagzVchTUKwe9EIkz14mCg=="
-Verbose
```

The URL of the Azure function is available from the **Integrate** tab, as shown in Figure 10.31:

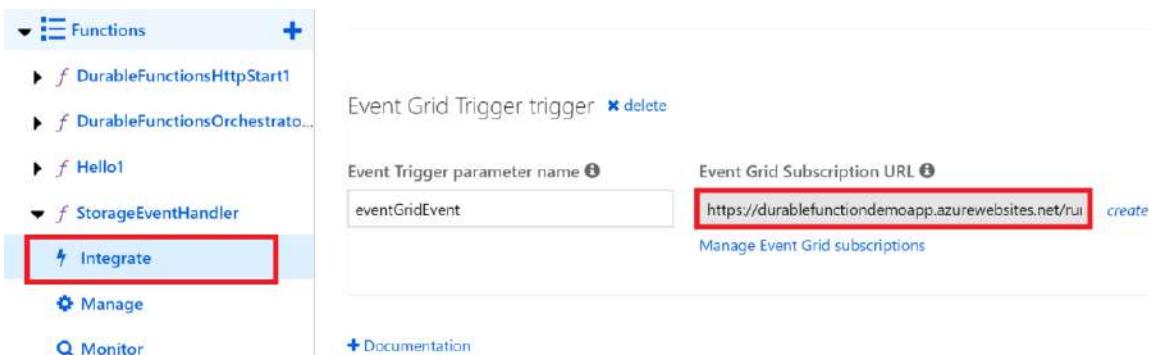


Figure 10.32: Event Grid Subscription URL in the Integrate tab

- By now, both the subscriber (event handler) and the publisher have been configured. The next step is to send and publish an event to the custom topic. The event data was already created in the previous step and, by using the **Invoke-WebRequest** cmdlet, the request is sent to the endpoint along with the body and the header:

```
Invoke-WebRequest -Uri $topicEndpoint -Body $finalBody -Headers $header
-Method Post
```

The API call will trigger the event and the Event Grid will message the endpoint we configured, which is the function app. With this activity, we are winding up this chapter.

Summary

The evolution of functions from traditional methods has led to the design of the loosely coupled, independently evolving, self-reliant serverless architecture that was only a concept in earlier days. Functions are a unit of deployment and provide an environment that does not need to be managed by the user at all. All they have to care about is the code written for the functionality. Azure provides a mature platform for hosting functions and integrating them seamlessly, based on events or on demand. Nearly every resource in Azure can participate in an architecture composed of Azure functions. The future is functions, as more and more organizations want to stay away from managing infrastructures and platforms. They want to offload this to cloud providers. Azure Functions is an essential feature to master for every architect dealing with Azure.

This chapter went into the details of Azure Functions, Functions as a Service, Durable Functions, and Event Grid. The next chapter will focus on Azure Logic Apps, and we will build a complete end-to-end solution combining multiple serverless services along with other Azure services, such as Azure Key Vault and Azure Automation.

11

Azure solutions using Azure Logic Apps, Event Grid, and Functions

This chapter continues from the previous chapter and will go into further depth about serverless services available within Azure. In the previous chapter, you learned in detail about Azure Functions, functions as a service, Durable Functions, and Event Grid. Going forward, this chapter will focus on understanding Logic Apps and then move on to creating a complete end-to-end serverless solution that combines multiple serverless and other kinds of services, such as Key Vault and Azure Automation.

In this chapter, we will further explore Azure services by covering the following topics:

- Azure Logic Apps
- Creating an end-to-end solution using serverless technologies

Azure Logic Apps

Logic Apps is a serverless workflow offering from Azure. It has all the features of serverless technologies, such as consumption-based costing and unlimited scalability. Logic Apps helps us to build a business process and workflow solution with ease using the Azure portal. It provides a drag-and-drop UI to create and configure workflows.

Using Logic Apps is the preferred way to integrate services and data, create business projects, and create a complete flow of logic. There are several important concepts that should be understood before building a logic app.

Activities

An activity is a single unit of work. Examples of activities include converting XML to JSON, reading blobs from Azure Storage, and writing to a Cosmos DB document collection. Logic Apps provides a workflow definition consisting of multiple co-related activities in a sequence. There are two types of activity in Logic Apps:

- **Trigger:** A trigger refers to the initiation of an activity. All logic apps have a single trigger that forms the first activity. It is the trigger that creates an instance of the logic app and starts the execution. Examples of triggers are the arrival of Event Grid messages, an email, an HTTP request, or a schedule.
- **Actions:** Any activity that is not a trigger is a step activity, and each of them is responsible to perform one task. Steps are connected to each other in a workflow. Each step will have an action that needs to be completed before going to the next step.

Connectors

Connectors are Azure resources that help connect a logic app to external services. These services can be in the cloud or on-premises. For example, there is a connector for connecting logic apps to Event Grid. Similarly, there is another connector to connect to Office 365 Exchange. Almost all types of connectors are available in Logic Apps, and they can be used to connect to services. Connectors contain connection information and also logic to connect to external services using this connection information.

The entire list of connectors is available at <https://docs.microsoft.com/connectors>.

Now that you know about connectors, you need to understand how they can be aligned in a step-by-step manner to make the workflow work as expected. In the next section, we will be focusing on the workings of a logic app.

The workings of a logic app

Let's create a Logic Apps workflow that gets triggered when an email account receives an email. It replies to the sender with a default email and performs sentiment analysis on the content of the email. For sentiment analysis, the Text Analytics resource from Cognitive Services should be provisioned before creating the logic app:

1. Navigate to the Azure portal, log in to your account, and create a **Text Analytics** resource in a resource group. Text Analytics is part of Cognitive Services and has features such as sentiment analysis, key phrase extraction, and language detection. You can find the service in the Azure portal, as shown in *Figure 11.1*:

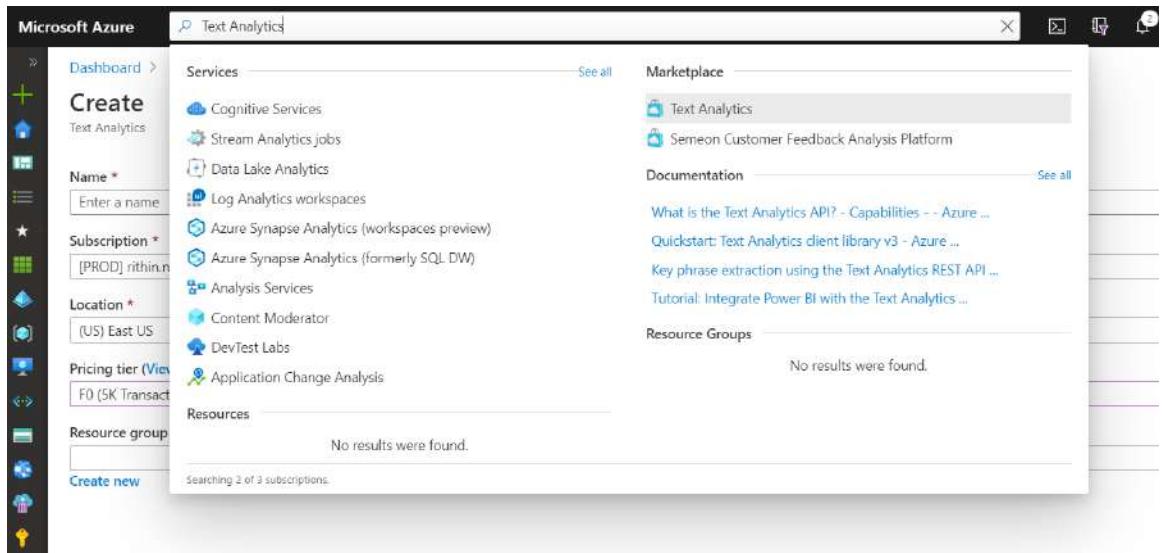


Figure 11.1: Navigating to the Text Analytics service from the Azure portal

2. Provide the **Name**, **Location**, **Subscription**, **Resource group**, and **Pricing tier** values. We'll be using the free tier (F0 tier) of this service for this demo.
3. Once the resource is provisioned, navigate to the **Overview** page, and copy the endpoint URL. Store it in a temporary location. This value will be required when configuring the logic app.

4. Navigate to the **Keys** page and copy the value from **Key 1** and store it in a temporary location. This value will be needed when configuring the logic app.
5. The next step is to create a logic app. To create a logic app, navigate to the resource group in the Azure portal in which the logic app should be created. Search for Logic App and create it by providing **Name**, **Location**, **Resource group**, and **Subscription** values.
6. After the logic app has been created, navigate to the resource, click on **Logic app designer** in the left-hand menu, and then select the **When a new email is received in Outlook.com** template to create a new workflow. The template provides a head start by adding boilerplate triggers and activities. This will add an Office 365 Outlook trigger automatically to the workflow.
7. Click on the **Sign in** button on the trigger; it will open a new Internet Explorer window. Then, sign in to your account. After successfully signing in, a new Office 365 mail connector will be created, containing the connection information to the account.
8. Click on the **Continue** button and configure the trigger with a 3-minute poll frequency, as shown in *Figure 11.2*:

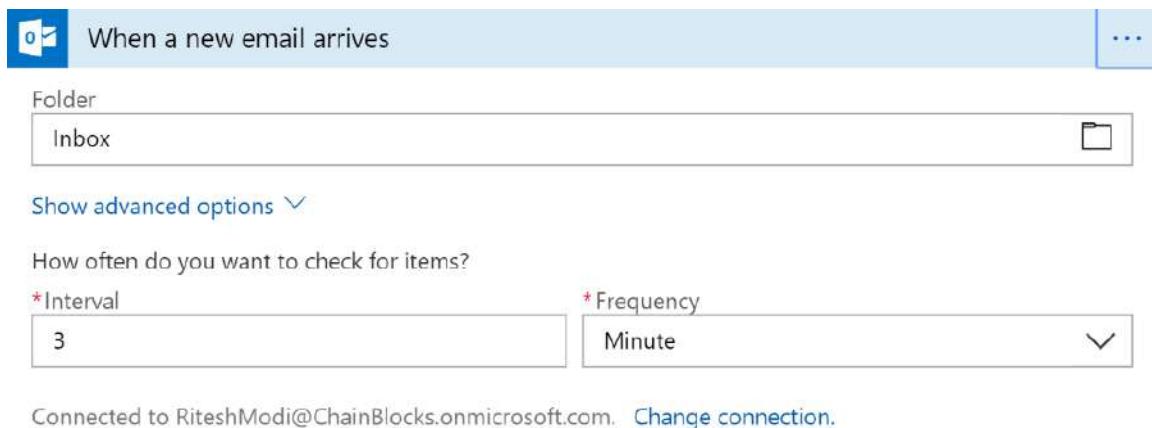


Figure 11.2: Configuring the trigger with a 3-minute poll frequency

9. Click on **Next step** to add another action and type the keyword **variable** in the search bar. Then, select the **Initialize variable** action, as demonstrated in *Figure 11.3*:

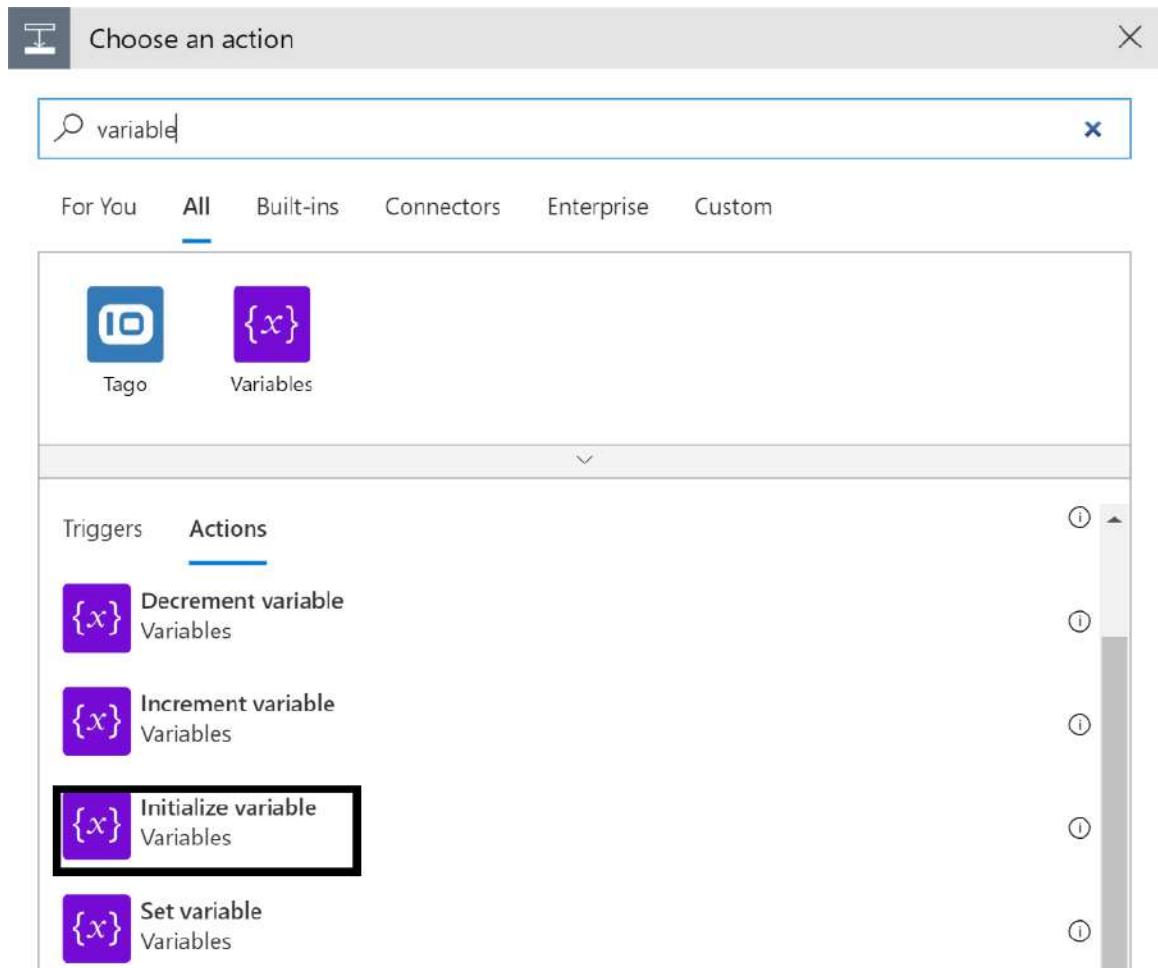


Figure 11.3: Adding the Initialize variable action

10. Next, configure the variable action. When the **Value** box is clicked on, a pop-up window appears that shows **Dynamic content** and **Expression**. Dynamic content refers to properties that are available to the current action and are filled with runtime values from previous actions and triggers. Variables help in keeping workflows generic. From this window, select **Body** from **Dynamic content**:

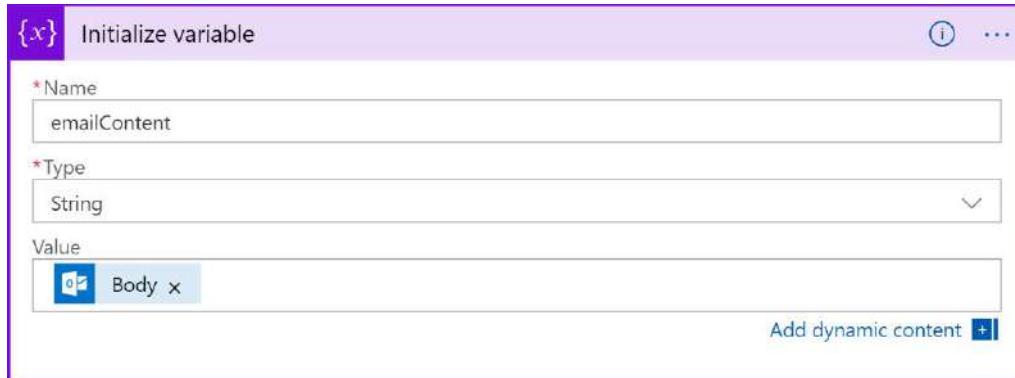


Figure 11.4: Configuring the variable action

11. Add another action by clicking on **Add step**, typing **outlook** in the search bar, and then selecting the **Reply to email** action:

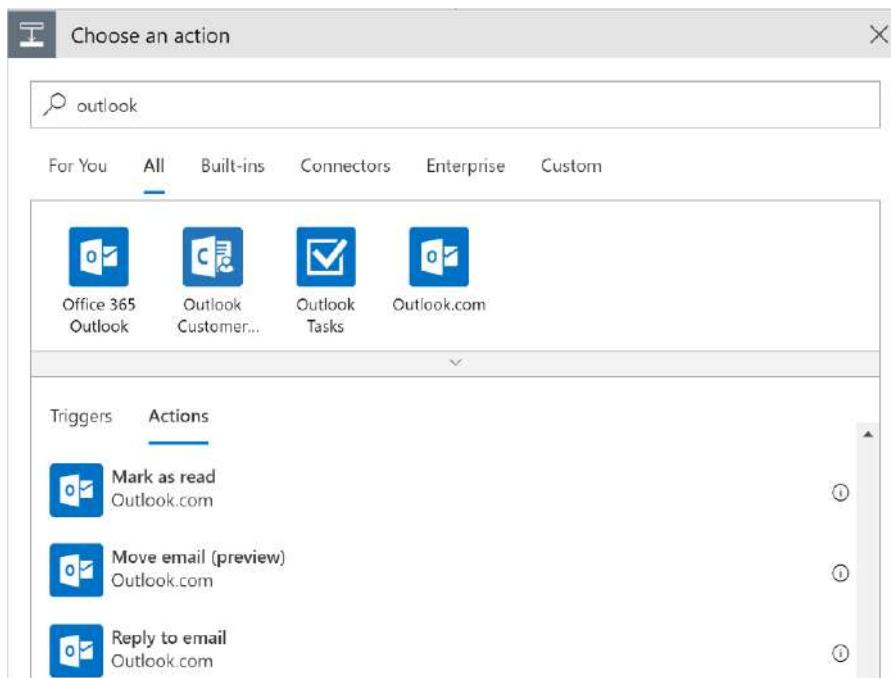


Figure 11.5: Adding the Reply to email action

12. Configure the new action. Ensure that **Message Id** is set with the dynamic content, **Message Id**, and then type the reply in the **Comment** box that you'd like to send to the recipient:

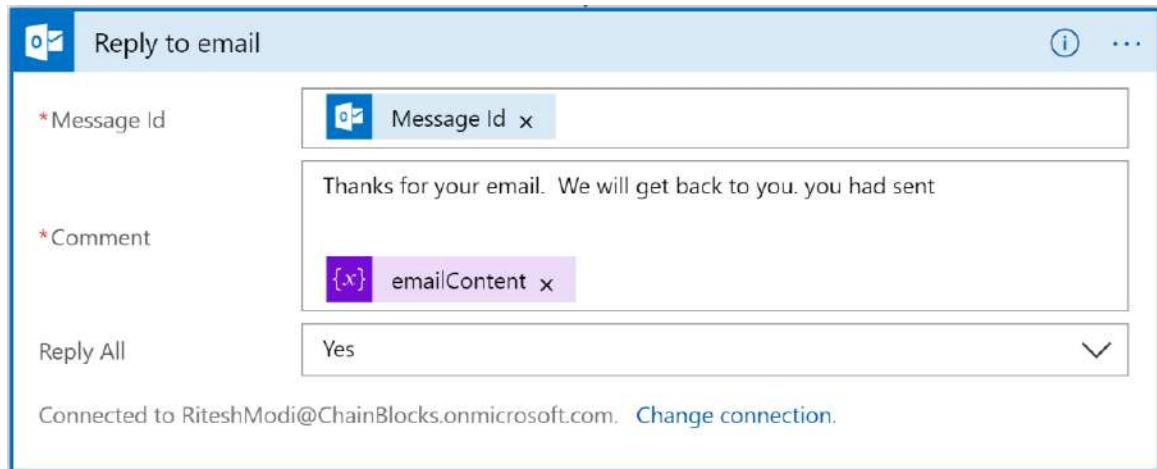


Figure 11.6: Configuring the Reply to email action

13. Add another action, type **text analytics** in the search bar, and then select **Detect Sentiment (preview)**:

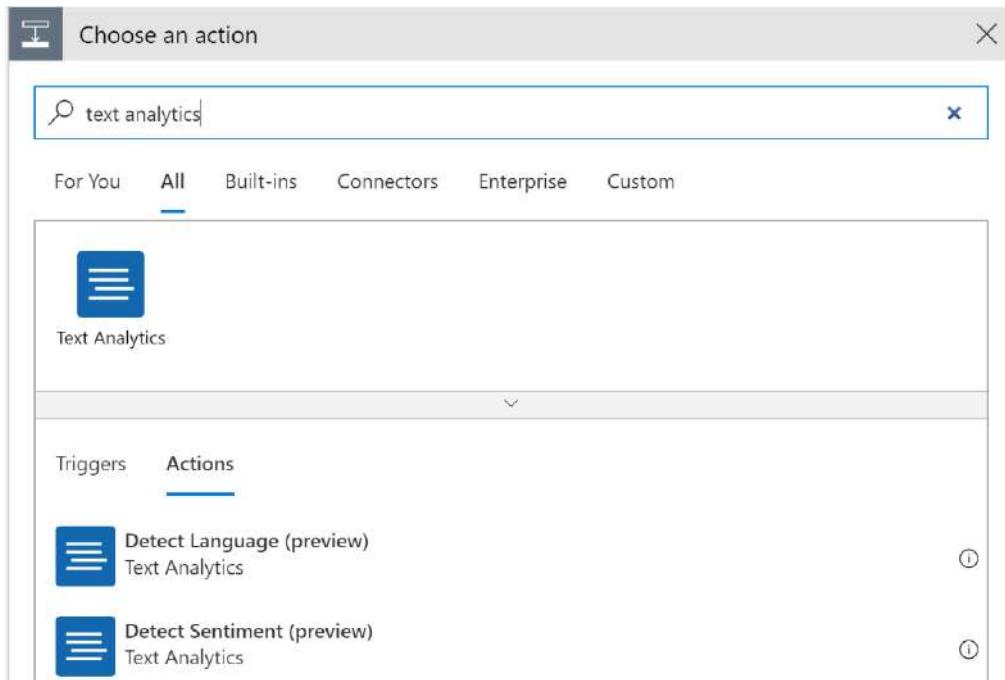


Figure 11.7: Adding the Detect Sentiment (preview) action

14. Configure the sentiment action as shown in *Figure 11.8*—both the endpoint and key values should be used here. Now click on the **Create** button, as demonstrated in *Figure 11.8*:

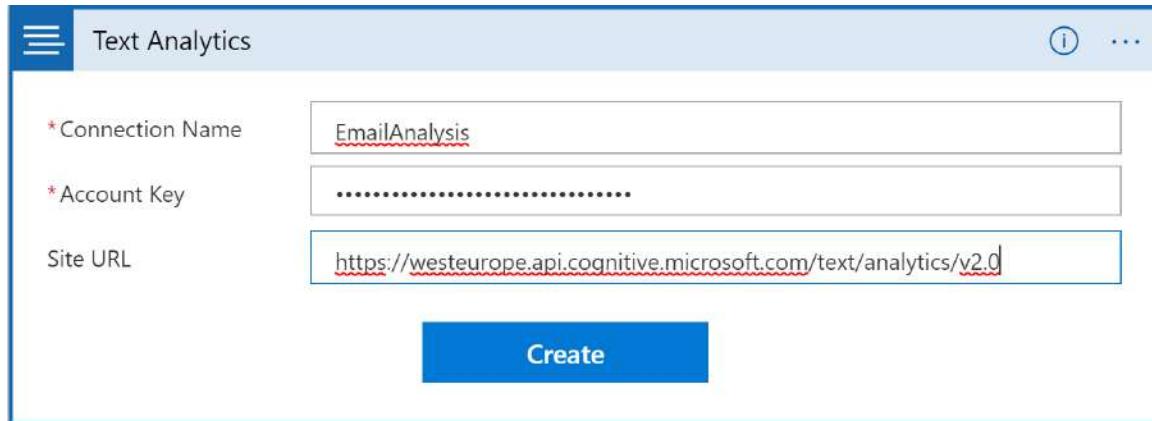


Figure 11.8: Configuring the Detect Sentiment (preview) action

15. Provide the text to the action by adding dynamic content and selecting the previously created variable, **emailContent**. Then, click on **Show advanced options** and select **en** for **Language**:

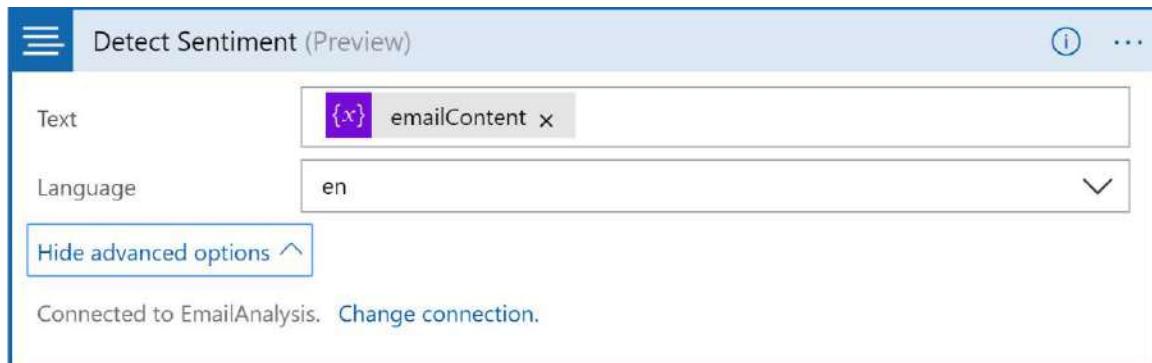


Figure 11.9: Selecting the language for the sentiment action

16. Next, add a new action by selecting **Outlook**, and then select **Send an email**. This action sends the original recipient the email content with the sentiment score in its subject. It should be configured as shown in *Figure 11.10*. If the score is not visible in the dynamic content window, click on the **See more** link beside it:

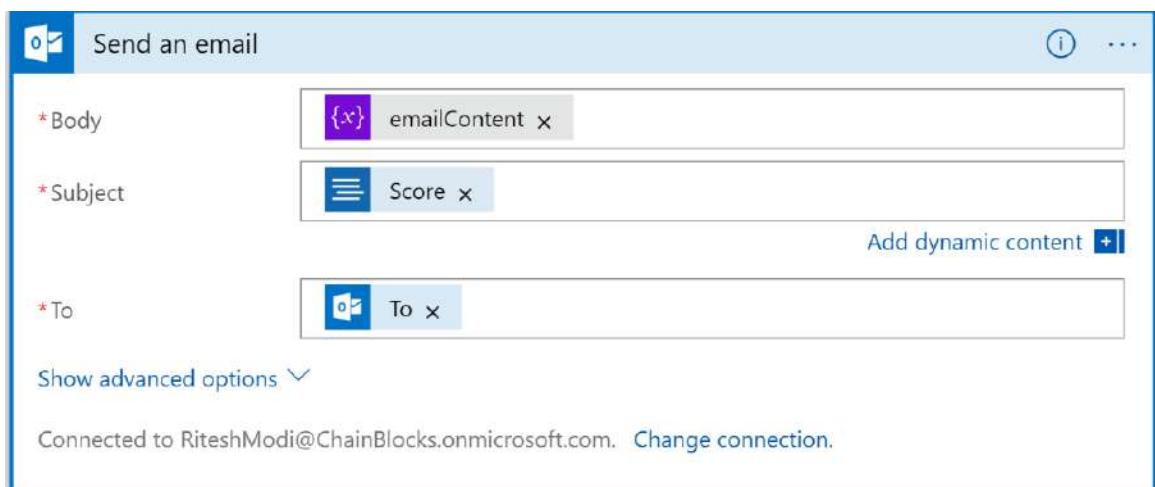


Figure 11.10: Adding the Send an email action

17. Save the logic app, navigate back to the overview page, and click on **Run trigger**. The trigger will check for new emails every 3 minutes, reply to the senders, perform sentiment analysis, and send an email to the original recipient. A sample email with negative connotations is sent to the given email ID:

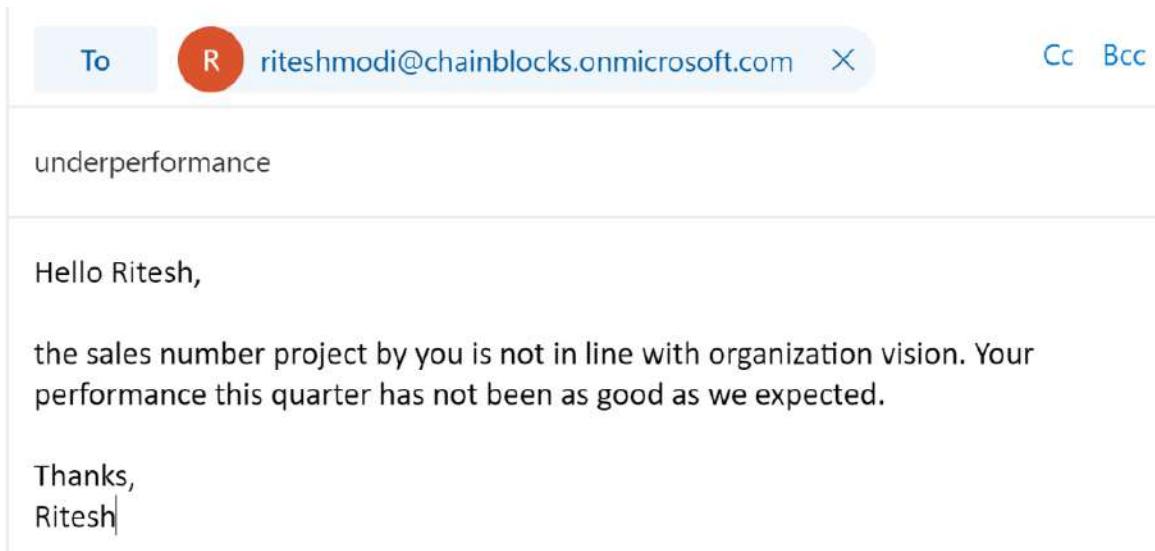


Figure 11.11: Sample email

18. After a few seconds, the logic app executes, and the sender gets the following reply:

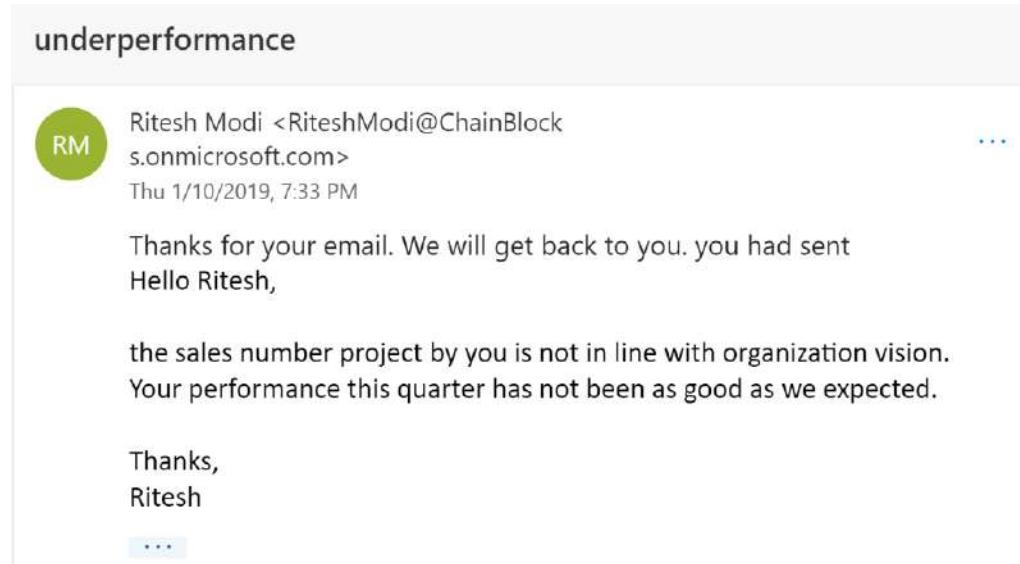


Figure 11.12: Reply email to the original sender

19. The original recipient gets an email with the sentiment score and the original email text, as shown in Figure 11.13:

0.731263041496277

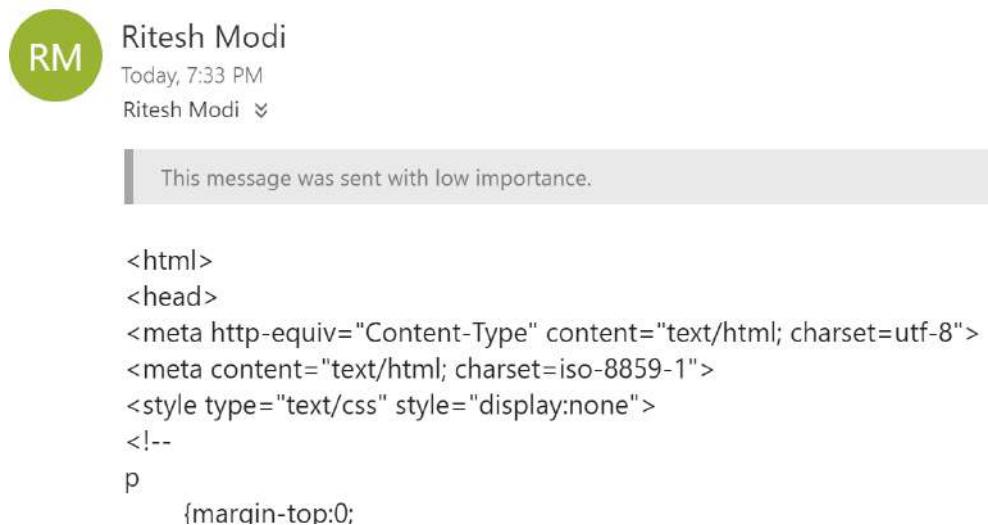


Figure 11.13: HTML view of the email message

From the activity, we were able to understand the workings of a logic app. The app was triggered when an email was received in the inbox of the user and the process followed the sequence of steps that were given in the logic app. In the next section, you will learn how to create an end-to-end solution using serverless technologies.

Creating an end-to-end solution using serverless technologies

In this section, we will create an end-to-end solution comprising serverless technologies that we discussed in the previous sections. The following example will give you an idea of how workflows can be intelligently implemented to avoid management overhead. In the next activity, we will create a workflow to notify the users when the keys, secrets, and certificates get stored in Azure Key Vault. We will take this as a problem statement, figure out a solution, architect the solution, and implement it.

The problem statement

The problem that we are going to solve here is that users and organizations are not notified regarding the expiration of secrets in their key vault, and applications stop working when they expire. Users are complaining that Azure does not provide the infrastructure to monitor Key Vault secrets, keys, and certificates.

Solution

The solution to this problem is to combine multiple Azure services and integrate them so that users can be proactively notified of the expiration of secrets. The solution will send notifications using two channels—email and SMS.

The Azure services used to create this solution include the following:

- Azure Key Vault
- **Azure Active Directory (Azure AD)**
- Azure Event Grid
- Azure Automation
- Logic Apps
- Azure Functions
- SendGrid
- Twilio SMS

Now that you know the services that will be used as part of the solution, let's go ahead and create an architecture for this solution.

Architecture

In the previous section, we explored the list of services that will be used in the solution. If we want to implement the solution, the services should be laid out in the proper order. The architecture will help us to develop the workflow and take a step closer to the solution.

The architecture of the solution comprises multiple services, as shown in Figure 11.14:

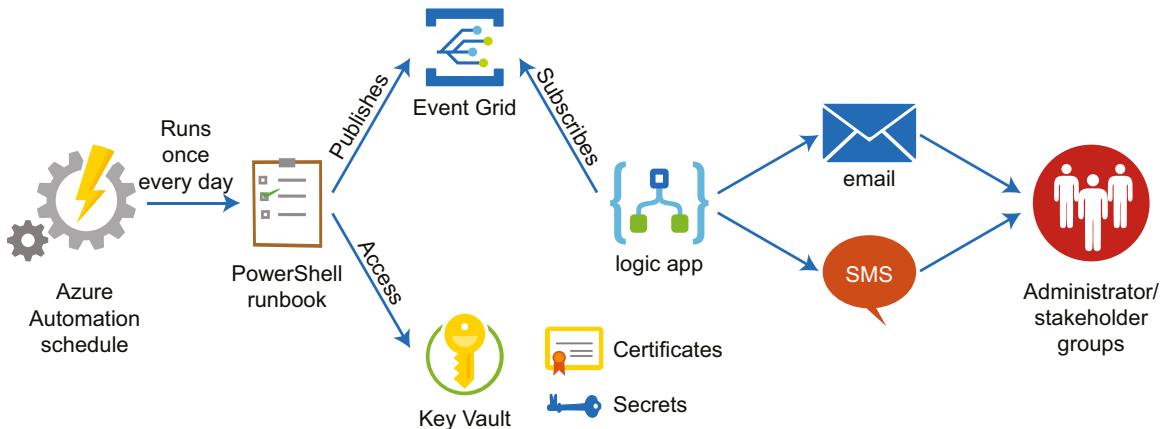


Figure 11.14: Solution architecture

Let's go through each of these services and understand their roles and the functionality that they provide in the overall solution.

Azure Automation

Azure Automation provides runbooks, and these runbooks can be executed to run logic using PowerShell, Python, and other scripting languages. Scripts can be executed either on-premises or in the cloud, which provides rich infrastructure and facilities to create scripts. These kinds of scripts are known as **runbooks**. Typically, runbooks implement a scenario such as stopping or starting a virtual machine, or creating and configuring storage accounts. It is quite easy to connect to the Azure environment from runbooks with the help of assets such as variables, certificates, and connections.

In the current solution, we want to connect to Azure Key Vault, read all the secrets and keys stored within it, and fetch their expiry dates. These expiry dates should be compared with today's date and, if the expiry date is within a month, the runbook should raise a custom event on Event Grid using an Event Grid custom topic.

An Azure Automation runbook using a PowerShell script will be implemented to achieve this. Along with the runbook, a scheduler will also be created that will execute the runbook once a day at 12.00 AM.

A custom Azure Event Grid topic

Once the runbook identifies that a secret or key is going to expire within a month, it will raise a new custom event and publish it to the custom topic created specifically for this purpose. Again, we will go into the details of the implementation in the next section.

Azure Logic Apps

A logic app is a serverless service that provides workflow capabilities. Our logic app will be configured to be triggered as and when an event is published on the custom Event Grid topic. After it is triggered, it will invoke the workflow and execute all the activities in it one after another. Generally, there are multiple activities, but for the purpose of this example, we will invoke one Azure function that will send both email and SMS messages. In a full-blown implementation, these notification functions should be implemented separately in separate Azure functions.

Azure Functions

Azure Functions is used to notify users and stakeholders about the expiration of secrets and keys using email and SMS. SendGrid is used to send emails, while Twilio is used to send SMS messages from Azure Functions.

In the next section, we will take a look at the prerequisites before implementing the solution.

Prerequisites

You will need an Azure subscription with contributor rights at the very least. As we are only deploying services to Azure and no external services are deployed, the subscription is the only prerequisite. Let's go ahead and implement the solution.

Implementation

A key vault should already exist. If not, one should be created.

This step should be performed if a new Azure Key Vault instance needs to be provisioned. Azure provides multiple ways in which to provision resources. Prominent among them are Azure PowerShell and the Azure CLI. The Azure CLI is a command-line interface that works across platforms. The first task will be to provision a key vault in Azure. In this implementation, we will use Azure PowerShell to provision the key vault.

Before Azure PowerShell can be used to create a key vault, it is important to log into Azure so that subsequent commands can be executed successfully to create the key vault.

Step 1: Provisioning an Azure Key Vault instance

The first step is to prepare the environment for the sample. This involves logging into the Azure portal, selecting an appropriate subscription, and then creating a new Azure resource group and a new Azure Key Vault resource:

1. Execute the **Connect-AzAccount** command to log into Azure. It will prompt for credentials in a new window.
2. After a successful login, if there are multiple subscriptions available for the login ID provided, they will all be listed. It is important to select an appropriate subscription—this can be done by executing the **Set-AzContext** cmdlet:

```
Set-AzContext -SubscriptionId xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

3. Create a new resource group in your preferred location. In this case, the name of the resource group is **IntegrationDemo** and it is created in the **West Europe** region:

```
New-AzResourceGroup -Name IntegrationDemo -Location "West Europe" -Verbose
```

4. Create a new Azure Key Vault resource—the name of the vault, in this case, is **keyvaultbook**, and it is enabled for deployment, template deployment, disk encryption, soft delete, and purge protection:

```
New-AzKeyVault -Name keyvaultbook -ResourceGroupName  
IntegrationDemo -Location "West Europe" -EnabledForDeployment  
-EnabledForTemplateDeployment -EnabledForDiskEncryption  
-EnablePurgeProtection -Sku Standard - Verbose
```

Please note that the key vault name needs to be unique. You may not be able to use the same name for two key vaults. The preceding command, when executed successfully, will create a new Azure Key Vault resource. The next step is to provide access to a service principal on the key vault.

Step 2: Creating a service principal

Instead of using an individual account to connect to Azure, Azure provides service principals, which are, in essence, service accounts that can be used to connect to Azure Resource Manager and run activities. Adding a user to an Azure directory/tenant makes them available everywhere, including in all resource groups and resources, due to the nature of security inheritance in Azure. Access must be explicitly revoked from resource groups for users if they are not allowed to access them. Service principals help by assigning granular access and control to resource groups and resources, and, if required, they can be given access to the subscription scope. They can also be assigned granular permissions, such as reader, contributor, or owner permissions.

In short, service principals should be the preferred mechanism to consume Azure services. They can be configured either with a password or with a certificate key. Service principals can be created using the **New-AzAdServicePrincipal** command, as shown here:

```
$sp = New-AzADServicePrincipal -DisplayName "keyvault-book" -Scope "/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" -Role Owner -StartDate ([datetime]::Now) -EndDate $([datetime]::now.AddYears(1)) -Verbose
```

The important configuration values are the scope and role. The scope determines the access area for the service application—it is currently shown at the subscription level. Valid values for scope are as follows:

```
/subscriptions/{subscriptionId}
/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}
/subscriptions/{subscriptionId}/resourcegroups/{resourceGroupName}/providers/{resourceProviderNamespace}/{resourceType}/{resourceName}
/subscriptions/{subscriptionId}/resourcegroups/{resourceGroupName}/providers/{resourceProviderNamespace}/{parentResourcePath}/{resourceType}/{resourceName}
```

The role provides permissions to the assigned scope. The valid values are as follows:

- Owner
- Contributor
- Reader
- Resource-specific permissions

In the preceding command, owner permissions have been provided to the newly created service principal.

We can also use certificates if needed. For simplicity, we will proceed with the password.

With the service principal we created, the secret will be hidden. To find out the secret, you can try the following commands:

```
$BSTR = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($sp.Secret)
$UnsecureSecret = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($BSTR)
```

\$UnsecureSecret will have your secret key.

Along with the service principal, an Application Directory application will be created. The application acts as the global representation of our application across directories and the principal is like a local representation of the application. We can create multiple principals using the same application in a different directory. We can get the details of the application created using the **Get-AzAdApplication** command. We will save the output of this command to a variable, **\$app**, as we will need this later:

```
$app = Get-AzAdApplication -DisplayName $sp.DisplayName
```

We have now created a service principal using a secret; another secure way of creating one is using certificates. In the next section, we will create a service principal using certificates.

Step 3: Creating a service principal using certificates

To create a service principal using certificates, the following steps should be executed:

1. **Create a self-signed certificate or purchase a certificate:** A self-signed certificate is used to create this example end-to-end application. For real-life deployments, a valid certificate should be purchased from a certificate authority.

To create a self-signed certificate, the following command can be run. The self-signed certificate is exportable and stored in a personal folder on the local machine—it also has an expiry date:

```
$currentDate = Get-Date  
$expiryDate = $currentDate.AddYears(1)  
$finalDate = $expiryDate.AddYears(1)  
$servicePrincipalName = "https://automation.book.com"  
$automationCertificate = New-SelfSignedCertificate -DnsName  
$servicePrincipalName -KeyExportPolicy Exportable -Provider "Microsoft  
Enhanced RSA and AES Cryptographic Provider" -NotAfter $finalDate  
-CertStoreLocation "Cert:\LocalMachine\My"
```

2. **Export the newly created certificate:** The new certificate must be exported to the filesystem so that later, it can be uploaded to other destinations, such as Azure AD, to create a service principal.

The commands used to export the certificate to the local filesystem are shown next. Please note that this certificate has both public and private keys, and so while it is exported, it must be protected using a password, and the password must be a secure string:

```
$securepfxpwd = ConvertTo-SecureString -String 'password' -AsPlainText
-Force # Password for the private key PFX certificate
$cert1 = Get-Item -Path Cert:\LocalMachine\My\$($automationCertificate.Thumbprint)
Export-PfxCertificate -Password $securepfxpwd -FilePath " C:\azureautomation.pfx" -Cert $cert1
```

The **Get-Item** cmdlet reads the certificate from the certificate store and stores it in the **\$cert1** variable. The **Export-PfxCertificate** cmdlet actually exports the certificate in the certificate store to the filesystem. In this case, it is in the **C:\book** folder.

3. **Read the content from the newly generated PFX file:** An object of **X509Certificate2** is created to hold the certificate in memory, and the data is converted to a Base64 string using the **System.Convert** function:

```
$newCert = New-Object System.Security.Cryptography.X509Certificates.X509Certificate2 -ArgumentList "C:\azureautomation.pfx", $securepfxpwd
$newcertdata = [System.Convert]::ToBase64String($newCert.GetRawCertData())
```

We will be using this same principal to connect to Azure from the Azure Automation account. It is important that the application ID, tenant ID, subscription ID, and certificate thumbprint values are stored in a temporary location so that they can be used to configure subsequent resources:

```
$adAppName = "azure-automation-sp"
$ServicePrincipal = New-AzADServicePrincipal -DisplayName $adAppName
-CertValue $newcertdata -StartDate $newCert.NotBefore -EndDate $newCert.NotAfter
Sleep 10
New-AzRoleAssignment -ServicePrincipalName $ServicePrincipal.ApplicationId
-RoleDefinitionName Owner -Scope /subscriptions/xxxxx-xxxxxx-xxxxxx-xxxxxx
```

We have our service principal ready. The key vault we created doesn't have an access policy set, which means no user or application will be able to access the vault. In the next step, we will grant permissions to the Application Directory application we created to access the key vault.

Step 4: Creating a key vault policy

At this stage, we have created the service principal and the key vault. However, the service principal still does not have access to the key vault. This service principal will be used to query and list all the secrets, keys, and certificates from the key vault, and it should have the necessary permissions to do so.

To provide the newly created service principal permission to access the key vault, we will go back to the Azure PowerShell console and execute the following command:

```
Set-AzKeyVaultAccessPolicy -VaultName keyvaultbook -ResourceGroupName IntegrationDemo -ObjectId $ServicePrincipal.Id -PermissionsToKeys get,list,create -PermissionsToCertificates get,list,import -PermissionsToSecrets get,list -Verbose
```

Referring to the previous command block, take a look at the following points:

- **Set-AzKeyVaultAccessPolicy** provides access permissions to users, groups, and service principals. It accepts the key vault name and the service principal object ID. This object is different from the application ID. The output of the service principal contains an **Id** property, as shown here:

```
PS C:\windows\system32> $ServicePrincipal
ServicePrincipalNames : {f48850c9-580a-41d4-a062-77cd623e519e, http://azure-automation-sp}
ApplicationId        : f48850c9-580a-41d4-a062-77cd623e519e
ObjectType           : ServicePrincipal
DisplayName          : azure-automation-sp
Id                   : c95ddd04-5906-4bd7-ab5a-d4e617e98946
Type                :
```

Figure 11.15: Finding the object ID of the service principal

- **PermissionsToKeys** provides access to keys in the key vault, and the **get**, **list**, and **create** permissions are provided to this service principal. There is no write or update permission provided to this principal.
- **PermissionsToSecrets** provides access to secrets in the key vault, and the **get** and **list** permissions are provided to this service principal. There is no write or update permission provided to this principal.
- **PermissionsToCertificates** provides access to secrets in the key vault, and **get**, **import**, and **list** permissions are provided to this service principal. There is no write or update permission provided to this principal.

At this point, we have configured the service principal to work with the Azure key vault. The next part of the solution is to create an Automation account.

Step 5: Creating an Automation account

Just like before, we will be using Azure PowerShell to create a new Azure Automation account within a resource group. Before creating a resource group and an Automation account, a connection to Azure should be established. However, this time, use the credentials for the service principal to connect to Azure. The steps are as follows:

1. The command to connect to Azure using the service application is as follows. The value is taken from the variables that we initialized in the previous steps:

```
Login-AzAccount -ServicePrincipal -CertificateThumbprint $newCert.  
Thumbprint -ApplicationId $ServicePrincipal.ApplicationId -Tenant "xxxx-  
xxxxxx-xxxxx-xxxx"
```

2. Make sure that you have access by checking **Get-AzContext** as shown here. Make a note of the subscription ID as it will be needed in subsequent commands:

```
Get-AzContext
```

3. After connecting to Azure, a new resource containing the resources for the solution and a new Azure Automation account should be created. You are naming the resource group **VaultMonitoring**, and creating it in the **West Europe** region. You will be creating the remainder of the resources in this resource group as well:

```
$IntegrationResourceGroup = "VaultMonitoring"  
$rgLocation = "West Europe"  
$automationAccountName = "MonitoringKeyVault"  
New-AzResourceGroup -name $IntegrationResourceGroup -Location $rgLocation  
New-AzAutomationAccount -Name $automationAccountName -ResourceGroupName  
$IntegrationResourceGroup -Location $rgLocation -Plan Free
```

4. Next, create three automation variables. The values for these, that is, the subscription ID, tenant ID, and application ID, should already be available using the previous steps:

```
New-AzAutomationVariable -Name "azuresubscriptionid"  
-AutomationAccountName $automationAccountName -ResourceGroupName  
$IntegrationResourceGroup -Value "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx "  
-Encrypted $true  
  
New-AzAutomationVariable -Name "azuretenantid" -AutomationAccountName  
$automationAccountName -ResourceGroupName $IntegrationResourceGroup -Value  
"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx " -Encrypted $true  
  
New-AzAutomationVariable -Name "azureappid" -AutomationAccountName  
$automationAccountName -ResourceGroupName $IntegrationResourceGroup -Value  
"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx " -Encrypted $true
```

- Now it's time to upload a certificate, which will be used to connect to Azure from Azure Automation:

```
$securepfxpwd = ConvertTo-SecureString -String 'password' -AsPlainText  
-Force # Password for the private key PFX certificate  
New-AzAutomationCertificate -Name "AutomationCertificate" -Path "C:\book\azureautomation.pfx" -Password $securepfxpwd -AutomationAccountName $automationAccountName -ResourceGroupName $IntegrationResourceGroup
```

- The next step is to install PowerShell modules related to Key Vault and Event Grid in the Azure Automation account, as these modules are not installed by default.
- From the Azure portal, navigate to the already-created **VaultMonitoring** resource group by clicking on **Resource Groups** in the left-hand menu.
- Click on the already-provisioned Azure Automation account, **MonitoringKeyVault**, and then click on **Modules** in the left-hand menu. The Event Grid module is dependent on the **Az.profile** module, and so we have to install it before the Event Grid module.
- Click on **Browse Gallery** in the top menu and type **Az.profile** in the search box, as shown in *Figure 11.16*:

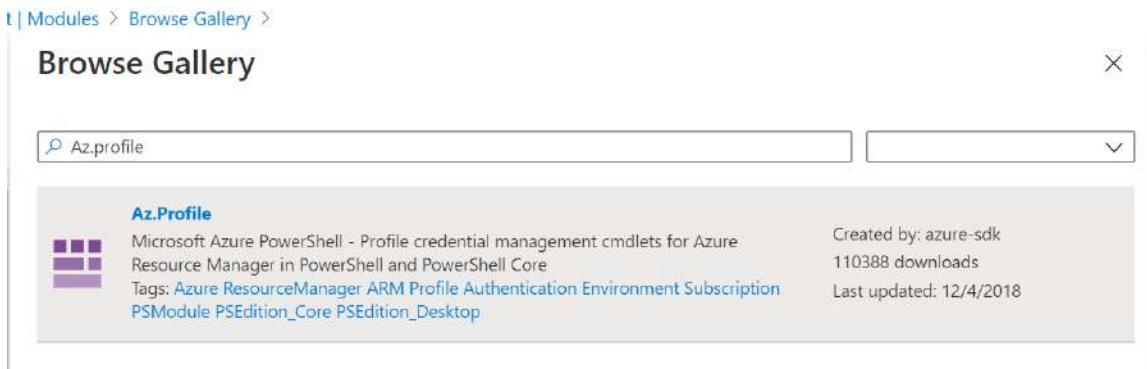
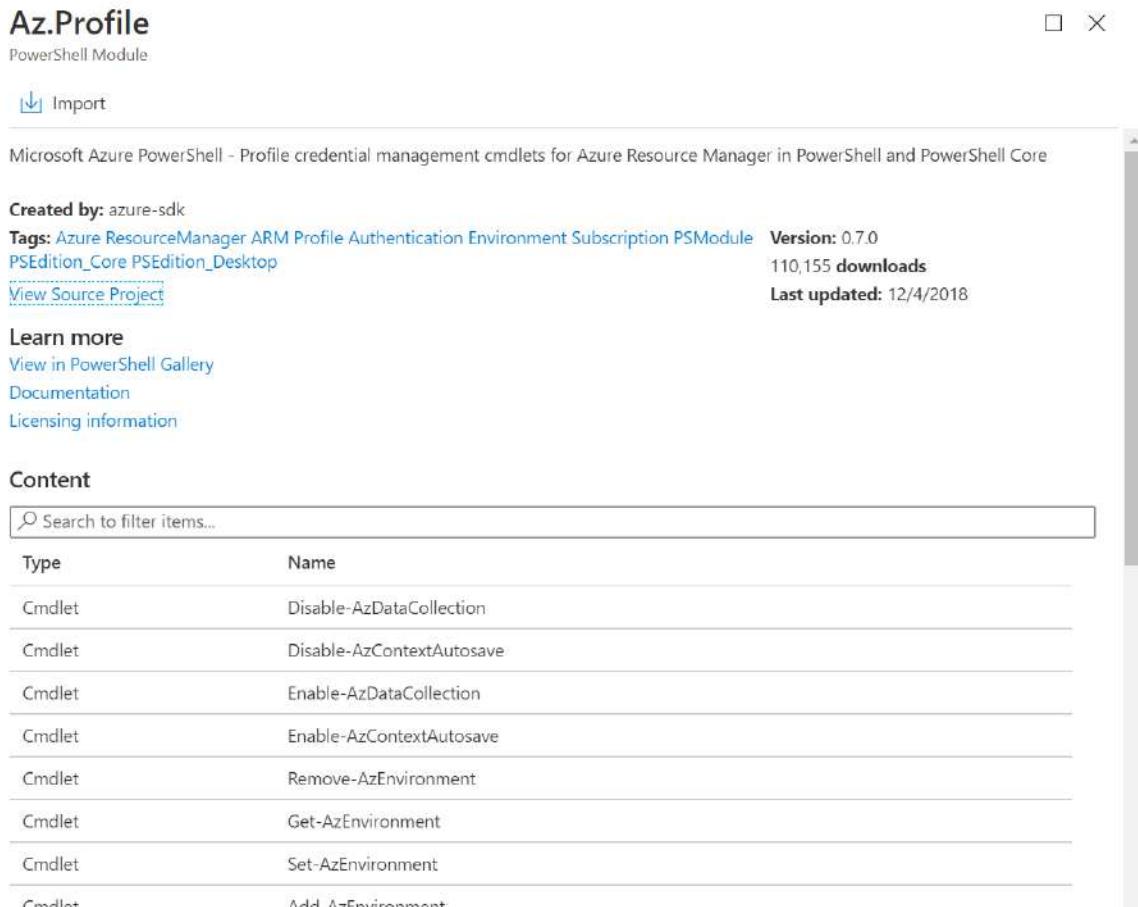


Figure 11.16: The Az.Profile module in the module gallery

- From the search results, select **Az.Profile** and click on the **Import** button in the top menu. Finally, click on the **OK** button. This step takes a few seconds to complete. After a few seconds, the module should be installed.

11. The status of the installation can be checked from the **Module** menu item.
Figure 11.17 demonstrates how we can import a module:



The screenshot shows the details of the **Az.Profile** PowerShell module. It is a **PowerShell Module**. There is a blue **Import** button. Below it, the description reads: "Microsoft Azure PowerShell - Profile credential management cmdlets for Azure Resource Manager in PowerShell and PowerShell Core". The module was **Created by:** azure-sdk. It has **Tags:** Azure, ResourceManager, ARM, Profile, Authentication, Environment, Subscription, PSModule, PSEdition_Core, PSEdition/Desktop. The **Version:** is 0.7.0, with **110,155 downloads** and last updated on **12/4/2018**. There are links to **View Source Project**, **Learn more**, **View in PowerShell Gallery**, **Documentation**, and **Licensing information**. A search bar at the top says "Search to filter items...". The **Content** section lists cmdlets:

Type	Name
Cmdlet	Disable-AzDataCollection
Cmdlet	Disable-AzContextAutosave
Cmdlet	Enable-AzDataCollection
Cmdlet	Enable-AzContextAutosave
Cmdlet	Remove-AzEnvironment
Cmdlet	Get-AzEnvironment
Cmdlet	Set-AzEnvironment
Cmdlet	Add-AzEnvironment

Figure 11.17: Az.Profile module status

12. Perform steps 9, 10, and 11 again in order to import and install the **Az.EventGrid** module. If you are warned to install any dependencies before proceeding, go ahead and install the dependencies first.
13. Perform steps 9, 10, and 11 again in order to import and install the **Az.KeyVault** module. If you are warned to install any dependencies before proceeding, go ahead and install the dependency first.

Since we have imported the necessary modules, let's go ahead and create the Event Grid topic.

Step 6: Creating an Event Grid topic

If you recall the architecture that we used, we need an Event Grid topic. Let's create one.

The command that's used to create an Event Grid topic using PowerShell is as follows:

```
New-AzEventGridTopic -ResourceGroupName VaultMonitoring -Name  
azureforarchitects-topic -Location "West Europe"
```

The process of creating an Event Grid topic using the Azure portal is as follows:

1. From the Azure portal, navigate to the already-created **Vaultmonitoring** resource group by clicking on **Resource Groups** in the left-hand menu.
2. Next, click on the **+Add** button and search for **Event Grid Topic** in the search box. Select it and then click on the **Create** button.
3. Fill in the appropriate values in the resultant form by providing a name, selecting a subscription, and selecting the newly created resource group, the location, and the event schema.

As we already discussed, the Event Grid topic provides an endpoint where the source will send the data. Since we have our topic ready, let's prepare the source Automation account.

Step 7: Setting up the runbook

This step will focus on creating an Azure Automation account and PowerShell runbooks that will contain the core logic of reading Azure key vaults and retrieving secrets stored within them. The steps required for configuring Azure Automation are as follows:

1. **Create the Azure Automation runbook:** From the Azure portal, navigate to the already-created **Vaultmonitoring** resource group by clicking on **Resource Groups** in the left-hand menu.
2. Click on the already-provisioned Azure Automation account, **MonitoringKeyVault**. Then, click on **Runbooks** in the left-hand menu, and click on **+Add a Runbook** from the top menu.
3. Click on **Create a new Runbook** and provide a name. Let's call this runbook **CheckExpiredAssets**, and then set **Runbook type** to **PowerShell**:

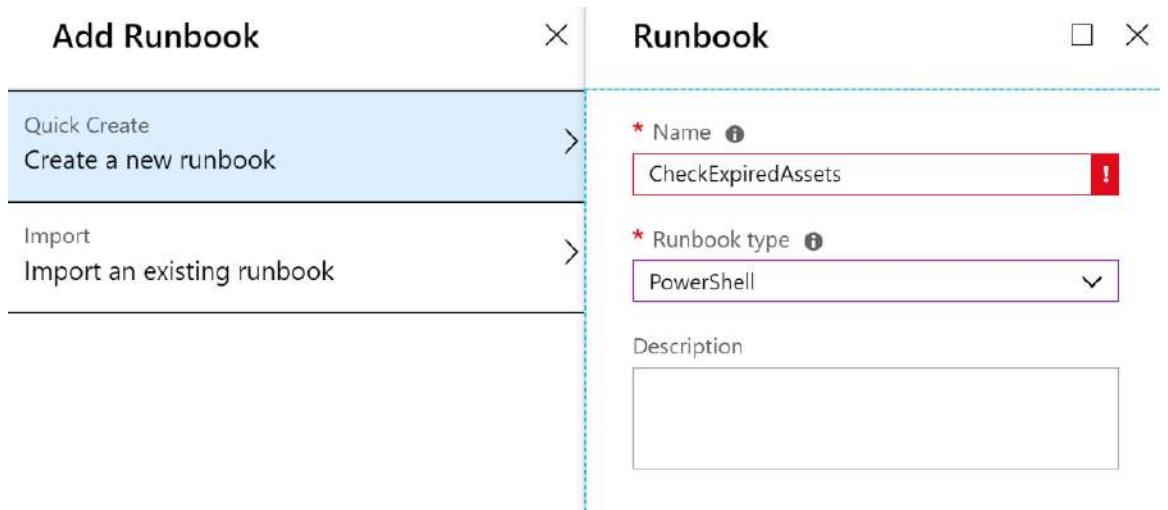


Figure 11.18: Creating a runbook

4. **Code the runbook:** Declare a few variables to hold the subscription ID, tenant ID, application ID, and certificate thumbprint information. These values should be stored in Azure Automation variables, and the certificate should be uploaded to Automation certificates. The key used for the uploaded certificate is **AutomationCertificate**. The values are retrieved from these stores and are assigned to the variables, as shown next:

```
$subscriptionID = get-AutomationVariable "azuresubscriptionid"
$tenantID = get-AutomationVariable "azuretenantid"
$applicationId = get-AutomationVariable "azureappid"
$cert = get-AutomationCertificate "AutomationCertificate"
$certThumbprint = ($cert.Thumbprint).ToString()
```

5. The next code within the runbook helps to log into Azure using the service principal with values from the variables declared previously. Also, the code selects an appropriate subscription. The code is shown next:

```
Login-AzAccount -ServicePrincipal -CertificateThumbprint $certThumbprint
-ApplicationId $applicationId -Tenant $tenantID
Set-AzContext -SubscriptionId $subscriptionID
```

Since Azure Event Grid was provisioned in step 6 of this section, its endpoint and keys are retrieved using the **Get-AzEventGridTopic** and **Get-AzEventGridTopicKey** cmdlets.

Azure Event Grid generates two keys—a primary and a secondary. The first key reference is taken as follows:

```
$eventGridName = "ExpiredAssetsKeyVaultEvents"  
$eventGridResourceGroup = "VaultMonitoring"  
$topicEndpoint = (Get-AzEventGridTopic -ResourceGroupName  
$eventGridResourceGroup -Name $eventGridName).Endpoint  
$keys = (Get-AzEventGridTopicKey -ResourceGroupName  
$eventGridResourceGroup -Name $eventGridName ).Key1
```

6. Next, all key vaults that were provisioned within the subscription are retrieved using iteration. While looping, all secrets are retrieved using the **Get-AzKeyVaultSecret** cmdlet.

The expiry date of each secret is compared to the current date, and if the difference is less than a month, it generates an Event Grid event and publishes it using the **Invoke-WebRequest** command.

The same steps are executed for certificates stored within the key vault. The cmdlet used to retrieve all the certificates is **Get-AzKeyVaultCertificate**.

The event that is published to Event Grid should be in the JSON array. The generated message is converted to JSON using the **ConvertTo-Json** cmdlet and then converted to an array by adding [and] as a prefix and suffix.

In order to connect to Azure Event Grid and publish the event, the sender should supply the key in its header. The request will fail if this data is missing in the request payload:

```
$keyvaults = Get-AzureRmKeyVault  
foreach($vault in $keyvaults) {  
$secrets = Get-AzureKeyVaultSecret -VaultName $vault.VaultName  
foreach($secret in $secrets) {  
if( ![string]::IsNullOrEmpty($secret.Expires) ) {  
if($secret.Expires.AddMonths(-1) -lt [datetime]::Now)  
{  
$secretDataMessage = @{  
id = [System.Guid]::NewGuid()  
subject = "Secret Expiry happening soon !!"  
eventType = "Secret Expiry"  
eventTime = [System.DateTime]::UtcNow  
data = @{  
"ExpiryDate" = $secret.Expires  
"SecretName" = $secret.Name.ToString()  
"VaultName" = $secret.VaultName.ToString()
```

```

"SecretCreationDate" = $secret.Created.ToString()
"IsSecretEnabled" = $secret.Enabled.ToString()
"SecretId" = $secret.Id.ToString()
}
}
...
Invoke-WebRequest -Uri $topicEndpoint -Body $finalBody -Headers $header
-Method Post -UseBasicParsing
}
}
Start-Sleep -Seconds 5
}
}

```

7. Publish the runbook by clicking on the **Publish** button, as shown in *Figure 11.19*:

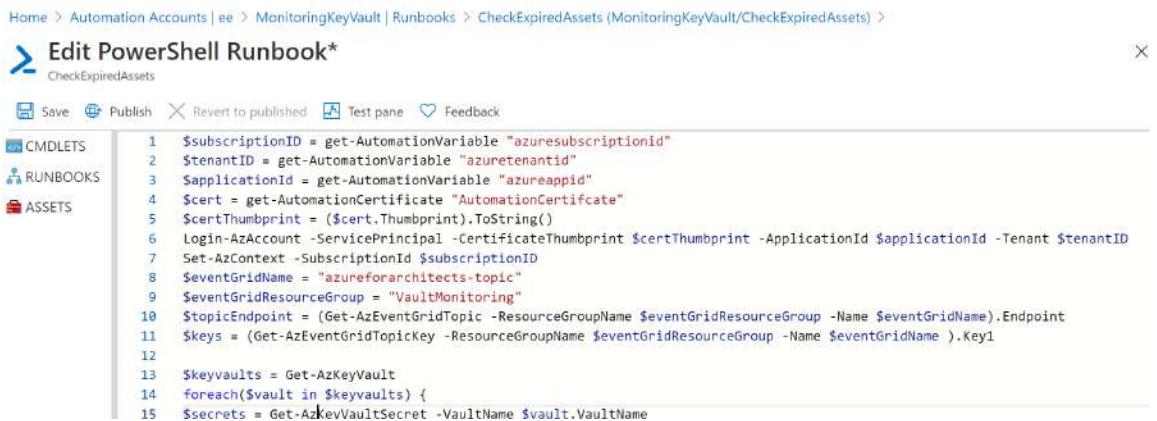


Figure 11.19: Publishing the runbook

8. **Scheduler:** Create an Azure Automation scheduler asset to execute this runbook once every day at 12.00 AM. Click on **Schedules** from the left-hand menu of Azure Automation and click on **+Add a schedule** in the top menu.
9. Provide scheduling information in the resulting form.

This should conclude the configuration of the Azure Automation account.

Step 8: Working with SendGrid

In this step, we will be creating a new SendGrid resource. The SendGrid resource is used to send emails from the application without needing to install a **Simple Mail Transfer Protocol (SMTP)** server. It provides a REST API and a **C# Software Development Kit (SDK)**, by means of which it is quite easy to send bulk emails. In the current solution, Azure Functions will be used to invoke the SendGrid APIs to send emails, and so this resource needs to be provisioned. This resource has separate costing and is not covered as part of the Azure cost—there is a free tier available that can be used for sending emails:

1. A **SendGrid** resource is created just like any other Azure resource. Search for **sendgrid**, and we will get **SendGrid Email Delivery** in the results.
2. Select the resource and click on the **Create** button to open its configuration form.
3. Select an appropriate pricing tier.
4. Provide the appropriate contact details.
5. Tick the **Terms of use** check box.
6. Complete the form and then click on the **Create** button.
7. After the resource is provisioned, click on the **Manage** button in the top menu—this will open the SendGrid website. The website may request email configuration. Then, select **API Keys** from the **Settings** section and click on the **Create API Key** button:

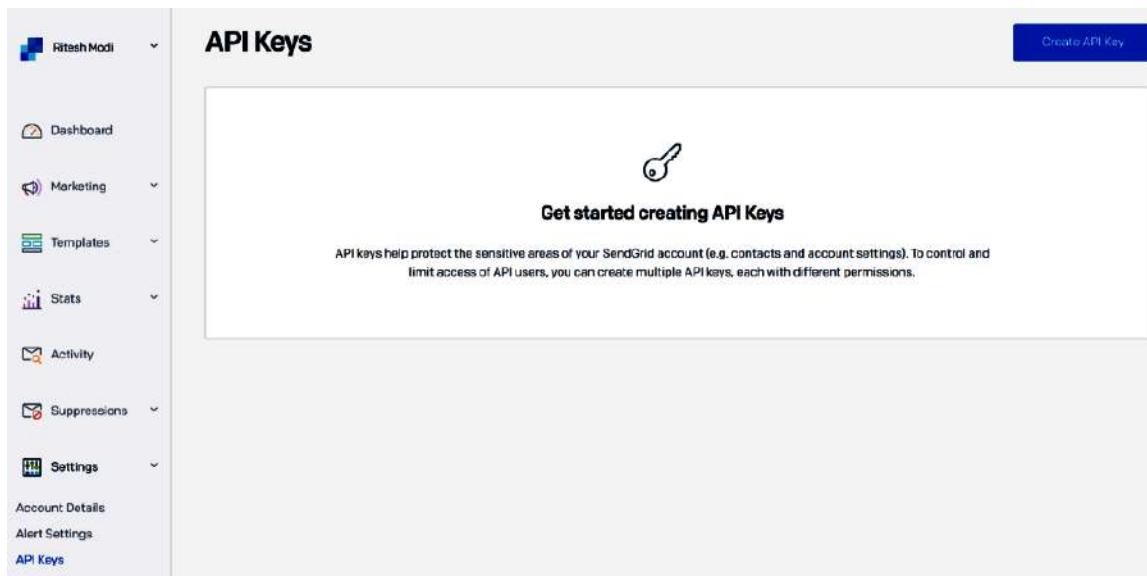


Figure 11.20: Creating API keys for SendGrid

8. From the resulting window, select **Full Access** and click on the **Create & View** button. This will create the key for the SendGrid resource; keep a note of this key, as it will be used with the Azure Functions configuration for SendGrid:

Create API Key

The screenshot shows a 'Create API Key' dialog box. At the top left is an 'API Key Name' input field with a red asterisk indicating it's required. To its right is a help icon. Below the name field is a section titled 'API Key Permissions*' with a help icon. Three access level options are listed: 'Full Access' (selected, indicated by a blue circle), 'Restricted Access' (indicated by an empty circle), and 'Billing Access' (indicated by an empty circle). Each option has a detailed description below it. At the bottom right are 'Cancel' and 'Create & View' buttons, with 'Create & View' being blue and bold.

API Key Name *

API Key Permissions* ⓘ

Full Access
Allows the API key to access GET, PATCH, PUT, DELETE, and POST endpoints for all parts of your account, excluding billing.

Restricted Access
Customize levels of access for all parts of your account, excluding billing.

Billing Access
Allows the API key to access billing endpoints for the account. (This is especially useful for Enterprise or Partner customers looking for more advanced account management.)

Create & View

Figure 11.21: Setting up the access level in the SendGrid portal

Now that we have configured access levels for SendGrid, let's configure another third-party service, which is called Twilio.

Step 9: Getting started with Twilio

In this step, we will be creating a new Twilio account. Twilio is used for sending bulk SMS messages. To create an account with Twilio, navigate to [twilio.com](https://www.twilio.com) and create a new account. After successfully creating an account, a mobile number is generated that can be used to send SMS messages to receivers:



Figure 11.22: Choosing a Twilio number

The Twilio account provides both production and test keys. Copy the test key and token to a temporary location, such as Notepad, as they will be required later within Azure Functions:

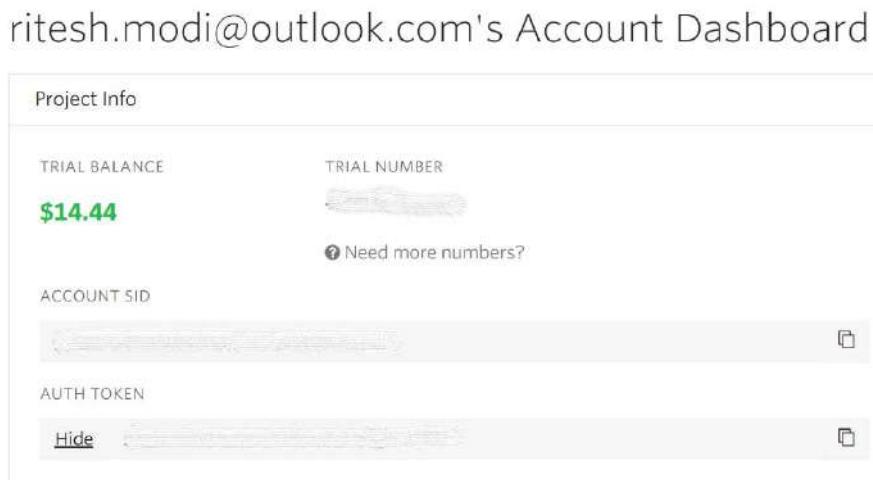


Figure 11.23: Setting up Twilio

We have SendGrid and Twilio in place for the notification service; however, we need something that can take the event and notify the users. Here comes the role of a function app. In the next section, we will be creating a function app that will help with sending SMS and emails.

Step 10: Setting up a function app

In this step, we will be creating a new function app responsible for sending emails and SMS notifications. The purpose of the function app within the solution is to send notification messages to users regarding the expiry of secrets in the key vault. A single function will be responsible for sending both emails and SMS messages—note that this could have been divided into two separate functions. The first step is to create a new function app and host a function within it:

1. As we have done before, navigate to your resource group, click on the **+Add** button in the top menu, and search for the **function app** resource. Then, click on the **Create** button to get the **Function App** form.
2. Fill in the **Function App** form and click on the **Create** button. The name of the function app must be unique across Azure.
3. Once the function app is provisioned, create a new function called **SMSandEMailFunction** by clicking on the **+** button next to the **Functions** item in the left-hand menu. Then, select **In-portal** from the central dashboard.
4. Select **HTTP trigger** and name it **SMSandEMailFunction**. Then, click on the **Create** button—the **Authorization level** option can be any value.
5. Remove the default code, replace it with the code shown in the following listing, and then click on the **Save** button in the top menu:

```
#r "SendGrid"
#r "Newtonsoft.Json"
#r "Twilio.Api"
using System.Net;
using System;
using SendGrid.Helpers.Mail;
using Microsoft.Azure.WebJobs.Host;
using Newtonsoft.Json;
using Twilio;
using System.Configuration;
public static HttpResponseMessage Run(HttpRequestMessage req, TraceWriter log, out Mail message,out SMSMessage sms)
{
    log.Info("C# HTTP trigger function processed a request.");
    string alldata = req.Content.ReadAsStringAsync().GetAwaiter().GetResult();
    message = new Mail();
    var personalization = new Personalization();
    personalization.AddBcc(new Email(ConfigurationManager.AppSettings["bccStakeholdersEmail"]));
}
```

```

personalization.AddTo(new Email(ConfigurationManager.
AppSettings["toStakeholdersEmail"]));
var messageContent = new Content("text/html", alldata);
message.AddContent(messageContent);
message.AddPersonalization(personalization);
message.Subject = "Key Vault assets Expiring soon..";
message.From = new Email(ConfigurationManager.AppSettings["serviceEmail"]);
string msg = alldata;
sms = new SMSMessage();
sms.Body = msg;
sms.To = ConfigurationManager.AppSettings["adminPhone"];
sms.From = ConfigurationManager.AppSettings["servicePhone"];
return req.CreateResponse(HttpStatusCode.OK, "Hello ");
}

```

6. Click on the function app name in the left-hand menu and click again on the **Application settings** link in the main window:

NotificationFunctionAppBook

Function Apps

RiteshSubscription

Function Apps

NotificationFunctionApp... (selected)

Functions

SMSandEMailFunction

Integrate

Manage

Monitor

Proxies

Slots (preview)

Overview

Platform features

Status: Running

Subscription: RiteshSubscription

Resource group: VaultMonitoring

Subscription ID: [REDACTED]

Location: West Europe

Configured features

Function app settings

Application settings

Application Insights

Figure 11.24: Navigating to Application settings

7. Navigate to the **Application settings** section, as shown in *Figure 11.24*, and add a few entries by clicking on **+ Add new setting** for each entry.

Note that the entries are in the form of key-value pairs, and the values should be actual real-time values. Both **adminPhone** and **servicePhone** should already be configured on the Twilio website. **servicePhone** is the phone number generated by Twilio that is used for sending SMS messages, and **adminPhone** is the phone number of the administrator to whom the SMS should be sent.

Also note that Twilio expects the destination phone number to be in a particular format depending on the country (for India, the format is **+91 xxxx xxxx**). Note the spaces and country code in the number.

We also need to add the keys for both SendGrid and Twilio within the application settings. These settings are mentioned in the following list. You may already have these values handy because of activities performed in earlier steps:

- The value of **SendGridAPIKeyAsAppSetting** is the key for SendGrid.
- **TwilioAccountSid** is the system identifier for the Twilio account. This value was already copied and stored in a temporary location in *Step 9: Getting started with Twilio*.
- **TwilioAuthToken** is the token for the Twilio account. This value was already copied and stored in a temporary place in an earlier step.

- Save the settings by clicking on the **Save** button in the top menu:

Application settings

Application Settings are encrypted at rest and transmitted over an encrypted connection.

[Hide Values](#) [Show Values](#)

APP SETTING NAME	VALUE
APPINSIGHTS_INSTRUMENTATIONKEY	<i>Hidden value. Click to edit.</i>
AzureWebJobsStorage	<i>Hidden value. Click to edit.</i>
FUNCTIONS_EXTENSION_VERSION	<i>Hidden value. Click to edit.</i>
FUNCTIONS_WORKER_RUNTIME	<i>Hidden value. Click to edit.</i>
WEBSITE_CONTENTAZUREFILECONNECTIO...	<i>Hidden value. Click to edit.</i>
WEBSITE_CONTENTSHARE	<i>Hidden value. Click to edit.</i>
WEBSITE_NODE_DEFAULT_VERSION	<i>Hidden value. Click to edit.</i>
adminPhone	<i>Hidden value. Click to edit.</i>
servicePhone	<i>Hidden value. Click to edit.</i>
serviceEmail	<i>Hidden value. Click to edit.</i>
bccStakeholdersEmail	<i>Hidden value. Click to edit.</i>
toStakeholdersEmail	<i>Hidden value. Click to edit.</i>
SendGridAPIKeyAsAppSetting	<i>Hidden value. Click to edit.</i>
TwilioAccountSid	<i>Hidden value. Click to edit.</i>
TwilioAuthToken	<i>Hidden value. Click to edit.</i>

[+ Add new setting](#)

Figure 11.25: Configuring application settings

9. Click on the **Integrate** link in the left-hand menu just below the name of the function, and click on **+ New Output**. This is to add an output for the SendGrid service:

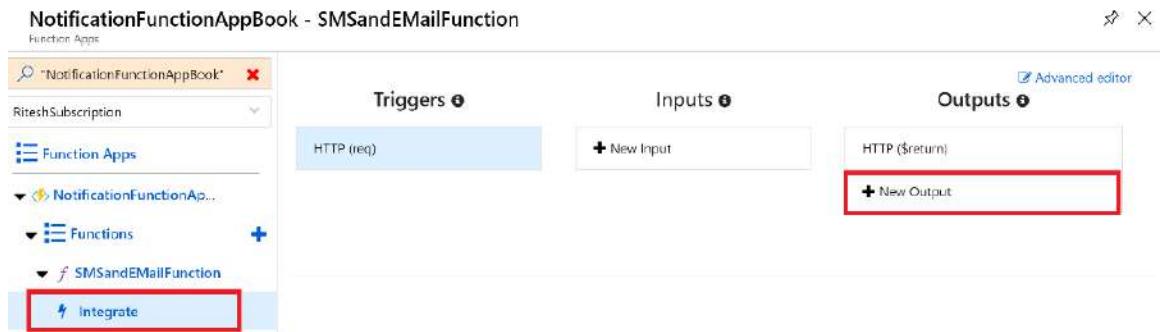


Figure 11.26: Adding an output to the function app

10. Next, select **SendGrid**; it might ask you to install the SendGrid extension. Install the extension, which will take a couple of minutes:

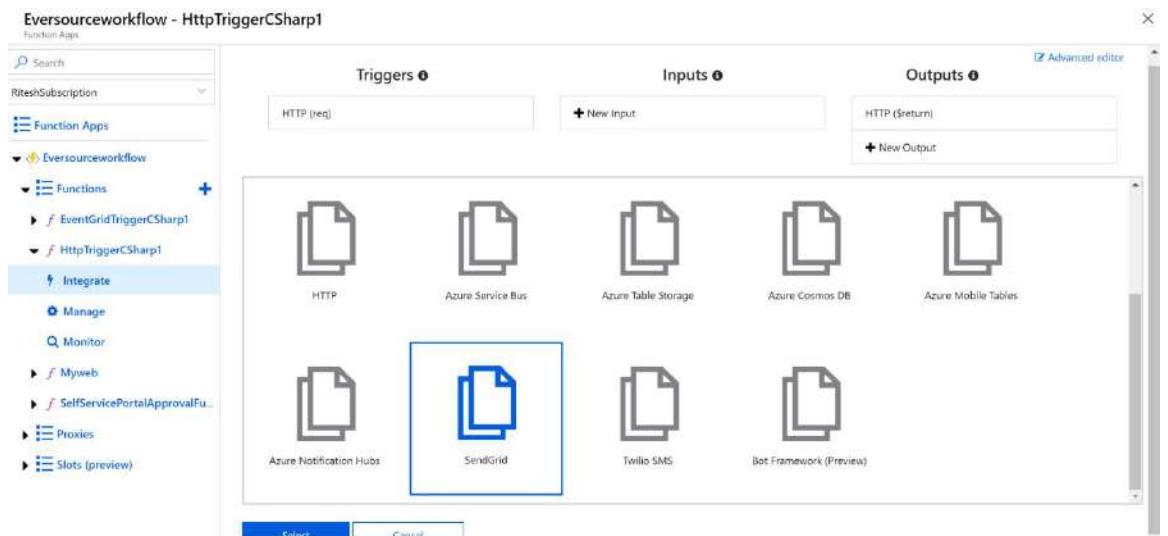


Figure 11.27: Configuring a function app

11. After installing the extension, the output configuration form appears. The important configuration items in this form are **Message parameter name** and **SendGrid API Key App Setting**. Leave the default value for **Message parameter name** and click on the drop-down list to select **SendGridAPIKeyAsAppSetting** as the API app setting key. This was already configured in a previous step within the app settings configuration. The form should be configured as shown in *Figure 11.28*, and then you need to click on the **Save** button:

SendGrid output

Message parameter name ⓘ
message

SendGrid API Key App Setting ⓘ show value
SendGridAPIKeyAsAppSetting new

From address ⓘ
From address

To address ⓘ
To address

Message subject ⓘ
Message subject

Message Text ⓘ
Message Text

Save **Cancel**

+ Documentation

Figure 11.28: Setting up SendGrid

12. Click on **+ New Output** again; this is to add an output for the Twilio service.
13. Then, select **Twilio SMS**. It might ask you to install the Twilio SMS extension. Install the extension, which will take a couple of minutes.
14. After installing the extension, the output configuration form appears. The important configuration items in this form are **Message parameter name**, **Account SID setting**, and **Auth Token setting**. Change the default value for **Message parameter name** to **sms**. This is done because the **message** parameter is already used for the SendGrid service parameter. Ensure that the value of **Account SID setting** is **TwilioAccountId** and that the value of **Auth Token setting** is **TwilioAuthToken**. These values were already configured in a previous step of the app settings configuration. The form should be configured as shown in *Figure 11.29*, and then you should click on **Save**:

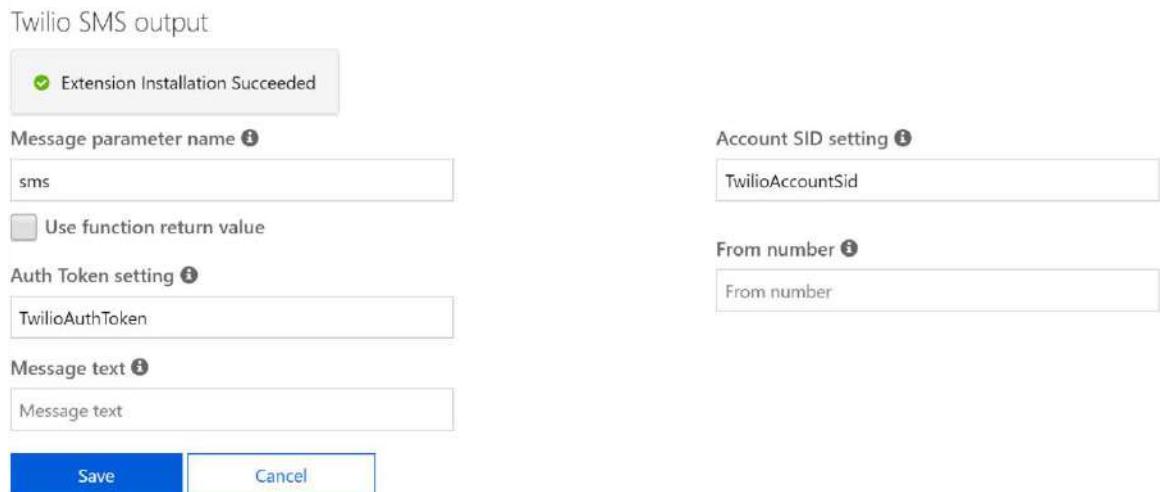


Figure 11.29: Setting up Twilio SMS output

Our SendGrid and Twilio accounts are ready. Now it's time to use the connectors and add them to the logic app. In the next part, we will create the logic app and will use connectors to work with the resources we have created so far.

Step 11: Creating a logic app

In this step, we will be creating a new logic app workflow. We have authored an Azure Automation runbook that queries all the secrets in all key vaults and publishes an event if it finds any of them expiring within a month. The logic app's workflow acts as a subscriber to these events:

1. The first step within the **Logic App** menu is to create a logic app workflow.
2. Fill in the resultant form after clicking on the **Create** button. We are provisioning the logic app in the same resource group as the other resources for this solution.
3. After the logic app is provisioned, it opens the designer window. Select **Blank Logic App** from the **Templates** section.
4. In the resultant window, add a trigger that can subscribe to Event Grid. Logic Apps provides a trigger for Event Grid, and you can search for this to see whether it's available.

5. Next, select the **When a resource event occurs (preview)** trigger:

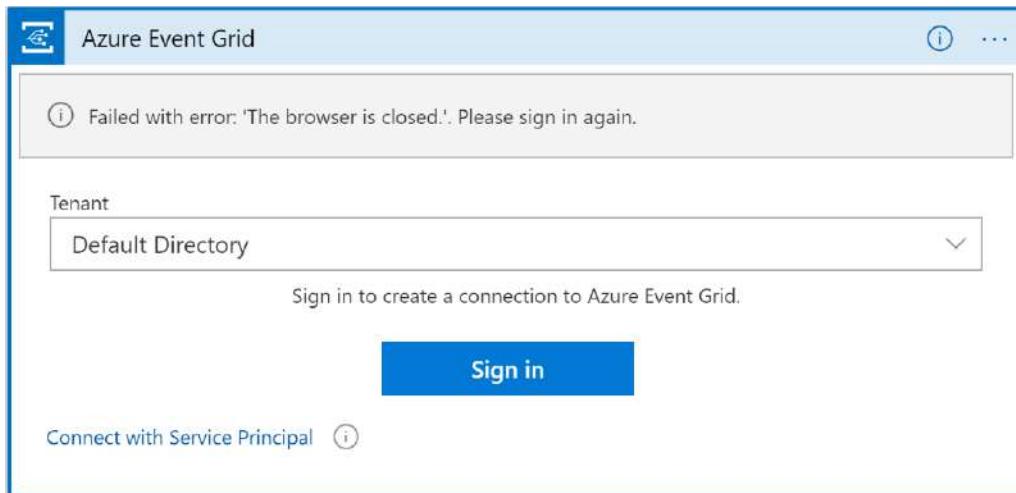


Figure 11.30: Selecting a trigger from Event Grid

6. In the resultant window, select **Connect with Service Principal**.

Provide the service principal details, including the application ID (**Client ID**), tenant ID, and password. This trigger does not accept a service principal that authenticates with the certificate—it accepts a service principal only with a password. Create a new service principal at this stage that authenticates with a password (the steps for creating a service principal based on password authentication were covered earlier, in step 2) and use the details of the newly created service principal for Azure Event Grid configuration, as shown in Figure 11.31:

The screenshot shows a "Azure Event Grid" configuration window. It includes fields for "Connection Name" (with placeholder text "New Connection"), "Client ID" (containing "0d2"), "Client Secret" (containing "*****"), and "Tenant" (containing "1e7e"). A large blue "Update" button is at the bottom. At the very bottom left, there is a "Connect with sign in" link.

Figure 11.31: Providing the service principal details for connection

- Select the subscription. Based on the scope of the service principal, this will get auto-filled. Select **Microsoft.EventGrid.Topics** as the **Resource Type** value and set the name of the custom topic as **ExpiredAssetsKeyVaultEvents**:

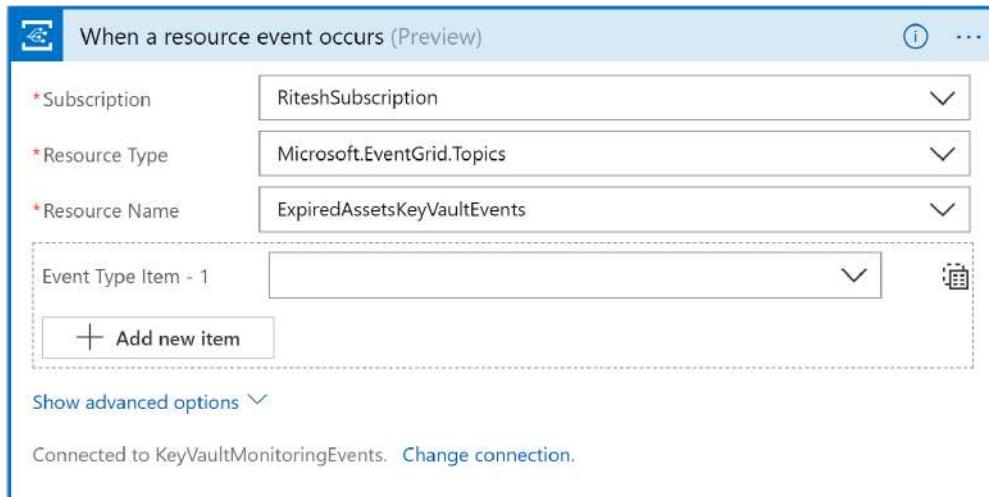


Figure 11.32: Providing Event Grid trigger details

- The previous step will create a connector, and the connection information can be changed by clicking on **Change connection**.
- The final configuration of the Event Grid trigger should be similar to Figure 11.33:

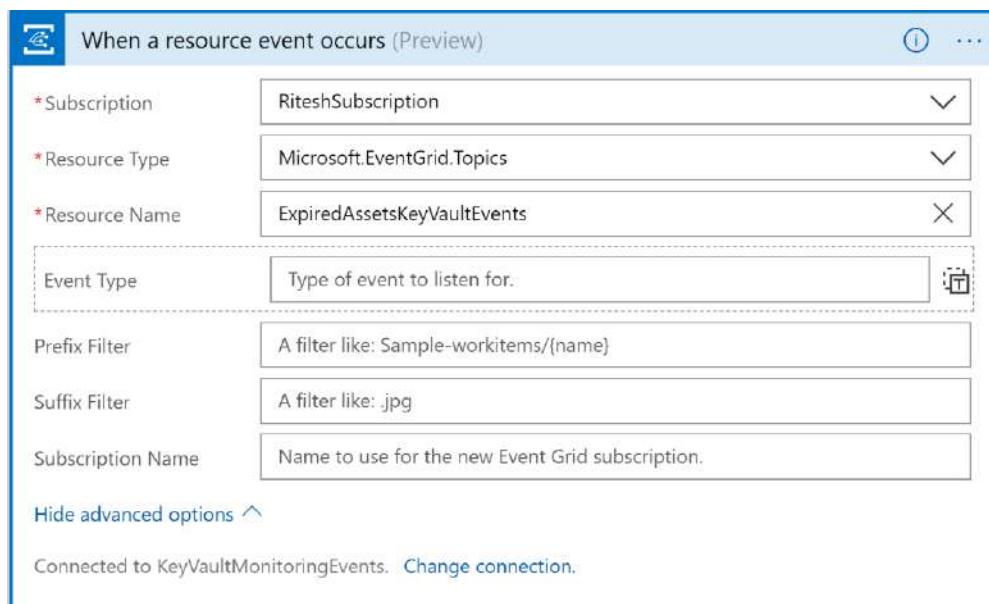


Figure 11.33: Event Grid trigger overview

10. Add a new **Parse JSON** activity after the Event Grid trigger—this activity needs the JSON schema. Generally, the schema is not available, but this activity helps generate the schema if valid JSON is provided to it:



Figure 11.34: Parse JSON activity

11. Click on **Use sample payload to generate schema** and provide the following data:

```
{
  "ExpiryDate": "",
  "SecretName": "",
  "VaultName": "",
  "SecretCreationDate": "",
  "IsSecretEnabled": "",
  "SecretId": ""
}
```

A question might arise here regarding the sample payload. At this stage, how do you calculate the payload that's generated by the Event Grid publisher? The answer to this lies in the fact that this sample payload is exactly the same as is used in the data element in the Azure Automation runbook. You can take a look at that code snippet again:

```
data = @{
  "ExpiryDate" = $certificate.Expires
  "CertificateName" = $certificate.Name.ToString()
  "VaultName" = $certificate.VaultName.ToString()
  "CertificateCreationDate" = $certificate.Created.ToString()
  "IsCertificateEnabled" = $certificate.Enabled.ToString()
  "CertificateId" = $certificate.Id.ToString()
}
```

12. The **Content** box should contain dynamic content coming out from the previous trigger, as demonstrated in *Figure 11.35*:

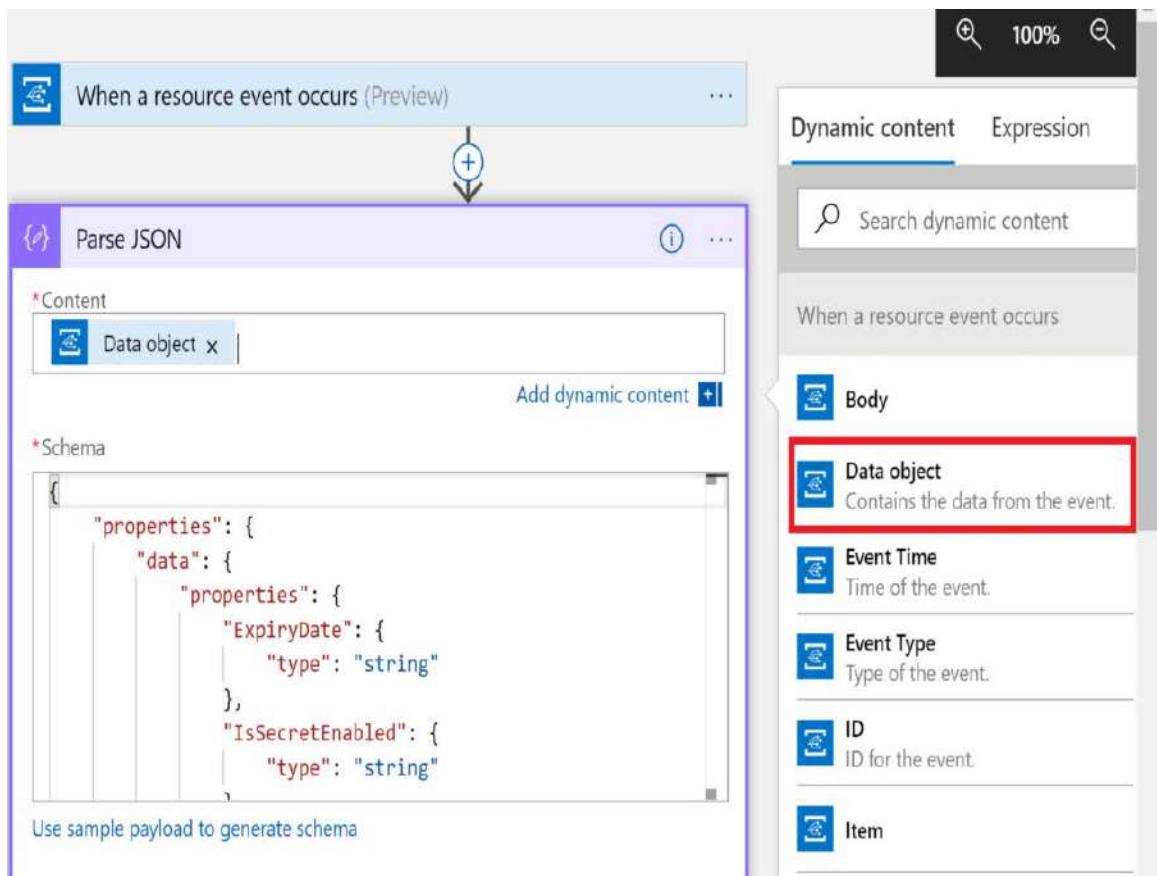


Figure 11.35: Providing dynamic content to the Parse JSON activity

13. Add another **Azure Functions** action after **Parse JSON**, and then select **Choose an Azure function**. Select the Azure function apps called **NotificationFunctionAppBook** and **SMSAndEmailFunction**, which were created earlier:

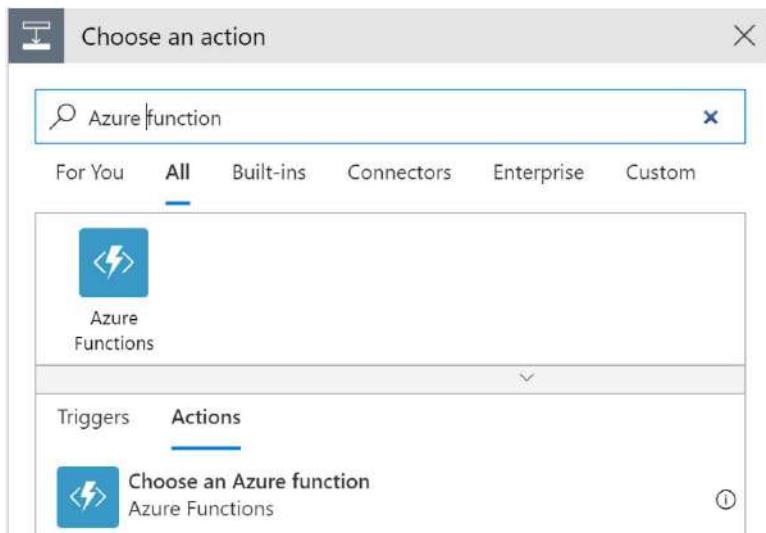


Figure 11.36: Adding an Azure Functions action

14. Click on the **Request Body** text area and fill it with the following code. This is done to convert the data into JSON before sending it to the Azure function:

```
{
  "alldata" :
}
```

15. Place the cursor after the ":" in the preceding code and click on **Add dynamic content | Body** from the previous activity:

Figure 11.37: Converting data to JSON before sending it to an Azure function

16. Save the entire logic app; it should look as follows:



Figure 11.38: Logic app workflow

Once you save the logic app, your solution is ready to be tested. If you don't have any keys or secrets, try adding them with an expiry date so that you can confirm whether your solution is working.

Testing

Upload some secrets and certificates that have expiry dates to Azure Key Vault and execute the Azure Automation runbook. The runbook is scheduled to run on a schedule. Additionally, the runbook will publish events to Event Grid. The logic app should be enabled, and it will pick the event and finally invoke the Azure function to send email and SMS notifications.

The email should look as follows:

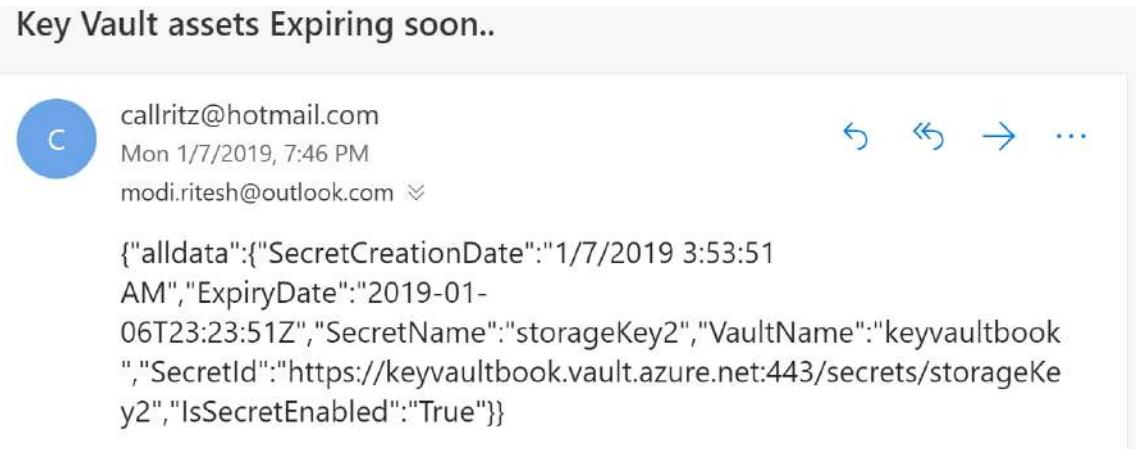


Figure 11.39: Email received regarding the expiring keys

In this exercise, we had a problem, we architected a solution, and we implemented it. This is exactly what happens in the role of an architect. Customers will have specific requirements and, based on those, you must develop a solution. On that note, we are concluding this chapter. Let's do a quick recap of what we have discussed.

Summary

This chapter introduced Logic Apps and demonstrated a complete end-to-end solution using multiple Azure services. The chapter focused heavily on creating an architecture that integrated multiple Azure services to create an end-to-end solution. The services used in the solution were Azure Automation, Azure Logic Apps, Azure Event Grid, Azure Functions, SendGrid, and Twilio. These services were implemented through the Azure portal and PowerShell using service principals as service accounts. The chapter also showed a number of ways of creating service principals with password and certificate authentication.

A solution to a problem can be found in multiple ways. You could use an Outlook trigger in a logic app instead of SendGrid. There will be many solutions to a problem—the one to go with depends on what approach you are taking. The more familiar you are with the services, the greater the number of options you will have. In the next chapter, you will learn about the importance of events in both Azure and Azure application architecture.

12

Azure Big Data eventing solutions

Events are everywhere! Any activity or task that changes the state of a work item generates an event. Due to a lack of infrastructure and the non-availability of cheap devices, there previously was not much traction for the **Internet of Things (IoT)**. Historically, organizations used hosted environments from **internet service providers (ISPs)** that just had monitoring systems on top of them. These monitoring systems raised events that were few and far between.

However, with the advent of the cloud, things are changing rapidly. With increased deployments on the cloud, especially of **Platform as a Service (PaaS)** services, organizations no longer need much control over the hardware and the platform, and now every time there is a change in an environment, an event is raised. With the emergence of cloud events, IoT has gained a lot of prominence and events have started to take center stage.

Another recent phenomenon has been the rapid burst of growth in the availability of data. The velocity, variety, and volume of data has spiked, and so has the need for solutions for storing and processing data. Multiple solutions and platforms have emerged, such as Hadoop, data lakes for storage, data lakes for analytics, and machine learning services.

Apart from storage and analytics, there is also a need for services that are capable of ingesting millions upon millions of events and messages from various sources. There is also a need for services that can work on temporal data, rather than working on an entire snapshot of data. For example, event data/IoT data is used in applications that make decisions based on real-time or near real-time data, such as traffic management systems or systems that monitor temperature.

Azure provides a plethora of services that help in capturing and analyzing real-time data from sensors. In this chapter, we will go through a couple of eventing services in Azure, as listed here:

- Azure Event Hubs
- Azure Stream Analytics

There are other eventing services, such as Azure Event Grid, that are not covered in this chapter; however, they are extensively covered in *Chapter 10, Azure Integration Services with Azure functions (Durable Functions and Proxy functions)*.

Introducing events

Events are important constructs in both Azure and Azure application architecture. Events are everywhere within the software ecosystem. Generally, any action that is taken results in an event that can be trapped, and then further action can be taken. To take this discussion forward, it is important to first understand the basics of events.

Events help in capturing the new state of a target resource. A message is a lightweight notification of a condition or a state change. Events are different than messages. Messages are related to business functionality, such as sending order details to another system. They contain raw data and can be large in size. In comparison, events are different; for instance, a virtual machine being stopped is an event. Figure 12.1 demonstrates this transition from the current state to the target state:



Figure 12.1: Transition of a state due to an event

Events can be stored in durable storage as historical data and events can also be used to find patterns that are emerging on an ongoing basis. Events can be thought of as data being streamed constantly. To capture, ingest, and perform analysis on a stream of data, special infrastructure components that can read a small window of data and provide insights are needed, and that is where the Stream Analytics service comes into the picture.

Event streaming

Processing events as they are ingested and streamed over a time window provides real-time insights about data. The time window could 15 minutes or an hour—the window is defined by the user and depends on the insights that are to be extracted from data. Take credit card swipes, for instance—millions of credit card swipes happen every minute, and fraud detection can be done over streamed events for a time window of one or two minutes.

Event streaming refers to services that can accept data as and when it arises, rather than accepting it periodically. For example, event streams should be capable of accepting temperature information from devices as and when they send it, rather than making the data wait in a queue or a staging environment.

Event streaming also has the capability of querying data while in transit. This is temporal data that is stored for a while, and the queries occur on the moving data; therefore, the data is not stationary. This capability is not available on other data platforms, which can only query stored data and not temporal data that has just been ingested.

Event streaming services should be able to scale easily to accept millions or even billions of events. They should be highly available such that sources can send events and data to them at any time. Real-time data ingestion and being able to work on that data, rather than data that's stored in a different location, is the key to event streaming.

But when we already have so many data platforms with advanced query execution capabilities, why do we need event streaming? One of the main advantages of event streaming is that it provides real-time insights and information whose usefulness is time-dependent. The same information found after a few minutes or hours might not be that useful. Let's consider some scenarios in which working on incoming data is quite important. These scenarios can't be effectively and efficiently solved by existing data platforms:

- **Credit card fraud detection:** This should happen as soon as and when a fraudulent transaction happens.
- **Telemetry information from sensors:** In the case of IoT devices sending vital information about their environments, the user should be notified as soon as and when an anomaly is detected.
- **Live dashboards:** Event streaming is needed to create dashboards that show live information.
- **Datacenter environment telemetry:** This will let the user know about any intrusions, security breaches, failures of components, and more.

There are many possibilities for applying event streaming within an enterprise, and its importance cannot be stressed enough.

Event Hubs

Azure Event Hubs is a streaming platform that provides functionality related to the ingestion and storage of streaming-related events.

It can ingest data from a variety of sources; these sources could be IoT sensors or any applications using the Event Hubs **Software Development Kit (SDK)**. It supports multiple protocols for ingesting and storing data. These protocols are industry standard, and they include the following:

- **HTTP:** This is a stateless option and does not require an active session.
- **Advanced Messaging Queuing Protocol (AMQP):** This requires an active session (that is, an established connection using sockets) and works with **Transport Layer Security (TLS)** and **Secure Socket Layer (SSL)**.
- **Apache Kafka:** This is a distributed streaming platform similar to Stream Analytics. However, Stream Analytics is designed to run real-time analytics on multiple streams of data from various sources, such as IoT sensors and websites.

Event Hubs is an event ingestion service. It can't query a request and output query results to another location. That is the responsibility of Stream Analytics, which is covered in the next section.

To create an Event Hubs instance from the portal, search for Event Hubs in Marketplace and click on **Create**. Select a subscription and an existing resource group (or create a new one). Provide a name for the Event Hubs namespace, the preferred Azure region to host it in, the pricing tier (Basic or Standard, explained later), and the number of throughput units (explained later):

Create Namespace

Event Hubs

Basics Features Tags Review + create

PROJECT DETAILS

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

Subscription * RiteshSubscription

Resource group * Create new

INSTANCE DETAILS

Enter required settings for this namespace, including a price tier and configuring the number of throughput units.

Namespace name * myeventhub.servicebus.windows.net

Location * West Central US

Pricing tier (View full pricing details) * Standard

Throughput Units * 1

Review + create **< Previous** **Next: Features >**

Figure 12.2: Creating an Event Hubs namespace

Event Hubs, being a PaaS service, is highly distributed, highly available, and highly scalable.

Event Hubs comes with the following two SKUs or pricing tiers:

- **Basic:** This comes with one consumer group and can retain messages for 1 day. It can have a maximum of 100 brokered connections.
- **Standard:** This comes with a maximum of 20 consumer groups and can retain messages for 1 day with additional storage for 7 days. It can have a maximum of 1,000 brokered connections. It is also possible to define policies in this SKU.

Figure 12.3 shows the different SKUs available while creating a new Event Hubs namespace. It provides an option to choose an appropriate pricing tier, along with other important details:

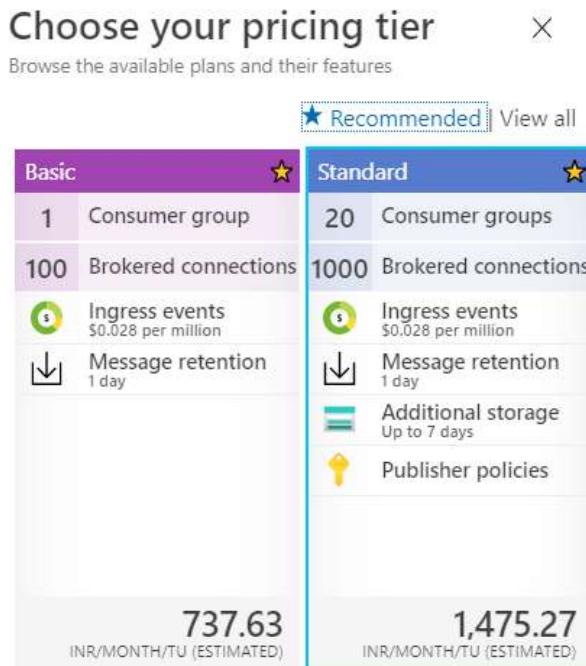


Figure 12.3: Event Hubs SKUs

Throughput can also be configured at the namespace level. Namespaces are containers that consist of multiple event hubs in the same subscription and region. The throughput is calculated as **throughput units (TUs)**. Each TU provides:

- Up to 1 MB per second of ingress or a maximum of 1,000 ingress events and management operations per second.
- Up to 2 MB per second of egress or a maximum of 4,096 events and management operations per second.
- Up to 84 GB of storage.

The TUs can range from 1 to 20 and they are billed on an hourly basis.

It is important to note that the SKU cannot be changed after provisioning an Event Hubs namespace. Due consideration and planning should be undertaken before selecting an SKU. The planning process should include planning the number of consumer groups required and the number of applications interested in reading events from the event hub.

Also, the Standard SKU is not available in every region. It should be checked for availability at the time of the design and implementation of the event hub. The URL for checking region availability is <https://azure.microsoft.com/global-infrastructure/services/?products=event-hubs>.

Event Hubs architecture

There are three main components of the Event Hubs architecture: The **Event Producers**, the **Event Hub**, and the **Event Consumer**, as shown in the following diagram:



Figure 12.4: Event Hubs architecture

Event Producers generate events and send them to the **Event Hub**. The **Event Hub** stores the ingested events and provides that data to the **Event Consumer**. The **Event Consumer** is whatever is interested in those events, and it connects to the **Event Hub** to fetch the data.

Event hubs cannot be created without an Event Hubs namespace. The Event Hubs namespace acts as a container and can host multiple event hubs. Each Event Hubs namespace provides a unique REST-based endpoint that is consumed by clients to send data to Event Hubs. This namespace is the same namespace that is needed for Service Bus artifacts, such as topics and queues.

The connection string of an Event Hubs namespace is composed of its URL, policy name, and key. A sample connection string is shown in the following code block:

```

Endpoint=sb://demoeventhubsbook.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=M/E4eeBsr7DALXcvw6ziFq1SDNbFX6E49Jfti8CRkbA=
  
```

This connection string can be found in the **Shared Access Signature (SAS)** menu item of the namespace. There can be multiple policies defined for a namespace, each having different levels of access to the namespace. The three levels of access are as follows:

- **Manage**: This can manage the event hub from an administrative perspective. It also has rights for sending and listening to events.
- **Send**: This can write events to Event Hubs.
- **Listen**: This can read events from Event Hubs.

By default, the **RootManageSharedAccessKey** policy is created when creating an event hub, as shown in Figure 12.5. Policies help in creating granular access control on Event Hubs. The key associated with each policy is used by consumers to determine their identity; additional policies can also be created with any combination of the three previously mentioned access levels:

The screenshot shows the Azure portal interface for managing shared access policies in an Event Hub namespace. On the left, a sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings, Shared access policies (which is selected and highlighted in grey), Scale, Geo-Recovery, and Networking. The main content area is titled "eventhubforbook | Shared access policies" and shows a table of policies. The table has two columns: "Policy" and "Claims". A single row is selected, showing "RootManageSharedAccessKey" in the Policy column and "Manage, Send, Listen" in the Claims column. Above the table, there are buttons for "Add" and "Search to filter items...". To the right, a detailed view of the "RootManageSharedAccessKey" policy is displayed under the heading "SAS Policy: RootManageShare...". This view includes fields for "Manage" (checked), "Send" (unchecked), and "Listen" (unchecked). It also displays the "Primary key" and "Secondary key" (both represented by long, encoded strings), and the "Connection string-primary key" and "Connection string-secondary key" (both represented by long, encoded strings).

Figure 12.5: Shared access policies in Event Hubs

Event hubs can be created from the Event Hubs namespace service by performing the following actions:

1. Click on **Event Hubs** in the left-hand menu and click on **+ Event Hub** in the resultant screen:

The screenshot shows the Azure portal interface for creating a new Event Hub. At the top, the URL is "Home > NoMarketplace | Overview > eventhubforbook | Event Hubs". On the left, a sidebar lists various settings and entities. Under "Entities", "Event Hubs" is selected and highlighted with a grey bar. In the main content area, there is a search bar ("Search (Ctrl+/)") and a "Event Hub" button with a plus sign. Below these are sections for "Overview", "Activity log", "Access control (IAM)", "Tags", "Diagnose and solve problems", and "Events". A "Settings" section follows, containing links for "Shared access policies", "Scale", "Geo-Recovery", "Networking", "Encryption", "Properties", "Locks", and "Export template". A "Name" input field is present, with the value "testeventhub" typed into it. A "Search to filter items..." input field is also visible.

Figure 12.6: Creating an event hub from the Azure portal

2. Next, provide values for the **Partition Count** and **Message Retention** fields, along with the name of your choice. Then, select **Off** for **Capture**, as demonstrated in *Figure 12.7*:

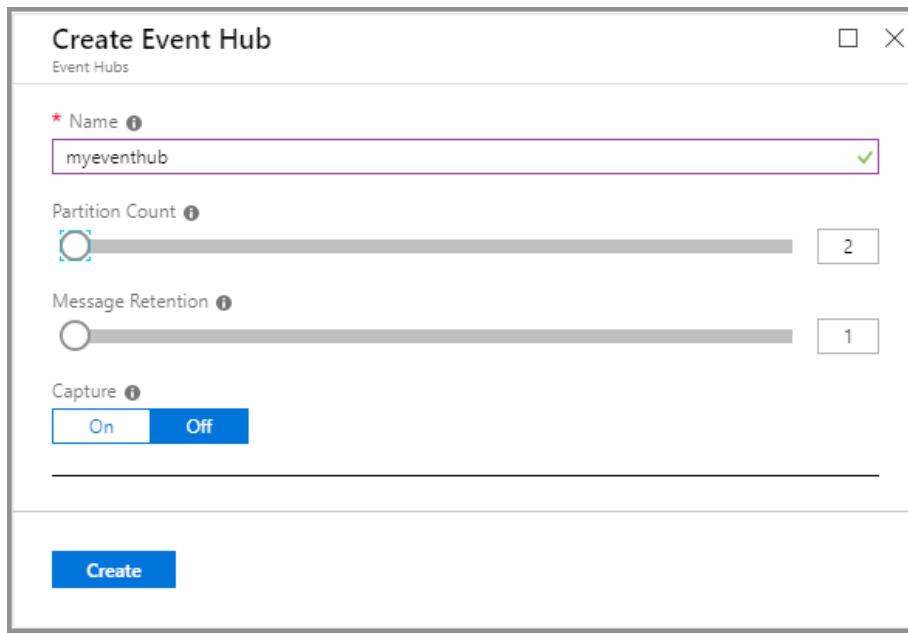


Figure 12.7: Creating a new event hub

After the event hub is created, you will see it in the list of event hubs, as shown in *Figure 12.8*:

The screenshot shows the Azure portal interface for 'eventrg' under 'Event Hubs Namespace'. On the left, there is a sidebar with icons for 'Scale', 'Geo-Recovery', 'Networking', 'Encryption', 'Properties', 'Locks', and 'Export template'. Below that is a section titled 'Entities' with a 'Event Hubs' icon. The main area has a search bar and a 'Refresh' button. A table lists the created event hub: 'myeventhub' (Status: Active, Message Retention: 2 days, Partition Count: 2).

Name	Status	Message Retention	Partition Count
myeventhub	Active	2 days	2

Figure 12.8: List of created event hubs

Event Hubs also allows the storage of events to a storage account or data lake directly using a feature known as Capture.

Capture helps in the automatic storage of ingested data to either an Azure Storage account or an Azure Data Lake Storage account. This feature ensures that the ingestion and storage of events happens in a single step, rather than transferring data into storage being a separate activity:

The screenshot shows the configuration options for the Capture feature. At the top, there is a toggle switch labeled "Capture" with "On" selected. Below it is a note: "Note: Enabling Capture will result in additional charges to this account. Learn more about our pricing here." Under "Capture Provider", "Azure Storage Account" is selected. In the "Azure Storage Container" section, there is a dropdown menu with "Select Container" and a placeholder text area. The "Storage Account" section has a dropdown menu. Below these, under "Sample Capture file name formats", there is a placeholder text area with the format: "{Namespace}/{EventHub}/{PartitionId}/{Year}/{Month}/{Day}/{Hour}/{Minute}/{Second}". In the "Capture file name format" section, there is another placeholder text area with the same format. At the bottom, there is an example: "e.g. eventhubforbook/undefined/0/2020/4/23/7/48/11".

Figure 12.9: Capture feature options

Separate policies can be assigned to each event hub by adding a new policy at the event hub level.

After creating the policy, the connection string is available from the **Secure Access Signature** left-menu item in the Azure portal.

Since a namespace can consist of multiple event hubs, the connection string for an individual event hub will be similar to the following code block. The difference here is in the key value and the addition of **EntityPath** with the name of the event hub:

```
Endpoint=sb://azurertwittereventdata.servicebus.windows
=rxEu5K4Y2qsi5wEe0Ku0vRnhtgW8xW35UBex4V1IKqg=;EntityPath=myeventhub
```

We had to keep the **Capture** option set to **Off** while creating the event hub, and it can be switched back on after creating the event hub. It helps to save events to Azure Blob storage or an Azure Data Lake Storage account automatically. The configuration for the size and time interval is shown in *Figure 12.10*:

The screenshot shows the Azure portal interface for managing an event hub. The left sidebar lists navigation items: Home, eventrg, eventhubforbook, Event Hubs, myeventhub (selected), Event Hubs Instance, Overview, Access control (IAM), Diagnose and solve problems, Settings, Shared access policies, Properties, Locks, Export template, Entities, Consumer groups, Features, Capture (selected), Process data, Support + troubleshooting, and New support request. The main content area is titled "myeventhub (eventhubforbook/myeventhub) | Capture". It contains the following configuration options:

- Capture**: A switch that is currently set to **Off**.
- Note:** Enabling Capture will result in additional charges to this account. Learn more about our pricing [here](#).
- Time window (minutes)**: A slider set to 5.
- Size window (MB)**: A slider set to 300.
- Do not emit empty files when no events occur during the Capture time window**: An unchecked checkbox.
- Capture Provider**: Set to "Azure Storage Account".
- Azure Storage Container ***: A dropdown menu with "Select Container" button.
- Storage Account**: A dropdown menu.
- Sample Capture file name formats**: A dropdown menu showing the format: "{Namespace}/{EventHub}/{PartitionId}/{Year}/{Month}/{Day}/{Hour}/{Minute}/{Second}".
- Capture file name format**: A dropdown menu showing the format: "{Namespace}/{EventHub}/{PartitionId}/{Year}/{Month}/{Day}/{Hour}/{Minute}/{Second}".
- e.g. eventhubforbook/myeventhub/0/2020/05/12/7/36/41**: An example of a capture file name.

Figure 12.10: Selecting the size and time interval for capturing events

We did not cover the concepts of partitions and message retention options while creating event hubs.

Partitioning is an important concept related to the scalability of any data store. Events are retained within event hubs for a specific period of time. If all events are stored within the same data store, then it becomes extremely difficult to scale that data store. Every event producer will connect to the same data store and send their events to it. Compare this with a data store that can partition the same data into multiple smaller data stores, each being uniquely identified with a value.

The smaller data store is called a **partition**, and the value that defines the partition is known as the **partition key**. This partition key is part of the event data.

Now the event producers can connect to the event hub, and based on the value of the partition key, the event hub will store the data in an appropriate partition. This will allow the event hub to ingest multiple events at the same time in parallel.

Deciding on the number of partitions is a crucial aspect of the scalability of an event hub. Figure 12.11 shows that ingested data is stored in the appropriate partition internally by Event Hubs using the partition key:

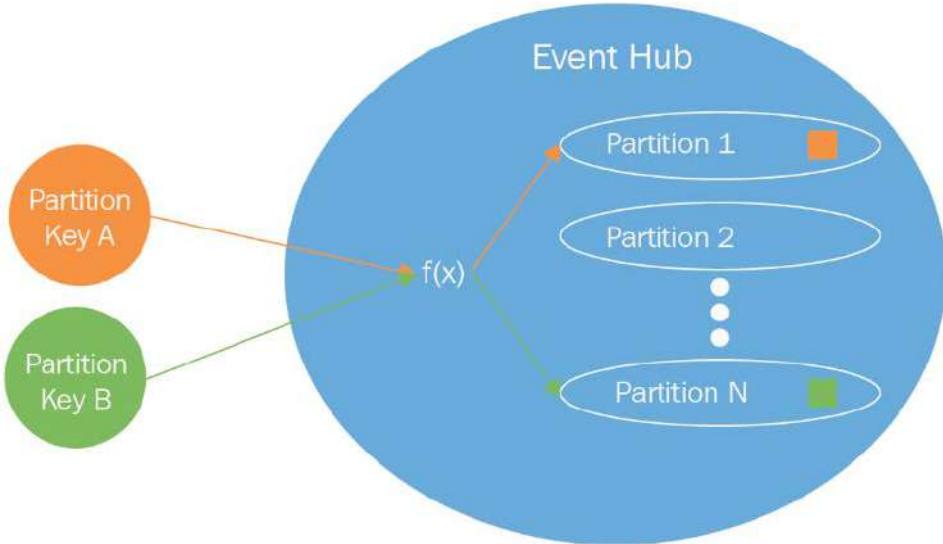


Figure 12.11: Partitioning in an event hub

It is important to understand that one partition might have multiple keys. The user decides how many partitions are required, and the event hub internally decides the best way to allocate the partition keys between them. Each partition stores data in an orderly way using a timestamp, and newer events are appended toward the end of the partition.

It is important to note that it is not possible to change the number of partitions once the event hub is created.

It is also important to remember that partitions also help in bringing parallelism and concurrency for applications reading the events. For example, if there are 10 partitions, 10 parallel readers can read the events without any degradation in performance.

Message retention refers to the time period for which events should be stored. After the expiry of the retention period, the events are discarded.

Consumer groups

Consumers are applications that read events from an event hub. Consumer groups are created for consumers to connect to in order to read the events. There can be multiple consumer groups for an event hub, and each consumer group has access to all the partitions within an event hub. Each consumer group forms a query on the events in events hubs. Applications can use consumer groups and each application will get a different view of the event hub events. A default **\$default** consumer group is created when creating an event hub. It is good practice for one consumer to be associated with one consumer group for optimal performance. However, it is possible to have five readers on each partition in a consumer group:

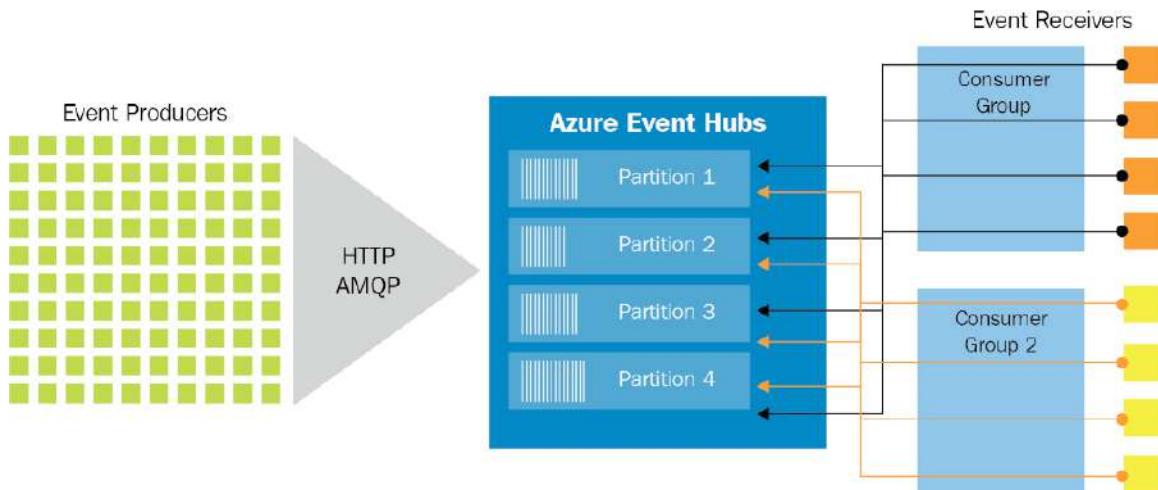


Figure 12.12: Event receivers in a consumer group

Now that you understand consumer groups, it is time to go deeper into the concept of Event Hubs throughput.

Throughput

Partitions help with scalability, while throughput helps with capacity per second. So, what is capacity in terms of Event Hubs? It is the amount of data that can be handled per second.

In Event Hubs, a single TU allows the following:

- 1 MB of ingestion data per second or 1,000 events per second (whichever happens first)
- 2 MB of egress data per second or 4,096 events per second (whichever happens first)

The auto-inflate option helps in increasing the throughput automatically if the number of incoming/outgoing events or the incoming/outgoing total size crosses a threshold. Instead of throttling, the throughput will scale up and down. The configuration of throughput at the time of the creation of the namespace is shown in *Figure 12.13*. Again, careful thought should go into deciding the TUs:

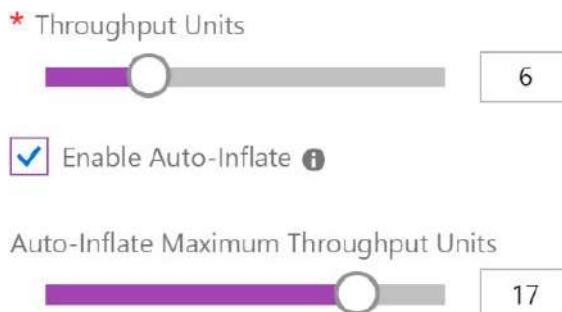


Figure 12.13: Selecting the TUs along with auto-inflate

A primer on Stream Analytics

Event Hubs is a highly scalable data streaming platform, so we need another service that can process these events as a stream rather than just as stored data. Stream Analytics helps in processing and examining a stream of big data, and Stream Analytics jobs help to execute the processing of events.

Stream Analytics can process millions of events per second and it is quite easy to get started with it. Azure Stream Analytics is a PaaS that is completely managed by Azure. Customers of Stream Analytics do not have to manage the underlying hardware and platform.

Each job comprises multiple inputs, outputs, and a query, which transforms the incoming data into new output. The whole architecture of Stream Analytics is shown in Figure 12.14:

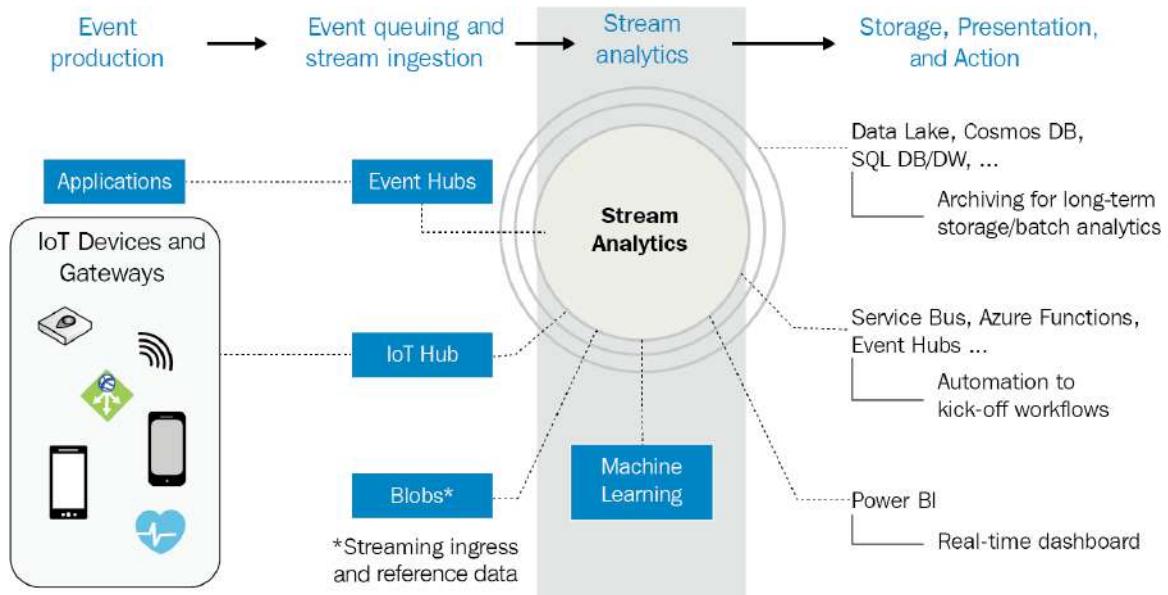


Figure 12.14: Azure Stream Analytics architecture

In Figure 12.14, the event sources are displayed on the extreme left. These are the sources that produce the events. They could be IoT devices, custom applications written in any programming language, or events coming from other Azure platforms, such as Log Analytics or Application Insights.

These events must first be ingested into the system, and there are numerous Azure services that can help to ingest this data. We've already looked at Event Hubs and how they help in ingesting data. There are other services, such as IoT Hub, that also help in ingesting device-specific and sensor-specific data. IoT Hub and ingestion are covered in detail in Chapter 11, Designing IoT Solutions. This ingested data undergoes processing as it arrives in a stream, and this processing is done using Stream Analytics. The output from Stream Analytics could be fed to a presentation platform, such as Power BI, to show real-time data to stakeholders, or a storage platform such as Cosmos DB, Data Lake Storage, or Azure Storage, from which the data can be read and actioned later by Azure Functions and Service Bus queues.

Stream Analytics helps in gathering insights from real-time ingested data within a time window frame and helps in identifying patterns.

It does so through three different tasks:

- **Input:** The data should be ingested within the analytics process. The data can originate from Event Hubs, IoT Hub, or Azure Blob storage. Multiple separate reference inputs using a storage account and SQL Database can be used for lookup data within queries.
- **Query:** This is where Stream Analytics does the core job of analyzing the ingested data and extracting meaningful insights and patterns. It does so with the help of JavaScript user-defined functions, JavaScript user-defined aggregates, Azure Machine Learning, and Azure Machine Learning studio.
- **Output:** The result of the queries can be sent to multiple different types of destinations, and prominent among them are Cosmos DB, Power BI, Synapse Analytics, Data Lake Storage, and Functions:

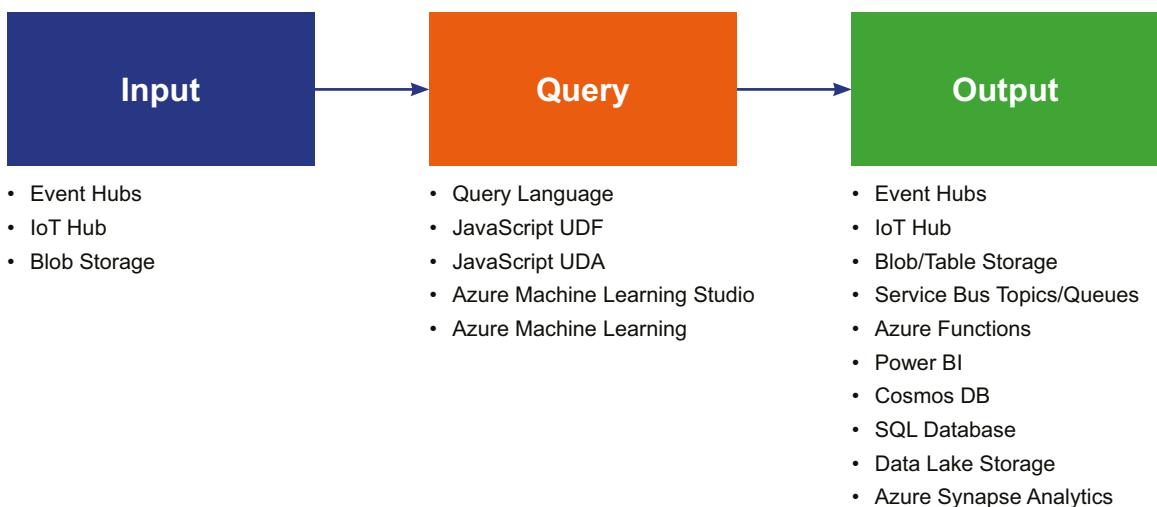


Figure 12.15: Stream Analytics process

Stream Analytics is capable of ingesting millions of events per second and can execute queries on top of them.

Input data is supported in any of the three following formats:

- **JavaScript Object Notation (JSON):** This is a lightweight, plaintext-based format that is human readable. It consists of name-value pairs; an example of a JSON event follows:

```
{
  "SensorId" : 2,
  "humidity" : 60,
  "temperature" : 26C
}
```

- **Comma-Separated Values (CSV)**: These are also plaintext values, which are separated by commas. An example of CSV is shown in *Figure 12.16*. The first row is the header, containing three fields, followed by two rows of data:

```
SensorID, humidity, temperature
2,60,26C
3,65,31C
```

Figure 12.16: Plaintext values

- **Avro**: This format is similar to JSON; however, it is stored in a binary format rather than a text format:

```
{
    "firstname": "Ritesh",
    "lastname": "Modi",
    "email": "rites.modi@outlook.com"
}
```

However, this does not mean that Stream Analytics can only ingest data using these three formats. It can also create custom .NET-based deserializers, using which any format of data can be ingested, depending upon the deserializers' implementation. The steps you can follow to write a custom deserializer are available at <https://docs.microsoft.com/azure/stream-analytics/custom-deserializer-examples>.

Not only can Stream Analytics receive events, but it also provides advanced query capability for the data that it receives. The queries can extract important insights from the temporal data streams and output them.

As shown in *Figure 12.17*, there is an input dataset and an output dataset; the query moves the events from the input to the output. The **INTO** clause refers to the output location, and the **FROM** clause refers to the input location. The queries are very similar to SQL queries, so the learning curve is not too steep for SQL programmers:

The screenshot shows the Azure Stream Analytics query editor interface. At the top, there are buttons for Save, Discard, and Test. Below that, the left sidebar shows 'Inputs (1)' with 'TwitterIncomingEvents' selected, and 'Outputs (1)' with 'TwitterData' selected. The main area contains a query script:

```
1  SELECT
2      *
3  INTO
4      [TwitterData]
5  FROM
6      [TwitterIncomingEvents]
```

A help message at the top right says: 'Need help with your query? Check out some of the most common Stream Analytics query patterns [here](#)'.

Figure 12.17: Stream Analytics query for receiving Twitter data

Event Hubs provides mechanisms for sending outputs from queries to target destinations. At the time of writing, Stream Analytics supports multiple destinations for events and query outputs, as shown before.

It is also possible to define custom functions that can be reused within queries. There are four options provided to define custom functions.

- Azure Machine Learning
- JavaScript user-defined functions
- JavaScript user-defined aggregates
- Azure Machine Learning studio

The hosting environment

Stream Analytics jobs can run on hosts that are running on the cloud, or they can run on IoT edge devices. IoT edge devices are devices that are near to IoT sensors, rather than on the cloud. *Figure 12.18* shows the **New Stream Analytics job** pane:

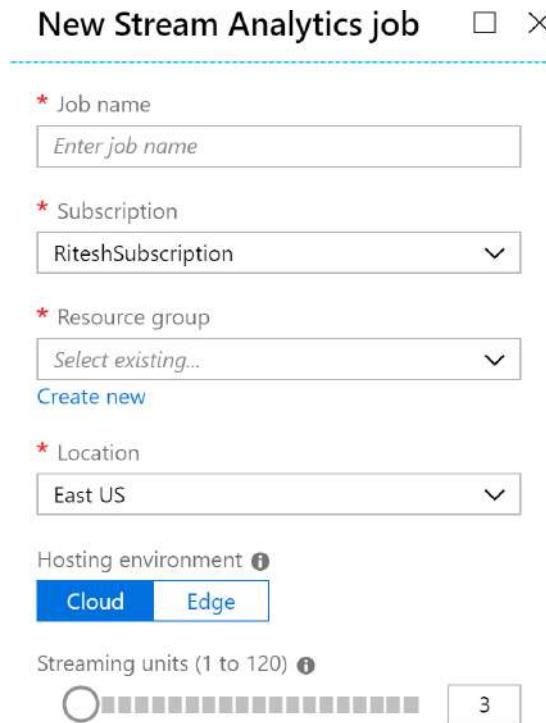


Figure 12.18: Creating a new Stream Analytics job

Let's check out streaming units in detail.

Streaming units

From Figure 12.18, you can see that the only configuration that is unique to Stream Analytics is streaming units. Streaming units refers to the resources (that is, CPU and memory) that are assigned for running a Stream Analytics job. The minimum and maximum streaming units are 1 and 120, respectively.

Streaming units must be pre-allocated according to the amount of data and the number of queries executed on that data; otherwise, the job will fail.

It is possible to scale streaming units up and down from the Azure portal.

A sample application using Event Hubs and Stream Analytics

In this section, we will be creating a sample application comprising multiple Azure services, including Azure Logic Apps, Azure Event Hubs, Azure Storage, and Azure Stream Analytics.

In this sample application, we will be reading all tweets containing the word "Azure" and storing them in an Azure storage account.

To create this solution, we first need to provision all the necessary resources.

Provisioning a new resource group

Navigate to the Azure portal, log in using valid credentials, click on **+ Create a resource**, and search for **Resource group**. Select **Resource group** from the search results and create a new resource group. Then, provide a name and choose an appropriate location. Note that all resources should be hosted in the same resource group and location so that it is easy to delete them:

Home > New > Marketplace > Everything > Resource group > Create a resource group

Create a resource group

Basics Tags Review + Create

Resource group - A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. [Learn more](#)

PROJECT DETAILS

* Subscription

* Resource group

RESOURCE DETAILS

* Region

Figure 12.19: Provisioning a new resource group in the Azure portal

Next, we will create an Event Hubs namespace.

Creating an Event Hubs namespace

Click on **+ Create a resource** and search for **Event Hubs**. Select **Event Hubs** from the search results and create a new event hub. Then, provide a name and location, and select a subscription based on the resource group that was created earlier. Select **Standard** as the pricing tier and also select **Enable Auto-inflate**, as shown in Figure 12.20:

The screenshot shows the 'Create Namespace' wizard for Event Hubs. The steps are as follows:

- Name:** AzureTwitterEventData (highlighted with a red border)
- Namespace already exists. Enter a different name.** (Error message)
- Pricing tier:** Standard (20 Consumer groups, 1000 B...)
- Enable Kafka:** Unchecked
- Make this namespace zone redundant:** Unchecked
- Subscription:** Visual Studio Enterprise
- Resource group:** TwitterAnalysis (with 'Create new' link)
- Location:** West Europe
- Throughput Units:** A slider set to 1, with a text input field showing '1'.
- Enable Auto-Inflate:** Checked (indicated by a checked checkbox)
- Auto-Inflate Maximum Throughput Units:** A slider set to 1, with a text input field showing '1'.
- Create** button at the bottom.

Figure 12.20: Creating an Event Hubs namespace

By now, an Event Hubs namespace should have been created. It is a pre-requisite to have a namespace before an event hub can be created. The next step is to provision an event hub.

Creating an event hub

From the Event Hubs namespace service, click on **Events Hubs** in the left-hand menu, and then click on **+ Event hubs** to create a new event hub. Name it **azuretwitterdata** and provide an optimal number of partitions and a **Message Retention** value:

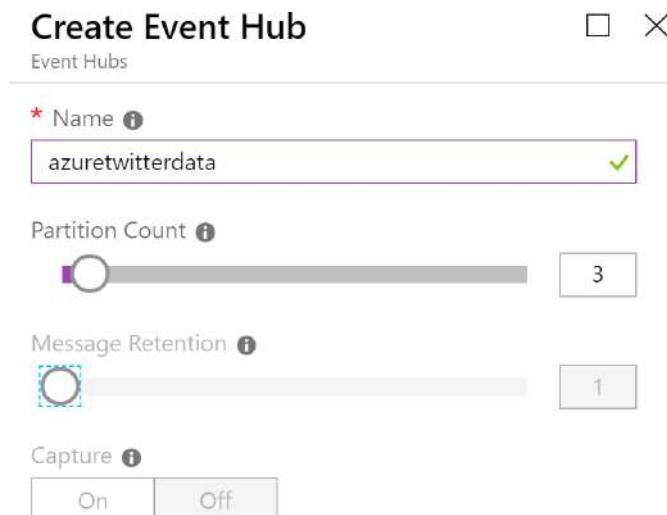


Figure 12.21: Creating the azuretwitterdata event hub

After this step, you will have an event hub that can be used to send event data, which is stored in durable storage such as a Data Lake Storage account or an Azure Storage account, to be used by downstream services.

Provisioning a logic app

After the resource group is provisioned, click on **+ Create a resource** and search for **Logic Apps**. Select **Logic Apps** from the search results and create a new logic app. Then, provide a name and location, and select a subscription based on the resource group created earlier. It is good practice to enable **Log Analytics**. Logic Apps is covered in more detail in *Chapter 11, Azure Solutions using Azure Logic Apps, Event Grid, and Functions*. The logic app is responsible for connecting to Twitter using an account and fetching all the tweets with **Azure** in them:

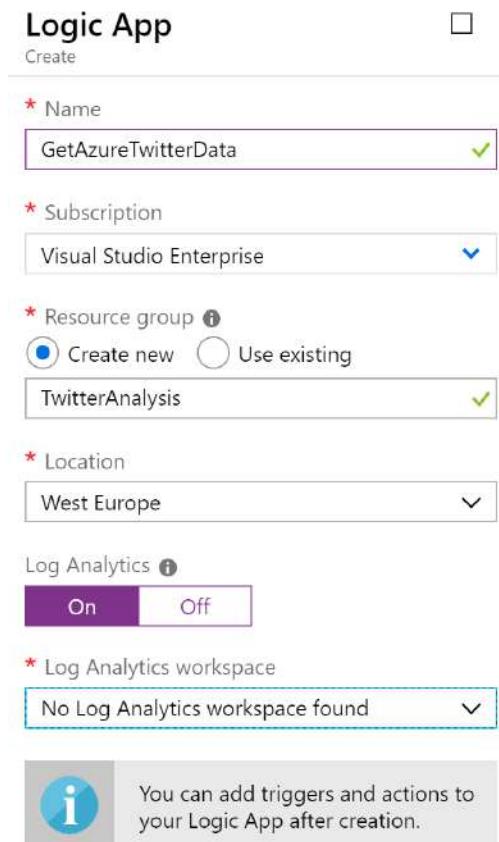


Figure 12.22: Creating a logic app

After the logic app is created, select the **When a new tweet is posted** trigger on the design surface, sign in, and then configure it as shown in Figure 12.23. You will need a valid Twitter account before configuring this trigger:

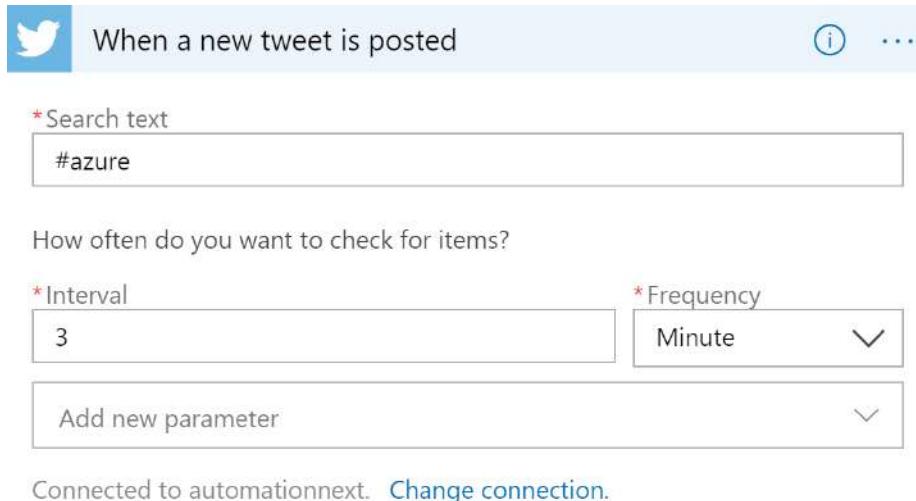
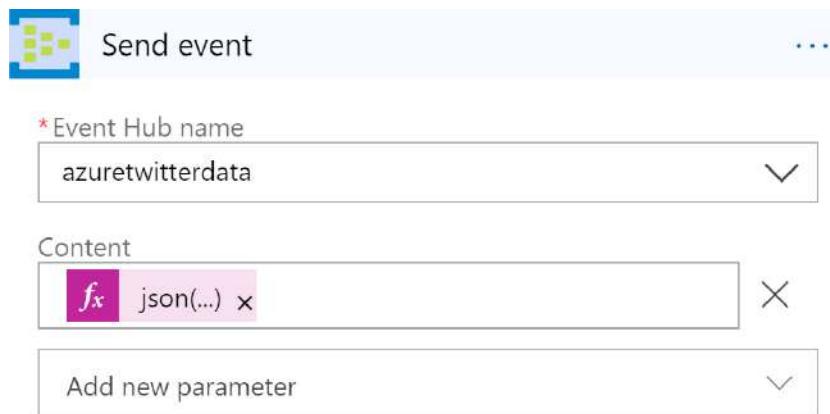


Figure 12.23: Configuring the frequency of incoming tweets

Next, drop a **Send event** action on the designer surface; this action is responsible for sending tweets to the event hub:



Connected to SendTwitterData. [Change connection](#).

Figure 12.24: Adding an action to send tweets to the event hub

Select the name of the event hub that was created in an earlier step.

The value specified in the content textbox is an expression that has been dynamically composed using Logic Apps-provided functions and Twitter data. Clicking on **Add dynamic content** provides a dialog through which the expression can be composed:

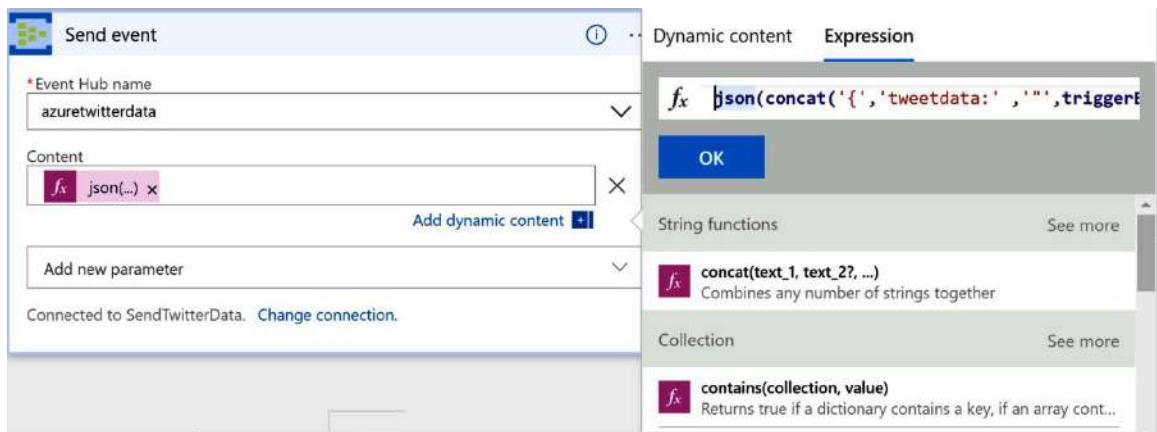


Figure 12.25: Configuring Logic Apps activity using dynamic expressions

The value of the expression is as follows:

```
json(concat('{', 'tweetdata:', ''', triggerBody()?['TweetText'], ''', '}'))
```

In the next section, we will provision the storage account.

Provisioning the storage account

Click on **+ Create a resource** and search for **Storage Account**. Select **Storage Account** from the search results and create a new storage account. Then, provide a name and location, and select a subscription based on the resource group that was created earlier. Finally, select **StorageV2** for **Account Kind**, **Standard** for **Performance**, and **Locally-redundant storage (LRS)** for the **Replication** field.

Next, we will create a Blob storage container to store the data coming out of Stream Analytics.

Creating a storage container

Stream Analytics will output the data as files, which will be stored within a Blob storage container. A container named **twitter** will be created within Blob storage, as shown in Figure 12.26:

The screenshot shows the Azure Storage account interface for 'twittereventdata - Blobs'. On the left, a sidebar lists navigation options: Overview, Activity log, Access control (IAM) (selected), Tags, Diagnose and solve problems, Events, and Storage Explorer (preview). The main area displays a table of containers. One container, 'twitter', is listed with the following details:

NAME	LAST MODIFIED	PUBLIC ACCESS L...	LEASE STATE
twitter	1/25/2019, 8:57:50 AM	Container	Available

Figure 12.26: Creating a storage container

Let's create a new Stream Analytics job with a hosting environment on the cloud and set the streaming units to the default settings.

Creating Stream Analytics jobs

The input for this Stream Analytics job comes from the event hub, and so we need to configure this from the **Inputs** menu:

The screenshot shows the Azure Stream Analytics job creation interface. The left sidebar includes options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (with Locks), Job topology (with Inputs selected), Functions, Query, Outputs, Configure (with Scale, Locale, Event ordering, Error policy, and Compatibility level), and Help. The main area is titled 'Add stream input' and shows the configuration for an input named 'TwitterIncomingEvents':

- NAME:** TwitterIncomingEvents
- SOURCE TYPE:** Event Hub
- SOURCE:** Empty
- Subscription:** RiteshSubscription
- Event Hub namespace:** AzureTwitterEventData
- Event Hub name:** azuretwitterdata
- Event Hub policy name:** RootManageSharedAccessKey
- Event Hub policy key:** (redacted)
- Event Hub consumer group:** (redacted)
- Event serialization format:** JSON
- Encoding:** UTF-8

Figure 12.27: Creating an input Stream Analytics job

The output for the Stream Analytics job is a Blob storage account, so you need to configure the output accordingly. Provide a path pattern that is suitable for this exercise; for example, **{datetime:ss}** is the path pattern that we are using for this exercise:

The screenshot shows the 'Outputs' blade for the 'StreamAzureTwitterJob'. On the left, a sidebar lists various settings like Overview, Activity log, and Tags. The 'Outputs' section is selected. A table lists one output named 'TwitterData' with a 'Sink' type of 'Blob storage'. To the right, the configuration pane is displayed for 'TwitterData'. It includes fields for 'Output alias' (set to 'TwitterData'), 'Subscription' (set to 'RiteshSubscription'), 'Storage account' (set to 'twittereventdata'), 'Storage account key' (redacted), 'Container' (set to 'Use existing' with value 'twitter'), 'Path pattern' (set to '{datetime:ss}'), 'Date format' (set to 'YYYY/MM/DD'), and 'Time format' (set to 'HH'). A 'Save' button is at the bottom.

Figure 12.28: Creating a Blob storage account as output

The query is quite simple; you are just copying the data from the input to the output:

The screenshot shows the 'Query' blade for the 'StreamAzureTwitterJob'. The sidebar has 'Query' selected. The main area shows the query structure with 'Inputs (1)' and 'Outputs (1)'. The 'Outputs (1)' section shows 'TwitterData'. Below the inputs, a code editor displays the following T-SQL query:

```

1 SELECT
2 *
3 INTO
4 [TwitterData]
5 FROM
6 [TwitterIncomingEvents]

```

Figure 12.29: Query for copying Twitter feeds

While this example just involves copying data, there can be more complex queries for performing transformation before loading data into a destination.

This concludes all the steps for the application; now you should be able to run it.

Running the application

The logic app should be enabled and Stream Analytics should be running. Now, run the logic app; it will create a job to run all the activities within it, as shown in Figure 12.30:

The screenshot shows the Azure Logic App interface for the 'GetAzureTwitterData' application. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Development Tools (Logic app designer, Logic app code view, Versions, API connections, Quick start guides, Release notes), Settings (Workflow settings, Access keys, Identity, Properties, Locks), and a search bar. The main content area displays the following details:

- Resource group (change):** TwitterAnalysis
- Location:** West Europe
- Subscription (change):** RiteshSubscription
- Subscription ID:** 9755ffce-e9cb-4332-9be8-1ade15e78909
- Status:** Enabled
- Runs last 24 hours:** 217 successful, 1275 failed
- Integration Account:** ---

Summary

Trigger	Actions
TWITTER When a new tweet is posted	COUNT 1 action View in Logic Apps designer
FREQUENCY Runs every 3 minutes	
EVALUATION Evaluated 2357 times, fired 1494 times in the last 24 hours See trigger history	

Runs history

STATUS	START TIME	IDENTIFIER	DURATION
Succeeded	1/26/2019, 10:16 AM	08586530910864161272447961552CU24	299 Milliseconds
Succeeded	1/26/2019, 10:15 AM	08586530911478695689296257377CU29	447 Milliseconds

Figure 12.30: Overview of the GetAzureTwitterData application

The **Storage Account** container should get data, as shown in *Figure 12.31*:

The screenshot shows the Azure Storage Explorer interface. On the left, there's a sidebar with options like Overview, Access Control (IAM), Properties, and Metadata. The main area shows a container named 'twitter'. Inside, there's a blob named '0_48c599701d414248959070c92c937b3a_1.json'. The properties for this blob are displayed on the right, including:

Properties	Value
URL	https://twittereventdata.blob.c...
LAST MODIFIED	1/25/2019, 9:06:25 AM
CREATION TIME	1/25/2019, 9:06:24 AM
TYPE	Block blob
SIZE	10.8 kB
SERVER ENCRYPTED	true
ETAG	0x8D682CE4AF8642B
CONTENT-MD5	-
LEASE STATUS	Unlocked
LEASE STATE	Available
LEASE DURATION	-
COPY STATUS	-
COPY COMPLETION TIME	-

Figure 12.31: Checking the Storage Account container data

As an exercise, you can extend this sample solution and evaluate the sentiment of the tweets every three minutes. The Logic Apps workflow for such an exercise would be as follows:

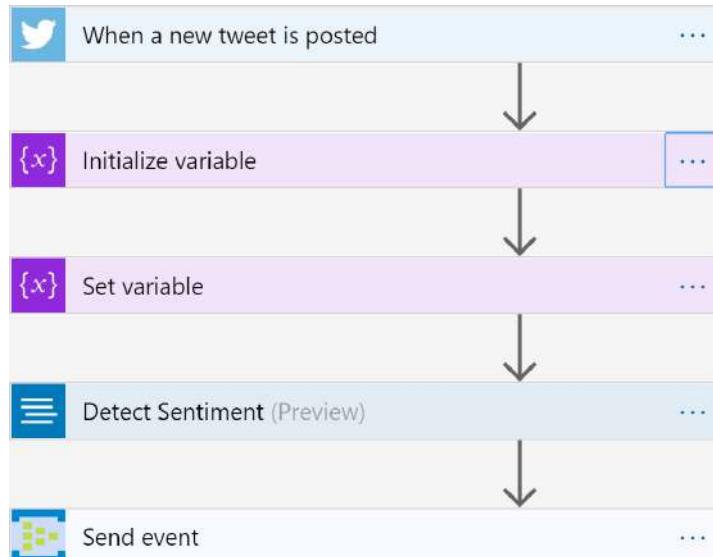


Figure 12.32: Flowchart for analyzing tweet sentiment

To detect sentiment, you'll need to use the Text Analytics API, which should be configured before being used in Logic Apps.

Summary

This chapter focused on topics related to the streaming and storage of events. Events have become an important consideration in overall solution architecture. We covered important resources, such as Event Hubs and Stream Analytics, and foundational concepts, such as consumer groups and throughputs, as well as creating an end-to-end solution using them along with Logic Apps. You learned that events are raised from multiple sources, and in order to get insights in real time about activities and their related events, services such as Event Hubs and Stream Analytics play a significant role. In the next chapter, we will learn about integrating Azure DevOps and Jenkins and implementing some of the industry's best practices while developing solutions.

13

Integrating Azure DevOps

In the previous chapter, you learned about big data eventing and its relationship with Azure's Event Hubs and Stream Analytics services. Software development is a complex undertaking comprising multiple processes and tools, and involving people from different departments. They all need to come together and work in a cohesive manner. With so many variables, the risks are high when you are delivering to end customers. One small omission or misconfiguration might lead to the application coming crashing down. This chapter is about adopting and implementing practices that reduce this risk considerably and ensure that high-quality software can be delivered to the customer over and over again.

Before getting into the details of DevOps, here is a list of the problems faced by software companies that DevOps addresses:

- Rigid organizations that don't welcome change
- Time-consuming processes
- Isolated teams working in silos
- Monolithic design and big bang deployments
- Manual execution
- A lack of innovation

In this chapter, we will cover the following topics:

- DevOps
- DevOps practices
- Azure DevOps
- DevOps preparation
- DevOps for PaaS solutions
- DevOps for virtual machine-based (IaaS) solutions
- DevOps for container-based (IaaS) solutions
- Azure DevOps and Jenkins
- Azure Automation
- Azure tools for DevOps

DevOps

There's currently no industry-wide consensus regarding the definition of DevOps. Organizations have formulated their own definition of DevOps and tried to implement it. They have their own perspective and think they've implemented DevOps once they implement automation and configuration management, and use Agile processes.

Based on my experience working on DevOps projects in industry, I have defined DevOps as the following: DevOps is about the delivery mechanism of software systems. It's about bringing people together, making them collaborate and communicate, working together toward a common goal and vision. It's about taking joint responsibility, accountability, and ownership. It's about implementing processes that foster collaboration and a service mindset. It enables delivery mechanisms that bring agility and flexibility to the organization. Contrary to popular belief, DevOps isn't about tools, technology, and automation. These are enablers that help with collaboration, the implementation of Agile processes, and faster and better delivery to the customer.

There are multiple definitions available on the internet for DevOps, and they aren't wrong. DevOps doesn't provide a framework or methodology. It's a set of principles and practices that, when employed within an organization, engagement, or project, achieve the goal and vision of both DevOps and the organization. These principles and practices don't mandate any specific processes, tools and technologies, or environments. DevOps provides guidance that can be implemented through any tool, technology, or process, although some of the technology and processes might be more applicable than others to achieve the vision of DevOps' principles and practices.

Although DevOps practices can be implemented in any organization that provides services and products to customers, going forward in this book, we'll look at DevOps from the perspective of software development and the operations department of any organization.

So, what is DevOps? DevOps is defined as a set of principles and practices bringing all teams, including developers and operations, together from the start of the project for faster, quicker, and more efficient end-to-end delivery of value to the end customer again and again, in a consistent and predictable manner, reducing time to market, thereby gaining a competitive advantage.

The preceding definition of DevOps doesn't indicate or refer to any specific processes, tools, or technology. It doesn't prescribe any methodology or environment.

The goal of implementing DevOps principles and practices in any organization is to ensure that the demands of stakeholders (including customers) and expectations are met efficiently and effectively.

Customer demands and expectations are met when the following happens:

- The customer gets the features they want
- The customer gets the features they want when they want them
- The customer gets faster updates on features
- The quality of delivery is high

When an organization can meet these expectations, customers are happy and remain loyal. This, in turn, increases the market competitiveness of the organization, which results in a bigger brand and market valuation. It has a direct impact on the top and bottom lines of the organization. The organization can invest further in innovation and customer feedback, bringing about continuous changes to its systems and services in order to stay relevant.

The implementation of DevOps principles and practices in any organization is guided by its surrounding ecosystem. This ecosystem is made up of the industry and domains the organization belongs to.

DevOps is based on a set of principles and practices. We'll look into the details of these principles and practices later in this chapter. The core principles of DevOps are:

- **Agility:** Being Agile increases the overall flexibility to changes and ensures that adaptability increases to every changing environment and being productive. Agile processes have a shorter work duration and it's easy to find issues earlier in the development life cycle rather than much later, thereby reducing the technical debt.
- **Automation:** The adoption of tools and automation increases the overall efficiency and predictability of the process and end product. It helps in doing things faster and in an easier and cheaper manner.
- **Collaboration:** Collaboration refers to a common repository, the rotation of work responsibilities, the sharing of data and information, and other aspects that improve the productivity of each member of the team, thereby supporting the overall effective delivery of the product.
- **Feedback:** This refers to quick and early feedback loops between multiple teams about things that work and things that don't work. It helps teams to prioritize issues and fix them in subsequent releases.

The core DevOps practices are:

- **Continuous integration:** This refers to the process of validating and verifying the quality and correctness of the code pushed within the repository by developers. It can be scheduled, manual, or continuous. Continuous means that the process will check for various quality attributes each time a developer pushes the code, while scheduled means on a given time schedule, the checks will be conducted. Manual refers to manual execution by an administrator or developer.
- **Configuration management:** This is an important facet of DevOps and provides guidance for configuring infrastructure and applications either by pulling configurations from configuration management servers or by pushing these configurations on a schedule. Configuration management should bring back the environment to the expected desired state every time it gets executed.
- **Continuous delivery:** Continuous delivery refers to the state of readiness of an application to be able to be deployed in any existing, as well as a new, environment. It is generally executed by means of a release definition in lower environments like development and testing.
- **Continuous deployment:** Continuous deployment refers to the ability to deploy the environment and application in production automatically. It is generally executed by means of a release definition in the production environment.
- **Continuous learning:** This refers to the process of understanding the issues faced by operations and customers and ensuring that they get communicated to development and testing teams such that they can fix those issues in subsequent releases to improve the overall health and usability of the application.

The essence of DevOps

DevOps is not a new paradigm; however, it's gaining a lot of popularity and traction. Its adoption is at its highest level, and more and more companies are undertaking this journey. I purposely mentioned DevOps as a journey because there are different levels of maturity within DevOps. While successfully implementing continuous deployment and delivery are considered the highest level of maturity in this journey, adopting source code control and Agile software development are considered the first step in the DevOps journey.

One of the first things DevOps talks about is breaking down the barriers between the development and the operations teams. It brings about close collaboration between multiple teams. It's about breaking the mindset that the developer is responsible for writing the code only and passing it on to operations for deployment once it's tested. It's also about breaking the mindset that operations have no role to play in development activities. Operations should influence the planning of the product and should be aware of the features coming up as releases. They should also continually provide feedback to the developers on operational issues such that they can be fixed in subsequent releases. They should influence the design of the system to improve the operational working of the system. Similarly, developers should help the operations team to deploy the system and solve incidents when they arise.

The definition of DevOps talks about faster and more efficient end-to-end delivery of systems to stakeholders. It doesn't talk about how fast or efficient the delivery should be. It should be fast enough for the organization's domain, industry, customer segmentation, and needs. For some organizations, quarterly releases are good enough, while for others it could be weekly. Both are valid from a DevOps point of view, and these organizations can deploy relevant processes and technologies to achieve their target release deadlines. DevOps doesn't mandate any specific time frame for **continuous integration/continuous deployment (CI/CD)**. Organizations should identify the best implementation of DevOps principles and practices based on their overall project, engagement, and organizational vision.

The definition also talks about end-to-end delivery. This means that everything from the planning and delivery of the system through to the services and operations should be part of DevOps adoption. Processes should allow greater flexibility, modularity, and agility in the application development life cycle. While organizations are free to use the best fitting process—Waterfall, Agile, Scrum, or another—typically, organizations tend to favor Agile processes with iteration-based delivery. This allows faster delivery in smaller units, which are far more testable and manageable compared to a large delivery.

DevOps repeatedly talks about end customers in a consistent and predictable manner. This means that organizations should continually deliver to customers with newer and upgraded features using automation. We can't achieve consistency and predictability without the use of automation. Manual work should be non-existent to ensure a high level of consistency and predictability. Automation should also be end-to-end, to avoid failures. This also indicates that the system design should be modular, allowing faster delivery on systems that are reliable, available, and scalable. Testing plays a big role in consistent and predictable delivery.

The end result of implementing these practices and principles is that the organization is able to meet the expectations and demands of customers. The organization is able to grow faster than the competition, and further increase the quality and capability of their product and services through continuous innovation and improvement.

Now that you understand the idea behind DevOps, it's time to look into core DevOps practices.

DevOps practices

DevOps consists of multiple practices, each providing a distinct functionality to the overall process. The following diagram shows the relationship between them. Configuration management, continuous integration, and continuous deployment form the core practices that enable DevOps. When we deliver software services that combine these three services, we achieve continuous delivery. Continuous delivery is the capability and level of maturity of an organization that's dependent on the maturity of configuration management, continuous integration, and continuous deployment. Continuous feedback, at all stages, forms the feedback loop that helps to provide superior services to customers. It runs across all DevOps practices. Let's deep dive into each of these capabilities and DevOps practices:

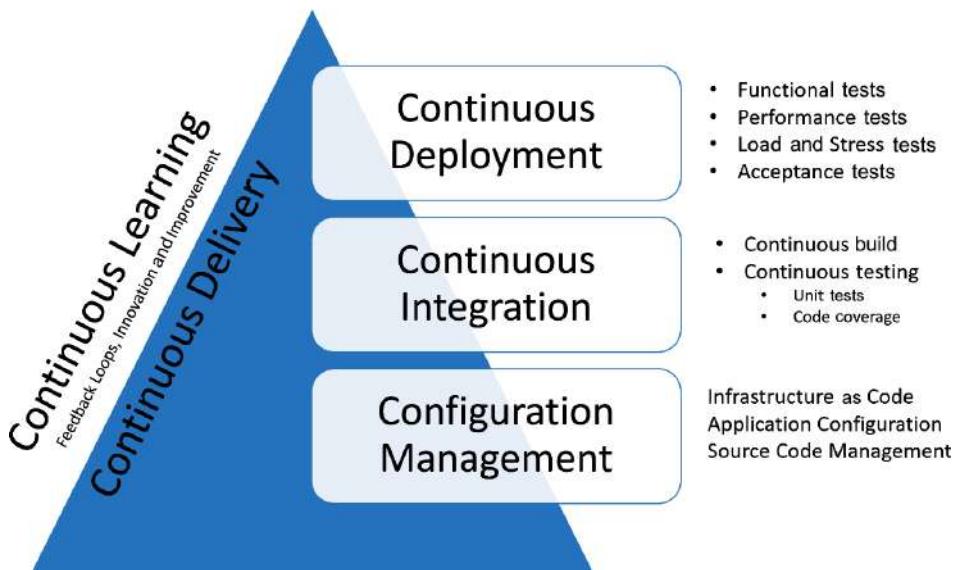


Figure 13.1: DevOps practices

Configuration management

Business applications and services need an environment in which they can be deployed. Typically, the environment is an infrastructure composed of multiple servers, computers, network, storage, containers, and many more services working together such that business applications can be deployed on top of them. Business applications are decomposed into multiple services running on multiple servers, either on-premises or on the cloud, and each service has its own configuration along with requirements related to the infrastructure's configuration. In short, both the infrastructure and the application are needed to deliver systems to customers, and both of them have their own configuration. If the configuration drifts, the application might not work as expected, leading to downtime and failure. Moreover, as the **Application Lifecycle Management (ALM)** process dictates the use of multiple stages and environments, an application would be deployed to multiple environments with different configurations. The application would be deployed to the development environment for developers to see the result of their work. It would then be deployed to multiple test environments with different configurations for functional tests, load and stress tests, performance tests, integration tests, and more; it would also be deployed to the preproduction environment to conduct user-acceptance tests, and finally into the production environment. It's important that an application can be deployed to multiple environments without undertaking any manual changes to its configuration.

Configuration management provides a set of processes and tools and they help to ensure that each environment and application gets its own configuration. Configuration management tracks configuration items, and anything that changes from environment to environment should be treated as a configuration item. Configuration management also defines the relationships between configuration items and how changes in one configuration item will impact other configuration items.

Usage of configuration management

Configuration management helps in the following places:

- **Infrastructure as Code:** When the process of provisioning infrastructure and its configuration is represented through code, and the same code goes through the application life cycle process, it's known as **Infrastructure as Code (IaC)**. IaC helps to automate the provisioning and configuration of infrastructure. It also represents the entire infrastructure in code that can be stored in a repository and version-controlled. This allows users to employ the previous environment's configurations when needed. It also enables the provisioning of an environment multiple times in a consistent and predictable manner. All environments provisioned in this way are consistent and equal in all ALM stages. There are many tools that help in achieving IaC, including ARM templates, Ansible, and Terraform.

- **Deploying and configuring the application:** The deployment of an application and its configuration is the next step after provisioning the infrastructure. Examples include deploying a `webdeploy` package on a server, deploying a SQL server schema and data (`bacpac`) on another server, and changing the SQL connection string on the web server to represent the appropriate SQL server. Configuration management stores values for the application's configuration for each environment on which it is deployed.

The configuration applied should also be monitored. The expected and desired configuration should be consistently maintained. Any drift from this expected and desired configuration would render the application unavailable. Configuration management is also capable of finding the drift and re-configuring the application and environment to its desired state.

With automated configuration management in place, nobody on the team has to deploy and configure environments and applications in production. The operations team isn't reliant on the development team or long deployment documentation.

Another aspect of configuration management is source code control. Business applications and services comprise code and other artifacts. Multiple team members work on the same files. The source code should always be up to date and should be accessible by only authenticated team members. The code and other artifacts by themselves are configuration items. Source control helps in collaboration and communication within the team since everybody is aware of what everyone else is doing and conflicts are resolved at an early stage.

Configuration management can be broadly divided into two categories:

- Inside the virtual machine
- Outside the virtual machine

Configuration management tools

The tools available for configuration management inside the virtual machine are discussed next.

Desired State Configuration

Desired State Configuration (DSC) is a configuration-management platform from Microsoft, built as an extension to PowerShell. DSC was originally launched as part of **Windows Management Framework (WMF) 4.0**. It's available as part of WMF 4.0 and 5.0 for all Windows Server operating systems before Windows 2008 R2. WMF 5.1 is available out of the box on Windows Server 2016/2019 and Windows 10.

Chef, Puppet, and Ansible

Apart from DSC, there's a host of configuration-management tools, such as Chef, Puppet, and Ansible, supported by Azure. Details about these tools aren't covered in this book. Read more about them here: <https://docs.microsoft.com/azure/virtual-machines/windows/infrastructure-automation>.

The tools available for configuration management outside of a virtual machine are mentioned next.

ARM templates

ARM templates are the primary means of provisioning resources in ARM. ARM templates provide a declarative model through which resources and their configuration, scripts, and extensions are specified. ARM templates are based on **JavaScript Object Notation (JSON)** format. It uses JSON syntax and conventions to declare and configure resources. JSON files are text-based, user friendly, and easily readable. They can be stored in a source code repository and have version control on them. They are also a means to represent infrastructure as code that can be used to provision resources in Azure resource groups over and over again, predictably, consistently, and uniformly.

Templates provide the flexibility to be generic and modular in their design and implementation. Templates give us the ability to accept parameters from users, declare internal variables, help define dependencies between resources, link resources within the same or different resource groups, and execute other templates. They also provide scripting language-type expressions and functions that make them dynamic and customizable at runtime. There are two chapters dedicated to ARM templates in this book: *Chapters 15, Cross Subscription Deployments Using ARM Templates*, and *Chapter 16, ARM Templates Modular Design and Implementation*.

Now, it's time to focus on the next important DevOps principle: continuous integration.

Continuous integration

Multiple developers write code that's eventually stored in a common repository. The code is normally checked in or pushed to the repository when the developers have finished developing their features. This can happen in a day or might take days or weeks. Some of the developers might be working on the same feature, and they might also follow the same practices of pushing/checking in code in days or weeks. This can create issues with the quality of the code. One of the tenets of DevOps is to fail fast. Developers should check in/push their code to the repository often and compile the code to check whether they've introduced bugs and that the code is compatible with the code written by their colleagues. If a developer doesn't follow this practice, the code on their machine will grow too large and will be difficult to integrate with other code. Moreover, if the compile fails, it's difficult and time-consuming to fix the issues that arise.

Code integration

Continuous integration solves these kinds of challenges. Continuous integration helps in compiling and validating the code pushed/checked in by a developer by taking it through a series of validation steps. Continuous integration creates a process flow that consists of multiple steps. Continuous integration is composed of continuous automated builds and continuous automated tests. Normally, the first step is compiling the code. After the successful compilation, each step is responsible for validating the code from a specific perspective. For example, unit tests can be executed on the compiled code, and then code coverage can be executed to check which code paths are executed by unit tests. These could reveal whether comprehensive unit tests are written or whether there's scope to add further unit tests. The end result of continuous integration is deployment packages that can be used by continuous deployment to deploy them to multiple environments.

Frequent code push

Developers are encouraged to check in their code multiple times a day, instead of doing so after days or weeks. Continuous integration initiates the execution of the entire pipeline as soon as the code is checked in or pushed. If compilation succeeds, code tests, and other activities that are part of the pipeline, are executed without error; the code is deployed to a test environment and integration tests are executed on it.

Increased productivity

Continuous integration increases developer productivity. They don't have to manually compile their code, run multiple types of tests one after another, and then create packages out of it. It also reduces the risk of getting bugs introduced in the code and the code doesn't get stale. It provides early feedback to the developers about the quality of their code. Overall, the quality of deliverables is high and they are delivered faster by adopting continuous integration practices. A sample continuous integration pipeline is shown here:

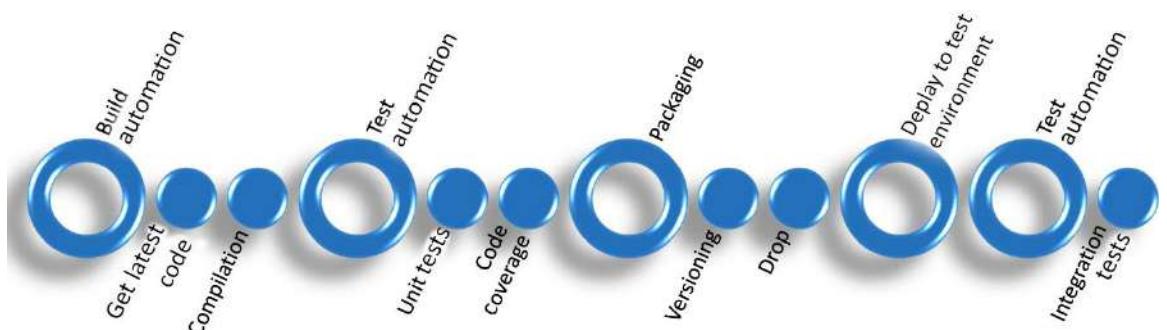


Figure 13.2: Continuous integration pipeline

Build automation

Build automation consists of multiple tasks executing in sequence. Generally, the first task is responsible for fetching the latest source code from the repository. The source code might comprise multiple projects and files. They are compiled to generate artifacts, such as executables, dynamic link libraries, and assemblies. Successful build automation reflects that there are no compile-time errors in the code.

There could be more steps to build automation, depending on the nature and type of the project.

Test automation

Test automation consists of tasks that are responsible for validating different aspects of code. These tasks are related to testing code from a different perspective and are executed in sequence. Generally, the first step is to run a series of unit tests on the code. Unit testing refers to the process of testing the smallest denomination of a feature by validating its behavior in isolation from other features. It can be automated or manual; however, the preference is toward automated unit testing.

Code coverage is another type of automated testing that can be executed on code to find out how much of the code is executed when running unit tests. It's generally represented as a percentage and refers to how much code is testable through unit testing. If the code coverage isn't close to 100%, it's either because the developer hasn't written unit tests for that behavior or the uncovered code isn't required at all.

The successful execution of test automation, resulting in no significant code failure, should start executing the packaging tasks. There could be more steps to test automation depending on the nature and type of the project.

Packaging

Packaging refers to the process of generating deployable artifacts, such as **MSI**, **NuGet**, and **webdeploy** packages, and database packages; versioning them; and then storing them in a location such that they can be consumed by other pipelines and processes.

Once the process of continuous integration completes, the process of continuous deployment starts, and that will be the focus of the next section.

Continuous deployment

By the time the process reaches continuous deployment, continuous integration has ensured that we have fully working bits of an application that can now be taken through different continuous deployment activities. Continuous deployment refers to the capability of deploying business applications and services to preproduction and production environments through automation. For example, continuous deployment could provision and configure the preproduction environment, deploy applications to it, and configure the applications. After conducting multiple validations, such as functional tests and performance tests on the preproduction environment, the production environment is provisioned, configured, and the application is deployed through automation. There are no manual steps in the deployment process. Every deployment task is automated. Continuous deployment can provision the environment and deploy the application from scratch, while it can just deploy delta changes to an existing environment if the environment already exists.

All environments are provisioned through automation using IaC. This ensures that all environments, whether development, test, preproduction, or production, are the same. Similarly, the application is deployed through automation, ensuring that it's also deployed uniformly across all environments. The configuration across these environments could be different for the application.

Continuous deployment is generally integrated with continuous integration. When continuous integration has done its work, by generating the final deployable packages, continuous deployment kicks in and starts its own pipeline. This pipeline is called the release pipeline. The release pipeline consists of multiple environments, with each environment consisting of tasks responsible for provisioning the environment, configuring the environment, deploying applications, configuring applications, executing operational validation on environments, and testing the application on multiple environments.

Employing continuous deployment provides immense benefits. There is a high level of confidence in the overall deployment process, which helps with faster and risk-free releases on production. The chances of anything going wrong decrease drastically. The team will be less stressed, and rollback to the previous working environment is possible if there are issues with the current release:

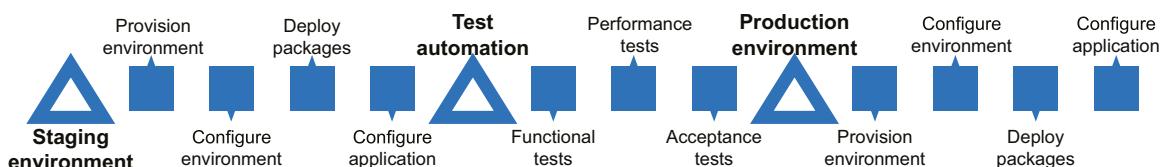


Figure 13.3: Continuous deployment pipeline

Although every system demands its own configuration of the release pipeline, an example of continuous deployment is shown in the preceding diagram. It's important to note that, generally, provisioning and configuring multiple environments is part of the release pipeline, and approvals should be sought before moving to the next environment. The approval process might be manual or automated, depending on the maturity of the organization.

Next, we will look into aspects related to the test environment.

Test environment deployment

The release pipeline starts once the drop is available from continuous integration and the first step it should take is to get all the artifacts from the drop. After this, the release pipeline might create a completely new bare-metal test environment or reuse an existing one. This is again dependent on the type of project and the nature of the testing planned to be executed in this environment. The environment is provisioned and configured. The application artifacts are deployed and configured.

Test automation

After deploying an application, a series of tests can be performed on the environment. One of the tests executed here is a functional test. Functional tests are primarily aimed at validating the feature completeness and functionality of the application. These tests are written from requirements gathered from the customer. Another set of tests that can be executed is related to the scalability and availability of the application. This typically includes load tests, stress tests, and performance tests. It should also include an operational validation of the infrastructure environment.

Staging environment deployment

This is very similar to the test environment deployment, the only difference being that the configuration values for the environment and application would be different.

Acceptance tests

Acceptance tests are generally conducted by application stakeholders, and these can be manual or automated. This step is a validation from the customer's point of view about the correctness and completeness of the application's functionality.

Deployment to production

Once the customer gives their approval, the same steps as that of the test and staging environment deployment are executed, the only difference being that the configuration values for the environment and application are specific to the production environment. A validation is conducted after deployment to ensure that the application is running according to expectations.

Continuous delivery is an important DevOps principle and closely resembles continuous deployment; however, there are a few differences. In the next section, we will look into continuous delivery.

Continuous delivery

Continuous delivery and continuous deployment might sound similar to you; however, they aren't the same. While continuous deployment talks about deployment to multiple environments and finally to the production environment through automation, continuous delivery is the ability to generate application packages that are readily deployable in any environment. To generate artifacts that are readily deployable, continuous integration should be used to generate the application artifacts; a new or existing environment should be used to deploy these artifacts and conduct functional tests, performance tests, and user-acceptance tests through automation. Once these activities are successfully executed without any errors, the application package is considered readily deployable. Continuous delivery includes continuous integration and deployment to an environment for final validations. It helps get feedback more quickly from both the operations and the end user. This feedback can then be used to implement subsequent iterations.

In the next section, we will look into continuous learning.

Continuous learning

With all the previously mentioned DevOps practices, it's possible to create great business applications and deploy them automatically to the production environment; however, the benefits of DevOps won't last for long if continuous improvement and feedback principles are not in place. It's of the utmost importance that real-time feedback about the application behavior is passed on as feedback to the development team from both end users and the operations team.

Feedback should be passed to the teams, providing relevant information about what's going well and what isn't.

An application's architecture and design should be built with monitoring, auditing, and telemetry in mind. The operations team should collect telemetry information from the production environment, capturing any bugs and issues, and pass it on to the development team so that it can be fixed for subsequent releases.

Continuous learning helps to make the application robust and resilient to failure. It helps in making sure that the application is meeting consumer requirements. Figure 13.4 shows the feedback loop that should be implemented between different teams:

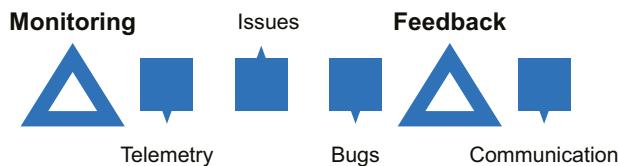


Figure 13.4: Feedback loop

After going through the important practices related to DevOps, now it's time to get into tools and services that make these possible.

Azure DevOps

Let's look at another top-of-the-line online service that enables continuous integration, continuous deployment, and continuous delivery seamlessly: Azure DevOps. In fact, it would be more appropriate to call it a suite of services available under a single name. Azure DevOps is a PaaS provided by Microsoft and hosted on the cloud. The same service is available as **Team Foundation Services (TFS)** on-premises. All examples shown in this book use Azure DevOps.

According to Microsoft, Azure DevOps is a cloud-based collaboration platform that helps teams to share code, track work, and ship software. Azure DevOps is a new name; earlier, it was known as **Visual Studio Team Services (VSTS)**. Azure DevOps is an enterprise software-development tool and service that enables organizations to provide automation facilities to their end-to-end application life cycle management process, from planning to deploying applications, and getting real-time feedback from software systems. This increases the maturity and capability of an organization to deliver high-quality software systems to their customers.

Successful software delivery involves efficiently bringing numerous processes and activities together. These include executing and implementing various Agile processes, increasing collaboration among teams, the seamless and automatic transition of artifacts from one phase of the ALM to another phase, and deployments to multiple environments. It's important to track and report on these activities to measure and improve delivery processes. Azure DevOps makes this simple and easy. It provides a whole suite of services that enables the following:

- Collaboration among every team member by providing a single interface for the entire application life cycle management
- Collaboration among development teams using source-code-management services
- Collaboration among test teams using test-management services
- Automatic validation of code and packaging through continuous integration using build-management services
- Automatic validation of application functionality, deployment, and configuration of multiple environments through continuous deployment and delivery using release-management services
- Tracking and work-item management using work-management services

The following table shows all the services available to a typical project from the **Azure DevOps** left navigation bar:

Service	Description
Boards	Boards helps in the planning of the project by displaying the current progress of tasks, backlogs, and user stories alongside sprint information. It also provides a Kanban process and helps depict the current tasks in progress and completed.
Repos	Repos helps in managing repositories. It provides support with creating additional branches, merging them, resolving code conflicts, and also managing permissions. There can be multiple repositories within a project.
Pipelines	Both release and build pipelines are created and managed from Pipelines. It helps in automating the build and release process. There can be multiple build and release pipelines within a project.
Test Plans	All testing-related artifacts along with their management are available from Test Plans.
Artifacts	NuGet packages and other artifacts are stored and managed here.

Table 13.1: A list of Azure DevOps services

An organization in Azure DevOps serves as a security boundary and logical container that provides all the services that are needed to implement a DevOps strategy. Azure DevOps allows the creation of multiple projects within a single organization. By default, a repository is created with the creation of a project; however, Azure DevOps allows the creation of additional repositories within a single project. The relationship between an **Azure DevOps Organization**, **Projects**, and a **Repository** is shown in *Figure 13.5*:

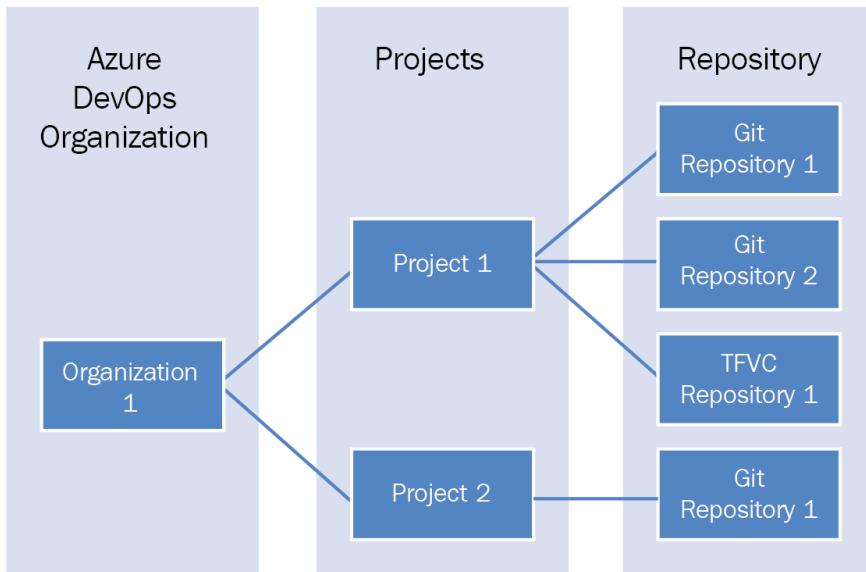


Figure 13.5: Relationship between Azure DevOps Organization, Projects, and Repository

Azure DevOps provides two types of repositories:

- Git
- Team Foundation Version Control (TFVC)

It also provides the flexibility to choose between the Git or TFVC source-control repository. There can be a combination of TFS and TFVC repositories available within a single project.

TFVC

TFVC is the traditional and centralized way of implementing version control, where there's a central repository and developers work on it directly in connected mode to check in their changes. If the central repository is offline or unavailable, developers can't check in their code and have to wait for it to be online and available. Other developers can see only the checked-in code. Developers can group multiple changes into a single changeset for checking in code changes that are logically grouped to form a single change. TFVC locks the code files that are undergoing edits. Other developers can read a locked file, but they can't edit it. They must wait for the prior edit to complete and release the lock before they can edit. The history of check-ins and changes is maintained on the central repository, while the developers have the working copy of the files but not the history.

TFVC works very well with large teams that are working on the same projects. This enables control over the source code at a central location. It also works best for long-duration projects since the history can be managed at a central location. TFVC has no issues working with large and binary files.

Git

Git, on the other hand, is a modern, distributed way of implementing version control, where developers can work on their own local copies of code and history in offline mode. Developers can work offline on their local clone of code. Each developer has a local copy of code and its entire history, and they work on their changes with this local repository. They can commit their code to the local repository. They can connect to the central repository for the synchronization of their local repository on a per-need basis. This allows every developer to work on any file since they would be working on their local copy. Branching in Git doesn't create another copy of the original code and is extremely fast to create.

Git works well with both small and large teams. Branching and merging is a breeze with the advanced options that Git has.

Git is the recommended way of using source control because of the rich functionality it provides. We'll use Git as the repository for our sample application in this book. In the next section, we will have a detailed overview of implementing automation through DevOps.

Preparing for DevOps

Going forward, our focus will be on process and deployment automation using different patterns in Azure. These include the following:

- DevOps for IaaS solutions
- DevOps for PaaS solutions
- DevOps for container-based solutions

Generally, there are shared services that aren't unique to any one application. These services are consumed by multiple applications from different environments, such as development, testing, and production. The life cycle of these shared services is different for each application. Therefore, they have different version-control repositories, a different code base, and build and release management. They have their own cycle of plan, design, build, test, and release.

The resources that are part of this group are provisioned using ARM templates, PowerShell, and DSC configurations.

The overall flow for building these common components is shown here:

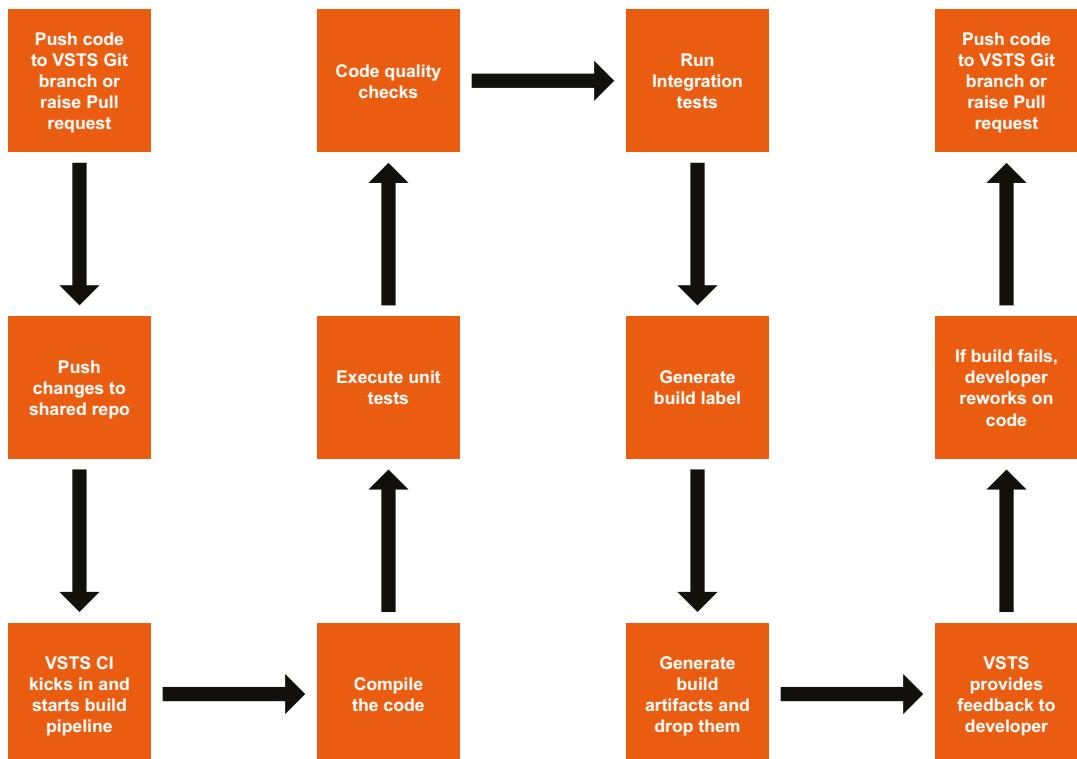


Figure 13.6: Overall flow for building common components

The release process is shown in Figure 13.7:

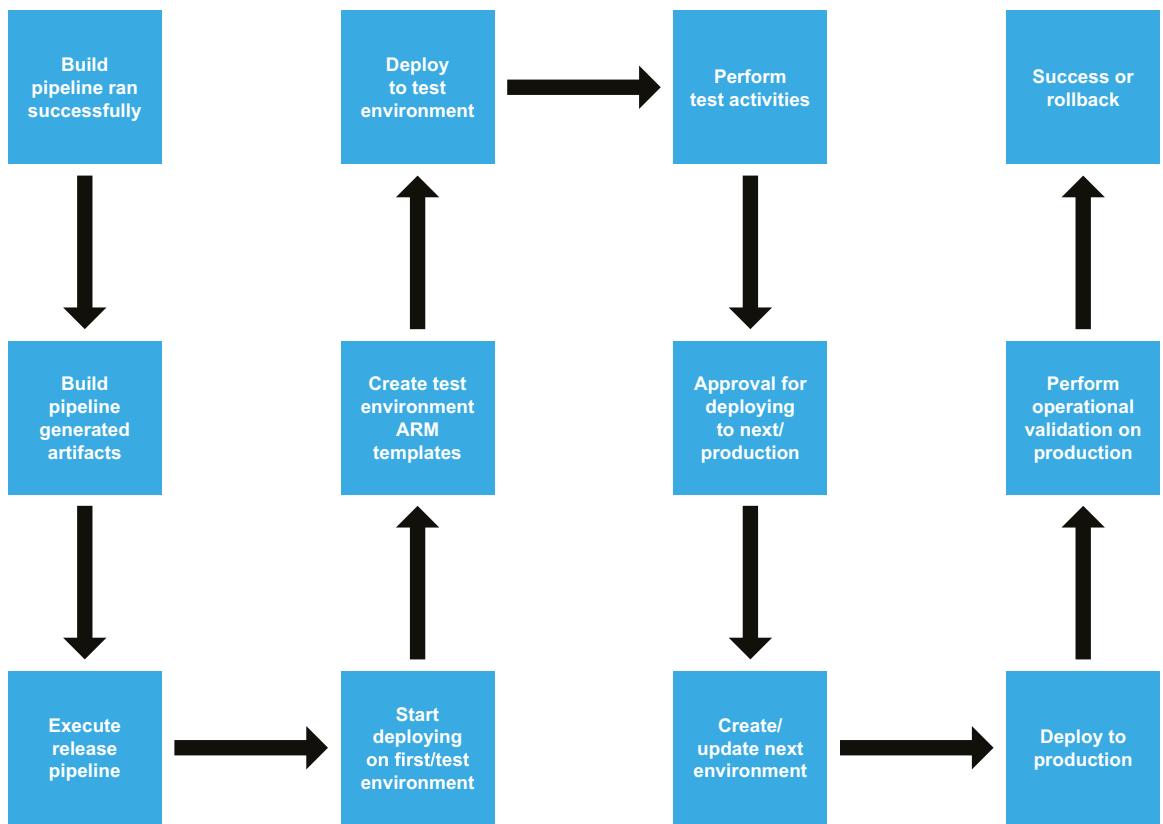


Figure 13.7: Release process

On the DevOps journey, it's important to understand and provision the common components and services before starting any software engagement, product, or service.

The first step in getting started with Azure DevOps is to provision an organization.

Azure DevOps organizations

A version-control system is needed to collaborate at the code level. Azure DevOps provides both centralized and decentralized versions of control systems. Azure DevOps also provides orchestration services for building and executing build and release pipelines. It's a mature platform that organizes all DevOps-related version control and builds and releases work-item-related artifacts. After an organization is provisioned in Azure DevOps, an Azure DevOps project should be created to hold all project-related artifacts.

An Azure DevOps organization can be provisioned by visiting <https://dev.azure.com>.

An Azure DevOps organization is the top-level administrative and management boundary that provides security, access, and collaboration between team members belonging to an organization. There can be multiple projects within an organization and each project comprises multiple teams.

Provisioning Azure Key Vault

It isn't advisable to store secrets, certificates, credentials, or other sensitive information in code configuration files, databases, or any other general storage system. It's advised to store this important data in a vault that's specifically designed for storing secrets and credentials. Azure Key Vault provides such a service. Azure Key Vault is available as a resource and service from Azure. Now, let's move on to exploring the storage options for configurations.

Provisioning a configuration-management server/service

A configuration-management server/service that provides storage for configurations and applies those configurations to different environments is always a good strategy for automating deployments. DSC on custom virtual machines and DSC from Azure Automation, Chef, Puppet, and Ansible are some options and can be used on Azure seamlessly for both Windows as well as Linux environments. This book uses DSC as a configuration-management tool for all purposes, and it provides a pull server that holds all configuration documents (MOF files) for the sample application. It also maintains the database of all virtual machines and containers that are configured and registered with the pull server to pull configuration documents from it. The local configuration manager on these targets virtual machines, and containers periodically check the availability of new configurations as well as drifts in the current configuration and report back to the pull server. It also has built-in reporting capabilities that provide information about nodes that are compliant, as well as those that are non-compliant, within a virtual machine. A pull server is a general web application that hosts the DSC pull server endpoint. In the next topic, we will discuss a technique to monitor processes in real time with Log Analytics.

Log Analytics

Log Analytics is an audit and monitoring service provided by Azure to get real-time information about all changes, drifts, and events occurring within virtual machines and containers. It provides a centralized workspace and dashboard for IT administrators for viewing, searching, and conducting drill-down searches on all changes, drifts, and events that occur on these virtual machines. It also provides agents that are deployed on target virtual machines and containers. Once deployed, these agents start sending all changes, events, and drifts to the centralized workspace. Let's check out the storage options for deploying multiple applications.

Azure Storage accounts

Azure Storage is a service provided by Azure to store files as blobs. All scripts and code for automating the provisioning, deployment, and configuration of the infrastructure and sample application are stored in the Azure DevOps Git repository and are packaged and deployed in an Azure Storage account. Azure provides PowerShell script-extension resources that can automatically download DSC and PowerShell scripts and execute them on virtual machines during the execution of ARM templates. This storage acts as common storage across all deployments for multiple applications. Storing scripts and templates in a Storage account ensures that they can be used across projects irrespective of projects in Azure DevOps. Let's move on to exploring the importance of images in the next section.

Docker and OS images

Both virtual machine and container images should be built as part of the common services build-and-release pipeline. Tools such as Packer and Docker Build can be used to generate these images.

Management tools

All management tools, such as Kubernetes, DC/OS, Docker Swarm, and ITIL tools, should be provisioned before building and deploying the solution.

We'll conclude this section on DevOps preparation with management tools. There are multiple choices for each activity within a DevOps ecosystem and we should enable them as part of the DevOps journey—it should not be an afterthought, but rather part of DevOps planning.

DevOps for PaaS solutions

The typical architecture for Azure PaaS app services is based on *Figure 13.8*:

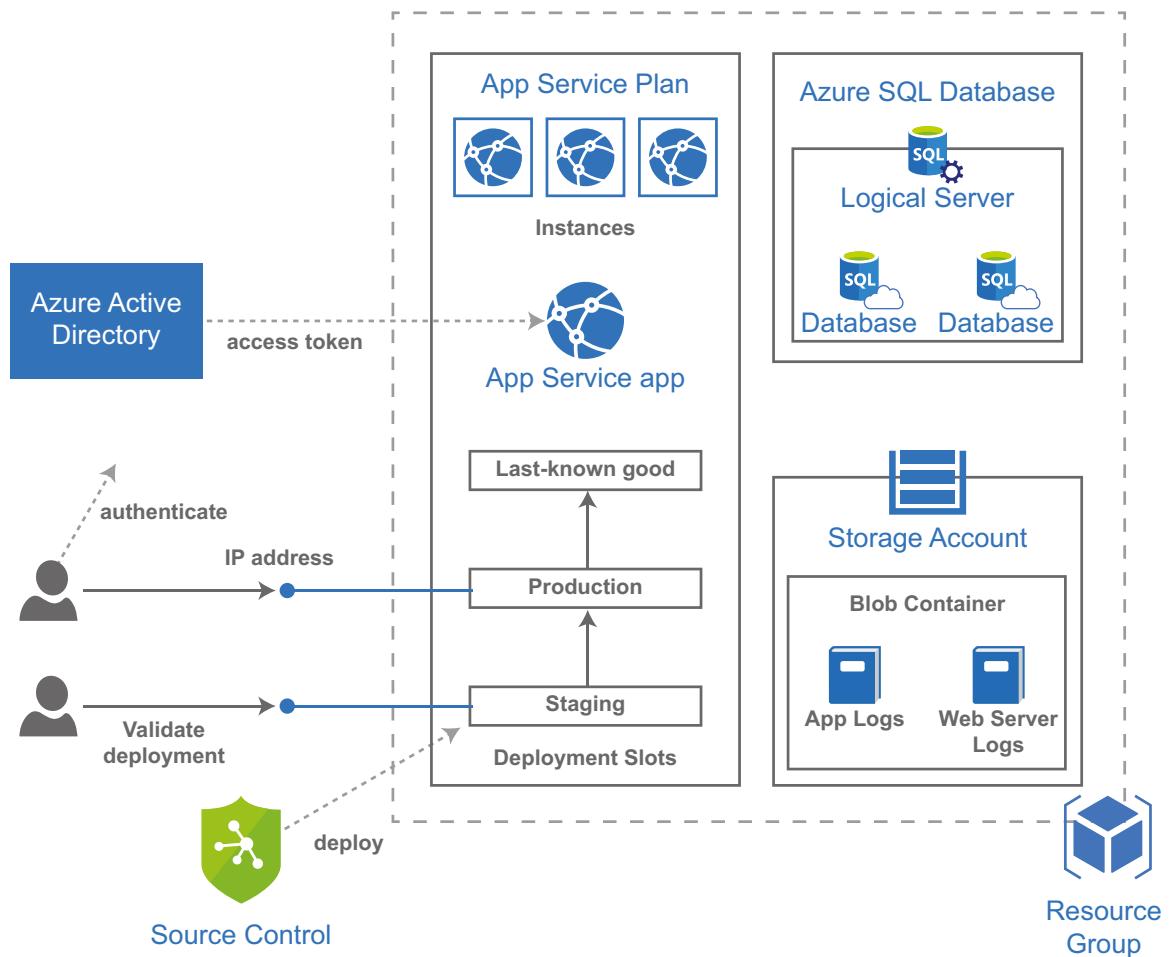


Figure 13.8: A typical Azure PaaS app service architecture

The architecture shows some of the important components—such as Azure SQL, Storage accounts, and the version control system—that participate in the Azure App Service-based cloud solution architecture. These artifacts should be created using ARM templates. These ARM templates should be part of the overall configuration management strategy. It can have its own build and release management pipelines, similar to the one shown in *Figure 13.9*:

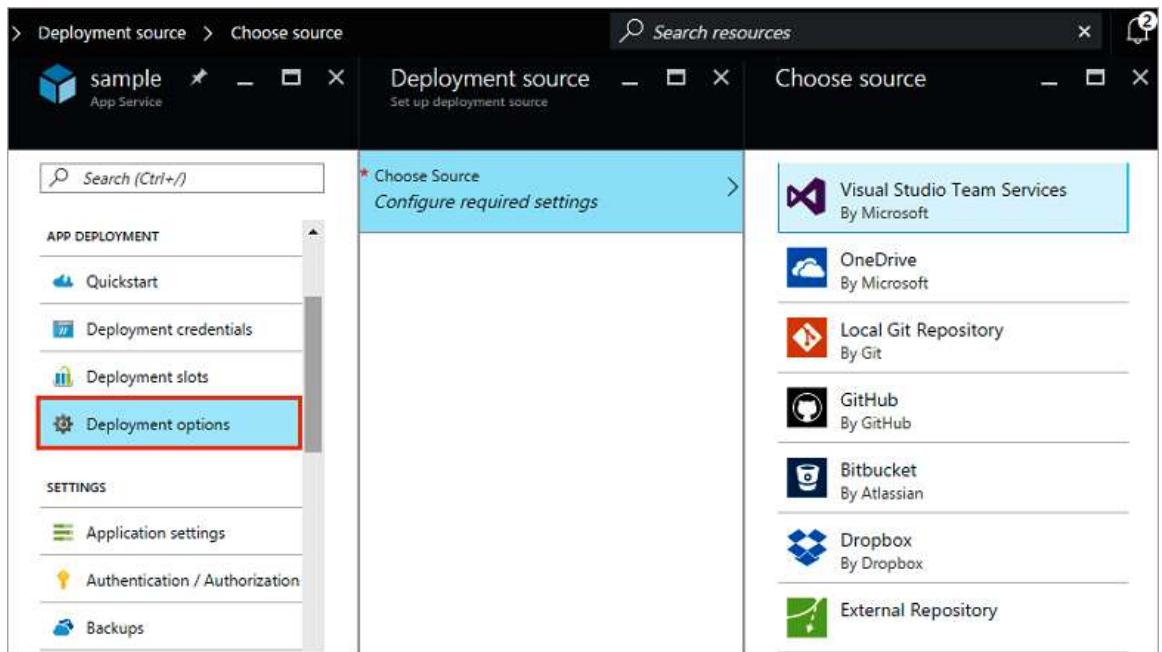


Figure 13.9: Choosing deployment options for the app service

Now that we have explored the various deployment source options, let's go ahead and dive into understanding how to deploy cloud solutions on Azure.

Azure App Service

Azure App Service provides managed hosting services for cloud solutions. It's a fully-managed platform that provisions and deploys cloud solutions. Azure App Service takes away the burden of creating and managing infrastructure and provides minimum **service-level agreements (SLAs)** for hosting your cloud solutions.

Deployment slots

Azure App Service provides deployment slots that make deployment to them seamless and easy. There are multiple slots, and swapping between slots is done at a DNS level. It means anything in the production slot can be swapped with a staging slot by just swapping the DNS entries. This helps in deploying the custom cloud solution to staging and, after all checks and tests, they can be swapped to production if found satisfactory. However, in the event of any issue in production after swapping, the previous good values from the production environment can be reinstated by swapping again. Let's move on to understanding Azure's database offering and some of its key features.

Azure SQL

Azure SQL is a SQL PaaS service provided by Azure to host databases. Azure provides a secure platform to host databases and takes complete ownership to manage the availability, reliability, and scalability of the service. With Azure SQL, there's no need to provision custom virtual machines, deploy a SQL server, and configure it. Instead, the Azure team does this behind the scenes and manages it on your behalf. It also provides a firewall service that enables security; only an IP address allowed by the firewall can connect the server and access the database. The virtual machines provisioned to host web applications have distinct public IP addresses assigned to them and they're added to Azure SQL firewall rules dynamically. Azure SQL Server and its database is created upon executing the ARM template. Next, we will cover build and release pipelines.

The build and release pipelines

In this section, a new build pipeline is created that compiles and validates an ASP.NET MVC application, and then generates packages for deployment. After package generation, a release definition ensures that deployment to the first environment happens in an App Service and Azure SQL as part of continuous deployment.

There are two ways to author build and release pipelines:

1. Using the classic editor
2. Using YAML files

YAML files provide more flexibility for authoring build and release pipelines.

The project structure of the sample application is shown in *Figure 13.10*:

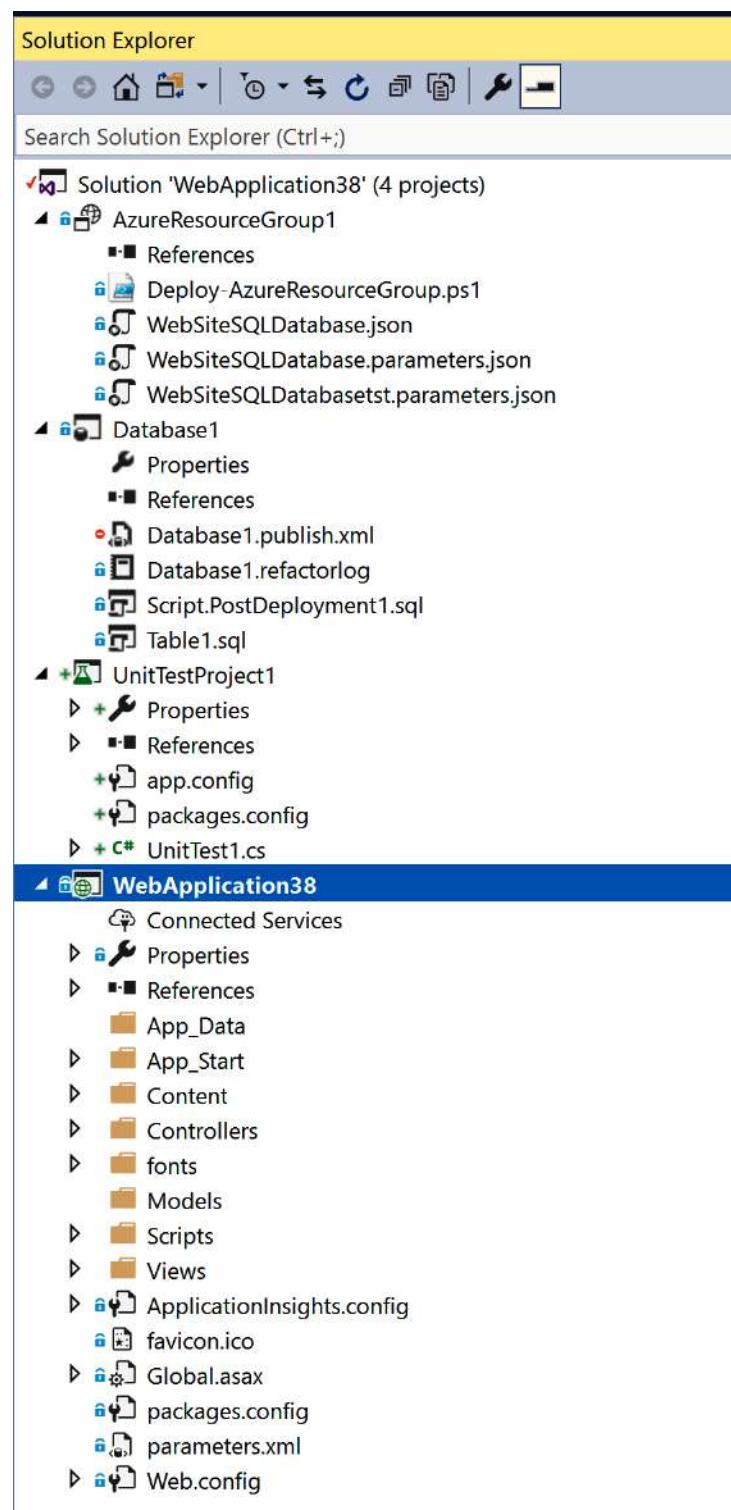


Figure 13.10: Project structure of a sample application

In this project, there's an ASP.NET MVC application—the main application—and it consists of application pages. Web Deploy packages will be generated out of this project from build pipelines and they will eventually be on Web Apps. There are other projects that are also part of the solution, as mentioned next:

- **Unit test project:** Code for unit-testing the ASP.NET MVC application. Assemblies from this project will be generated and executed in the build execution.
- **SQL Database project:** Code related to the SQL database schema, structure, and master data. **DacPac** files will be generated out of this project using the build definition.
- **Azure resource group project:** ARM templates and parameter code to provision the entire Azure environment on which the ASP.NET MVC application and the SQL tables are created.

The build pipeline is shown in *Figure 13.11*:

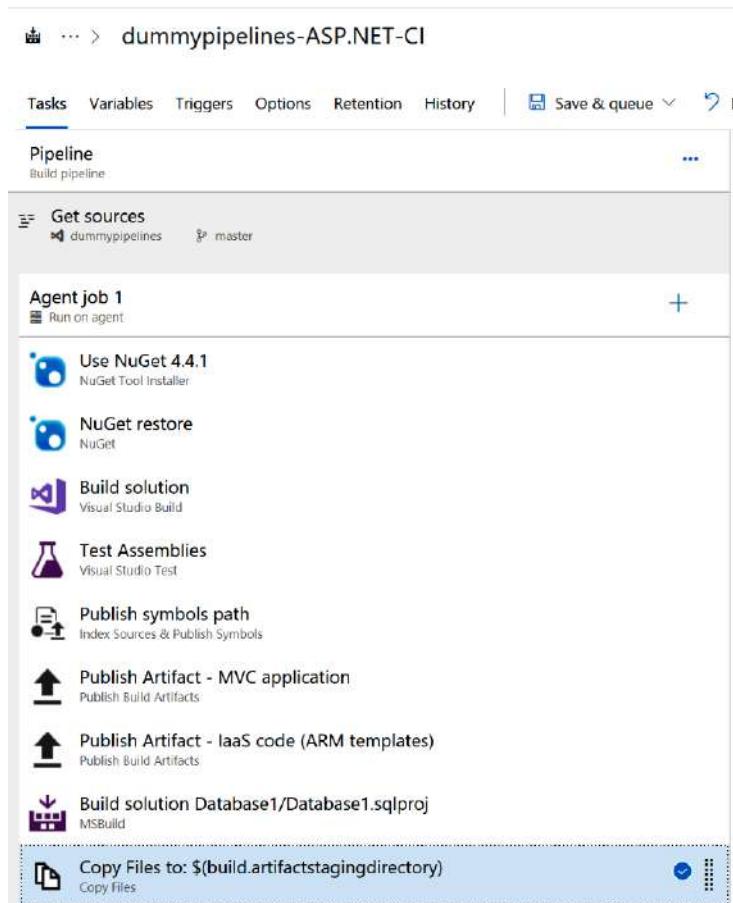


Figure 13.11: Build pipeline of the ASP.NET MVC application

The configuration of each task is shown in Table 13.2:

Task name	Task configuration
Use NuGet 4.4.1	<p>NuGet Tool Installer ⓘ</p> <p>Version <input type="text" value="0.*"/> ⏺</p> <p>Display name *</p> <p><input type="text" value="Use NuGet 4.4.1"/></p> <p>Version of NuGet.exe to install * ⓘ</p> <p><input type="text" value="4.4.1"/></p> <p><input type="checkbox"/> Always download the latest matching version ⓘ</p> <p>Control Options ⏺</p> <p>Output Variables ⏺</p> <p>Link settings View YAML Remove</p>
NuGet restore	<p>NuGet ⓘ</p> <p>Version <input type="text" value="2.*"/> ⏺</p> <p>Display name *</p> <p><input type="text" value="NuGet restore"/></p> <p>Command * ⓘ</p> <p><input type="text" value="restore"/></p> <p>Path to solution, packages.config, or project.json *</p> <p><input type="text" value="***.sln"/></p> <p>Link settings View YAML Remove</p>
Build solution	<p>Visual Studio Build ⓘ</p> <p>Version <input type="text" value="1.*"/> ⏺</p> <p>Display name *</p> <p><input type="text" value="Build solution"/></p> <p>Solution * ⓘ</p> <p><input type="text" value="WebApplication38/WebApplication38.csproj"/> ...</p> <p>Visual Studio Version ⓘ</p> <p><input type="text" value="Latest"/></p> <p>MSBuild Arguments ⓘ</p> <p><input \$(build.artifactstagingdirectory)\""="" type="text" value="/p:DeployOnBuild=true /p:WebPublishMethod=Package /p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true /p:PackageLocation="/></p> <p>Platform ⓘ</p> <p><input type="text" value="\${BuildPlatform}"/></p> <p>Configuration ⓘ</p> <p><input type="text" value="\${BuildConfiguration}"/></p> <p>Link settings View YAML Remove</p>

Task name	Task configuration
Test Assemblies	<p>Visual Studio Test ⓘ</p> <p>Version <input type="text" value="2.*"/> ▾</p> <p>Display name * Test Assemblies</p> <p>Test selection ^</p> <p>Select tests using * ⓘ</p> <p>Test assemblies</p> <p>Test files * ⓘ **\\$(BuildConfiguration)*test*.dll !**\obj**</p> <p>Search folder * ⓘ \$(System.DefaultWorkingDirectory)</p>
Publish symbols path	<p>Index Sources & Publish Symbols ⓘ</p> <p>Version <input type="text" value="2.*"/> ▾</p> <p>Display name * Publish symbols path</p> <p>Path to symbols folder ⓘ \$(Build.SourcesDirectory)</p> <p>Search pattern * ⓘ **\bin***.pdb</p> <p><input checked="" type="checkbox"/> Index sources ⓘ <input type="checkbox"/> Publish symbols ⓘ</p>
Publish Artifact - MVC application	<p>Publish Build Artifacts ⓘ</p> <p>Version <input type="text" value="1.*"/> ▾</p> <p>Display name * Publish Artifact - MVC application</p> <p>Path to publish * ⓘ \$(build.artifactstagingdirectory) ...</p> <p>Artifact name * drop</p>

Task name	Task configuration
Publish Artifact - IaaS code (ARM templates)	<p>Publish Build Artifacts ⓘ</p> <p>Version <input type="text" value="1.*"/> ⏺</p> <p>Display name *</p> <input type="text" value="Publish Artifact - IaaS code (ARM templates)"/> <p>Path to publish * ⓘ</p> <input type="text" value="AzureResourceGroup1"/> ... <p>Artifact name * ⓘ</p> <input type="text" value="drop1"/> <p>Artifact publish location * ⓘ</p> <input type="text" value="Azure Pipelines/TFS"/> ⏺ <p>Link settings View YAML Remove</p>
Build solution Database1/Database1.sqlproj	<p>MSBuild ⓘ</p> <p>Version <input type="text" value="1.*"/> ⏺</p> <p>Display name *</p> <input type="text" value="Build solution Database1/Database1.sqlproj"/> <p>Project * ⓘ</p> <input type="text" value="Database1/Database1.sqlproj"/> ... <p>MSBuild ⓘ</p> <p><input checked="" type="radio"/> Version <input type="radio"/> Specify Location</p> <p>MSBuild Version ⓘ</p> <input type="text" value="Latest"/> ⏺ <p>MSBuild Architecture ⓘ</p> <input type="text" value="MSBuild x64"/> ⏺ <p>Platform ⓘ</p> <input type="text"/> ⏺ <p>Configuration ⓘ</p> <input type="text"/> ⏺ <p>MSBuild Arguments ⓘ</p> <input type="text" value="/t:build /p:CmdLineInMemoryStorage=True"/>

Task name	Task configuration
Copy Files to: \$(build.artifactstagingdirectory)	<p>Copy Files ①</p> <p>Version 2.*</p> <p>Display name *</p> <p>Copy Files to: \$(build.artifactstagingdirectory)</p> <p>Source Folder ①</p> <p>Contents *</p> <p>***.dacpac</p> <p>Target Folder *</p> <p>\$(build.artifactstagingdirectory)</p>

Table 13.2: Configuration of the build pipeline tasks

The build pipeline is configured to execute automatically as part of continuous integration, as shown in *Figure 13.12*:

The screenshot shows the 'Triggers' tab of a build pipeline named 'dummypipelines-ASP.NET-CI'. The 'Continuous integration' section has the 'dummypipelines' trigger enabled. Under 'Branch filters', 'master' is selected. Under 'Path filters', there is an '+ Add' button.

Figure 13.12: Enabling continuous integration in the build pipeline

The release definition consists of multiple environments, such as development, testing, **System Integration Testing (SIT)**, **User Acceptance Testing (UAT)**, preproduction, and production. The tasks are pretty similar in each environment, with the addition of tasks specific to that environment. For example, a test environment has additional tasks related to the UI, and functional and integration testing, compared to a development environment.

The release definition for such an application is shown in *Figure 13.13*:

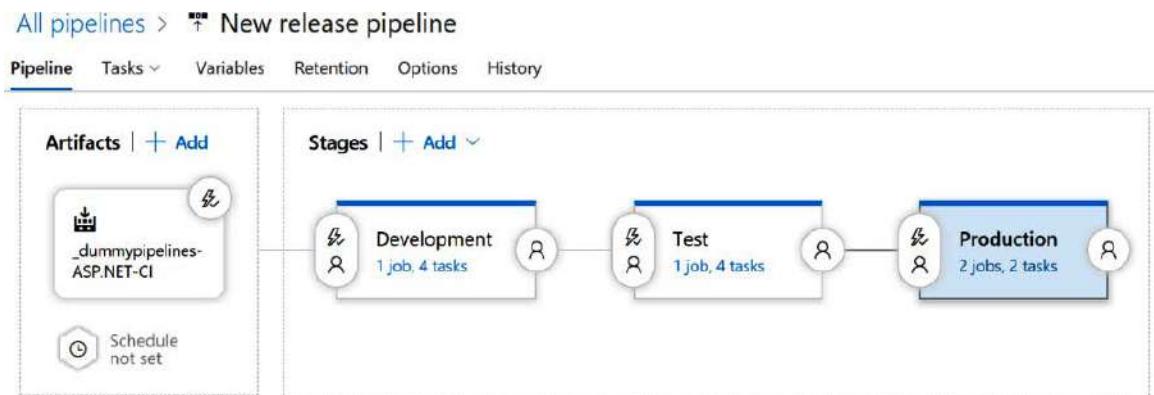


Figure 13.13: Release definition

The release tasks for a single environment are shown in *Figure 13.14*:

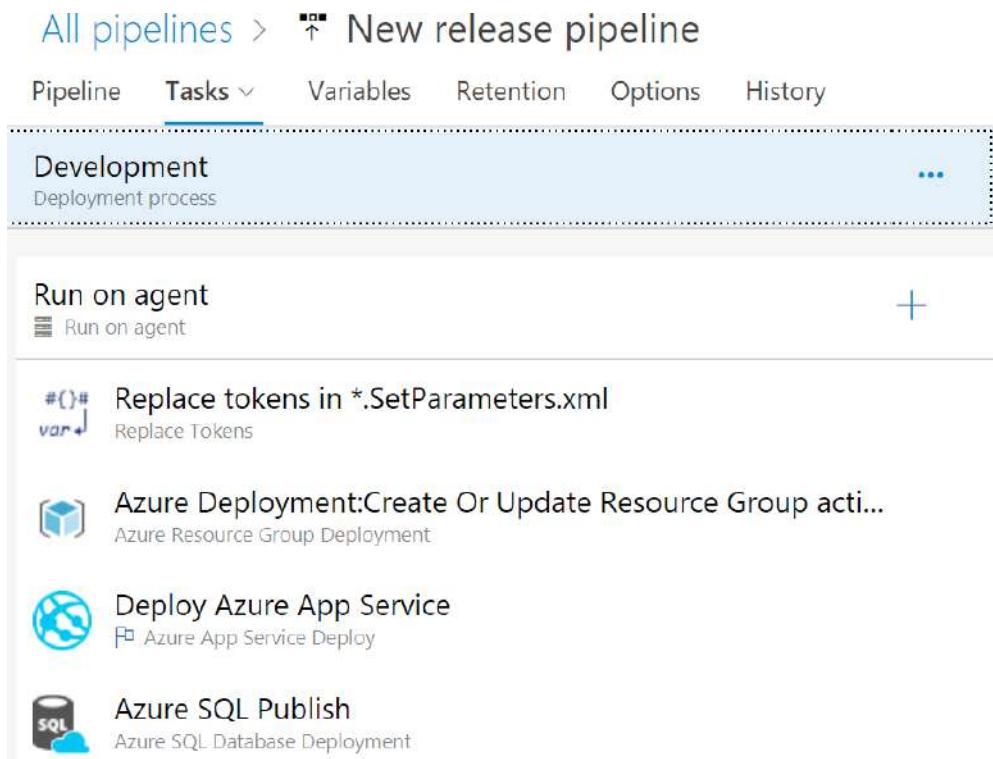


Figure 13.14: Release tasks for a single environment

The configuration for each of the tasks is listed here:

Task name	Task configuration
Replace tokens in *.SetParameters.xml (This is a task installed from Marketplace.)	<p>Version 3.*</p> <p>Display name *</p> <p>Replace tokens in *.SetParameters.xml</p> <p>Root directory ⓘ</p> <p>\$(System.DefaultWorkingDirectory)/_dummypipelines-ASP.NET-CI/drop</p> <p>Target files * ⓘ</p> <p>*.SetParameters.xml</p> <p>Files encoding * ⓘ</p> <p>auto</p> <p><input checked="" type="checkbox"/> Write unicode BOM ⓘ</p> <p>Missing variables ^</p> <p>Action * ⓘ</p> <p>log warning</p> <p><input checked="" type="checkbox"/> Keep token ⓘ</p> <p>Advanced ^</p> <p>Token prefix * ⓘ</p> <p>—</p> <p>Token suffix * ⓘ</p> <p>—</p> <p>Empty value ⓘ</p> <p>(empty)</p> <p>Escape values type ⓘ</p> <p>no escaping</p>

Task name	Task configuration
Azure Deployment: Create Or Update Resource Group action on devRG	<p>Azure Resource Group Deployment ⓘ X Remove</p> <p>Version <input type="text" value="2.*"/> ▾</p> <p>Display name * <input type="text" value="Azure Deployment:Create Or Update Resource Group action on devRG"/></p> <p>Azure Details ▾</p> <p>Azure subscription * ⓘ Manage <input type="text" value="myconnection"/> ▾ ↻</p> <p>Scoped to subscription 'Visual Studio Enterprise'</p> <p>Action * ⓘ <input type="text" value="Create or update resource group"/> ▾ ↻</p> <p>Resource group * ⓘ <input type="text" value="devRG"/> ▾ ↻</p> <p>Location * ⓘ <input type="text" value="West Europe"/> ▾ ↻</p> <p>Template ▾</p> <p>Template location * <input type="text" value="Linked artifact"/> ▾ ...</p> <p>Template * ⓘ <input type="text" value="\$(System.DefaultWorkingDirectory)/_dummypipelines-ASP.NET-CI/drop1/WebSiteSQLDatabase.json"/> ▾ ...</p> <p>Template parameters <input type="text" value="\$(System.DefaultWorkingDirectory)/_dummypipelines-ASP.NET-CI/drop1/WebSiteSQLDatabase.parameters.json"/> ▾ ...</p> <p>Override template parameters <input type="text" value="-sqlserverName \$(SQLServerName) -hostingPlanName \$(AppServiceName)"/> ▾ ...</p> <p>Deployment mode * ⓘ <input type="text" value="Incremental"/> ▾ ...</p>

Task name	Task configuration
Deploy Azure App Service	<p>Azure App Service Deploy ⓘ</p> <p>Version 3.*</p> <p>Display name *</p> <p>Deploy Azure App Service</p> <p>Azure subscription * ⓘ Manage ↗</p> <p>myconnection</p> <p>Scoped to subscription 'Visual Studio Enterprise'</p> <p>App type * ⓘ</p> <p>webApp</p> <p>App Service name *</p> <p>\$(AppServiceName)</p> <p><input type="checkbox"/> Deploy to slot ⓘ</p> <p>Virtual application ⓘ</p> <p>Package or folder *</p> <p>\$(System.DefaultWorkingDirectory)/_dummypipelines-ASP.NET-CI/drop/WebApplication38.zip</p>

Task name	Task configuration
Azure SQL Publish	<p>Azure SQL Database Deployment ⓘ X Remove</p> <p>Version <input type="text" value="1.*"/> ▾</p> <p>Display name * <input type="text" value="Azure SQL Publish"/></p> <p>Azure Service Connection Type <input type="text" value="Azure Resource Manager"/> ▾</p> <p>Azure Subscription * ⓘ Manage ↗ <input type="text" value="myconnection"/> ▾ ↻</p> <p>Scoped to subscription 'Visual Studio Enterprise'</p> <p>SQL DB Details ^</p> <p>Azure SQL Server Name * ⓘ <input type="text" value="mdemoclasssql.database.windows.net"/></p> <p>Database Name * ⓘ <input type="text" value="myecommerce"/></p> <p>Server Admin Login * ⓘ <input type="text" value="citynextadmin"/></p> <p>Password * ⓘ <input type="text" value="citynext!1234"/></p> <p>Deployment Package ^</p> <p>Action * ⓘ <input type="text" value="Publish"/> ▾</p> <p>Type <input type="text" value="SQL DACPAC File"/> ▾</p> <p>DACPAC File * ⓘ <input type="text" value="\$(System.DefaultWorkingDirectory)/_dummypipelines-ASP.NET-Ci/drop2/Database1/bin/Debug/Database1.dacpac"/> ...</p>

Table 13.3: Configuration of the release pipeline tasks

In this section, you saw ways to configure build and release pipelines in Azure DevOps. In the next section onward, the focus will be on different architectures, such as IaaS, containers, and different scenarios.

DevOps for IaaS

IaaS involves the management and administration of base infrastructure and applications together and there are multiple resources and components that need to be provisioned, configured, and deployed on multiple environments. It is important to understand the architecture before going ahead.

The typical architecture for an IaaS virtual machine-based solution is shown here:

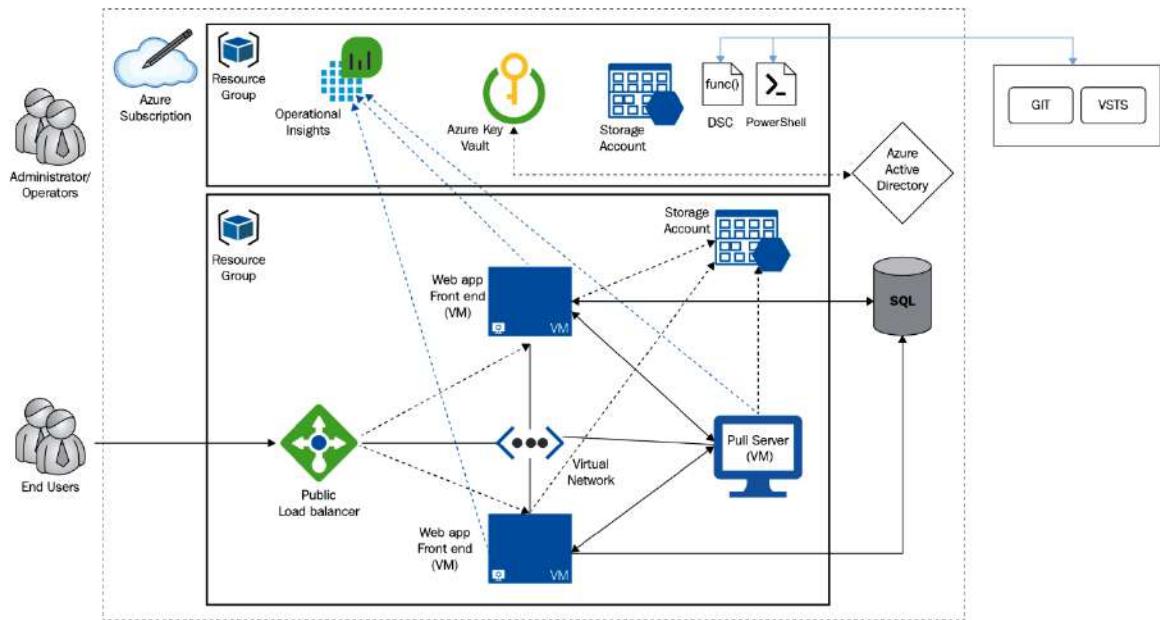


Figure 13.15: Architecture for an IaaS virtual machine-based solution

Each of the components listed in the architecture is discussed from the next section onward.

Azure virtual machines

Azure virtual machines that host web applications, application servers, databases, and other services are provisioned using ARM templates. They're attached to a virtual network and have a private IP address from the same network. The public IP for virtual machines is optional since they're attached to a public load balancer. Operational Insights agents are installed on virtual machines to monitor the virtual machines. PowerShell scripts are also executed on these virtual machines, downloaded from a Storage account available in another resource group to open relevant firewall ports, download appropriate packages, and install local certificates to secure access through PowerShell. The web application is configured to run on the provided port on these virtual machines. The port number for the web application and all its configuration is pulled from the DSC pull server and dynamically assigned.

Azure public load balancers

A public load balancer is attached to some of the virtual machines for sending requests to them in a round-robin fashion. This is generally needed for front-end web applications and APIs. A public IP address and DNS name can be assigned to a load balancer such that it can serve internet requests. It accepts HTTP web requests on a different port and routes them to the virtual machines. It also probes certain ports on HTTP protocols with some provided application paths. **Network Address Translation (NAT)** rules can also be applied such that they can be used to log into the virtual machines using remote desktops.

An alternative resource to the Azure public Load Balancer is the Azure Application Gateway. Application gateways are layer-7 load balancers and provide features such as SSL termination, session affinity, and URL-based routing. Let's discuss the build pipeline in the next section.

The build pipeline

A typical build pipeline for an IaaS virtual machine-based solution is shown next. A release pipeline starts when a developer pushes their code to the repository. The build pipeline starts automatically as part of continuous integration. It compiles and builds the code, executes unit tests on it, checks code quality, and generates documentation from code comments. It deploys the new binaries into the development environment (note that the development environment is not newly created), changes configuration, executes integration tests, and generates build labels for easy identification. It then drops the generated artifacts into a location accessible by the release pipeline. If there are issues during the execution of any step in this pipeline, this is communicated to the developer as part of the build pipeline feedback so that they can rework and resubmit their changes. The build pipeline should fail or pass based on the severity of issues found, and that varies from organization to organization. A typical build pipeline is shown in *Figure 13.16*:

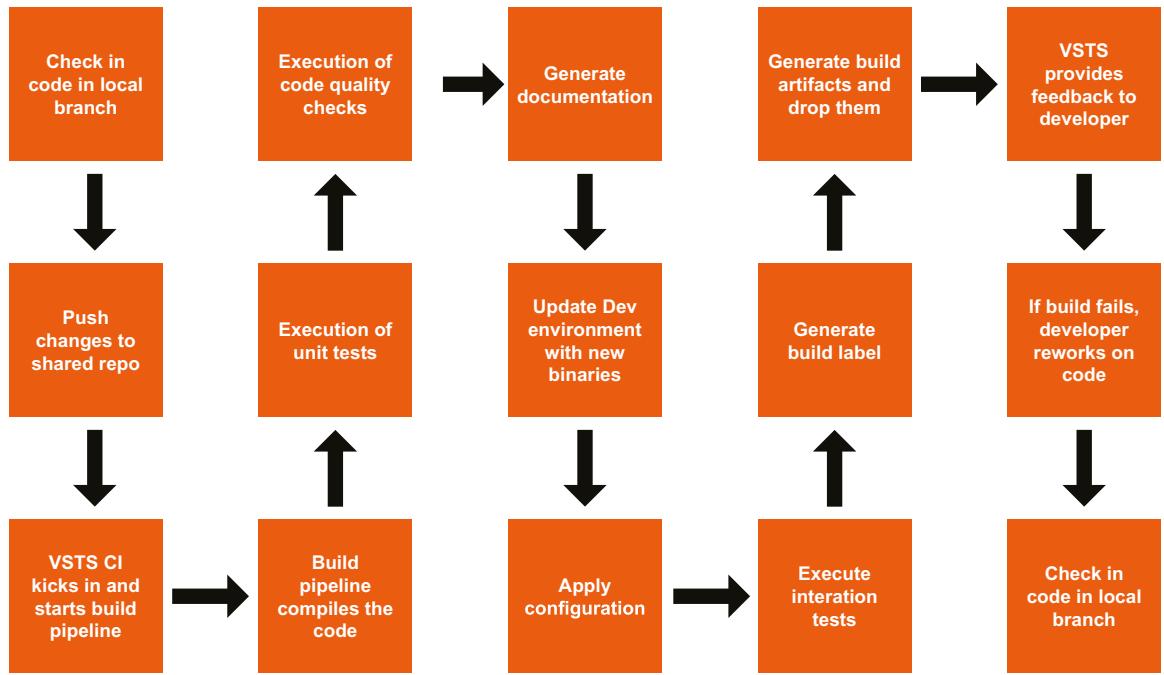


Figure 13.16: A typical IaaS build pipeline

Similar to the build pipeline, let's learn about the implementation of a release pipeline.

The release pipeline

A typical release pipeline for an IaaS virtual machine-based deployment is shown next. A release pipeline starts after the completion of the build pipeline. The first step in the release pipeline is to gather the artifacts generated by the build pipeline. They are generally deployable assemblies, binaries, and configuration documents. The release pipeline executes and creates or updates the first environment, which generally is a test environment. It uses ARM templates to provision all IaaS and PaaS services and resources on Azure and configures them as well. They also help in executing scripts and DSC configuration after virtual machines are created as post-creation steps. This helps to configure the environment within the virtual machine and the operating system. At this stage, application binaries from the build pipeline are deployed and configured. Different automated tests are performed to check the solution and, if found satisfactory, the pipeline moves the deployment to the next environment after obtaining the necessary approvals. The same steps are executed in the next environment, including the production environment. Finally, the operational validation tests are executed in production to ensure that the application is working as expected and there are no deviations.

At this stage, if there are any issues or bugs, they should be rectified and the entire cycle should be repeated; however, if this doesn't happen within a stipulated time frame, the last-known snapshot should be restored in the production environment to minimize downtime. A typical release pipeline is shown in *Figure 13.17*:

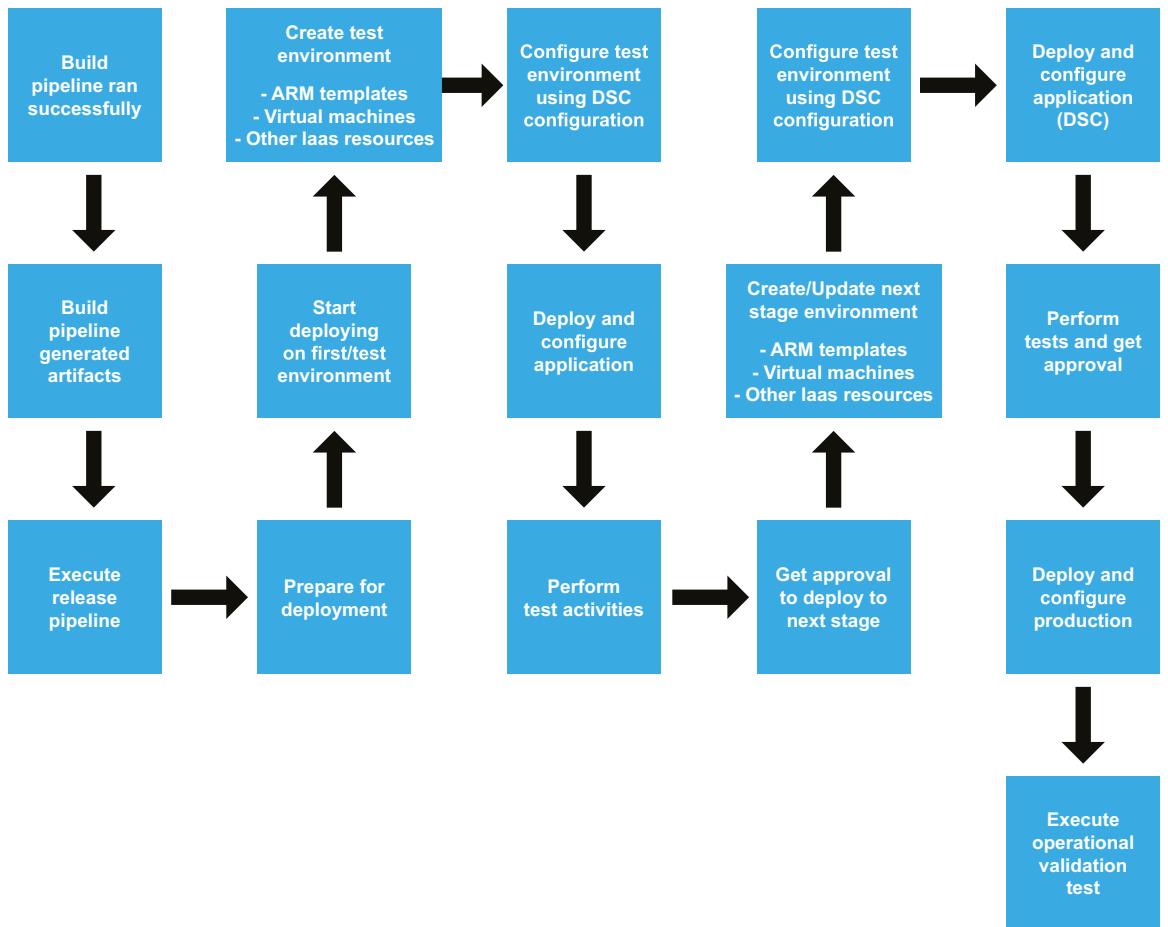


Figure 13.17: A typical IaaS release pipeline

This section concludes the DevOps process for IaaS solutions and the next chapter will focus on containers on virtual machines. Please note that containers can also run on PaaS like App Service and Azure Functions.

DevOps with containers

In a typical architecture, container runtimes are deployed on virtual machines and containers are run within them. The typical architecture for IaaS container-based solutions is shown here:

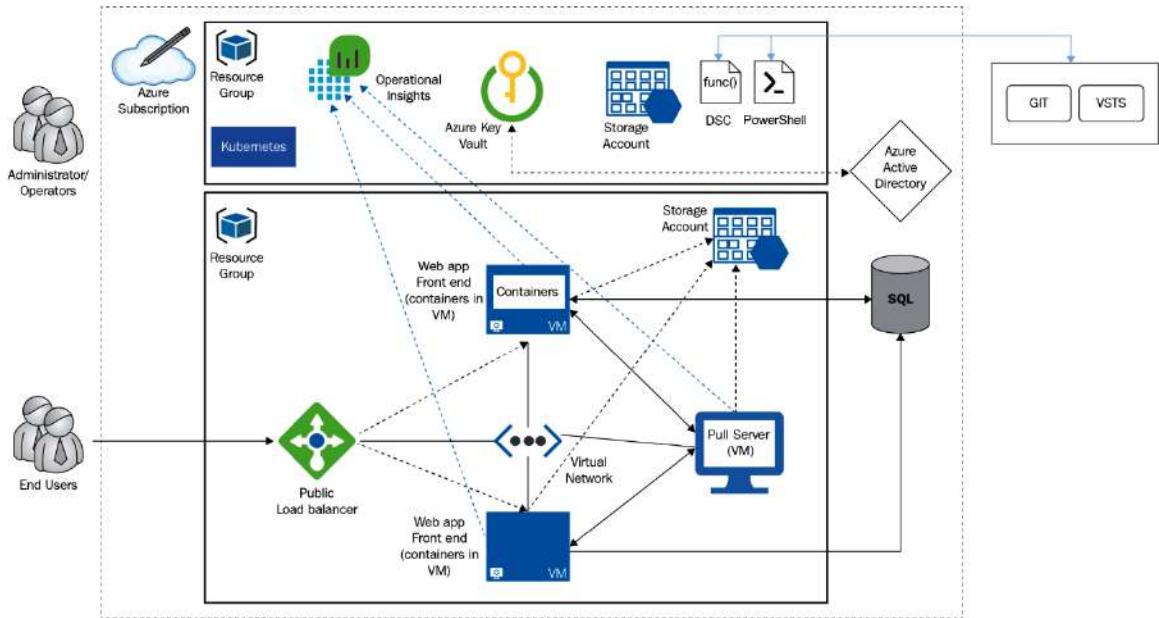


Figure 13.18: Architecture for IaaS container-based solutions

These containers are managed by container orchestrators such as Kubernetes. Monitoring services are provided by Log Analytics and all secrets and keys are stored in Azure Key Vault. There is also a pull server, which could be on a virtual machine or Azure Automation, providing configuration information to the virtual machines.

Containers

Containers are a virtualization technology; however, they don't virtualize physical servers. Instead, containers are an operating system-level virtualization. This means that containers share the operating system kernel provided by their host among themselves and with the host. Running multiple containers on a host (physical or virtual) shares the host operating system kernel. There's a single operating system kernel provided by the host and used by all containers running on top of it.

Containers are also completely isolated from their host and other containers, much like a virtual machine. Containers use operating system namespaces, control groups on Linux, to provide the perception of a new operating system environment, and use specific operating system virtualization techniques on Windows. Each container gets its own copy of the operating system resources.

Docker

Docker provides management features to containers. It comprises two executables:

- The Docker daemon
- The Docker client

The Docker daemon is the workhorse for managing containers. It's a management service that's responsible for managing all activities on the host related to containers. The Docker client interacts with the Docker daemon and is responsible for capturing inputs and sending them to the Docker daemon. The Docker daemon provides the runtime; libraries; graph drivers; the engines to create, manage, and monitor containers; and images on the host server. It can also create custom images that are used for building and shipping applications to multiple environments.

The Dockerfile

The **Dockerfile** is the primary building block for creating container images. It's a simple text-based human-readable file without an extension and is even named **Dockerfile**. Although there's a mechanism to name it differently, generally it is named **Dockerfile**. The Dockerfile contains instructions to create a custom image using a base image. These instructions are executed sequentially from top to bottom by the Docker daemon. The instructions refer to the command and its parameters, such as **COPY**, **ADD**, **RUN**, and **ENTRYPOINT**. The Dockerfile enables IaC practices by converting the application deployment and configuration into instructions that can be versioned and stored in a source code repository. Let's check out the build steps in the following section.

The build pipeline

There's no difference, from the build perspective, between the container and a virtual-machine-based solution. The build step remains the same. A typical release pipeline for an IaaS container-based deployment is shown next.

The release pipeline

The only difference between a typical release pipeline for an IaaS container-based deployment and the release pipeline is the container-image management and the creation of containers using Dockerfile and Docker Compose. Advanced container-management utilities, such as Docker Swarm, DC/OS, and Kubernetes, can also be deployed and configured as part of release management. However, note that these container management tools should be part of the shared services release pipeline, as discussed earlier. *Figure 13.19* shows a typical release pipeline for a container-based solution:

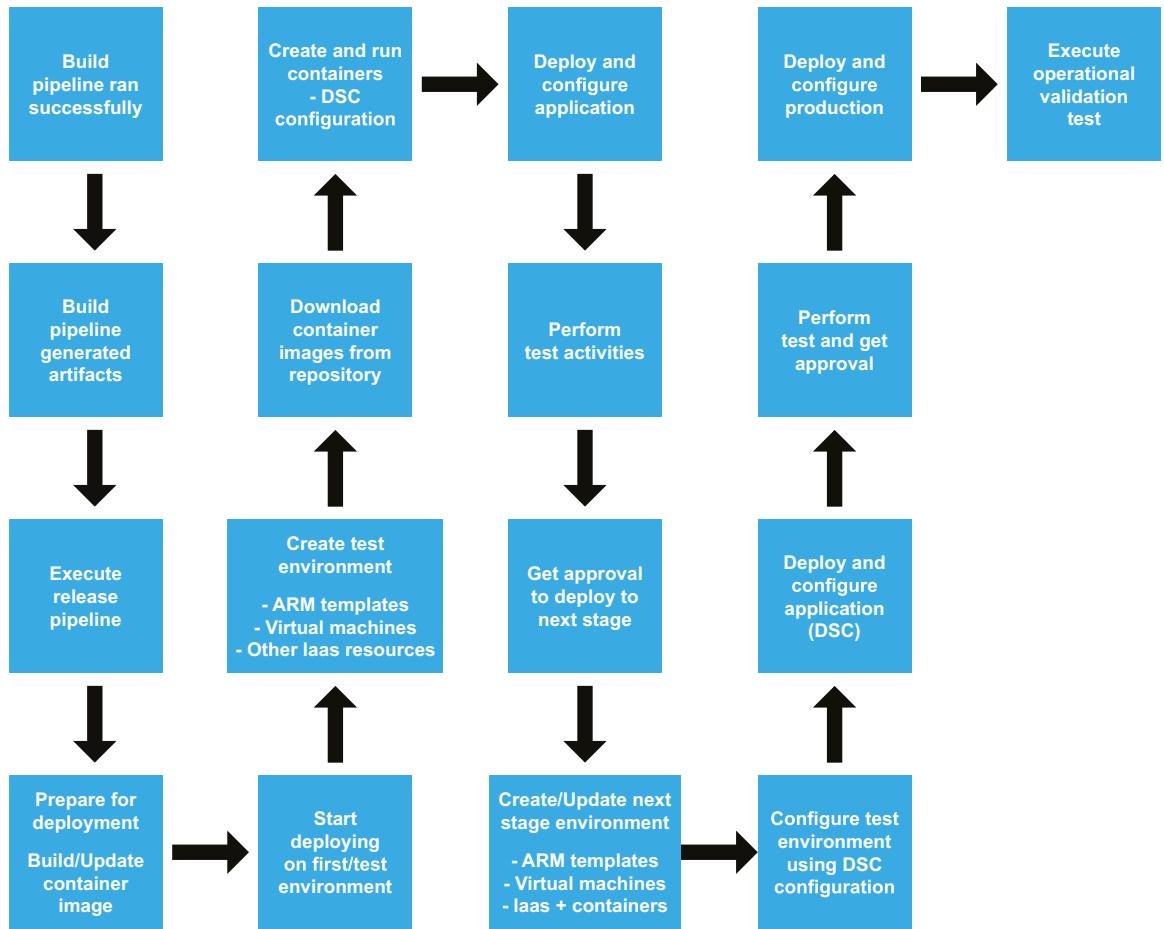


Figure 13.19: Container-based release pipeline

The focus of the next section is integration with other toolsets, such as Jenkins.

Azure DevOps and Jenkins

Azure DevOps is an open platform orchestrator that integrates with other orchestrator tools seamlessly. It provides all the necessary infrastructure and features that integrate well with Jenkins, as well. Organizations with well-established CI/CD pipelines built on Jenkins can reuse them with the advanced but simple features of Azure DevOps to orchestrate them.

Jenkins can be used as a repository and can execute CI/CD pipelines in Azure DevOps, while it's also possible to have a repository in Azure DevOps and execute CI/CD pipelines in Jenkins.

The Jenkins configuration can be added in Azure DevOps as service hooks, and whenever any code change is committed to the Azure DevOps repository, it can trigger pipelines in Jenkins. Figure 13.20 shows the configuration of Jenkins from the Azure DevOps service hook configuration section:

The screenshot shows the 'Project Settings' page for a project named 'learnarmtemplates / dummpipelines'. The 'Service hooks' section is highlighted in the sidebar. The main area displays a brief description: 'Integrate with your favorite services by notifying them when events happen in your project.' A blue button labeled '+ Create subscription' is visible. The sidebar also includes links for General, Overview, Teams, Security, Notifications, Service hooks (which is selected and highlighted in blue), and Dashboards.

Figure 13.20: Configuration of Jenkins

There are multiple triggers that execute the pipelines in Jenkins; one of them is **Code pushed**, as shown in Figure 13.21:

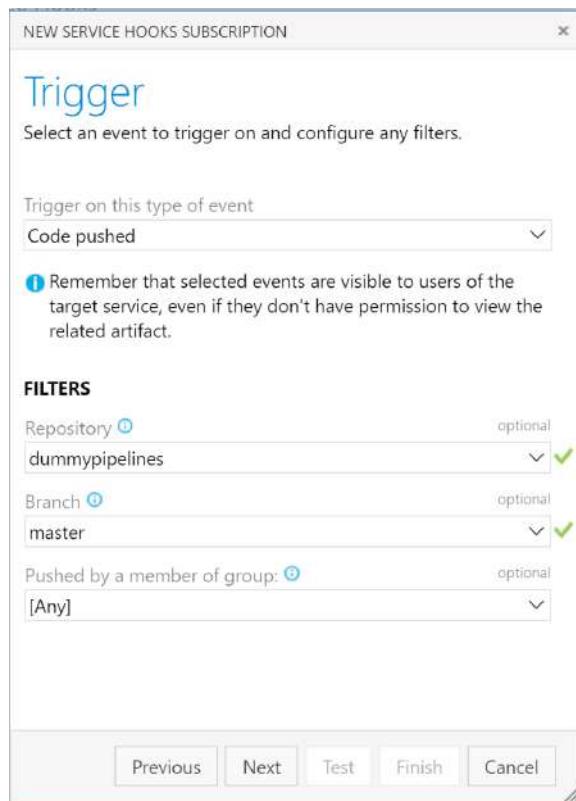


Figure 13.21: Code pushed trigger executed

It's also possible to deploy to Azure VM and execute Azure DevOps release pipelines, as explained here: <https://docs.microsoft.com/azure/virtual-machines/linux/tutorial-build-deploy-jenkins>.

Jenkins should already be deployed before using it in any scenario. The deployment process on Linux can be found at <https://docs.microsoft.com/azure/virtual-machines/linux/tutorial-jenkins-github-docker-cicd>.

The next section will be more focused on tools and services related to configuration management. Azure automation provides DSC-related services such as the pull server.

Azure Automation

Azure Automation is Microsoft's platform for all automation implementation with regard to cloud, on-premises, and hybrid deployments. Azure Automation is a mature automation platform that provides rich capabilities in terms of the following:

- Defining assets, such as variables, connections, credentials, certificates, and modules
- Implementing runbooks using Python, PowerShell scripts, and PowerShell workflows
- Providing UIs to create runbooks
- Managing the full runbook life cycle, including building, testing, and publishing
- Scheduling runbooks
- The ability to run runbooks anywhere—on cloud or on-premises
- DSC as a configuration-management platform
- Managing and configuring environments—Windows and Linux, applications, and deployment
- The ability to extend Azure Automation by importing custom modules

Azure Automation provides a DSC pull server that helps to create a centralized configuration management server that consists of configurations for nodes/virtual machines and their constituents.

It implements the hub and spoke pattern wherein nodes can connect to the DSC pull server and download configurations assigned to them, and reconfigure themselves to reflect their desired state. Any changes or deviations within these nodes are autocorrected by DSC agents the next time they run. This ensures that administrators don't need to actively monitor the environment to find any deviations.

DSC provides a declarative language in which you define the intent and configuration, but not how to run and apply those configurations. These configurations are based on the PowerShell language and ease the process of configuration management.

In this section, we'll look into a simple implementation of using Azure Automation DSC to configure a virtual machine to install and configure the web server (IIS) and create an `index.htm` file that informs users that the website is under maintenance.

Next, you will learn how to provision an Azure Automation account.

Provisioning an Azure Automation account

Create a new Azure Automation account from the Azure portal or PowerShell within an existing or new resource group. You may notice in Figure 13.22 that Azure Automation provides menu items for DSC:

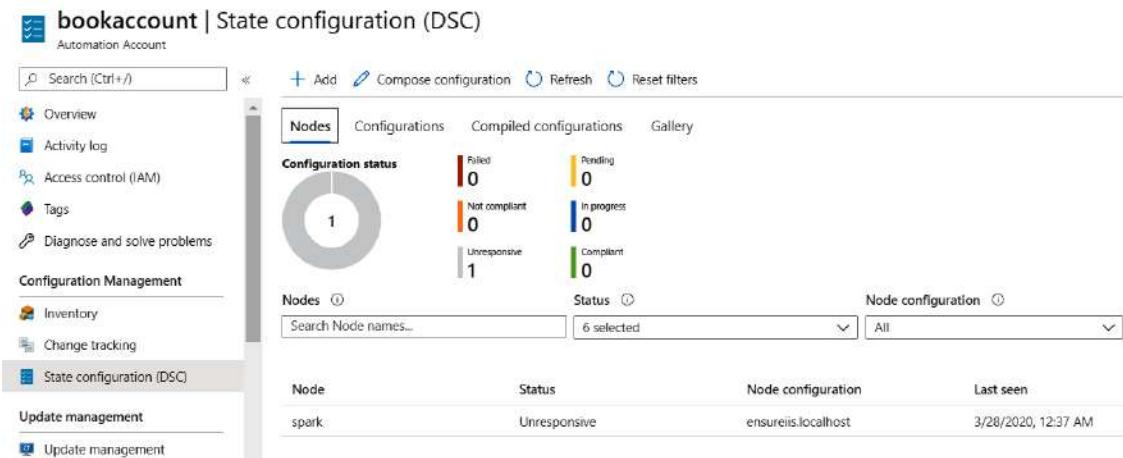


Figure 13.22: DSC in an Azure Automation account

It provides the following:

- **DSC nodes:** These list all the virtual machines and containers that are enlisted with the current Azure Automation DSC pull server. These virtual machines and containers are managed using configurations from the current DSC pull server.
- **DSC configurations:** These list all the raw PowerShell configurations imported and uploaded to the DSC pull server. They are in human-readable format and aren't in a compiled state.
- **DSC node configurations:** These list all compiles of DSC configurations available on the pull server to be assigned to nodes—virtual machines and containers. A DSC configuration produces MOF files after compilations and they're eventually used to configure nodes.

After provisioning an Azure Automation account, we can create a sample DSC configuration, as shown in the next section.

Creating a DSC configuration

The next step is to write a DSC configuration using any PowerShell editor to reflect the intent of the configuration. For this sample, a single configuration, **ConfigureSiteOnIIS**, is created. It imports the base DSC module, **PSDesiredStateConfiguration**, which consists of resources used within the configuration. It also declares a node web server. When this configuration is uploaded and compiled, it will generate a DSC configuration named **ConfigureSiteOnIISwebserver**. This configuration can then be applied to nodes.

The configuration consists of a few resources. These resources configure the target node. The resources install a web server, ASP.NET, and framework, and create an **index.htm** file within the **inetpub\wwwroot** directory with content to show that the site is under maintenance. For more information about writing DSC configuration, refer to <https://docs.microsoft.com/powershell/scripting/dsc/getting-started/wingettingstarted?view=powershell-7>.

The next code listing shows the entire configuration described in the previous paragraph. This configuration will be uploaded to the Azure Automation account:

```
Configuration ConfigureSiteOnIIS {
    Import-DscResource -ModuleName 'PSDesiredStateConfiguration'
    Node WebServer {
        WindowsFeature IIS
        {
            Name = "Web-Server"
            Ensure = "Present"
        }
        WindowsFeature AspDotNet
        {
            Name = "net-framework-45-Core"
            Ensure = "Present"
            DependsOn = "[WindowsFeature]IIS"
        }
    }
}
```

```

WindowsFeature AspNet45
{
    Ensure          = "Present"
    Name            = "Web-Asp-Net45"
    DependsOn      = "[WindowsFeature]AspNetDotNet"
}

File IndexFile
{
    DestinationPath = "C:\inetpub\wwwroot\index.htm"
    Ensure          = "Present"
    Type            = "File"
    Force           = $true
    Contents        = "<HTML><HEAD><Title> Website under construction.</
Title></HEAD><BODY> '"
                    <h1>If you are seeing this page, it means the website is under
maintenance and DSC Rocks !!!!!</h1></BODY></HTML>"
}

}
}
}

```

After creating a sample DSC configuration, it should be imported within Azure Automation as shown in the next section.

Importing the DSC configuration

The DSC configuration still isn't known to Azure Automation. It's available on some local machines. It should be uploaded to Azure Automation DSC configurations. Azure Automation provides the **Import-AzureRMAutomationDscConfiguration** cmdlet to import the configuration to Azure Automation:

```
Import-AzureRmAutomationDscConfiguration -SourcePath "C:\DSC\AA\DSConfigurations\
ConfigureSiteOnIIS.ps1" -ResourceGroupName "omsauto" -AutomationAccountName
"datacenterautomation" -Published -Verbose
```

The commands will import the configuration within Azure Automation. After importing, the DSC configuration should be compiled so that it can be assigned to servers for compliance checks and autoremediation.

Compiling the DSC configuration

After the DSC configuration is available in Azure Automation, it can be asked to compile. Azure Automation provides another cmdlet for this. Use the **Start-AzureRmAutomationDscCompilationJob** cmdlet to compile the imported configuration. The configuration name should match the name of the uploaded configuration. Compilation creates an MOF file named after the configuration and node name together, which in this case is the **ConfigureSiteOnIIS** web server. The execution of the command is shown here:

```
Start-AzureRmAutomationDscCompilationJob -ConfigurationName ConfigureSiteOnIIS  
-ResourceGroupName "omsauto" -AutomationAccountName "datacenterautomation"  
-Verbose
```

Now you have accomplished DSC node configuration. In the next section, you will learn to assign configurations to nodes.

Assigning configurations to nodes

The compiled DSC configurations can be applied to nodes. Use **Register-AzureRmAutomationDscNode** to assign the configuration to a node.

The **NodeConfigurationName** parameter identifies the configuration name that should be applied to the node. This is a powerful cmdlet that can also configure the DSC agent, which is **localconfigurationmanager**, on nodes before they can download configurations and apply them. There are multiple **localconfigurationmanager** parameters that can be configured—details are available at <https://devblogs.microsoft.com/powershell/understanding-meta-configuration-in-windows-powershell-desired-state-configuration/>.

Let's heck out the configuration below:

```
Register-AzureRmAutomationDscNode -ResourceGroupName "omsauto"  
-AutomationAccountName "datacenterautomation" -AzureVMName testtwo  
-ConfigurationMode ApplyAndAutocorrect -ActionAfterReboot ContinueConfiguration  
-AllowModuleOverwrite $true -AzureVMResourceGroup testone -AzureVMLocation  
"West Central US" -NodeConfigurationName "ConfigureSiteOnIIS.WebServer"  
-Verbose
```

Now, we can test whether the configuration has been applied to the servers by browsing the newly deployed website using a browser. After the testing has completed successfully, let's move on to validating the connections.

Validation

If appropriate, network security groups and firewalls are opened and enabled for port 80, and a public IP is assigned to the virtual machine. The default website can be browsed using the IP address. Otherwise, log into the virtual machine that's used to apply the DSC configuration and navigate to <http://localhost>.

It should show the following page:



Figure 13.23: Localhost

This is the power of configuration management: without writing any significant code, authoring a configuration once can be applied multiple times to the same and multiple servers, and you can be assured that they will run in the desired state without any manual intervention. In the next section, we will check out the various tools available for Azure DevOps.

Tools for DevOps

As mentioned before, Azure is a rich and mature platform that supports the following:

- Multiple choices of languages
- Multiple choices of operating systems
- Multiple choices of tools and utilities
- Multiple patterns for deploying solutions (such as virtual machines, app services, containers, and microservices)

With so many options and choices, Azure offers the following:

- **Open cloud:** It is open to open source, Microsoft, and non-Microsoft products, tools, and services.
- **Flexible cloud:** It is easy enough for both end users and developers to use it with their existing skills and knowledge.
- **Unified management:** It provides seamless monitoring and management features.

All the services and capabilities mentioned here are important for the successful implementation of DevOps. Figure 13.24 shows the open source tools and utilities that can be used for different phases of managing the application life cycle and DevOps in general:

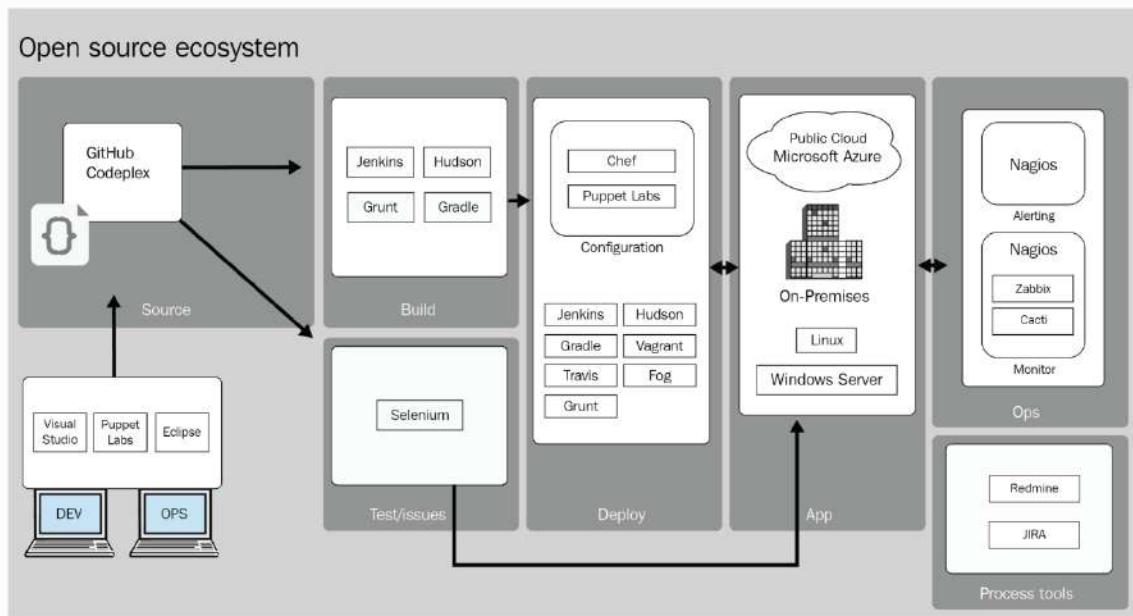


Figure 13.24: Open source tools and utilities

Figure 13.24 shows the Microsoft tools and utilities that can be used for different phases of managing the application life cycle and DevOps in general. Again, this is just a small representation of all the tools and utilities—there are many more options available, such as the following:

- Azure DevOps build orchestration for constructing a build pipeline
- Microsoft Test Manager and Pester for testing
- DSC, PowerShell, and ARM templates for deployment or configuration management
- Log Analytics, Application Insights, and **System Center Operations Manager (SCOM)** for alerting and monitoring
- Azure DevOps and System Center Service Manager for managing processes:

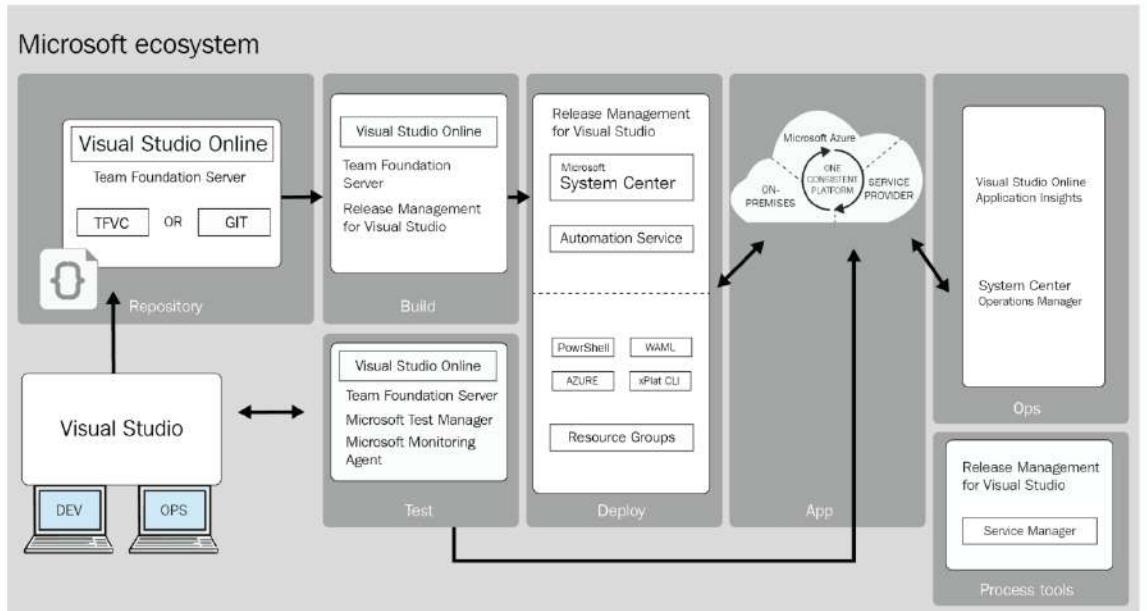


Figure 13.25: Microsoft tools and utilities

There are many tools available for each of the DevOps practices and in this section, you saw some of the tools and the way to configure them.

Summary

DevOps is gaining a lot of traction and momentum in the industry. Most organizations have realized its benefits and are looking to implement DevOps. This is happening while most of them are moving to the cloud. Azure, as a cloud platform, provides rich and mature DevOps services, making it easy for organizations to implement DevOps.

In this chapter, we discussed DevOps along with its core practices, such as configuration management, continuous integration, continuous delivery, and deployment. We also discussed different cloud solutions based on PaaS, a virtual machine IaaS, and a container IaaS, along with their respective Azure resources, the build and release pipelines.

Configuration management was also explained in the chapter, along with DSC services from Azure Automation and using pull servers to configure virtual machines automatically. Finally, we covered Azure's openness and flexibility regarding the choice of languages, tools, and operating systems.

In the next chapter, we will go through the details of Kubernetes and its components and interactions, in addition to application design and deployment considerations on Kubernetes.

14

Architecting Azure Kubernetes solutions

Containers are one of the most talked-about infrastructure components of the last decade. Containers are not a new technology; they have been around for quite some time. They have been prevalent in the Linux world for more than two decades. Containers were not well known in the developer community due to their complexity and the fact that there was not much documentation regarding them. However, around the beginning of this decade, in 2013, a company was launched known as Docker that changed the perception and adoption of containers within the developer world.

Docker wrote a robust API wrapper on top of existing Linux LXC containers and made it easy for developers to create, manage, and destroy containers from the command-line interface. When containerizing applications, the number of containers we have can increase drastically over time, and we can reach a point where we need to manage hundreds or even thousands of containers. This is where container orchestrators play a role, and Kubernetes is one of them. Using Kubernetes, we can automate the deployment, scaling, networking, and management of containers.

In this chapter, we will look at:

- The introductory concepts of containers
- The concepts of Kubernetes
- The important elements that make Kubernetes work
- Architecting solutions using Azure Kubernetes Service

Now that you know what Kubernetes is used for, let's start from scratch and discuss what containers are, how they are orchestrated using Kubernetes, and more.

Introduction to containers

Containers are referred to as operating system-level virtualization systems. They are hosted on an operating system running either on a physical server or a virtual server. The nature of the implementation depends on the host operating system. For example, Linux containers are inspired by cgroups; on the other hand, Windows containers are almost lightweight virtual machines with a small footprint.

Containers are truly cross-platform. Containerized applications can run on any platform, such as Linux, Windows, or Mac, uniformly without any changes being needed, which makes them highly portable. This makes them a perfect technology for organizations to adopt as they are platform-agnostic.

In addition, containers can run in any cloud environment or on-premises environment without changes being needed. This means that organizations are also not tied to a single cloud provider if they implement containers as their hosting platform on the cloud. They can move their environment from on-premises and lift and shift to the cloud.

Containers provide all the benefits that are typically available with virtual machines. They have their own IP addresses, DNS names, identities, networking stacks, filesystems, and other components that give users the impression of using a pristine new operating system environment. Under the hood, the Docker runtime virtualizes multiple operating system kernel-level components to provide that impression.

All these benefits provide immense benefits for organizations adopting container technology, and Docker is one of the forerunners in this regard. There are other container runtime options available, such as CoreOS Rkt (pronounced as Rocket, out of production), Mesos Containerizer, and LXC containers. Organizations can adopt the technology that they feel comfortable with.

Containers were previously not available in the Windows world, only becoming available for Windows 10 and Windows Server 2016. However, containers are now first-class citizens in the Windows world.

As mentioned in the introduction, containers should be monitored, governed, and managed well, just like any other infrastructural component within an ecosystem. It's necessary to deploy an orchestrator, such as Kubernetes, that can help you to do so easily. In the next section, you will learn about the fundamentals of Kubernetes, including what its advantages are.

Kubernetes fundamentals

Many organizations still ask, "Do we need Kubernetes, or indeed any container orchestrator?" When we think about container management on a large scale, we need to think about several points, such as scaling, load balancing, life cycle management, continuous delivery, logging and monitoring, and more.

You might ask, "Aren't containers supposed to do all that?" The answer is that containers are only a low-level piece of the puzzle. The real benefits are gained through the tools that sit on top of the containers. At the end of the day, we need something to help us with orchestration.

Kubernetes is a Greek word, κυβερνήτης, which means "helmsman" or "captain of the ship." Keeping the maritime theme of Docker containers, Kubernetes is the captain of the ship. Kubernetes is often denoted as K8s, where 8 represents the eight letters between "K" and "s" in the word "Kubernetes."

As mentioned before, containers are more agile than virtual machines. They can be created within seconds and destroyed equally quickly. They have a similar life cycle to virtual machines; however, they need to be monitored, governed, and managed actively within an environment.

It is possible to manage them using your existing toolset; even so, specialized tools, such as Kubernetes, can provide valuable benefits:

- Kubernetes is self-healing in nature. When a Pod (read as "container" for now) goes down within a Kubernetes environment, Kubernetes will ensure that a new Pod is created elsewhere either on the same node or on another node, to respond to requests on behalf of the application.
- Kubernetes also eases the process of upgrading an application. It provides out-of-the-box features to help you perform multiple types of upgrades with the original configuration.
- It helps to enable blue-green deployments. In this type of deployment, Kubernetes will deploy the new version of the application alongside the old one, and once it is confirmed that the new application works as expected, a DNS switch will be made to switch to the new version of the application. The old application deployment can continue to exist for rollback purposes.
- Kubernetes also helps to implement a rolling-upgrade deployment strategy. Here, Kubernetes will deploy the new version of the application one server at a time, and tear down the old deployment one server at a time. It will carry on this activity until there are no more servers left from the old deployment.
- Kubernetes can be deployed on an on-premises data center or on the cloud using the **infrastructure as a service (IaaS)** paradigm. This means that developers first create a group of virtual machines and deploy Kubernetes on top of it. There is also the alternative approach of using Kubernetes as a **platform as a service (PaaS)** offering. Azure provides a PaaS service known as **Azure Kubernetes Service (AKS)**, which provides an out-of-the-box Kubernetes environment to developers.

When it comes to Deployment, Kubernetes can be deployed in two ways:

- **Unmanaged clusters:** Unmanaged clusters can be created by installing Kubernetes and any other relevant packages on a bare-metal machine or a virtual machine. In an unmanaged cluster, there will be master and worker nodes, formerly known as minions. The master and worker nodes work hand-in-hand to orchestrate the containers. If you are wondering how this is achieved, later in this chapter, we will be exploring the complete architecture of Kubernetes. Right now, just know that there are master and worker nodes.

- **Managed clusters:** Managed clusters are normally provided by the cloud provider; the cloud provider manages the infrastructure for you. In Azure, this service is called AKS. Azure will provide active support regarding patching and managing the infrastructure. With IaaS, organizations have to ensure the availability and scalability of the nodes and the infrastructure on their own. In the case of AKS, the master component will not be visible as it is managed by Azure. However, the worker nodes (minions) will be visible and will be deployed to a separate resource group, so you can access the nodes if needed.

Some of the key benefits of using AKS over unmanaged clusters are:

- If you are using unmanaged clusters, you need to work to make the solution highly available and scalable. In addition to that, you need to have proper update management in place to install updates and patches. On the other hand, in AKS, Azure manages this completely, enabling developers to save time and be more productive.
- Native integration with other services, such as Azure Container Registry to store your container images securely, Azure DevOps to integrate CI/CD pipelines, Azure Monitor for logging, and Azure Active Directory for security.
- Scalability and faster startup speed.
- Support for virtual machine scale sets.

While there is no difference in terms of the basic functionality of these two deployments, the IaaS form of deployment provides the flexibility to add new plugins and configuration immediately that might take some time for the Azure team to make available with AKS. Also, newer versions of Kubernetes are available within AKS quite quickly, without much delay.

We have covered the basics of Kubernetes. At this point, you might be wondering how Kubernetes achieves all this. In the next section, we will be looking at the components of Kubernetes and how they work hand-in-hand.

Kubernetes architecture

The first step in understanding Kubernetes is understanding its architecture. We will go into the details of each component in the next section, but getting a high-level overview of the architecture will help you to understand the interaction between the components.

Kubernetes clusters

Kubernetes needs physical or virtual nodes for installing two types of components:

- Kubernetes control plane components, or master components
- Kubernetes worker nodes (minions), or non-master components

Figure 14.1 is a diagram that offers a high-level overview of Kubernetes' architecture. We will get into the components in more detail later on:

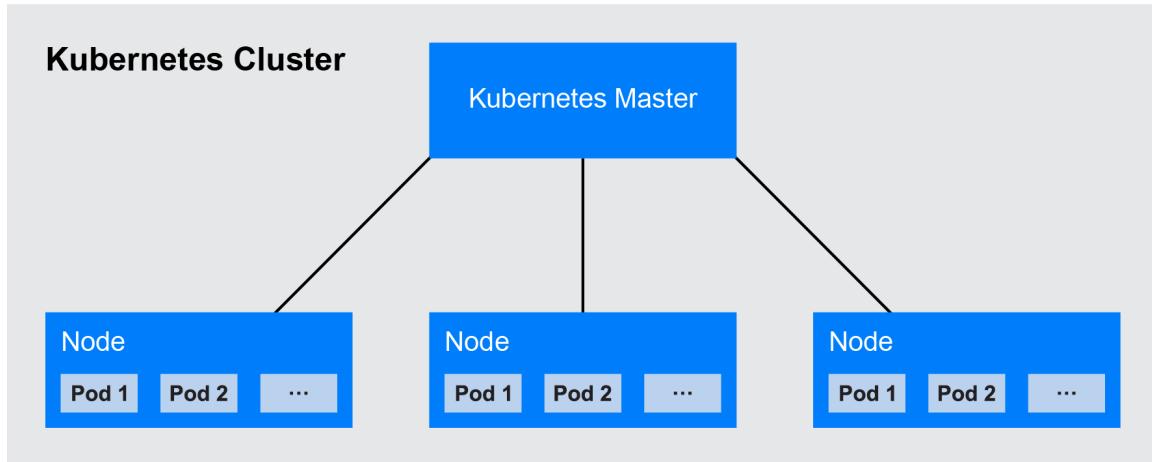


Figure 14.1: Kubernetes cluster overview

The control plane components are responsible for managing and governing the Kubernetes environment and Kubernetes minions.

All nodes together—the master as well as the minions—form the cluster. A cluster, in other words, is a collection of nodes. They are virtual or physical, connected to each other, and reachable using the TCP networking stack. The outside world will have no clue about the size or capability of your cluster, or even the names of the worker nodes. The only thing the nodes are aware of is the address of the API server through which they interact with the cluster. For them, the cluster is one large computer that runs their applications.

It is Kubernetes that internally decides an appropriate strategy, using controllers, to choose a valid, healthy node that can run the application smoothly.

The control plane components can be installed in a high-availability configuration. So far, we have discussed clusters and how they work. In the next section, we will be taking a look at the components of a cluster.

Kubernetes components

Kubernetes components are divided into two categories: master components and node components. The master components are also known as the control plane of the cluster. The control plane is responsible for managing the worker nodes and the Pods in the cluster. The decision-making authority of a cluster is the control plane, and it also takes care of detection and responses related to cluster events. Figure 14.2 describes the complete architecture of a Kubernetes cluster:

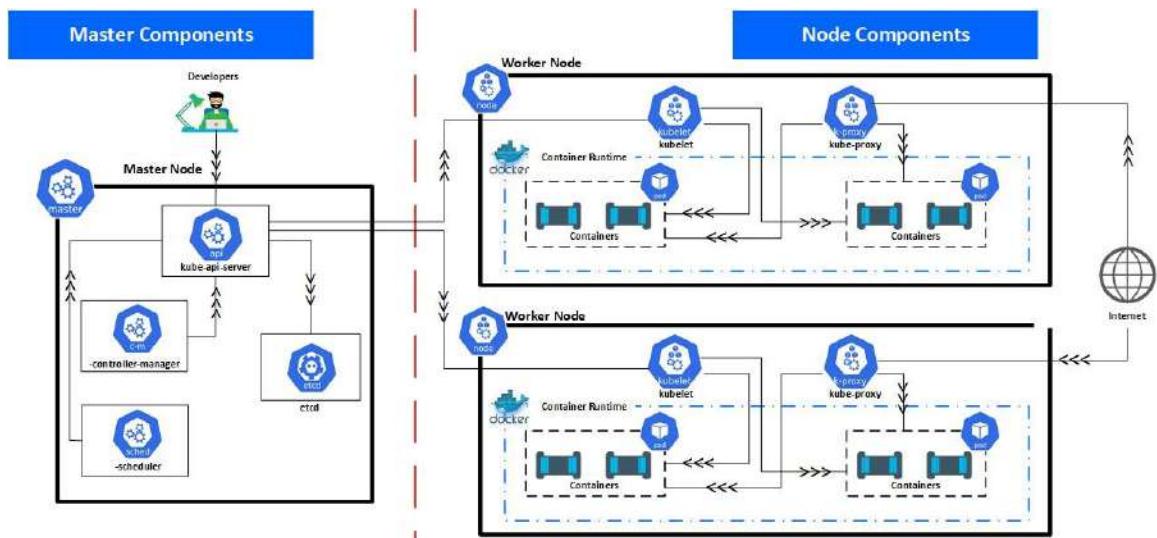


Figure 14.2: Kubernetes architecture

You need to understand each of these components to administer a cluster correctly. Let's go ahead and discuss what the master components are:

- **API server:** The API server is undoubtedly the brain of Kubernetes. It is the central component that enables all activities within Kubernetes. Every client request, with few exceptions, ends up with the API server, which decides the flow for the request. It is solely responsible for interacting with the etcd server.
- **etcd:** etcd is the data store for Kubernetes. Only the API server is allowed to communicate with etcd, and the API server can perform **Create, Read, Update** and **Delete (CRUD)** activities on etcd. When a request ends up with the API server, after validation, the API server can perform any CRUD operations, depending on the etcd request. etcd is a distributed, highly available data store. There can be multiple installations of etcd, each with a copy of the data, and any of them can serve the requests from the API server. In Figure 14.3, you can see that there are multiple instances running in the control plane to provide high availability:

kubeadm HA topology - stacked etcd

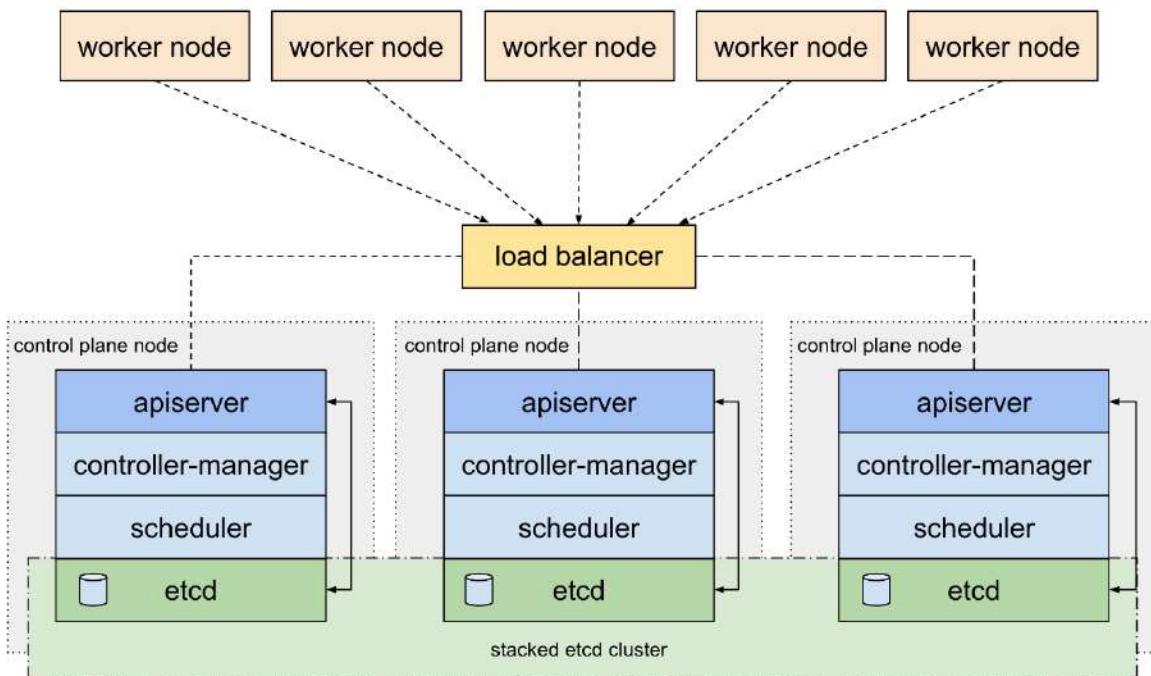


Figure 14.3: Making the control plane highly available

- **Controller manager:** The controller manager is the workhorse of Kubernetes. While the API server receives the requests, the actual work in Kubernetes is done by the controller manager. The controller manager, as the name suggests, is the manager of the controllers. There are multiple controllers in a Kubernetes master node, and each is responsible for managing a single controller.

The main responsibility of a controller is managing a single resource in a Kubernetes environment. For example, there is a replication controller manager for managing replication controller resources, and a ReplicaSet controller to manage ReplicaSets in a Kubernetes environment. The controller keeps a watch on the API server, and when it receives a request for a resource managed by it, the controller performs its job.

One of the main responsibilities of controllers is to keep running in a loop and ensure that Kubernetes is in the desired state. If there is any deviation from the desired state, the controllers should bring it back to the desired state. A deployment controller watches for any new deployment resources created by the API server. If a new deployment resource is found, the deployment controller creates a new ReplicaSet resource and ensures that the ReplicaSet is always in the desired state. A replication controller keeps running in a loop and checks whether the actual number of Pods in the environment matches the desired number of Pods. If a Pod dies for any reason, the replication controller will find that the actual count has gone down by one and it will schedule a new Pod in the same or another node.

- **Scheduler:** The job of a scheduler is to schedule the Pods on Kubernetes minion nodes. It is not responsible for creating Pods. It is purely responsible for assigning Pods to Kubernetes minion nodes. It does so by taking into account the current state of nodes, how busy they are, their available resources, and also the definition of the Pod. A Pod might have a preference regarding a specific node, and the scheduler will keep these requests in consideration while scheduling Pods to nodes.

We will now explore the node components that are deployed in each of the worker nodes in the cluster:

- **Kubelet:** While the API server, scheduler, controllers, and etcd are deployed on master nodes, kubelets are deployed on minion nodes. They act as agents for the Kubernetes master components and are responsible for managing Pods locally on the nodes. There is one kubelet on each node. A kubelet takes commands from the master components and also provides health, monitoring, and update information about nodes and Pods to the master components, such as the API server and the controller manager. They are the conduit for administrative communication between the master and minion nodes.
- **kube-proxy:** kube-proxy, just like kubelets, is deployed on minion nodes. It is responsible for monitoring Pods and Services, as well as updating the local iptables and netfilter firewall rules with any change in the availability of Pods and Services. This ensures that the routing information on nodes is updated as and when new Pods and Services are created or existing Pods and Services are deleted.

- **Container runtime:** There are many container vendors and providers in the ecosystem today. Docker is the most famous of them all, though others are also gaining popularity. That's why, in our architecture, we denoted the container runtime with the Docker logo. Kubernetes is a generic container orchestrator. It cannot be tightly coupled with any single container vendor, such as Docker. It should be possible to use any container runtime on the minion nodes to manage the life cycle of containers.

To run containers in Pods, an industry-based standard known as a **container runtime interface (CRI)** has been developed and is used by all leading companies. The standard provides rules that should be followed to achieve interoperability with orchestrators such as Kubernetes. Kubelets do not know which container binaries are installed on the nodes. They could be Docker binaries or any other binaries.

As these container runtimes are developed with a common industry-based standard, irrespective of which runtime you are using, kubelets will be able to communicate with the container runtime. This decouples container management from Kubernetes cluster management. The responsibilities of the container runtime include the creation of containers, managing the networking stack of the containers, and managing the bridge network. Since the container management is separate from the cluster management, Kubernetes will not interfere in the responsibilities of the container runtime.

The components we discussed are applicable to both unmanaged as well as managed AKS clusters. However, the master components are not exposed to the end user, as Azure manages all that in the case of AKS. Later in this chapter, we will cover the architecture of AKS. You will learn about unmanaged clusters and come to understand the differences between these systems more clearly.

Next, you will learn about some of the most important Kubernetes resources, also known as the primitives, knowledge that is applicable to both unmanaged and AKS clusters.

Kubernetes primitives

You have learned that Kubernetes is an orchestration system used to deploy and manage containers. Kubernetes defines a set of building blocks, which are also known as primitives. These primitives together can help us to deploy, maintain, and scale containerized applications. Let's take a look at each of the primitives and understand their roles.

Pod

Pods are the most basic unit of Deployment in Kubernetes. The immediate question that arises to a curious mind is how is a Pod different to a container? Pods are wrappers on top of containers. In other words, containers are contained within Pods. There can be multiple containers within a Pod; however, best practice is to have a one-Pod-one-container relationship. This does not mean we cannot have more than one container in a Pod. Multiple containers in a Pod is also fine, as long as there is one main container and the rest are ancillary containers. There are also patterns, such as sidecar patterns, that can be implemented with multi-container Pods.

Each Pod has its own IP address and networking stack. All containers share the network interface and the stack. All containers within a Pod can be reached locally using the hostname.

A simple Pod definition in YAML format is shown in the following lines of code:

```
---
```

```
apiVersion: v1
kind: Pod
metadata:
  name: tappdeployment
  labels:
    appname: tapp
    ostype: linux
spec:
  containers:
    - name: mynewcontainer
      image: "tacracr.azurecr.io/tapp:latest"
      ports:
        - containerPort: 80
          protocol: TCP
          name: http
```

The Pod definition shown has a name and defines a few labels, which can be used by the Service resource to expose to other Pods, nodes and external custom resources. It also defines a single container based on a custom image stored in Azure Container Registry and opens port **80** for the container.

Services

Kubernetes allows creating Pods with multiple instances. These Pods should be reachable from any Pod or node within a cluster. It is possible to use the IP address of a Pod directly and access the Pod. However, this is far from ideal. Pods are ephemeral and they might get a new IP address if the previous Pod has gone down. In such cases, the application will break easily. Kubernetes provides Services, which decouple Pod instances from their clients. Pods may get created and torn down, but the IP address of a Kubernetes Service remains constant and stable. Clients can connect to the Service IP address, which in turn has one endpoint for each Pod it can send requests to. If there are multiple Pod instances, each of their IP addresses will be available to the Service as an endpoint object. When a Pod goes down, the endpoints are updated to reflect the current Pod instances along with their IP addresses.

Services are highly decoupled with Pods. The main intention of Services is to queue for Pods that have labels in their Service selector definitions. A Service defines label selectors, and based on label selectors, Pod IP addresses are added to the Service resource. Pods and Services can be managed independently of each other.

A Service provides multiple types of IP address schemes. There are four types of Services: ClusterIP, NodePort, LoadBalancer, and Ingress Controller using Application Gateway.

The most fundamental scheme is known as ClusterIP, and it is an internal IP address that can be reached only from within the cluster. The ClusterIP scheme is shown in Figure 14.4:

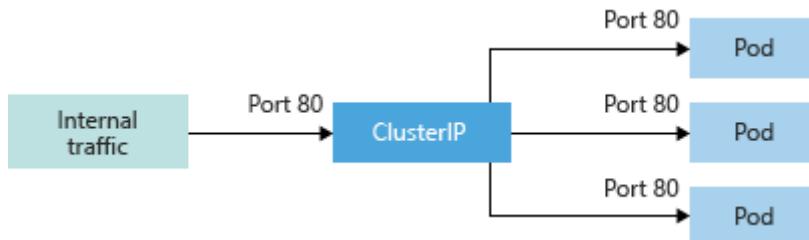


Figure 14.4: The workings of ClusterIP

ClusterIP also allows the creation of NodePort, using which it gets a ClusterIP. However, it can also open a port on each of the nodes within a cluster. The Pods can be reached using ClusterIP addresses as well as by using a combination of the node IP and node port:

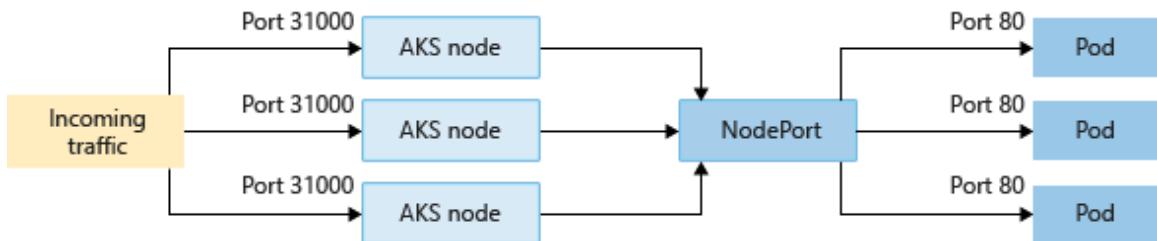


Figure 14.5: The workings of NodePort

Services can refer not only to Pods but to external endpoints as well. Finally, Services also allow the creation of a load balancer-based service that is capable of receiving requests externally and redirecting them to a Pod instance using ClusterIP and NodePort internally:

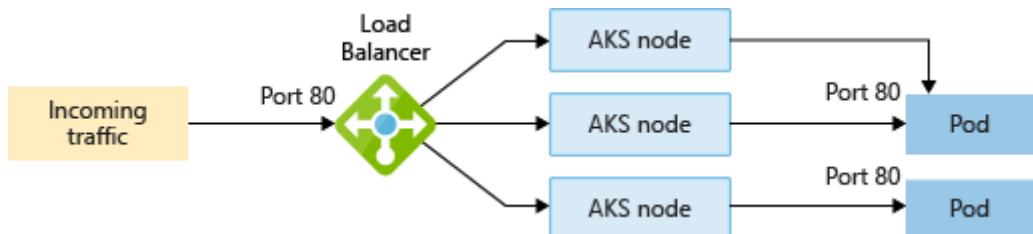


Figure 14.6: The workings of Load Balancer

There is one final type of service known as Ingress Controller, which provides advanced functionalities such as URL-based routing, as shown in Figure 14.7:

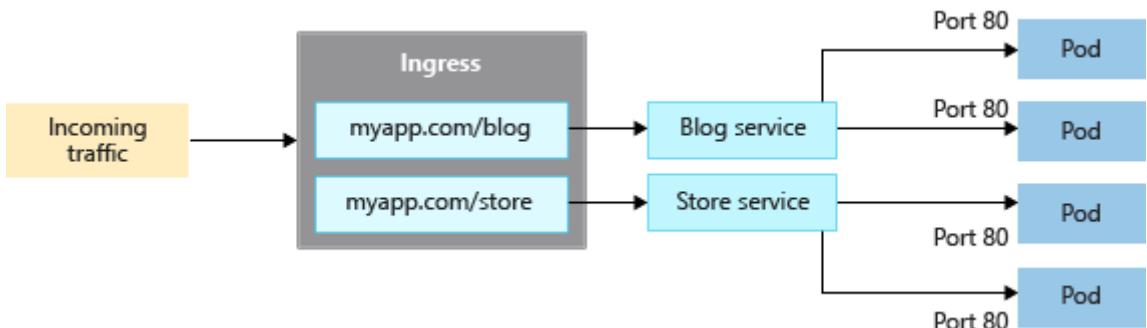


Figure 14.7: The workings of Ingress Controller

A service definition in YAML format is shown here:

```
apiVersion: v1
kind: Service
metadata:
  name: tappservice
  labels:
    appname: tapp
    ostype: linux
spec:
  type: LoadBalancer
  selector:
    appname: myappnew
  ports:
  - name: http
    port: 8080
    targetPort: 80
    protocol: TCP
```

This service definition creates a load balancer-based service using label selectors.

Deployments

Kubernetes Deployments are higher-level resources in comparison to ReplicaSets and Pods. Deployments provide functionality related to the upgrading and release of an application. Deployment resources create a ReplicaSet, and the ReplicaSet manages the Pod. It is important to understand the need for deployment resources when ReplicaSets already exist.

Deployments play a significant role in upgrading applications. If an application is already in production and a new version of the application needs to be deployed, there are a few choices for you:

1. Delete existing Pods and create new Pods – in this method, there is downtime for the application, so this method should only be used if downtime is acceptable.
There is a risk of increased downtime if the Deployment contains bugs and you have to roll back to a previous version.

2. Blue-green deployment – In this method, the existing Pods continue to run and a new set of Pods is created with the new version of the application. The new Pods are not reachable externally. Once the tests have successfully completed, Kubernetes starts pointing to the new set of Pods. The old Pods can stay as-is or can be subsequently deleted.
3. Rolling upgrades – In this method, existing Pods are deleted one at a time while new Pods for the new application version are created one at a time. The new Pods are incrementally deployed while the old Pods are incrementally reduced, until they reach a count of zero.

All these approaches would have to be carried out manually without a Deployment resource. A Deployment resource automates the entire release and upgrade process. It can also help to automatically roll back to a previous version if there are any issues with the current Deployment.

A Deployment definition is shown in the following code listing:

```
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tappdeployment
  labels:
    appname: tapp
    ostype: linux
spec:
  replicas: 3
  selector:
    matchLabels:
      appname: myappnew
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  template:
```

```
metadata:  
  name: mypod  
  labels:  
    appname: myappnew  
spec:  
  containers:  
    - name: mynewcontainer  
      image: "tacracr.azurecr.io/tapp:latest"  
      ports:  
        - containerPort: 80  
          protocol: TCP  
          name: http
```

It is important to note that a Deployment has a **strategy** property, which determines whether the **recreate** or **RollingUpdate** strategy is used. **recreate** will delete all existing Pods and create new Pods. It also contains configuration details related to **RollingUpdate** by providing the maximum number of Pods that can be created and destroyed in a single execution.

Replication controller and ReplicaSet

Kubernetes' replication controller resource ensures that a specified desired number of Pod instances are always running within a cluster. Any deviation from the desired state is watched for by the replication controller, and it creates new Pod instances to meet the desired state.

ReplicaSets are the new version of the replication controller. ReplicaSets provide the same functionality as that of replication controllers, with a few advanced functionalities. The main one among these is the rich capability for defining the selectors associated with Pods. With ReplicaSets, it is possible to define the dynamic expressions that were missing with replication controllers.

It is recommended to use ReplicaSets rather than replication controllers.

The next code listing shows an example of defining a **ReplicaSet** resource:

```
---  
apiVersion: apps/v1  
kind: ReplicaSet  
metadata:
```

```
name: tappdeployment
labels:
  appname: tapp
  ostype: linux
spec:
  replicas: 3
  selector:
    matchLabels:
      appname: myappnew
  template:
    metadata:
      name: mypod
    labels:
      appname: myappnew
  spec:
    containers:
      - name: mynewcontainer
        image: "tacracr.azurecr.io/tapp:latest"
        ports:
          - containerPort: 80
            protocol: TCP
            name: http
```

It is important to note that ReplicaSets have a **replicas** property, which determines the count of Pod instances, a **selector** property, which defines the Pods that should be managed by ReplicaSet, and finally the **template** property, which defines the Pod itself.

ConfigMaps and Secrets

Kubernetes provides two important resources to store configuration data. ConfigMaps are used to store general configuration data that is not security-sensitive. Generic application configuration data, such as folder names, volume names, and DNS names, can be stored in ConfigMaps. On the other hand, sensitive data, such as credentials, certificates, and secrets, should be stored within Secrets resources. This Secrets data is encrypted and stored within the Kubernetes etcd data store.

Both ConfigMaps and Secrets data can be made available as environment variables or volumes within Pods.

The definition of the Pod that wants to consume these resources should include a reference to them. We have now covered the Kubernetes primitives and the roles of each of the building blocks. Next, you will be learning about the architecture of AKS.

AKS architecture

In the previous section, we discussed the architecture of an unmanaged cluster. Now, we will be exploring the architecture of AKS. When you have read this section, you will be able to point out the major differences between the architecture of unmanaged and managed (AKS, in this case) clusters.

When an AKS instance is created, the worker nodes only are created. The master components are managed by Azure. The master components are the API server, the scheduler, etcd, and the controller manager, which we discussed earlier. The kubelets and kube-proxy are deployed on the worker nodes. Communication between the nodes and master components happens using kubelets, which act as agents for the Kubernetes clusters for the node:

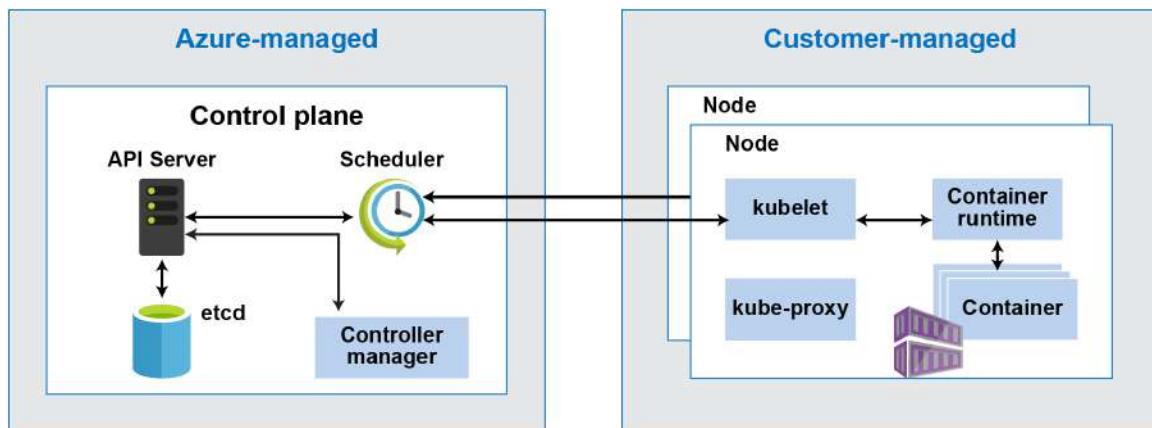


Figure 14.8: AKS architecture

When a user requests a Pod instance, the user request lands with the API server. The API server checks and validates the request details and stores in etcd (the data store for the cluster) and also creates the deployment resource (if the Pod request is wrapped around a deployment resource). The deployment controller keeps a watch on the creation of any new deployment resources. If it sees one, it creates a ReplicaSet resource based on the definition provided in the user request.

The ReplicaSet controller keeps a watch on the creation of any new ReplicaSet resources, and upon seeing a resource being created, it asks the scheduler to schedule the Pods. The scheduler has its own procedure and rules for finding an appropriate node for hosting the Pods. The scheduler informs the kubelet of the node and the kubelet then fetches the definition for the Pod and creates the Pods using the container runtime installed on the nodes. The Pod finally creates the containers within its definition.

kube-proxy helps in maintaining the list of IP addresses of Pod and Service information on local nodes, as well as updating the local firewall and routing rules. To do a quick recap of what we have discussed so far, we started off with the Kubernetes architecture and then moved on to primitives, followed by the architecture of AKS. Since you are clear on the concepts, let's go ahead and create an AKS cluster in the next section.

Deploying an AKS cluster

AKS can be provisioned using the Azure portal, the Azure **CLI (command-line interface)**, Azure PowerShell cmdlets, ARM templates, **SDKs (software development kits)** for supported languages, and even Azure ARM REST APIs.

The Azure portal is the simplest way of creating an AKS instance; however, to enable DevOps, it is better to create an AKS instance using ARM templates, the CLI, or PowerShell.

Creating an AKS cluster

Let's create a resource group to deploy our AKS cluster. From the Azure CLI, use the **az group create** command:

```
az group create -n AzureForArchitects -l southeastasia
```

Here, **-n** denotes the name of the resource group and **-l** denotes the location. If the request was successful, you will see a similar response to this:

```
rithin az group create -n AzureForArchitects -l southeastasia
{
  "id": "/subscriptions/.../resourceGroups/AzureForArchitects",
  "location": "southeastasia",
  "managedBy": null,
  "name": "AzureForArchitects",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": null
}
```

Figure 14.9: Resource group creation

Now that we have the resource group ready, we will go ahead and create the AKS cluster using the `az aks create` command. The following command will create a cluster named **AzureForArchitects-AKS** in the **AzureForArchitects** resource group with a node count of 2. The `--generate-ssh-keys` parameter will allow the creation of **RSA (Rivest-Shamir-Adleman)** key pairs, a public-key cryptosystem:

```
az aks create --resource-group AzureForArchitects \
--name AzureForArchitects-AKS \
--node-count 2 \
--generate-ssh-keys
```

If the command succeeded, you will be able to see a similar output to this:

```
[2] rithin az aks create --resource-group AzureForArchitects \
> --name AzureForArchitects-AKS \
> --node-count 2 \
> --generate-ssh-keys
{
  "aadProfile": null,
  "addonProfiles": null,
  "agentPoolProfiles": [
    {
      "availabilityZones": null,
      "count": 2,
      "enableAutoScaling": null,
      "maxCount": null,
      "maxPods": 110,
      "minCount": null,
      "name": "nodepool1",
      "orchestratorVersion": "1.15.11",
      "osDiskSizeGb": 100,
      "osType": "Linux",
      "provisioningState": "Succeeded",
      "type": "AvailabilitySet",
      "vmSize": "Standard_DS2_v2",
      "vnetSubnetId": null
    }
  ],
  "apiServerAuthorizedIpRanges": null,
  "dnsPrefix": "AzureForAr-AzureForArchitec-1b2287",
  "enablePodSecurityPolicy": null,
  "enableRbac": true,
  "fqdn": "azureforar-azureforarchitec-1b2287-72aecee5.hcp.southeastasia.azmk8s.io",
```

Figure 14.10: Creating the cluster

Going through the cluster, you will see a line item that says "`nodeResourceGroup`": "`MC_AzureForArchitects_AzureForArchitects-AKS_southeastasia`". When creating an AKS cluster, a second resource is automatically created to store the node resources.

Our cluster is provisioned. Now we need to connect to the cluster and interact with it. To control the Kubernetes cluster manager, we will be using kubectl. In the next section, we will take a quick look at kubectl.

Kubectl

Kubectl is the main component through which developers and infrastructure consultants can interact with AKS. Kubectl helps in creating a REST request containing the HTTP header and body, and submitting it to the API server. The header contains the authentication details, such as a token or username/password combination. The body contains the actual payload in JSON format.

The kubectl command provides rich log details when used along with the verbose switch. The switch takes an integer input that can range from 0 to 9, which can be viewed from the details logs.

Connecting to the cluster

To connect to the cluster locally, we need to install kubectl. Azure Cloud Shell has kubectl already installed. If you want to connect locally, use **az aks install-cli** to install kubectl.

In order to configure kubectl to connect to our Kubernetes cluster, we need to download the credentials and configure the CLI with them. This can be done using the **az aks get-credentials** command. Use the command as shown here:

```
az aks get-credentials \
--resource-group AzureForArchitects \
--name AzureForArchitects-AKS
```

Now, we need to verify whether we're connected to the cluster. As mentioned earlier, we'll be using kubectl to communicate with the cluster, and **kubectl get nodes** will show a list of nodes in the cluster. During creation, we set the node count to 2, so the output should have two nodes. Also, we need to make sure that the status of the node is **Ready**. The output should be something like Figure 14.11:

rithin kubectl get nodes				
NAME	STATUS	ROLES	AGE	VERSION
aks-nodepool1-31051128-0	Ready	agent	64m	v1.15.11
aks-nodepool1-31051128-1	Ready	agent	64m	v1.15.11

Figure 14.11: Getting the list of nodes

Since our node is in the **Ready** state, let's go ahead and create a Pod. There are two ways in which you can create resources in Kubernetes. They are:

- **Imperative:** In this method, we use the **kubectl run** and **kubectl expose** commands to create the resources.
- **Declarative:** We describe the state of the resource via JSON or a YAML file. While we were discussing Kubernetes primitives, you saw a lot of YAML files for each of the building blocks. We will pass the file to the **kubectl apply** command to create the resources, and the resources declared in the file will be created.

Let's take the imperative approach first, to create a Pod with the name **webserver**, running an NGINX container with port **80** exposed:

```
kubectl run webserver --restart=Never --image nginx --port 80
```

Upon successful completion of the command, the CLI will let you know the status:

```
[root@rithin ~]# kubectl run webserver --restart=Never --image nginx --port 80
pod/webserver created
```

Figure 14.12: Creating a Pod

Now that we have tried the imperative method, let's follow the declarative method. You can use the structure of the YAML file we discussed in the Pod subsection of the *Kubernetes primitives* section and modify it as per your requirements.

We will be using the NGINX image, and the Pod will be named **webserver-2**.

You can use any text editor and create the file. The final file will look similar to this:

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver-2
  labels:
    appname: nginx
    ostype: linux
spec:
```

```

containers:
  - name: wenserver-2-container
    image: nginx
    ports:
      - containerPort: 80
        protocol: TCP
      name: http

```

In the **kubectl apply** command, we will pass the filename to the **-f** parameter, as shown in *Figure 14.13*, and you can see that the Pod has been created:

```

rithin kubectl apply -f nginx.yaml
pod/webserver-2 created

```

Figure 14.13: Creating a Pod using the declarative method.

Since we have created the Pods, we can use the **kubectl get pods** command to list all the Pods. Kubernetes uses the concept of namespaces for the logical isolation of resources. By default, all commands are pointing to the **default** namespace. If you want to perform an action on a specific namespace, you can pass the namespace name via the **-n** parameter. In *Figure 14.14*, you can see that **kubectl get pods** returns the Pods we created in the previous example, which reside in the default namespace. Also, when we use **--all-namespaces**, the output returns pods in all namespaces:

```

rithin kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
webserver   1/1     Running   0          70m
webserver-2  1/1     Running   0          58m
rithin kubectl get pods --all-namespaces
NAMESPACE      NAME           READY   STATUS    RESTARTS   AGE
default        webserver      1/1     Running   0          70m
default        webserver-2    1/1     Running   0          58m
kube-system   coredns-698c77c5d7-12wbd  1/1     Running   0          145m
kube-system   coredns-698c77c5d7-v96gq  1/1     Running   0          142m
kube-system   coredns-autoscaler-5bd7c6759b-7kpzq  1/1     Running   0          145m
kube-system   kube-proxy-95jmg       1/1     Running   0          142m
kube-system   kube-proxy-qsfwq       1/1     Running   0          143m
kube-system   kubernetes-dashboard-74d8c675bc-jzhcf  1/1     Running   1          145m
kube-system   metrics-server-7d654ddc8b-txbt9       1/1     Running   1          145m
kube-system   tunnelfront-79fc68b7f-qwzvj        1/1     Running   0          145m

```

Figure 14.14: Listing all Pods

Now we will create a simple Deployment that runs NGINX and with a load balancer that exposes it to the internet. The **YAML** file will look like this:

```
#Creating a deployment that runs six replicas of nginx
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-server
spec:
  replicas: 6
  selector:
    matchLabels:
      app: nginx-server
  template:
    metadata:
      labels:
        app: nginx-server
    spec:
      containers:
        - name: nginx-server
          image: nginx
          ports:
            - containerPort: 80
              name: http
---
#Creating Service
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
```

```

spec:
  ports:
    - port: 80
  selector:
    app: nginx-server
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-lb
spec:
  type: LoadBalancer
  ports:
    - port: 80
  selector:
    app: nginx-server

```

We'll be using the **kubectl apply** command and passing the YAML file to the **-f** parameter.

Upon success, all three Services will be created, and if you execute the **kubectl get deployment nginx-server** command, you will see six replicas running, as shown in Figure 14.15, to make the Service highly available:

rithin kubectl get deployment nginx-server					
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
nginx-server	6/6	6	6	8m17s	
rithin kubectl get pods					
NAME	READY	STATUS	RESTARTS	AGE	
nginx-server-777ff65664-j7lgw	1/1	Running	0	8m25s	
nginx-server-777ff65664-kxtzx	1/1	Running	0	8m25s	
nginx-server-777ff65664-mh8hj	1/1	Running	0	8m25s	
nginx-server-777ff65664-wn79x	1/1	Running	0	8m25s	
nginx-server-777ff65664-xc7kc	1/1	Running	0	8m25s	
nginx-server-777ff65664-xl4nk	1/1	Running	0	8m25s	

Figure 14.15: Checking the deployment

Since our Deployment is provisioned, we need to check what the public IP of the load balancer that we created is. We can use the `kubectl get service nginx-lb --watch` command. When the load balancer is initializing, `EXTERNAL-IP` will show as `<pending>`, the `--wait` parameter will let the command run in the foreground, and when the public IP is allocated, we will be able to see a new line, as shown here:

rithin kubectl get service nginx-lb --watch					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nginx-lb	LoadBalancer	10.0.140.48	<pending>	80:30686/TCP	68s
nginx-lb	LoadBalancer	10.0.140.48	52.187.70.177	80:30686/TCP	115s

Figure 14.16: Finding public IP of the load balancer

Now that we have the public IP, we can go to the browser and should see the NGINX landing page, as shown in *Figure 14.17*:



Figure 14.17: NGINX landing page

Similarly, you can use the `YAML` files we discussed in the *Kubernetes primitives* section to create different types of resources.

There are a lot of commands, such as `logs`, `describe`, `exec`, and `delete`, that administrators need to use with the `kubectl` command. The objective of this section was to enable you to create an AKS cluster, connect to the cluster, and deploy a simple web application.

In the next section, we will be discussing AKS networking.

AKS networking

Networking forms a core component within a Kubernetes cluster. The master components should be able to reach the minion nodes and the Pods running on top of them, while the worker nodes should be able to communicate among themselves as well as with the master components.

It might come as a surprise that core Kubernetes does not manage the networking stack at all. It is the job of the container runtime on the nodes.

Kubernetes has prescribed three important tenets that any container runtime should adhere to. These are as follows:

- Pods should be able to communicate with other Pods without any transformation in their source or destination addresses, something that is performed using **network address translation (NAT)**.
- Agents such as kubelets should be able to communicate with Pods directly on the nodes.
- Pods that are directly hosted on the host network still should be able to communicate with all Pods in the cluster.

Every Pod gets a unique IP address within the Kubernetes cluster, along with a complete network stack, similar to virtual machines. They all are connected to the local bridge network created by the **Container Networking Interface (CNI)** component. The CNI component also creates the networking stack of the Pod. The bridge network then talks to the host network and becomes the conduit for the flow of traffic from Pods to network and vice versa.

CNI is a standard managed and maintained by the **Cloud Native Computing Foundation (CNCF)**, and there are many providers that provide their own implementation of the interface. Docker is one of these providers. There are others, such as rkt (read as rocket), weave, calico, and many more. Each has its own capabilities and independently decides the network capabilities, while ensuring that the main tenets of Kubernetes networking are followed completely.

AKS provides two distinct networking models:

- Kubenet
- Azure CNI

Kubenet

Kubenet is the default networking framework in AKS. Under Kubenet, each node gets an IP address from the subnet of the virtual network they are connected with. The Pods do not get IP addresses from the subnet. Instead, a separate addressing scheme is used to provide IP addresses to Pods and Kubernetes Services. While creating an AKS instance, it is important to set the IP address range for Pods and Services. Since Pods are not on the same network as that of nodes, requests from Pods and to Pods are always NATed/routed to replace the source Pod IP with the node IP address and vice versa.

In user-defined routing, Azure can support up to 400 routes, and you also cannot have a cluster larger than 400 nodes. Figure 14.18 shows how the AKS node receives an IP address from the virtual network, but not the Pods created in the node:

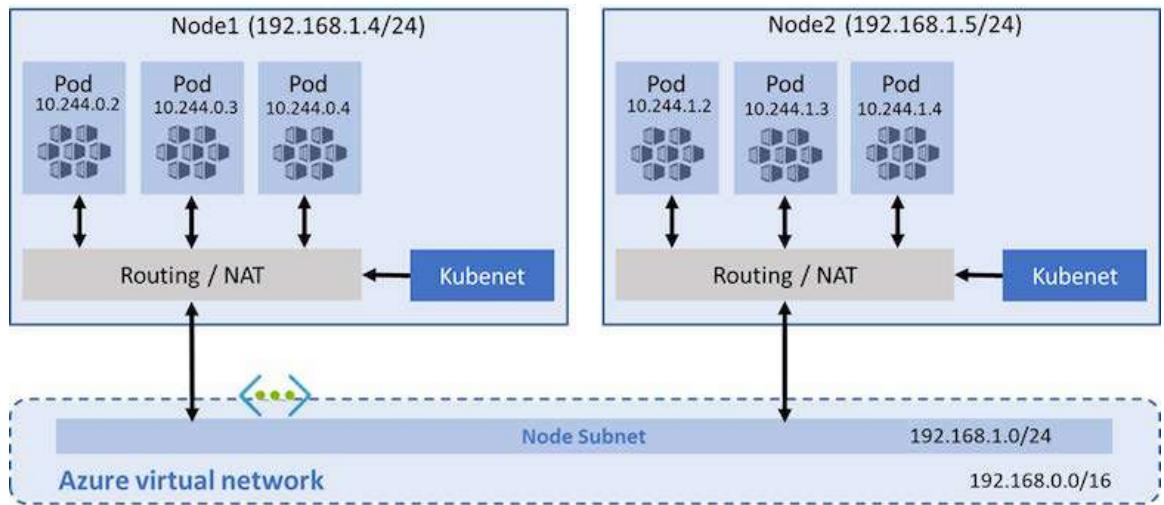


Figure 14.18: Networking in AKS

By default, this Kubenet is configured with 110 Pods per node. This means there can be a maximum of $110 * 400$ Pods in a Kubernetes cluster by default. The maximum number of Pods per node is 250.

This scheme should be used when IP address availability and having user-defined routing are not a constraint.

In the Azure CLI, you can execute the following command to create an AKS instance using this networking stack:

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--node-count 3 \
--network-plugin kubenet \
--service-cidr 10.0.0.0/16 \
--dns-service-ip 10.0.0.10 \
--pod-cidr 10.244.0.0/16 \
--docker-bridge-address 172.17.0.1/16 \
--vnet-subnet-id $SUBNET_ID \
--service-principal <appId> \
--client-secret <password>
```

Notice how all the IP addresses are explicitly provided for Service resources, Pods, nodes, and Docker bridges. These are non-overlapping IP address ranges. Also notice that Kubenet is used as a network plugin.

Azure CNI (advanced networking)

With Azure CNI, each node and Pod gets an IP address assigned from the network subnet directly. This means there can be as many Pods as there are unique IP addresses available on a subnet. This makes IP address range planning much more important under this networking strategy.

It is important to note that Windows hosting is only possible using the Azure CNI networking stack. Moreover, some of AKS components, such as virtual nodes and virtual kubelets, are also dependent on the Azure CNI stack. There is a need to reserve IP addresses in advance, depending on the number of Pods that will be created. There should always be extra IP addresses available on the subnet, to avoid exhaustion of IP addresses or to avoid the need to rebuild the cluster for a larger subnet due to application demand.

By default, this networking stack is configured for 30 Pods per node and it can be configured with 250 Pods as the maximum number of Pods per node.

The command to execute to create an AKS instance using this networking stack is shown here:

```
az aks create \
  --resource-group myResourceGroup \
  --name myAKScluster \
  --network-plugin azure \
  --vnet-subnet-id <subnet-id> \
  --docker-bridge-address 172.17.0.1/16 \
  --dns-service-ip 10.2.0.10 \
  --service-cidr 10.2.0.0/24 \
  --generate-ssh-keys
```

Notice how all the IP addresses are explicitly provided for Service resources, Pods, nodes, and Docker bridges. These are non-overlapping IP address ranges. Also, notice that Azure is used as a network plugin.

So far, you have learned how to deploy a solution and manage the networking of an AKS cluster. Security is another important factor that needs to be addressed. In the next section, we will be focusing on access and identity options for AKS.

Access and identity for AKS

Kubernetes clusters can be secured in multiple ways.

The service account is one of the primary user types in Kubernetes. The Kubernetes API manages the service account. Authorized Pods can communicate with the API server using the credentials of service accounts, which are stored as Kubernetes Secrets. Kubernetes does not have any data store or identity provider of its own. It delegates the responsibility of authentication to external software. It provides an authentication plugin that checks for the given credentials and maps them to available groups. If the authentication is successful, the request passes to another set of authorization plugins to check the permission levels of the user on the cluster, as well as the namespace-scoped resources.

For Azure, the best security integration would be to use Azure AD. Using Azure AD, you can also bring your on-premises identities to AKS to provide centralized management of accounts and security. The basic workflow of Azure AD integration is shown in Figure 14.19:

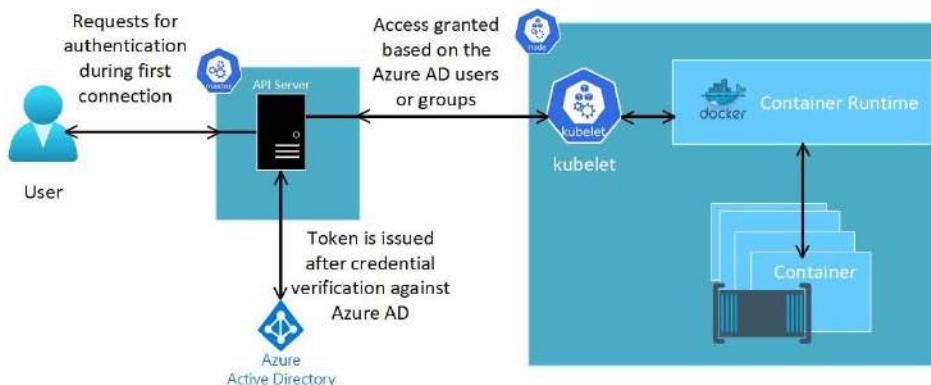


Figure 14.19: Basic workflow of Azure AD integration

Users or groups can be granted access to resources within a namespace or across a cluster. In the previous section, we used the `az aks get-credential` command to get the credentials and the kubectl configuration context. When the user tries to interact with kubectl, they are prompted to sign in using their Azure AD credentials. Azure AD validates the credentials and a token is issued for the user. Based on the access level they have, they can access the resources in the cluster or the namespace.

Additionally, you can leverage Azure **Role-Based Access Control (RBAC)** to limit access to the resources in the resource group.

In the next section, we will be discussing virtual kubelet, which is one of the quickest ways to scale out a cluster.

Virtual kubelet

Virtual kubelet is currently in preview and is managed by the CNCF organization. It is quite an innovative approach that AKS uses for scalability purposes. Virtual kubelet is deployed on the Kubernetes cluster as a Pod. The container running within the Pod uses the Kubernetes SDK to create a new node resource and represents itself to the entire cluster as a node. The cluster components, including the API server, scheduler, and controllers, think of it and treat it as a node and schedule Pods on it.

However, when a Pod is scheduled on this node that is masquerading as a node, it communicates to its backend components, known as providers, to create, delete, and update the Pods. One of the main providers on Azure is Azure Container Instances. Azure Batch can also be used as a provider. This means the containers are actually created on Container Instances or Azure Batch rather than on the cluster itself; however, they are managed by the cluster. The architecture of virtual kubelet is shown in Figure 14.20:

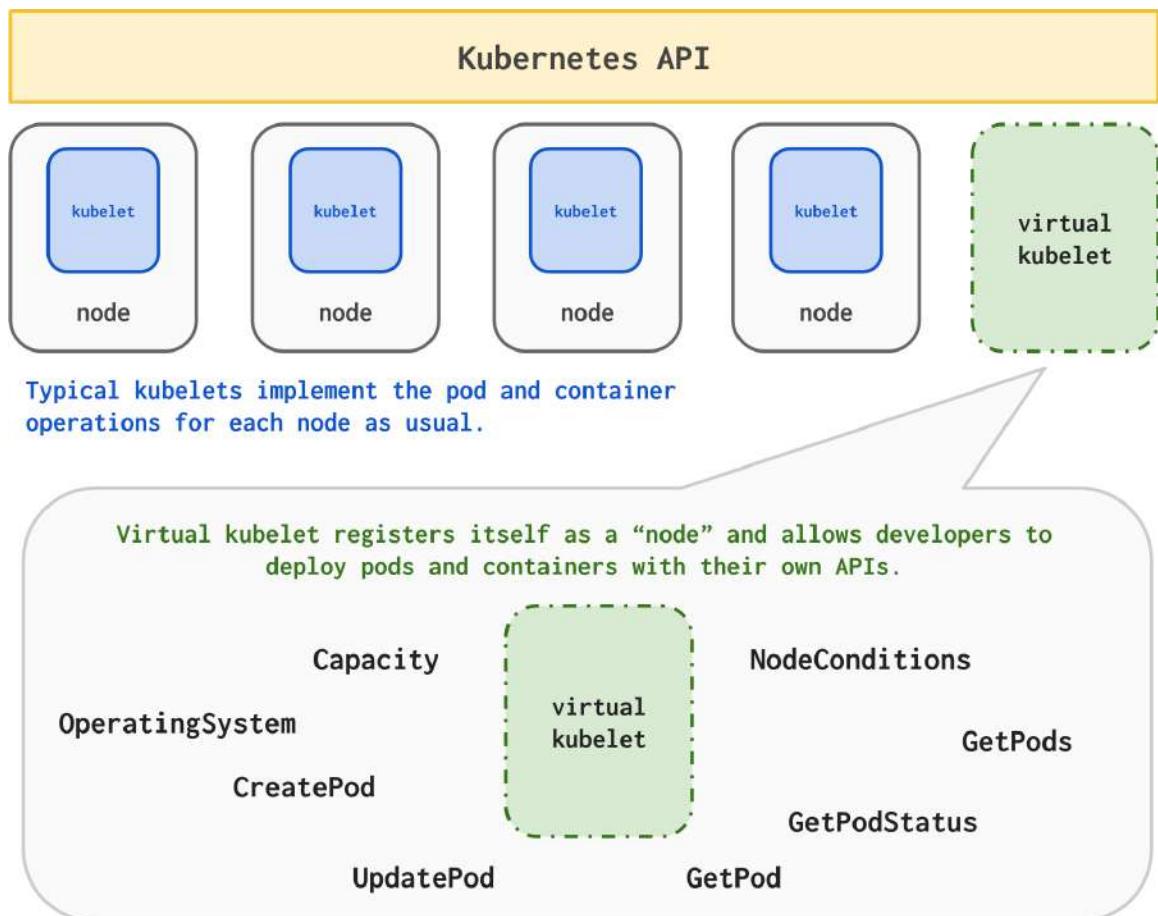


Figure 14.20: Virtual kubelet architecture

Notice that virtual kubelet is represented as a node within the cluster and can help in hosting and managing Pods, just like a normal kubelet would. However, virtual kubelet has one limitation; this is what we are going to discuss in the next section.

Virtual nodes

One of the limitations of virtual kubelet is that the Pods deployed on virtual kubelet providers are isolated and do not communicate with other Pods in the cluster. If there is a need for the Pods on these providers to talk to other Pods and nodes in the cluster and vice versa, then virtual nodes should be created. Virtual nodes are created on a different subnet on the same virtual network that is hosting Kubernetes cluster nodes, which can enable communication between Pods. Only the Linux operating system is supported, at the time of writing, for working with virtual nodes.

Virtual nodes give a perception of a node; however, the node does not exist. Anything scheduled on such a node actually gets created in Azure Container Instances. Virtual nodes are based on virtual kubelet but have the extra functionality of seamless to-and-fro communication between the cluster and Azure Container Instances.

While deploying Pods on virtual nodes, the Pod definition should contain an appropriate node selector to refer to virtual nodes, and also tolerations, as shown in next code snippet:

```
nodeSelector:  
    kubernetes.io/role: agent  
    beta.kubernetes.io/os: linux  
    type: virtual-kubelet  
  
tolerations:  
  - key: virtual-kubelet.io/provider  
    operator: Exists  
  - key: azure.com/aci  
    effect: NoSchedule
```

Here, the node selector is using the **type** property to refer to virtual kubelet and the **tolerations** property to inform Kubernetes that nodes with taints, **virtual-kubelet.io/provider**, should allow the Deployment of these Pods on them.

Summary

Kubernetes is the most widely used container orchestrator and works with different container and network runtimes. In this chapter, you learned about the basics of Kubernetes, its architecture, and some of the important infrastructure components, such as etcd, the API server, controller managers, and the scheduler, along with their purpose. Plus, we looked at important resources that can be deployed to manage applications, such as Pods, replication controllers, ReplicaSets, Deployments, and Services.

AKS provides a couple of different networking stacks—Azure CNI and Kubenet. They provide different strategies for assigning IP addresses to Pods. While Azure CNI provides IP addresses to Pods from the underlying subnet, Kubenet uses virtual IP addresses only.

We also covered some of the features provided exclusively by Azure, such as virtual nodes, and concepts around virtual kubelet. In the next chapter, we will learn about the provisioning and configuring resources with ARM templates.

15

Cross-subscription deployments using ARM templates

Azure Resource Manager (ARM) templates are the preferred mechanism for provisioning resources and configuring them on Azure.

ARM templates help to implement a relatively new paradigm known as **Infrastructure as Code (IaC)**. ARM templates convert the infrastructure and its configuration into code, which has numerous advantages. IaC brings a high level of consistency and predictability to deployments across environments. It also ensures that environments can be tested before going to production, and, finally, it gives a high level of confidence in the deployment process, maintenance, and governance.

The following topics will be covered in this chapter:

- ARM templates
- Deploying resource groups with ARM templates
- Deploying resources across subscriptions and resource groups
- Deploying cross-subscription and resource group deployments using linked templates
- Creating ARM templates for PaaS, data, and IaaS solutions

ARM templates

A prominent advantage of IaC is that it can be version controlled. It can also be reused across environments, which provides a high degree of consistency and predictability in deployments, and ensures that the impact and result of deploying an ARM template is the same no matter the number of times the template is deployed. This feature is known as **idempotency**.

ARM templates debuted with the introduction of the ARM specification and have been getting richer in features and growing in maturity since then. It's important to understand that there's generally a feature gap of a few weeks to a couple of months between the actual resource configuration and the availability of the configuration in ARM templates.

Each resource has its own configuration. This configuration can be affected in a multitude of ways, including using Azure PowerShell, the Azure CLI, Azure SDKs, REST APIs, and ARM templates.

Each of these techniques has its own development and release life cycle, which is different from the actual resource development. Let's try to understand this with the help of an example.

The Azure Databricks resource has its own cadence and development life cycle. The consumers of this resource have their own development life cycle, in turn, which is different from the actual resource development. If Databricks gets its first release on December 31, the Azure PowerShell cmdlets for it might not be available on the same date and might even be released on January 31 of the next year; similarly, the availability of these features in the REST API and ARM templates might be around January 15.

ARM templates are JSON-based documents that, when executed, invoke a REST API on the Azure management plane and submit the entire document to it. The REST API has its own development life cycle, and the JSON schema for the resource has its own life cycle too.

This means the development of a feature within a resource needs to happen in at least three different components before they can be consumed from ARM templates. These include:

- The resource itself
- The REST API for the resource
- The ARM template resource schema

Each resource in the ARM template has the **apiVersion** property. This property helps to decide the REST API version that should be used to provision and deploy the resource. *Figure 15.1* shows the flow of requests from the ARM template to resource APIs that are responsible for the creation, updating, and deletion of resources:



Figure 15.1: Request flow

A resource configuration, such as a storage account in an ARM template, looks as follows:

```
{  
  "type": "Microsoft.Storage/storageAccounts",  
  "apiVersion": "2019-04-01",  
  "name": "[variables('storage2')]",  
  "location": "[resourceGroup().location]",  
  "kind": "Storage",  
  "sku": {  
    "name": "Standard_LRS"  
  }  
}
```

In the preceding code, the availability of this schema for defining **sku** is based on the development of the ARM template schema. The availability of the REST API and its version number is determined by **apiVersion**, which happens to be **2019-04-01**. The actual resource is determined by the **type** property, which has the following two parts:

- **Resource-provider namespace:** Resources in Azure are hosted within namespaces and related resources are hosted within the same namespace.
- **Resource type:** Resources are referenced using their type name.

In this case, the resource is identified by its provider name and type, which happens to be **Microsoft.Storage/storageaccounts**.

Previously, ARM templates expected resource groups to be available prior to deployment. They were also limited to deploying to a single resource group within a single subscription.

This meant that, until recently, an ARM template could deploy all resources within a single resource group. Azure ARM templates now have added functionality for deploying resources to multiple resource groups within the same subscription or multiple subscriptions simultaneously. It's now possible to create resource groups as part of ARM templates, which means it's now possible to deploy resources in multiple regions into different resource groups.

Why would we need to create resource groups from within ARM templates, and why would we need to have cross-subscription and resource group deployments simultaneously?

To appreciate the value of creating a resource group and cross-subscription deployments, we need to understand how deployments were carried out prior to these features being available.

To deploy an ARM template, a resource group is a prerequisite. Resource groups should be created prior to the deployment of a template. Developers use PowerShell, the Azure CLI, or the REST API to create resource groups and then initiate the deployment of ARM templates. This means that any end-to-end deployment consists of multiple steps. The first step is to provision the resource group and the next step is the deployment of the ARM template to this newly created resource group. These steps could be executed using a single PowerShell script or individual steps from the PowerShell command line. The PowerShell script should be complete with regard to code related to exception handling, taking care of edge cases, and ensuring that there are no bugs in it before it can be said to be enterprise-ready. It is important to note that resource groups can be deleted from Azure, and the next time the script runs, they might be expected to be available. It would fail because it might assume that the resource group exists. In short, the deployment of the ARM template to a resource group should be an atomic step rather than multiple steps.

Compare this with the ability to create resource groups and their constituent resources together within the same ARM templates. Whenever you deploy the template, it ensures that the resource groups are created if they don't yet exist and continues to deploy resources to them after creation.

Let's also look at how these new features can help to remove some of the technical constraints related to disaster recovery sites.

Prior to these features, if you had to deploy a solution that was designed with disaster recovery in mind, there were two separate deployments: one deployment for the primary region and another deployment for the secondary region. For example, if you were deploying an ASP.NET MVC application using App Service, you would create an app service and configure it for the primary region, and then you would conduct another deployment with the same template to another region using a different **parameters** file. When deploying another set of resources in another region, as mentioned before, the parameters used with the template should be different to reflect the differences between the two environments. The parameters would include changes such as a SQL connection string, domain and IP addresses, and other configuration items unique to an environment.

With the availability of cross-subscription and resource group deployment, it's possible to create the disaster recovery site at the same time as the primary site. This eliminates two deployments and ensures that the same configuration can be used on multiple sites.

Deploying resource groups with ARM templates

In this section, an ARM template will be authored and deployed, which will create a couple of resource groups within the same subscription.

To use PowerShell to deploy templates that contain resource groups and cross-subscription resources, the latest version of PowerShell should be used. At the time of writing, Azure module version 3.3.0 is being used:

```
PS C:\WINDOWS\system32> get-module -Name az

ModuleType Version      Name
----- 3.3.0        az
```

Figure 15.2: Verifying the latest Azure module version

If the latest Azure module is not installed, it can be installed using the following command:

```
install-module -Name az -Force
```

It's time to create an ARM template that will create multiple resource groups within the same subscription. The code for the ARM template is as follows:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/
deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "resourceGroupInfo": {
      "type": "array"
    },
    "multiLocation": {
      "type": "array"
    }
  },
  "resources": [
    {
      "type": "Microsoft.Resources/resourceGroups",
      "location": "[parameters('multiLocation')[copyIndex()]]",
      "name": "[parameters('resourceGroupInfo')[copyIndex()]]",
      "apiVersion": "2019-10-01",
      "copy": {
        "name": "allResourceGroups",
        "count": "[length(parameters('resourceGroupInfo'))]"
      },
      "properties": {}
    }
  ],
  "outputs": {}
}
```

The first section of the code is about parameters that the ARM templates expect. These are mandatory parameters, and anybody deploying these templates should provide values for them. Array values must be provided for both the parameters.

The second major section is the **resources** JSON array, which can contain multiple resources. In this example, we are creating resource groups, so it is declared within the **resources** section. Resource groups are getting provisioned in a loop because of the use of the **copy** element. The **copy** element ensures that the resource is run a specified number of times and creates a new resource in every iteration. If we send two values for the **resourceGroupInfo** array parameter, the length of the array would be two, and the **copy** element will ensure that the **resourceGroup** resource is executed twice.

All resource names within a template should be unique for a resource type. The **copyIndex** function is used to assign the current iteration number to the overall name of the resource and make it unique. Also, we want the resource groups to be created in different regions using distinct region names sent as parameters. The assignment of a name and location for each resource group is done using the **copyIndex** function.

The code for the **parameters** file is shown next. This code is pretty straightforward and provides array values to the two parameters expected by the previous template. The values in this file should be changed for all parameters according to your environment:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/
deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "resourceGroupInfo": {
            "value": [ "firstResourceGroup", "SeocndResourceGroup" ]
        },
        "multiLocation": {
            "value": [
                "West Europe",
                "East US"
            ]
        }
    }
}
```

Deploying ARM templates

To deploy this template using PowerShell, log in to Azure with valid credentials using the following command:

```
Login-AzAccount
```

The valid credentials could be a user account or a service principal. Then, use a newly released **New-AzDeployment** cmdlet to deploy the template. The deployment script is available in the **multipleResourceGroups.ps1** file:

```
New-AzDeployment -Location "West Europe" -TemplateFile "c:\users\rites\source\repos\CrossSubscription\CrossSubscription\multipleResourceGroups.json" -TemplateParameterFile "c:\users\rites\source\repos\CrossSubscription\CrossSubscription\multipleResourceGroups.parameters.json" -Verbose
```

It's important to understand that the **New-AzResourceGroupDeployment** cmdlet can't be used here because the scope of the **New-AzResourceGroupDeployment** cmdlet is a resource group and it expects a resource group to be available as a prerequisite. For deploying resources at the subscription level, Azure had released a new cmdlet that can work above the resource group scope. The new cmdlet, **new-AzDeployment**, works at the subscription level. It is also possible to have a deployment at the management group level. Management groups are at a higher level than subscriptions using the **New-AzManagementGroupDeployment** cmdlet.

Deployment of templates using Azure CLI

The same template can also be deployed using the Azure CLI. Here are the steps to deploy it using the Azure CLI:

1. Use the latest version of the Azure CLI to create resource groups using the ARM template. At the time of writing, version 2.0.75 was used for deployment, as shown here:

```
C:\Users\Ritesh>az --version
azure-cli          2.0.75 *
command-modules-nspkg    2.0.3
core                2.0.75 *
nspkg               3.0.4
telemetry           1.0.4

Extensions:
aks-preview         0.4.18
azure-devops        0.16.0

Python location 'C:\Program Files (x86)\Microsoft SDKs\Azure\CLI2\python.exe'
Extensions directory 'C:\Users\Ritesh\.azure\cliextensions'

Python (Windows) 3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 02:47:15) [MSC v.1900 32 bit (Intel)]
Legal docs and information: aka.ms/AzureCliLegal
```

Figure 15.3: Checking the version of the Azure CLI

2. Log in to Azure using the following command and select the right subscription for use:

```
az login
```

3. If the login has access to multiple subscriptions, select the appropriate subscription using the following command:

```
az account set -subscription xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

4. Execute the deployment using the following command. The deployment script is available in the **multipleResourceGroupsCLI.txt** file:

```
C:\Users\Ritesh>az deployment create--location westus--template-file "C:\users\rites\source\repos\CrossSubscription\CrossSubscription\azuredeploy.json--parameters @"C:\users\rites\source\repos\CrossSubscription\CrossSubscription\azuredeploy.parameters.json"--verbose
```

Once the command is executed, the resources defined within the ARM template should be reflected on the Azure portal.

Deploying resources across subscriptions and resource groups

In the last section, resource groups were created as part of ARM templates. Another feature in Azure is the provision of resources into multiple subscriptions simultaneously from a single deployment using a single ARM template. In this section, we will provide a new storage account to two different subscriptions and resource groups. The person deploying the ARM template would select one of the subscriptions as the base subscription, using which they would initiate the deployment and provision the storage account into the current and another subscription. The prerequisite for deploying this template is that the person doing the deployment should have access to at least two subscriptions and that they have contributor rights on these subscriptions. The code listing is shown here and is available in the **CrossSubscriptionStorageAccount.json** file within the accompanying code:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/  
deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "storagePrefix1": {  
            "type": "string",  
            "defaultValue": "st01"  
        ...  
        "type": "string",  
        "defaultValue": "rg01"  
    },  
    "remoteSub": {  
        "type": "string",  
        "defaultValue": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"  
    }  
...  
        }  
    }  
],  
    "outputs": {}  
}  
}  
}  
]  
,"outputs": {}  
}
```

It is important to note that the names of the resource group used within the code should already be available in the respective subscriptions. The code will throw an error if the resource groups are not available. Moreover, the names of the resource group should precisely match those in the ARM template.

The code for deploying this template is shown next. In this case, we use **New-AzResourceGroupDeployment**, because the scope of the deployment is a resource group. The deployment script is available in the **CrossSubscriptionStorageAccount.ps1** file within the code bundle:

```
New-AzResourceGroupDeployment -TemplateFile "<< path to your  
CrossSubscriptionStorageAccount.json file >>" -ResourceGroupName "<<provide  
your base subscription resource group name>>" -storagePrefix1 <<provide prefix  
for first storage account>> -storagePrefix2 <<provide prefix for first storage  
account>> -verbose
```

Once the command is executed, the resources defined within the ARM template should be reflected in the Azure portal.

Another example of cross-subscription and resource group deployments

In this section, we create two storage accounts in two different subscriptions, resource groups, and regions from one ARM template and a single deployment. We will use the nested templates approach along with the **copy** element to provide different names and locations to these resource groups in different subscriptions.

However, before we can execute the next set of ARM templates, an Azure Key Vault instance should be provisioned as a prerequisite and a secret should be added to it. This is because the names of the storage accounts are retrieved from Azure Key Vault and passed as parameters to ARM templates to provision the storage account.

To provision Azure Key Vault using Azure PowerShell, the next set of commands can be executed. The code for the following commands is available in the **CreateKeyVaultandSetSecret.ps1** file:

```
New-AzResourceGroup -Location <>replace with location of your key vault>>
-Name <>replace with name of your resource group for key vault>> -verbose
New-AzureRmKeyVault -Name <>replace with name of your key vault>>
-ResourceGroupName <>replace with name of your resource group for
key vault>> -Location <>replace with location of your key vault>>
-EnabledForDeployment -EnabledForTemplateDeployment -EnabledForDiskEncryption
-EnableSoftDelete -EnablePurgeProtection -Sku Standard -Verbose
```

You should note that the **ResourceID** value should be noted from the result of the **New-AzKeyVault** cmdlet. This value will need to be replaced in the **parameters** file. See **Figure 15.4** for details:

```
VERBOSE: Performing the operation "Create key vault" on target "testkeyvaultbook".
Vault Name : testkeyvaultbook
Resource Group Name : rg01
Location : west europe
Resource ID : /subscriptions/.../resourceGroups/rg01/providers/Microsoft.KeyVault/vaults/testkeyvaultbook
Vault URL : https://testkeyvaultbook.vault.azure.net/
Tenant ID : ...
SKU : Standard
Enabled For Deployment? : True
Enabled For Template Deployment? : True
Enabled For Disk Encryption? : True
Soft Delete Enabled? : True
Access Policies : ...
    Tenant ID : ...
    Object ID : ...
    Application ID : ...
    Display Name : ...
    Permissions to Keys : ...
    Permissions to Secrets : ...
    Permissions to Certificates : ...
    SetIssuers, Recover, Backup, Restore : ...
    Permissions to (Key Vault Managed) Storage : ...
        : delete, deletesas, get, getsas, list, listsas, regeneratekey, set, setse, update
Network Rule Set : ...
    Default Action : ...
    Bypass : ...
    IP Rules : ...
    Virtual Network Rules : ...

```

Figure 15.4: Creating a Key Vault instance

Execute the following command to add a new secret to the newly created Azure Key Vault instance:

```
Set-AzKeyVaultSecret -VaultName <><replace with name of your key vault>> -Name <><replace with name of yoursecret>> -SecretValue $(ConvertTo-SecureString -String <><replace with value of your secret>> -AsPlainText -Force ) -Verbose
```

The code listing is available in the **CrossSubscriptionNestedStorageAccount.json** file within the code bundle:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "hostingPlanNames": {
      "type": "array",
      "minLength": 1
    },
    ...
    "type": "Microsoft.Resources/deployments",
    "name": "deployment01",
    "apiVersion": "2019-10-01",
    "subscriptionId": "[parameters('subscriptions')[copyIndex()]]",
    "resourceGroup": "[parameters('resourceGroups')[copyIndex()]]",
    "copy": {
      "count": "[length(parameters('hostingPlanNames'))]",
      "name": "mywebsites",           "mode": "Parallel"
    },
    ...
    "kind": "Storage",
    "properties": {
    }
  }
}
...
...
```

Here's the code for the **parameters** file. It is available in the **CrossSubscriptionNestedStorageAccount.parameters.json** file:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/  
deploymentParameters.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "hostingPlanNames": {  
            ...  
            "storageKey": {  
                "reference": {  
                    "keyVault": { "id": "<>replace it with the value of Key vault  
ResourceId noted before>>" },  
                    "secretName": "<>replace with the name of the secret available in  
Key vault>>"  
                }  
            }  
        }  
    }  
}
```

Here's the PowerShell code for deploying the previous template. The deployment script is available in the **CrossSubscriptionNestedStorageAccount.ps1** file:

```
New-AzResourceGroupDeployment -TemplateFile "c:\users\rites\source\repos\  
CrossSubscription\CrossSubscription\CrossSubscriptionNestedStorageAccount.  
json" -ResourceGroupName rg01 -TemplateParameterFile "c:\  
users\rites\source\repos\CrossSubscription\CrossSubscription\  
CrossSubscriptionNestedStorageAccount.parameters.json" -Verbose
```

Once the command gets executed, the resources defined within the ARM template should be reflected in the Azure portal.

Deploying cross-subscription and resource group deployments using linked templates

The previous example used nested templates to deploy to multiple subscriptions and resource groups. In the next example, we will deploy multiple App Service plans in separate subscriptions and resource groups using linked templates. The linked templates are stored in Azure Blob storage, which is protected using policies. This means that only the holder of the storage account key or a valid shared access signature can access this template. The access key is stored in Azure Key Vault and is accessed from the **parameters** file using references under the **storageKey** element. You should upload the **website.json** file to a container in Azure Blob storage. The **website.json** file is a linked template responsible for provisioning an App Service plan and an app service. The file is protected using the **Private (no anonymous access)** policy, as shown in Figure 15.5. A privacy policy ensures that anonymous access is not allowed. For this instance, we have created a container named **armtemplates** and set it with a private policy:

The screenshot shows the Azure portal interface for managing a blob container. On the left, there's a sidebar with links like Home, Resource groups, and Storage accounts. The main area shows a storage account named 'st02gwidc xm5suwe - Blobs'. Under 'Containers', there's a list with one item: 'armtemplates'. To the right, a modal window titled 'Access policy' is open. It shows a table with one row for 'armtemplates'. The 'Public access level' dropdown is set to 'Private (no anonymous access)'. There's also a 'Save' button at the top of the modal.

Figure 15.5: Setting a private policy for the container

This file can only be accessed using the **Shared Access Signature (SAS)** keys. The SAS keys can be generated from the Azure portal for a storage account using the **Shared access signature** item in the left menu shown in Figure 15.6. You should click on the **Generate SAS and connection string** button to generate the SAS token. It is to be noted that an SAS token is displayed once and not stored within Azure. So, copy it and store it somewhere so that it can be uploaded to Azure Key Vault. Figure 15.6 shows the generation of the SAS token:

The screenshot shows the Azure Storage account settings for 'st02gvwlcdxm5suwe'. The left sidebar lists various account management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Storage Explorer (preview), Settings, Access keys, Geo-replication, CORS, Configuration, Encryption, Shared access signature (selected), Firewalls and virtual networks, Advanced Threat Protection, Properties, Locks, and Automation script. The main pane displays the 'Shared access signature' configuration. It includes sections for Allowed services (Blob, File, Queue, Table checked), Allowed resource types (Service, Container, Object checked), Allowed permissions (Read, Write, Delete, List, Add, Create, Update, Process checked), Start and expiry date/time (Start: 2019-01-30, End: 2019-01-30, Timezone: (UTC-05:00) --- Current Time Zone ---), Allowed IP addresses (example: 168.1.5.65 or 168.1.5.65-168.1.5.70), and Allowed protocols (HTTPS only selected). A 'Generate SAS and connection string' button is at the bottom.

Figure 15.6: Generating an SAS token in the Azure portal

We will use the same Key Vault instance that was created in the previous section. We just have to ensure that there are two secrets available within the Key Vault instance. The first secret is **StorageName** and the other one is **StorageKey**. The commands to create these secrets in the Key Vault instance are as follows:

```
Set-AzKeyVaultSecret -VaultName "testkeyvaultbook" -Name "storageName"
-SecretValue $(ConvertTo-SecureString -String "uniquename" -AsPlainText
-Force ) -Verbose
```

```
Set-AzKeyVaultSecret -VaultName "testkeyvaultbook" -Name "storageKey"
-SecretValue $(ConvertTo-SecureString -String "?sv=2020-03-28&ss=bfqt&srt=sc
o&sp=rwdlacup&se=2020-03-30T21:51:03Z&st=2020-03-30T14:51:03Z&spr=https&sig=
gTynGhj20er6pDl7Ab%2Bpc29W03%2BJhvi%2Bff%2F6rHYWp4g%3D" -AsPlainText -Force
) -Verbose
```

You are advised to change the names of the Key Vault instance and the secret key value based on your storage account.

After ensuring that the Key Vault instance has the necessary secrets, the ARM template file code can be used to deploy the nested templates across subscriptions and resource groups.

The ARM template code is available in the **CrossSubscriptionLinkedStorageAccount.json** file and is also shown here. You are advised to change the value of the **templateUrl** variable within this file. It should be updated with a valid Azure Blob storage file location:

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/  
deploymentTemplate.json#",  
  "contentVersion": "1.0.0.0",  
  "parameters": {  
    "hostingPlanNames": {  
      "type": "array",  
      "minLength": 1  
    },  
    ...  
    "type": "Microsoft.Resources/deployments",  
    "name": "fsdfsdf",  
    "apiVersion": "2019-10-01",  
    "subscriptionId": "[parameters('subscriptions')[copyIndex()]]",  
    "resourceGroup": "[parameters('resourceGroups')[copyIndex()]]",  
    "copy": {  
      "count": "[length(parameters('hostingPlanNames'))]",  
      "name": "mywebsites",  
      "mode": "Parallel"  
    },  
    ...  
  }  
}
```

The code for the **parameters** file is shown next. You are advised to change the values of the parameters, including the **resourceid** of the Key Vault instance and the secret name. The names of app services should be unique, or the template will fail to deploy. The code for the **parameters** file is available in the **CrossSubscriptionLinkedStorageAccount.parameters.json** code file:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/  
deploymentParameters.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "hostingPlanNames": {  
            "value": [ "firstappservice", "secondappservice" ]  
        },  
        ...  
        "storageKey": {  
            "reference": {  
                "keyVault": { "id": "/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-  
xxxxxxxxxxxx/resourceGroups/keyvaluedemo/providers/Microsoft.KeyVault/  
vaults/forsqlvault1" },  
                "secretName": "storageKey"  
            }  
        }  
    }  
}
```

Here's the command to deploy the template. The deployment script is available in the **CrossSubscriptionLinkedStorageAccount.ps1** file:

```
New-AzureRmResourceGroupDeployment -TemplateFile "c:\users\rites\source\repos\CrossSubscription\CrossSubscription\CrossSubscriptionLinkedStorageAccount.json" -ResourceGroupName <>replace  
with the base subscription resource group name >> -TemplateParameterFile  
"c:\users\rites\source\repos\CrossSubscription\CrossSubscription\CrossSubscriptionLinkedStorageAccount.parameters.json" -Verbose
```

Once the command gets executed, the resources defined within the ARM template should be reflected in the Azure portal.

Now that you know how to provision resources across resource groups and subscriptions, we will look at some of the solutions that can be created using ARM templates.

Virtual machine solutions using ARM templates

Infrastructure as a service (IaaS) resources and solutions can be deployed and configured using ARM templates. The major resources related to IaaS are virtual machine resources.

Creating a virtual machine resource is dependent on multiple other resources in Azure. Some of the resources that are needed to create a virtual machine include:

- A storage account or a managed disk for hosting the operating system and data disk
- A virtual network along with subnets
- A network interface card

There are other resources that are optional, including:

- Azure Load Balancer
- Network security groups
- Public IP address
- Route tables and more

This section will deal with the process of creating virtual machines using ARM templates. As mentioned before in this section, we need to create a few resources, upon which the virtual machine resource will depend, prior to creating the virtual machine resource itself.

It is important to note that it is not always necessary to create the dependent resources. They should be created only if they do not exist already. If they already are available within the Azure subscription, the virtual machine resource can be provisioned by referencing those dependent resources.

The template is dependent on a few parameters that should be supplied to it at the time of executing the template. These variables relate to the location of the resources and some of their configuration values. These values are taken from parameters because they might change from one deployment to another, so using parameters helps keep the template generic.

The first step is to create a storage account, as shown in the following code:

```
{  
  "type": "Microsoft.Storage/storageAccounts",  
  "name": "[variables('storageAccountName')]",  
  "apiVersion": "2019-04-01",  
  "location": "[parameters('location')]",  
  "sku": {  
    "name": "Standard_LRS"  
  },  
  "kind": "Storage",  
  "properties": {}  
},
```

After creating a storage account, a virtual network should be defined within the ARM template. It is important to note that there is no dependency between a storage account and a virtual network. They can be created in parallel. The virtual network resource has a subnet as its sub-resource. These are both configured with their IP ranges; the subnet typically has a smaller range than the virtual network IP range:

```
{  
  "apiVersion": "2019-09-01",  
  "type": "Microsoft.Network/virtualNetworks",  
  "name": "[variables('virtualNetworkName')]",  
  "location": "[parameters('location')]",  
  "properties": {  
    "addressSpace": {  
      "addressPrefixes": [  
        "[variables('addressPrefix')]"  
      ]  
    },  
    "subnets": [  
      {  
        "name": "[variables('subnetName')]",  
        "properties": {  
          "addressPrefix": "[variables('subnetPrefix')]"  
        }  
      }  
    ]  
  }  
},
```

```
    }
]
},
},
```

If the virtual machine needs to be accessed over the public internet, a public IP address can also be created, as shown in the following code. Again, it is a completely independent resource and can be created in parallel with the storage account and virtual network:

```
{
  "apiVersion": "2019-11-01",
  "type": "Microsoft.Network/publicIPAddresses",
  "name": "[variables('publicIPAddressName')]",
  "location": "[parameters('location')]",
  "properties": {
    "publicIPAllocationMethod": "Dynamic",
    "dnsSettings": {
      "domainNameLabel": "[parameters('dnsLabelPrefix')]"
    }
  }
},
```

After creating the virtual network, storage account, and public IP address, a network interface can be created. A network interface is dependent on a virtual network and subnet resource. It can optionally also be associated with a public IP address as well. This is shown in the following code:

```
{
  "apiVersion": "2019-11-01",
  "type": "Microsoft.Network/networkInterfaces",
  "name": "[variables('nicName')]",
  "location": "[parameters('location')]",
  "dependsOn": [
```

```
    "[resourceId('Microsoft.Network/publicIPAddresses/',
variables('publicIPAddressName'))]",
    "[resourceId('Microsoft.Network/virtualNetworks/',
variables('virtualNetworkName'))]"
],
"properties": {
    "ipConfigurations": [
        {
            "name": "ipconfig1",
            "properties": {
                "privateIPAllocationMethod": "Dynamic",
                "publicIPAddress": {
                    "id": "[resourceId('Microsoft.Network/
publicIPAddresses',variables('publicIPAddressName'))]"
                },
                "subnet": {
                    "id": "[variables('subnetRef')]"
                }
            }
        }
    ]
},
},
```

It is important to note that both the public IP address and the subnet are referred to by their unique Azure identifiers.

After the creation of the network interface, we have all the resources that are needed to create a virtual machine. The next code block shows how to create a virtual machine using an ARM template. It has a dependency on the network card and storage account. This indirectly creates dependencies on the virtual network, subnet, and the public IP address.

For the virtual machine, we configure the mandatory resource configuration, including **type**, **apiVersion**, **location**, and **name**, along with any dependencies, as shown in the following code:

```
{  
  "apiVersion": "2019-07-01",  
  "type": "Microsoft.Compute/virtualMachines",  
  "name": "[variables('vmName')]",  
  "location": "[resourceGroup().location]",  
  "tags": {  
    "displayName": "VirtualMachine"  
  },  
  "dependsOn": [  
    "[concat('Microsoft.Storage/storageAccounts/',  
    variables('storageAccountName'))]",  
    "[concat('Microsoft.Network/networkInterfaces/', variables('nicName'))]"  
  ],  
  "properties": {  
    "hardwareProfile": { "vmSize": "[variables('vmSize')]" },  
    "availabilitySet": {  
      "id": "[resourceId('Microsoft.Compute/availabilitySets',  
      parameters('adAvailabilitySetName'))]"  
    },  
    "osProfile": {  
      "computerName": "[variables('vmName')]",  
      "adminUsername": "[parameters('adminUsername')]",  
      "adminPassword": "[parameters('adminPassword')]"  
    },  
    "storageProfile": {  
      "imageReference": {
```

```
"publisher": "[variables('imagePublisher')]",  
"offer": "[variables('imageOffer')]",  
"sku": "[parameters('windowsOSVersion')]",  
"version": "latest"  
},  
"osDisk": { "createOption": "FromImage" },  
"copy": [  
    {  
        "name": "dataDisks",  
        "count": 3,  
        "input": {  
            "lun": "[copyIndex('dataDisks')]",  
            "createOption": "Empty",  
            "diskSizeGB": "1023",  
            "name": "[concat(variables('vmName'), '-datadisk', copyIndex('dataDisks'))]"  
        }  
    }  
]  
},  
"networkProfile": {  
    "networkInterfaces": [  
        {  
            "id": "[resourceId('Microsoft.Network/networkInterfaces',  
variables('nicName'))]"  
        }  
    ]  
}  
}
```

In the preceding code, the virtual machine is configured with:

- A hardware profile—the size of the virtual machine.
- An OS profile—the name and credentials for logging in to the virtual machine.
- A storage profile—the storage account on which to store the **Virtual Hard Disk (VHD)** file for the virtual machine, including data disks.
- A network profile—the reference to the network interface card.

The next section will show an example of using ARM templates to provision a Platform as a Service solution.

PaaS solutions using ARM templates

Platform as a service (PaaS) resources and solutions can be deployed using ARM templates. One of the main resources related to PaaS is Azure Web Apps, and in this section, we will focus on creating web apps on Azure using ARM templates.

The template expects a few parameters to be supplied while executing it. The parameters needed are the SKU for the App Service plan, the Azure region hosting the resources, and the SKU capacity of the App Service plan.

There are a couple of variables declared within the template to make it generic and maintainable. The first one, **hostingPlanName**, is for the App Service plan name, and the next one, **webSiteName**, is for the app service itself.

There are at minimum two resources that should be declared and provisioned for a working web app in Azure. They are the following:

- The Azure App Service plan
- Azure App Service

The first step in creating a web app on Azure is defining the configuration for an Azure App Service plan. The following code defines a new App Service plan. It is important to note that the resource type is **Microsoft.Web/serverfarms**. Most of the configuration values of the plan, such as **location**, **name**, and **capacity**, come as parameters to the ARM template:

```
{  
  "apiVersion": "2019-08-01",  
  "name": "[variables('hostingPlanName')]",  
  "type": "Microsoft.Web/serverfarms",  
  "location": "[parameters('location')]",  
  "tags": {  
    "displayName": "HostingPlan"  
  },  
  "sku": {  
    "name": "[parameters('skuName')]",  
    "capacity": "[parameters('skuCapacity')]"  
  },  
  "properties": {  
    "name": "[variables('hostingPlanName')]"  
  }  
},
```

The next resource that should be provisioned after a plan is the app service itself. It is important that a dependency between both these resources is created such that a plan is already created before the app service itself is created:

```
{  
  "apiVersion": "2019-08-01",  
  "name": "[variables('webSiteName')]",  
  "type": "Microsoft.Web/sites",  
  "location": "[parameters('location')]",  
  "dependsOn": [  
    "[variables('hostingPlanName')]"  
  ],  
  "properties": {  
    "name": "[variables('webSiteName')]",  
    "serverFarmId": "[resourceId('Microsoft.Web/serverfarms',  
      variables('hostingPlanName'))]"  
  },  
  "resources": [  
  ]
```

```
{  
  "apiVersion": "2019-08-01",  
  "type": "config",  
  "name": "connectionstrings",  
  "dependsOn": [  
    "[variables('webSiteName')]"  
  ],  
  "properties": {  
    "DefaultConnection": {  
      "value": "[concat( 'sql connection string here')]",  
      "type": "SQLAzure"  
    }  
  }  
}  
]  
}
```

In the preceding code, a resource of type **Microsoft.Web/sites** is defined and it has a dependency on the plan. It is also using the App Service plan and is associated with it using **serverFarmId**. It further declares a connection string that can be used for connecting to SQL Server.

This section showed an example of creating a PaaS solution on Azure using an ARM template. Similarly, other PaaS solutions, including Azure Function apps, Kubernetes Service, and Service Fabric, among many others, can be created using ARM templates.

Data-related solutions using ARM templates

There are many resources in Azure related to data management and storage. Some of the important data-related resources include Azure SQL, Azure Cosmos DB, Azure Data Lake Storage, Data Lake Analytics, Azure Synapsis, Databricks, and Data Factory.

All of these resources can be provisioned and configured using an ARM template. In this section, we will create an ARM template to provision a Data Factory resource responsible for migrating data from Azure Blob storage to Azure SQL Database using stored procedures.

You will find the parameters file along with the template. These values might change from one deployment to another; we'll keep the template generic so that you can customize and use it easily with other deployments as well.

The entire code for this section can be found at <https://github.com/Azure/azure-quickstart-templates/blob/master/101-data-factory-blob-to-sql-copy-stored-proc>.

The first step is to declare the configuration for the data factory in the ARM template, as shown in the following code:

```
"name": "[variables('dataFactoryName')]",  
"apiVersion": "2018-06-01",  
"type": "Microsoft.DataFactory/datafactories",  
"location": "[parameters('location')]",
```

Each data factory has multiple linked services. These linked services act as connectors to get data into the data factory, or the data factory can send data to them. The following code listing creates a linked service for the Azure storage account from which the blobs will be read into the data factory, and another linked service for Azure SQL Database:

```
{  
  "type": "linkedservices",  
  "name": "[variables('storageLinkedServiceName')]"  
  "apiVersion": "2018-06-01",  
  "dependsOn": [  
    "[variables('dataFactoryName')]"  
  ],  
  "properties": {  
    "type": "AzureStorage",  
    "description": "Azure Storage Linked Service",  
    "typeProperties": {  
      "connectionString":  
        "[concat('DefaultEndpointsProtocol=https;AccountName=', parameters('storageAccountName'), ';AccountKey=', parameters('storageAccountKey'))]"  
    },  
    }  
  },  
  {  
    "type": "linkedservices",
```

```
"name": "[variables('sqlLinkedServiceName')]",  
"apiVersion": "2018-06-01",  
"dependsOn": [  
    "[variables('dataFactoryName')]"  
],  
"properties": {  
    "type": "AzureSqlDatabase",  
    "description": "Azure SQL linked service",  
    "typeProperties": {  
        "connectionString": "[concat('Data Source=tcp:', parameters('sqlServerName'),  
        '.database.windows.net,1433;Initial Catalog=', parameters('sqlDatabaseName'),  
        ';Integrated Security=False;User ID=', parameters('sqlUserId'), ', Password=',  
        parameters('sqlPassword'), ', Connect Timeout=30;Encrypt=True')]"  
    }  
},  
},
```

After linked services, it's time to define the datasets for Azure Data Factory. Datasets help in identifying the data that should be read and placed in the data factory. They could also represent the temporary data that needs to be stored by the Data Factory during the transformation, or even the destination location where the data will be written. The next code block creates three datasets—one for each of the aspects of datasets that were just mentioned.

The read dataset is shown in the following code block:

```
{  
    "type": "datasets",  
    "name": "[variables('storageDataset')]",  
    "dependsOn": [  
        "[variables('dataFactoryName')]",  
        "[variables('storageLinkedServiceName')]"  
    ],  
    "apiVersion": "2018-06-01",  
    "properties": {  
        "type": "AzureBlob",  
        "linkedServiceName": "[variables('storageLinkedServiceName')]",
```

```
"typeProperties": {  
    "folderPath": "[concat(parameters('sourceBlobContainer'), '/')]",  
    "fileName": "[parameters('sourceBlobName')]",  
    "format": {  
        "type": "TextFormat"  
    },  
    "availability": {  
        "frequency": "Hour",  
        "interval": 1  
    },  
    "external": true  
},  
},
```

The intermediate dataset is shown in the following lines of code:

```
{  
    "type": "datasets",  
    "name": "[variables('intermediateDataset')]",  
    "dependsOn": [  
        "[variables('dataFactoryName')]",  
        "[variables('sqlLinkedServiceName')]"  
    ],  
    "apiVersion": "2018-06-01",  
    "properties": {  
        "type": "AzureSqlTable",  
        "linkedServiceName": "[variables('sqlLinkedServiceName')]",  
        "typeProperties": {  
            "tableName": "[variables('intermediateDataset')]"  
        },  
        "availability": {  
            "frequency": "Hour",  
            "interval": 1  
        }  
    }  
}
```

```
    }
}
},
```

Finally, the dataset used for the destination is shown here:

```
{
  "type": "datasets",
  "name": "[variables('sqlDataset')]",
  "dependsOn": [
    "[variables('dataFactoryName')]",
    "[variables('sqlLinkedServiceName')]"
  ],
  "apiVersion": "2018-06-01",
  "properties": {
    "type": "AzureSqlTable",
    "linkedServiceName": "[variables('sqlLinkedServiceName')]",
    "typeProperties": {
      "tableName": "[parameters('sqlTargetTable')]"
    },
    "availability": {
      "frequency": "Hour",
      "interval": 1
    }
  }
},
```

Finally, we need a pipeline in Data Factory that can bring together all the datasets and linked services, and help in creating extract-transform-load data solutions. A pipeline consists of multiple activities, each fulfilling a particular task. All these activities can be defined within the ARM template, as you'll see now. The first activity copies the blobs in the storage account to an intermediate SQL Server, as shown in the following code:

```
{
  "type": "dataPipelines",
  "name": "[variables('pipelineName')]",
  "dependsOn": [
```

```
"[variables('dataFactoryName')]",
"[variables('storageLinkedServiceName')]",
"[variables('sqlLinkedServiceName')]",
"[variables('storageDataset')]",
"[variables('sqlDataset')]"
],
"apiVersion": "2018-06-01",
"properties": {
"description": "Copies data from Azure Blob to Sql DB while invoking stored procedure",
"activities": [
{
"name": "BlobtoSqlTableCopyActivity",
"type": "Copy",
"typeProperties": {
"source": {
"type": "BlobSource"
},
"sink": {
"type": "SqlSink",
"writeBatchSize": 0,
"writeBatchTimeout": "00:00:00"
}
},
"inputs": [
{
"name": "[variables('storageDataset')]"
}
],
"outputs": [
{
"name": "[variables('intermediateDataset')]"
}
```

```
        }
    ],
},
{
"name": "SqlTabletoSqlDbSprocActivity",
"type": "SqlServerStoredProcedure",
"inputs": [
    {
        "name": "[variables('intermediateDataset')]"
    }
],
"outputs": [
    {
        "name": "[variables('sqlDataset')]"
    }
],
"typeProperties": {
    "storedProcedureName": "[parameters('sqlWriterStoredProcName')]"
},
"scheduler": {
    "frequency": "Hour",
    "interval": 1
},
"policy": {
    "timeout": "02:00:00",
    "concurrency": 1,
    "executionPriorityOrder": "NewestFirst",
    "retry": 3
}
},
],
],
```

```
"start": "2020-10-01T00:00:00Z",
"end": "2020-10-02T00:00:00Z"
}
}
]
}
```

The last activity copies data from the intermediate dataset to the final destination dataset.

There are also start and end times during which the pipeline should be running.

This section focused on creating an ARM template for a data-related solution. In the next section, we will deal with ARM templates for creating datacenters on Azure with Active Directory and DNS.

Creating an IaaS solution on Azure with Active Directory and DNS

Creating an IaaS solution on Azure means creating multiple virtual machines, promoting a virtual machine to be a domain controller, and making other virtual machines join the domain controller as domain-joined nodes. It also means installing a DNS server for name resolution and, optionally, a jump server for accessing these virtual machines securely.

The template creates an Active Directory forest on the virtual machines. It creates multiple virtual machines based on the parameters supplied.

The template creates:

- A couple of availability sets
- A virtual network
- Network security groups to define the allowed and disallowed ports and IP addresses

The template then does the following:

- Provisions one or two domains. The root domain is created by default; the child domain is optional
- Provisions two domain controllers per domain
- Executes the desired state configuration scripts to promote a virtual machine to be a domain controller

We can create multiple virtual machines using the approach discussed in the *Virtual machine solutions using ARM templates* section. However, these virtual machines should be part of an availability set if they need to be highly available. It is to be noted that availability sets provide 99.95% availability for applications deployed on these virtual machines, while Availability Zones provide 99.99% availability.

An availability set can be configured as shown in the following code:

```
{  
  "name": "[variables('adAvailabilitySetNameRoot')]",  
  "type": "Microsoft.Compute/availabilitySets",  
  "apiVersion": "2019-07-01",  
  "location": "[parameters('location')]",  
  "sku": {  
    "name": "Aligned"  
  },  
  "properties": {  
    "PlatformUpdateDomainCount": 3,  
    "PlatformFaultDomainCount": 2  
  }  
},
```

Once the availability set is created, an additional profile should be added to the virtual machine configuration to associate the virtual machine with the availability set, as shown in the following code:

```
"availabilitySet" : {  
  "id": "[resourceId('Microsoft.Compute/availabilitySets',  
  parameters('adAvailabilitySetName'))]"  
}
```

You should note that availability sets are mandatory in order to use load balancers with virtual machines.

Another change needed in the virtual network configuration is adding DNS information, as shown in the following code:

```
{  
  "name": "[parameters('virtualNetworkName')]",  
  "type": "Microsoft.Network/virtualNetworks",  
  "location": "[parameters('location')]",  
  "apiVersion": "2019-09-01",  
  "properties": {  
    "addressSpace": {  
      "addressPrefixes": [  
        "[parameters('virtualNetworkAddressRange')]"  
      ]  
    },  
    "dhcpOptions": {  
      "dnsServers": "[parameters('DNSServerAddress')]"  
    },  
    "subnets": [  
      {  
        "name": "[parameters('subnetName')]",  
        "properties": {  
          "addressPrefix": "[parameters('subnetRange')]"  
        }  
      }  
    ]  
  }  
},
```

Finally, to convert a virtual machine into Active Directory, a PowerShell script or **desired state configuration (DSC)** script should be executed on the virtual machine. Even for joining other virtual machines to the domain, another set of scripts should be executed on those virtual machines.

Scripts can be executed on the virtual machine using the **CustomScriptExtension** resource, as shown in the following code:

```
{  
  "type": "Microsoft.Compute/virtualMachines/extensions",  
  "name": "[concat(parameters('adNextDCVMName'), '/PrepareNextDC')]",  
  "apiVersion": "2018-06-01",  
  "location": "[parameters('location')]",  
  "properties": {  
    "publisher": "Microsoft.Powershell",  
    "type": "DSC",  
    "typeHandlerVersion": "2.21",  
    "autoUpgradeMinorVersion": true,  
    "settings": {  
      "modulesURL": "[parameters('adNextDCConfigurationModulesURL')]",  
      "configurationFunction": "[parameters('adNextDCConfigurationFunction')]",  
      "properties": {  
        "domainName": "[parameters('domainName')]",  
        "DNSServer": "[parameters('DNSServer')]",  
        "DNSForwarder": "[parameters('DNSServer')]",  
        "adminCreds": {  
          "userName": "[parameters('adminUserName')]",  
          "password": "privateSettingsRef:adminPassword"  
        }  
      }  
    },  
    "protectedSettings": {  
      "items": {  
        "adminPassword": "[parameters('adminPassword')]"  
      }  
    }  
  },  
},
```

In this section, we created a datacenter on Azure using the IaaS paradigm. We created multiple virtual machines and converted one of them into domain controller, installed DNS, and assigned a domain to it. Now, other virtual machines on the network can be joined to this domain and they can form a complete datacenter on Azure.

Please refer to <https://github.com/Azure/azure-quickstart-templates/tree/master/301-create-ad-forest-with-subdomain> for the complete code listing for creating a datacenter on Azure.

Summary

The option to deploy resources using a single deployment to multiple subscriptions, resource groups, and regions provides enhanced abilities to deploy, reduce bugs in deployment, and access advanced benefits, such as creating disaster recovery sites and achieving high availability.

In this chapter, you saw how to create a few different kinds of solution using ARM templates. This included creating an infrastructure-based solution comprising virtual machines; a platform-based solution using Azure App Service; a data-related solution using the Data Factory resource (including its configuration); and a datacenter on Azure with virtual machines, Active Directory, and DNS installed on top of the virtual machine.

In the next chapter, we will focus on creating modular ARM templates, an essential skill for architects who really want to take their ARM templates to the next level. The chapter will also show you various ways to design ARM templates and create reusable and modular ARM templates.

16

ARM template modular design and implementation

We know that there are multiple ways to author an **Azure Resource Manager (ARM)** template. It is quite easy to author one that provisions all of the necessary resources in Azure using Visual Studio and Visual Studio Code. A single ARM template can consist of all the required resources for a solution on Azure. This single ARM template could be as small as a few resources, or it could be a larger one consisting of many resources.

While authoring a single template consisting of all resources is quite tempting, it is advisable to plan an ARM template implementation divided into multiple smaller ARM templates beforehand, so that future troubles related to them can be avoided.

In this chapter, we will look at how to write ARM templates in a modular way so that they can evolve over a period of time with minimal involvement in terms of changes and effort in testing and deployment.

However, before writing modular templates, it is best to understand the problems solved by writing them in a modular fashion.

The following topics will be covered in this chapter:

- Problems with a single template
- Understanding nested and linked deployment
- Linked templates
- Nested templates
- Free-flow configurations
- Known configurations

Now, let's explore the aforementioned topics in detail, which will help you to write modular templates using industry best practices.

Problems with the single template approach

On the surface, it might not sound like a single large template consisting of all resources will have problems, but there are issues that could arise in the future. Let's discuss the issues that might arise when using single large templates.

Reduced flexibility in changing templates

Using a single large template with all resources makes it difficult to change it in the future. With all dependencies, parameters, and variables in a single template, changing the template can take a considerable amount of time compared to smaller templates. The change could have an impact on other sections of the template, which might go unnoticed, as well as introducing bugs.

Troubleshooting large templates

Large templates are difficult to troubleshoot. This is a known fact. The larger the number of resources in a template, the more difficult it is to troubleshoot the template. A template deploys all the resources in it, and finding a bug involves deploying the template quite often. Developers would have reduced productivity while waiting for the completion of template deployment.

Also, deploying a single template is more time-consuming than deploying smaller templates. Developers have to wait for resources containing errors to be deployed before taking any action.

Dependency abuse

The dependencies between resources also tend to become more complex in larger templates. It is quite easy to abuse the usage of the `dependsOn` feature in ARM templates because of the way they work. Every resource in a template can refer to all its prior resources rather than building a tree of dependencies. ARM templates do not complain if a single resource is dependent on all other resources in the ARM template, even though those other resources might have inter-dependencies within themselves. This makes changing ARM templates bug prone and, at times, it is not even possible to change them.

Reduced agility

Generally, there are multiple teams in a project, with each owning their own resources in Azure. These teams will find it difficult to work with a single ARM template because a single developer should be updating them. Updating a single template with multiple teams might induce conflict and difficult-to-solve merges. Having multiple smaller templates can enable each team to author their own piece of an ARM template.

No reusability

If you have a single template, then that's all that you have, and using this template means deploying all resources. There is no possibility, out of the box, to select individual resources without some maneuvering, such as adding conditional resources. A single large template loses reusability because you take all the resources or none of them.

Knowing that single large templates have so many issues, it is good practice to author modular templates so that we get benefits such as the following:

- Multiple teams can work on their templates in isolation.
- Templates can be reused across projects and solutions.
- Templates are easy to debug and troubleshoot.

Now that we have covered some of the issues with single large templates, in the next section, we will consider the crux of modular templates and how they may help developers to implement efficient deployments.

Understanding the Single Responsibility Principle

The **Single Responsibility Principle** is one of the core principles of the SOLID design principles. It states that a class or code segment should be responsible for a single function and that it should own that functionality completely. The code should change or evolve only if there is a functional change or bug in the current functionality and not otherwise. This code should not change because of changes in some other component or code that is not part of the current component.

Applying the same principle to ARM templates helps us to create templates that have the sole responsibility of deploying a single resource or functionality instead of deploying all resources and a complete solution.

Using this principle will help you create multiple templates, each responsible for a single resource or a smaller group of resources rather than all resources.

Faster troubleshooting and debugging

Each template deployment is a distinct activity within Azure and is a separate instance consisting of inputs, outputs, and logs. When multiple templates are deployed for deploying a solution, each template deployment has separate log entries along with its input and output descriptions. It is much easier to isolate bugs and troubleshoot issues using these independent logs from multiple deployments compared to a single large template.

Modular templates

When a single large template is decomposed into multiple templates where each smaller template takes care of its own resources, and those resources are solely owned, maintained, and are the responsibility of the template containing it, we can say we have modular templates. Each template within these templates follows the Single Responsibility Principle.

Before learning how to divide a large template into multiple smaller reusable templates, it is important to understand the technology behind creating smaller templates and how to compose them to deploy complete solutions.

Deployment resources

ARM provides a facility to link templates. Although we have already gone through linked templates in detail, I will mention it here to help you understand how linking templates helps us achieve modularity, composition, and reusability.

ARM templates provide specialized resources known as **deployments**, which are available from the **Microsoft.Resources** namespace. A deployment resource in an ARM template looks very similar to the code segment that follows:

```
"resources": [
  {
    "apiVersion": "2019-10-01",
    "name": "linkedTemplate",
    "type": "Microsoft.Resources/deployments",
    "properties": {
      "mode": "Incremental",
      <nested-template-or-external-template>
    }
  }
]
```

This template is self-explanatory, and the two most important configurations in the template resource are the type and the properties. The type here refers to the deployment resource rather than any specific Azure resource (storage, virtual machine, and so on) and the properties specify the deployment configuration, including a linked template deployment or a nested template deployment.

However, what does the deployment resource do? The job of a deployment resource is to deploy another template. Another template could be an external template in a separate ARM template file, or it could be a nested template. It means that it is possible to invoke other templates from a template, just like a function call.

There can be nested levels of deployments in ARM templates. What this means is that a single template can call another template, and the called template can call another template, and this can go on for five levels of nested callings:

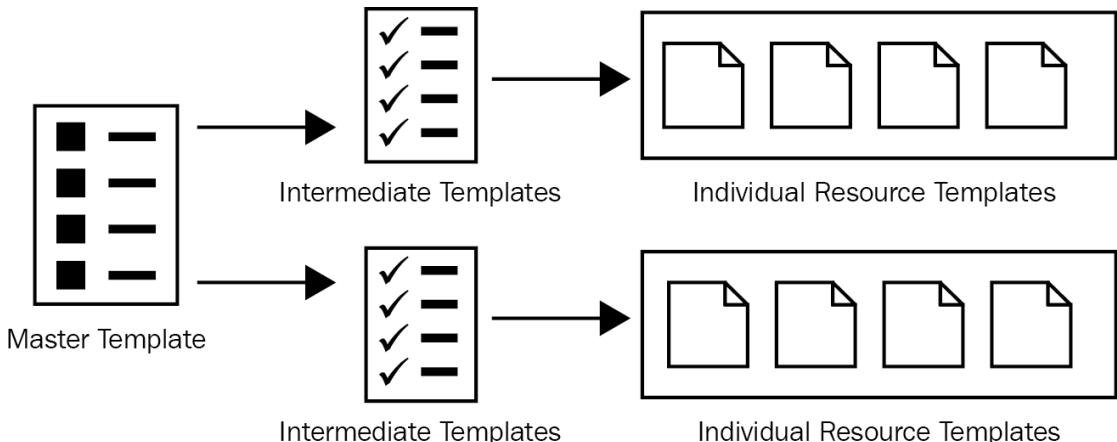


Figure 16.1: Template decomposition into smaller templates

Now that we understand that large templates can be modular with separate resources in separate templates, we need to link and bring them together to deploy resources on Azure. Linked and nested templates are ways to compose multiple templates together.

Linked templates

Linked templates are templates that invoke external templates. External templates are stored in different ARM template files. An example of linked templates follows:

```
"resources": [  
  {  
    "apiVersion": "2019-10-01",  
    "name": "linkedTemplate",  
    "type": "Microsoft.Resources/deployments",  
    "properties": {  
      "mode": "Incremental",  
      "templateLink": {  
        "uri": "https://mystorageaccount.blob.core.windows.net/  
AzureTemplates/newStorageAccount.json",  
        "contentVersion": "1.0.0.0"  
      },  
      "parametersLink": {  
        "uri": "https://mystorageaccount.blob.core.windows.net/  
AzureTemplates/newStorageAccount.parameters.json",  
        "contentVersion": "1.0.0.0"  
      }  
    }  
  }]
```

```
        }
    }
}
]
```

Important additional properties in this template compared to the previous template are **templateLink** and **parametersLink**. Now, **templateLink** refers to the actual URL of the location of the external template file, and **parametersLink** is the URL location for the corresponding **parameters** file. It is important to note that the caller template should have access rights to the location of the called template. For example, if the external templates are stored in Azure Blob storage, which is protected by keys, then the appropriate **Secure Access Signature (SAS)** keys must be available to the caller template to be able to access the linked templates.

It is also possible to provide explicit inline parameters instead of the **parametersLink** value, as shown here:

```
"resources": [
{
    "apiVersion": "2019-10-01",
    "name": "linkedTemplate",
    "type": "Microsoft.Resources/deployments",
    "properties": {
        "mode": "Incremental",
        "templateLink": {
            "uri": "https://mystorageaccount.blob.core.windows.net/AzureTemplates/newStorageAccount.json",
            "contentVersion": "1.0.0.0"
        },
        "parameters": {
            "StorageAccountName": {"value": "
                [parameters('StorageAccountName')]"
        }
    }
}
]
```

You now have a good understanding of linked templates. A closely related topic is nested templates, which the next section will discuss in detail.

Nested templates

Nested templates are a relatively new feature in ARM templates compared to external linked templates.

Nested templates do not define resources in external files. The resources are defined within the caller template itself and within the deployment resource, as shown here:

```
"resources": [
  {
    "apiVersion": "2019-10-01",
    "name": "nestedTemplate",
    "type": "Microsoft.Resources/deployments",
    "properties": {
      "mode": "Incremental",
      "template": {
        "$schema": "https://schema.management.azure.com/schemas/2015-
          01-01/deploymentTemplate.json#",
        "contentVersion": "1.0.0.0",
        "resources": [
          {
            "type": "Microsoft.Storage/storageAccounts",
            "name": "[variables('storageName')]",
            "apiVersion": "2019-04-01",
            "location": "West US",
            "properties": {
              "accountType": "Standard_LRS"
            }
          }
        ]
      }
    }
  }
]
```

In this code segment, we can see that the storage account resource is nested within the original template as part of the deployments resource. Instead of using the **templateLink** and **parametersLink** attributes, a **resources** array is used to create multiple resources as part of a single deployment. The advantage of using a nested deployment is that resources within a parent can be used to reconfigure them by using their names. Usually, a resource with a name can exist only once within a template. Nested templates allow us to use them within the same template and ensure that all templates are self-sufficient rather than being stored separately, and they may or may not be accessible to those external files.

Now that we understand the technology behind modular ARM templates, how should we divide a large template into smaller templates?

There are multiple ways a large template can be decomposed into smaller templates. Microsoft recommends the following pattern for the decomposition of ARM templates:

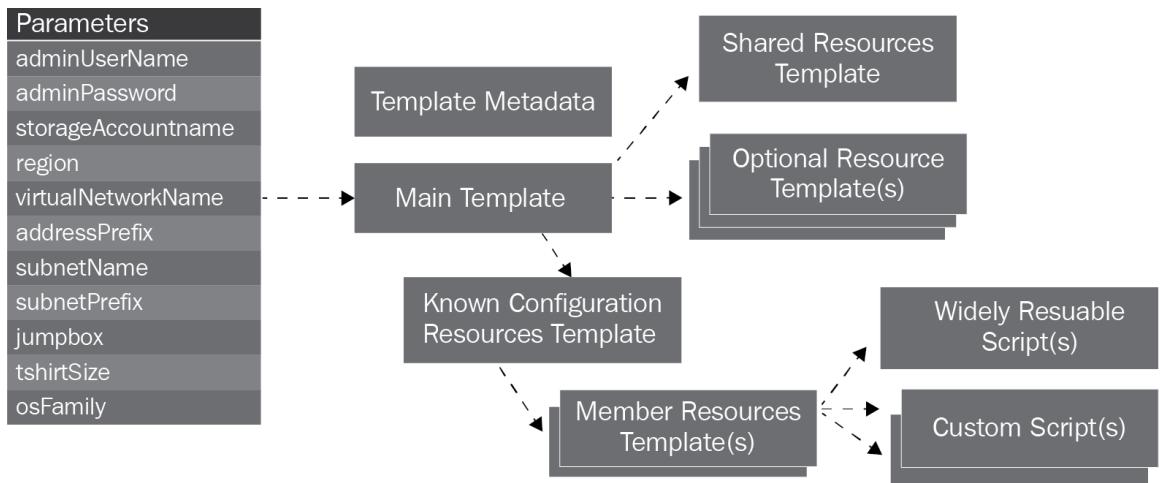


Figure 16.2: Template decomposition strategy

When we decompose a large template into smaller templates, there is always the main template, which is used for deploying the solution. This main or master template internally invokes other nested or linked templates and they, in turn, invoke other templates, and finally, the templates containing Azure resources are deployed.

The main template can invoke a known configuration resource template, which, in turn, will invoke templates comprising Azure resources. The known configuration resource template is specific to a project or solution and it does not have many reusable factors associated with it. The member resource templates are reusable templates invoked by the known configuration resource template.

Optionally, the master template can invoke shared resource templates and other resource templates if they exist.

It is important to understand known configurations. Templates can be authored as known configurations or as free-flow configurations.

Free-flow configurations

ARM templates can be authored as generic templates where most, if not all, of the values assigned to variables are obtained as parameters. This allows the person using the template to pass any value they deem necessary to deploy resources in Azure. For example, the person deploying the template could choose a virtual machine of any size, any number of virtual machines, and any configuration for its storage and networks. This is known as free-flow configuration, where most of the configuration is allowed and the templates come from the user instead of being declared within the template.

There are challenges with this kind of configuration. The biggest one is that not all configurations are supported in every Azure region and datacenter in Azure. The templates will fail to create resources if those resources are not allowed to be created in specific locations or regions. Another issue with free-flow configuration is that users can provide any value they deem necessary and a template will honor them, thereby increasing both the cost and deployment footprint even though they are not completely required.

Known configurations

Known configurations, on the other hand, are specific pre-determined configurations for deploying an environment using ARM templates. These pre-determined configurations are known as **T-shirt sizing configurations**. Similar to the way a T-shirt is available in a pre-determined configuration such as small, medium, and large, ARM templates can be pre-configured to deploy a small, medium, or large environment depending on the requirements. This means that users cannot determine any random custom size for the environment, but they can choose from various provided options, and ARM templates executed during runtime will ensure that an appropriate configuration of the environment is provided.

So, the first step in creating a modular ARM template is deciding on the known configurations for an environment.

As an example, here is the configuration of a datacenter deployment on Azure:

T-Shirt Size	ARM Template Configuration
Small	Four virtual machines with 7 GB of memory along with four CPU cores
Medium	Eight virtual machines with 14 GB of memory along with eight CPU cores
Large	16 virtual machines with 28 GB of memory along with eight CPU cores

Table 16.1: Configuration of a datacenter deployment on Azure

Now that we know the configurations, we can create modular ARM templates.

There are two ways to write modular ARM templates:

- **Composed templates:** Composed templates link to other templates. Examples of composed templates are master and intermediate templates.
- **Leaf-level templates:** Leaf-level templates are templates that contain a single Azure resource.

ARM templates can be divided into modular templates based on the following:

- Technology
- Functionality

An ideal way to decide on the modular method to author an ARM template is as follows:

- Define resource- or leaf-level templates consisting of single resources. In the upcoming diagram, the extreme right templates are leaf-level templates. Within the diagram, virtual machines, virtual network, storage, and others in the same column represent leaf-level templates.
- Compose environment-specific templates using leaf-level templates. These environment-specific templates provide an Azure environment, such as a SQL Server environment, an App Service environment, or a datacenter environment. Let's drill down a bit more into this topic. Let's take the example of an Azure SQL environment. To create an Azure SQL environment, multiple resources are needed. At a bare minimum, a logical SQL Server, a SQL database, and a few SQL firewall resources should be provisioned. All these resources are defined in individual templates at the leaf level. These resources can be composed together in a single template that has the capability to create an Azure SQL environment. Anybody wanting to create an SQL environment can use this composed template. *Figure 16.3* has **Data center**, **Messaging**, and **App Service** as environment-specific templates.
- Create templates with higher abstraction composing multiple environment-specific templates into solutions. These templates are composed of environment-specific templates that were created in the previous step. For example, to create an e-commerce inventory solution that needs an App Service environment and a SQL environment, two environment templates, App Service and SQL Server, can be composed together. *Figure 16.3* has **Functional 1** and **Functional 2** templates, which are composed of child templates.
- Finally, a master template should be created, which should be composed of multiple templates where each template is capable of deploying a solution.

The preceding steps for creating a modular designed template can be easily understood by means of Figure 16.3:

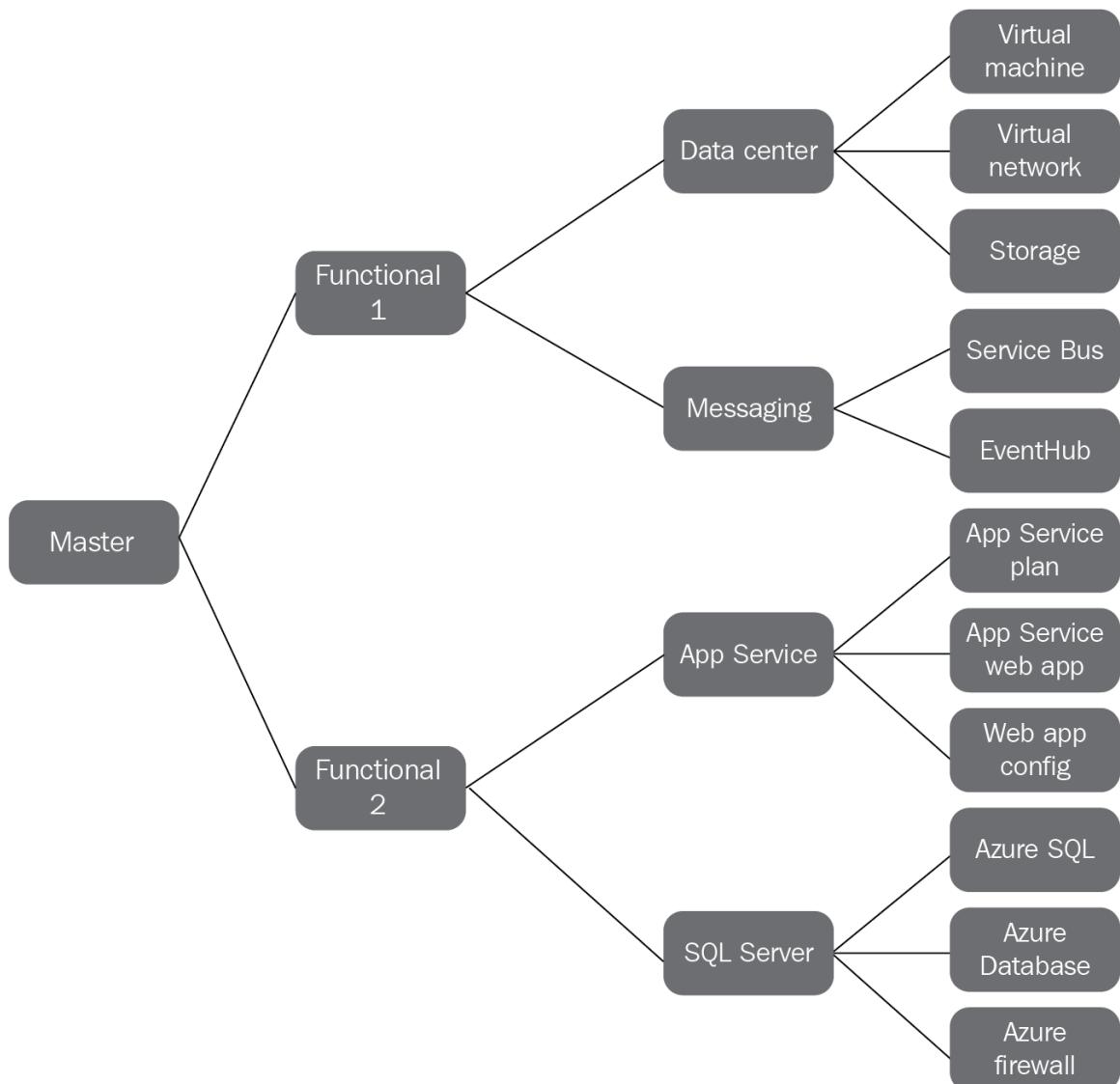


Figure 16.3: Template and resource mapping

Now, let's implement a part of the functionality shown in the previous diagram. In this implementation, we will provide a virtual machine with a script extension using a modular approach. The custom script extension deploys Docker binaries and prepares a container environment on a Windows Server 2016 virtual machine.

Now, we are going to create a solution using ARM templates using a modular approach. As mentioned before, the first step is to create individual resource templates. These individual resource templates will be used to compose additional templates capable of creating an environment. These templates will be needed to create a virtual machine. All ARM templates shown here are available in the accompanying chapter code. The names and code of these templates are as follows:

- **Storage.json**
- **virtualNetwork.json**
- **PublicIPAddress.json**
- **NIC.json**
- **VirtualMachine.json**
- **CustomScriptExtension.json**

First, let's look at the code for the **Storage.json** template. This template provides a storage account, which every virtual machine needs for storing its OS and data disk files:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-  
01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "storageAccountName": {  
            "type": "string",  
            "minLength": 1  
        },  
        "storageType": {  
            "type": "string",  
            "minLength": 1  
        },  
        ...  
    },  
    "outputs": {  
        "resourceDetails": {  
            "type": "object",  
            "value": "[reference(parameters('storageAccountName'))]"  
        }  
    }  
}
```

Next, let's look at the code for the public IP address template. A virtual machine that should be accessible over the internet needs a public IP address resource assigned to its network interface card. Although exposing a virtual machine to the internet is optional, this resource might get used for creating a virtual machine. The following code is available in the **PublicIPAddress.json** file:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-  
01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "publicIPName": {  
            "type": "string",  
            "minLength": 1  
        },  
        "publicIPType": {  
            "type": "string",  
            "minLength": 1  
        },  
        ...  
    },  
    "outputs": {  
        "resourceDetails": {  
            "type": "object",  
            "value": "[reference(parameters('publicIPName'))]"  
        }  
    }  
}
```

Next, let's look at the code for the virtual network. Virtual machines on Azure need a virtual network for communication. This template will be used to create a virtual network on Azure with a pre-defined address range and subnets. The following code is available in the **virtualNetwork.json** file:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-  
01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "virtualNetworkName": {  
            "type": "string",  
            "minLength": 1  
        ...  
    },  
        "subnetPrefix": {  
            "type": "string",  
            "minLength": 1  
        },  
        "resourceLocation": {  
            "type": "string",  
            "minLength": 1  
        }  
    ...  
        "subnets": [  
            {  
                "name": "[parameters('subnetName')]",  
                "properties": {  
                    "addressPrefix": "[parameters('subnetPrefix')]"  
                }  
            }  
        ]  
    }  
},  
    "outputs": {  
        "resourceDetails": {  
            "type": "object",  
            "value": "[reference(parameters('virtualNetworkName'))]"  
        }  
    }  
}
```

Next, let's look at the code for the network interface card. A virtual network card is needed by a virtual machine to connect to a virtual network and to accept and send requests to and from the internet. The following code is available in the **NIC.json** file:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-  
01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "nicName": {  
            "type": "string",  
            "minLength": 1  
        },  
        "publicIpReference": {  
            "type": "string",  
            "minLength": 1  
        },  
        ...  
        [resourceId(subscription().subscriptionId,resourceGroup().name, 'Microsoft.  
Network/publicIPAddresses', parameters('publicIpReference'))]",  
        "vnetRef": "[resourceId(subscription().  
subscriptionId,resourceGroup().name, 'Microsoft.Network/virtualNetworks',  
parameters('virtualNetworkReference'))]",  
        "subnet1Ref": "[concat(variables('vnetRef'), '/subnets/',  
parameters('subnetReference'))]"  
    },  
    ...  
        "id": "[variables('subnet1Ref')]"  
    }  
},  
}  
]  
}  
]  
}  
],  
"outputs": {  
    "resourceDetails": {  
        "type": "object",  
        "value": "[reference(parameters('nicName'))]"  
    }  
}  
}
```

Next, let's look at the code for creating a virtual machine. Each virtual machine is a resource in Azure, and note that this template has no reference to storage, network, public IP addresses, or other resources created earlier. This reference and composition will happen later in this section using another template. The following code is available in the **VirtualMachine.json** file:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01  
01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "vmName": {  
            "type": "string",  
            "minLength": 1  
        },  
        ...  
    },  
    "imageOffer": {  
        "type": "string",  
        "minLength": 1  
    },  
    "windowsOSVersion": {  
        "type": "string",  
        "minLength": 1  
    },  
    ...  
    "outputs": {  
        "resourceDetails": {  
            "type": "object",  
            "value": "[reference(parameters('vmName'))]"  
        }  
    }  
}
```

Next, let's look at the code for creating a custom script extension. This resource executes a PowerShell script on a virtual machine after it is provisioned. This resource provides an opportunity to execute post-provisioning tasks in Azure Virtual Machines. The following code is available in the `CustomScriptExtension.json` file:

```
{  
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/  
deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "VMName": {  
            "type": "string",  
            "defaultValue": "sqldock",  
            "metadata": {  
                ...  
                "commandToExecute": "[concat('powershell -ExecutionPolicy  
Unrestricted -file docker.ps1')]"  
            },  
            "protectedSettings": {  
                ...  
            }  
        }  
    },  
    "outputs": {}  
}
```

Next, we'll look at the custom script extension PowerShell code that prepares the Docker environment. Please note that a virtual machine reboot might happen while executing the PowerShell script, depending on whether the Windows containers feature is already installed or not. The following script installs the NuGet package, the `DockerMsftProvider` provider, and the Docker executable. The `docker.ps1` file is available with the accompanying chapter code:

```
#  
# docker.ps1  
#  
Install-PackageProvider -Name Nuget -Force -ForceBootstrap -Confirm:$false  
Install-Module -Name DockerMsftProvider -Repository PSGallery -Force  
-Confirm:$false -verboseInstall-Package -Name docker -ProviderName  
DockerMsftProvider -Force -ForceBootstrap -Confirm:$false
```

All the previously seen linked templates should be uploaded to a container within an Azure Blob storage account. This container can have a private access policy applied, as you saw in the previous chapter; however, for this example, we will set the access policy as **container**. This means these linked templates can be accessed without an SAS token.

Finally, let's focus on writing the master template. Within the master template, all the linked templates are composed together to create a solution—to deploy a virtual machine and execute a script within it. The same approach can be used for creating other solutions, such as providing a datacenter consisting of multiple inter-connected virtual machines. The following code is available in the **Master.json** file:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/  
deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "storageAccountName": {  
            "type": "string",  
            "minLength": 1  
        ...  
    },  
        "subnetName": {  
            "type": "string",  
            "minLength": 1  
    },  
        "subnetPrefix": {  
            "type": "string",  
            "minLength": 1  
    ...  
        "windowsOSVersion": {  
            "type": "string",  
            "minLength": 1  
    },  
        "vhdStorageName": {  
            "type": "string",  
            "minLength": 1  
    },  
        "vhdStorageContainerName": {  
            "type": "string",  
            "minLength": 1  
    ...[concat('https://',parameters('storageAccountName'),'arntfiles.blob.  
core.windows.net/',variables('containerName'), '/Storage.json')]]",  
        "contentVersion": "1.0.0.0"
```

```
},
"parameters": {
    "storageAccountName": {
        "value": "[parameters('storageAccountName')]"
    },
    "storageType": {
        "value": "[parameters('storageType')]"
    },
    "resourceLocation": {
        "value": "[resourceGroup().location]"
    }
    ...
},
"outputs": {
    "resourceDetails": {
        "type": "object",
        "value": "[reference('GetVM').outputs.resourceDetails.value]"
    }
}
}
```

The master templates invoke the external templates and also co-ordinate inter-dependencies among them.

The external templates should be available in a well-known location so that the master template can access and invoke them. In this example, the external templates are stored in the Azure Blob storage container and this information was passed to the ARM template by means of parameters.

The external templates in Azure Blob storage could be access-protected by setting up access policies. The command used to deploy the master template is shown next. It might look like a complex command, but a majority of the values are used as parameters. You are advised to change the values of these parameters before running it. The linked templates have been uploaded to a storage account named **st02gvwlcdcxm5suwe** within the **armtemplates** container. The resource group should be created if it does not currently exist. The first command is used to create a new resource group in the **West Europe** region:

```
New-AzResourceGroup -Name "testvmrg" -Location "West Europe" -Verbose
```

The rest of the parameter values are needed to configure each resource. The storage account name and the **dnsNameForPublicIP** value should be unique within Azure:

```
New-AzResourceGroupDeployment -Name "testdeploy1" -ResourceGroupName testvmrg -Mode Incremental -TemplateFile "C:\chapter 05\Master.json" -storageAccountName "st02gvwlcdxm5suwe" -storageType "Standard_LRS" -publicIPAddressName "uniipaddname" -publicIPAddressType "Dynamic" -dnsNameForPublicIP "azureforarchitectsbook" -virtualNetworkName vnetwork01 -addressPrefix "10.0.1.0/16" -subnetName "subnet01" -subnetPrefix "10.0.1.0/24" -nicName nic02 -vmSize "Standard_DS1" -adminUsername "sysadmin" -adminPassword $(ConvertTo-SecureString -String sysadmin@123 -AsPlainText -Force) -vhdStorageName oddnewuniqueacc -vhdStorageContainerName vhds -OSDiskName mynewvm -vmName vm10 -windowsOSVersion 2012-R2-Datacenter -imagePublisher MicrosoftWindowsServer -imageOffer WindowsServer -containerName armtemplates -Verbose
```

In this section, we covered best practices for decomposing large templates into smaller reusable templates and combining them together at runtime to deploy complete solutions on Azure. As we progress through the book, we will modify the ARM template step by step until we have explored its core parts. We used Azure PowerShell cmdlets to initiate the deployment of templates on Azure.

Let's move on to the topic of **copy** and **copyIndex**.

Understanding copy and copyIndex

There are many times when multiple instances of a particular resource or a group of resources are needed. For example, you may need to provision 10 virtual machines of the same type. In such cases, it is not prudent to deploy templates 10 times to create these instances. A better alternate approach is to use the **copy** and **copyIndex** features of ARM templates.

copy is an attribute of every resource definition. This means it can be used to create multiple instances of any resource type.

Let's understand this with the help of an example of creating multiple storage accounts within a single ARM template deployment.

The next code snippet creates 10 storage accounts serially. They could have been created in parallel by using **Parallel** instead of **Serial** for the **mode** property:

```
"resources": [
    {
        "apiVersion": "2019-06-01",
        "type": "Microsoft.Storage/storageAccounts",
        "location": "[resourceGroup().location]",
        "name": "[concat(variables('storageAccountName'), copyIndex())]",
        "tags": {
            "displayName": "[variables('storageAccountName')]"
        },
        "sku": {
            "name": "Premium_ZRS"
        },
        "kind": "StorageV2",
        "copy": {
            "name": "storageInstances",
            "count": 10,
            "mode": "Serial"
        }
    }
],
```

In the preceding code, **copy** has been used to provision 10 instances of the storage account serially, that is, one after another. The storage account names must be unique for all 10 instances, and **copyIndex** has been used to make them unique by concatenating the original storage name with the index value. The value returned by the **copyIndex** function changes in every iteration; it will start at 0 and go on for 10 iterations. This means it will return **9** for the last iteration.

Now that we have learned how to create multiple instances of an ARM template, let's dive into securing these templates from known vulnerabilities.

Securing ARM templates

Another important aspect related to creating enterprise ARM templates is securing them appropriately. ARM templates contain the resource configuration and vital information about infrastructure, and so they should not be compromised or accessible to unauthorized people.

The first step in securing ARM templates is storing them in storage accounts and stopping any anonymous access to the storage account container. Moreover, SAS tokens should be generated for storage accounts and used in ARM templates to consume linked templates. This will ensure that only the holders of SAS tokens can access the templates. Moreover, these SAS tokens should be stored in Azure Key Vault instead of being hardcoded into ARM templates. This will ensure that even the people responsible for deployment do not have access to the SAS token.

Another step in securing ARM templates is ensuring that any sensitive information and secrets, such as database connection strings, Azure subscription and tenant identifiers, service principal identifiers, IP addresses, and so on, should not be hardcoded in ARM templates. They should all be parameterized, and the values should be fetched at runtime from Azure Key Vault. However, before using this approach, it is important that these secrets are stored in Key Vault prior to executing any ARM templates.

The following code shows one of the ways that values can be extracted from Azure Key Vault at runtime using the parameters file:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2016-01-01/
deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "storageAccountName": {
            "reference": {
                "keyVault": {
                    "id": "/subscriptions/--subscription id --/
resourceGroups/rgname/providers/Microsoft.KeyVault/vaults/keyvaultbook"),
                    "secretName": "StorageAccountName"
                }
            }
        }
    }
}
```

In this code listing, a parameter is defined that references Azure Key Vault to fetch values at runtime during deployment. The Azure Key Vault identifier and the secret name have been provided as input values.

Now that you have learned how to secure ARM templates, let's take a look at identifying the various dependencies between them and how we can enable communication between multiple templates.

Using outputs between ARM templates

One of the important aspects that can easily be overlooked while using linked templates is that there might be resource dependencies within linked templates. For example, a SQL Server resource might be in a linked template that is different to that of a virtual machine resource. If we want to open the SQL Server firewall for the virtual machine IP address, then we should be able to dynamically pass this information to the SQL Server firewall resource after provisioning the virtual machine.

This could be done using the simple method of referring to the IP address resource using the **REFERENCES** function if the SQL Server and virtual machine resources are in the same template.

It becomes slightly more complex in the case of linked templates if we want to share runtime property values from one resource to another when they are in different templates.

ARM templates provide an **outputs** configuration, which is responsible for generating outputs from the current template deployment and returning them to the user. For example, we might output a complete object, as shown in the following code listing, using the **reference** function, or we might just output an IP address as a string value:

```
"outputs": {  
    "storageAccountDetails": {  
        "type": "object",  
        "value": "[reference(resourceid  
            ('Microsoft.Storage/storageAccounts',  
            variables('storageAccountName')))]",  
    },  
    "virtualMachineIPAddress": {  
        "type": "string",  
        "value": "[reference(variables  
            ('publicIPPropertyName')).properties.ipAddress]"  
    },  
}
```

Parameters within a linked template can be utilized by the master template. When a linked template is called, the output is available to the master template that can be supplied as a parameter to the next linked or nested template. This way, it is possible to send the runtime configuration values of resources from one template to another.

The code in the master template would be similar to what's shown here; this is the code that's used to call the first template:

```
{  
    "type": "Microsoft.Resources/deployments",  
    "apiVersion": "2017-05-10",  
    "name": "createvm",  
    "resourceGroup": "myrg",  
    "dependsOn": [  
        "allResourceGroups"  
    ],  
    "properties": {  
        "mode": "Incremental",  
        "templateLink": {  
            "uri": "[variables(  
                'templateRefSharedServicesTemplateUri')]",  
            "contentVersion": "1.0.0.0"  
        },  
        "parameters": {  
            "VMName": {  
                "value": "[variables('VmName')]"  
            }  
        }  
    }  
}
```

The preceding code snippet from the master template is calling a nested template responsible for provisioning a virtual machine. The nested template has an output section that provides the IP address of the virtual machine. The master template will have another deployment resource in its template that will take the output value and send it as a parameter to the next nested template, passing the IP address at runtime. This is shown in the following code:

```
{  
  "type": "Microsoft.Resources/deployments",  
  "apiVersion": "2017-05-10",  
  "name": "createSQLServer",  
  "resourceGroup": "myrg",  
  "dependsOn": [  
    "createvm"  
,  
  "properties": {  
    "mode": "Incremental",  
    "templateLink": {  
      "uri": "[variables('templateRefsql')]",  
      "contentVersion": "1.0.0.0"  
    },  
    "parameters": {  
      "VMName": {  
        "value": "[reference  
('createvm').outputs.virtualMachineIPAddress.value]"  
      }  
    }  
  }  
}
```

In the preceding code listing, a nested template is being invoked and a parameter is being passed to it. The value of the parameter is derived from the previous linked template's output, which is named **virtualMachineIPAddress**. Now, the nested template will get the IP address of the virtual machine dynamically and it can use it as a whitelisted IP address.

Using this approach, we can pass runtime values from one nested template to another.

Summary

ARM templates are the preferred means of provisioning resources in Azure. They are idempotent in nature, bringing consistency, predictability, and reusability to environment creation. In this chapter, we looked at how to create a modular ARM template. It is important for teams to spend quality time designing ARM templates in an appropriate way, so that multiple teams can work on them together. They are highly reusable and require minimal changes to evolve. In this chapter, we learned how to create templates that are secure by design, how to provision multiple resource instances in a single deployment, and how to pass outputs from one nested template to another using the outputs section of ARM templates.

The next chapter will move on to a different and very popular strand of technology known as serverless within Azure. Azure Functions is one of the major serverless resources of Azure, and this will be covered in complete depth, including Durable Functions.

17

Designing IoT solutions

In the previous chapter, you learned about ARM templates, and so far, we have been dealing with architectural concerns and their solutions in Azure in general. However, this chapter is not based on generalized architecture. In fact, it explores one of the most disruptive technologies of this century. This chapter will discuss the details of the **Internet of Things (IoT)** and Azure.

Azure IoT refers to a collection of Microsoft-managed cloud services that can connect, monitor, and control billions of IoT assets. In other words, an IoT solution comprises one or more IoT devices that constantly communicate with one or more back-end servers in the cloud.

This chapter will cover the following topics:

- Azure and IoT
- An overview of Azure IoT
- Device management
- Registering devices
- Device-to-IoT-hub communication
- Scaling IoT solutions
- High availability for IoT solutions
- IoT protocols
- Using message properties to route messages

IoT

The internet was invented in the 1980s and later became widely available. Almost everyone moved toward having a presence on the internet and started creating their own static web pages. Eventually, the static content became dynamic and could be generated on the fly, based on context. In nearly all cases, a browser was needed to access the internet. There were a plethora of browsers available, and without them, using the internet was a challenge.

During the first decade of this century, there was an interesting development that was emerging—the rise of handheld devices, such as mobile phones and tablets. Mobile phones started becoming cheaper and were available ubiquitously. The hardware and software capabilities of these handheld devices were improving considerably, and so much so that people started using browsers on their mobile devices rather than on their desktops. But one particularly distinct change was the rise of mobile apps. These mobile apps were downloaded from a store and connected to the internet to talk to back-end systems. Toward the end of the last decade, there were millions of apps available with almost every conceivable functionality built into them. The back-end system for these apps was built on the cloud so that they could be scaled rapidly. This was the age of connecting applications and servers.

But was this the pinnacle of innovation? What was the next evolution of the internet? Well, another paradigm has now been taking center stage: IoT. Instead of just mobile and tablet devices connecting to the internet, why can't other devices connect to the internet? Previously, such devices were available only in select markets; they were costly, not available to the masses, and had limited hardware and software capabilities. However, since the first part of this decade, the commercialization of these devices has been growing on a grand scale. These devices are becoming smaller and smaller, are more capable in terms of hardware and software, have more storage and compute power, can connect to the internet on various protocols, and can be attached to almost anything. This is the age of connecting devices to servers, applications, and other devices.

This has led to the formulation of the idea that IoT applications can change the way that industries are operating. Newer solutions that were previously unheard of are beginning to be realized. Now, these devices can be attached to anything; they can get information and send it to a back-end system that can assimilate information from all the devices and either take action on or report incidents.

IoT sensors and controls can be leveraged in many business use cases. For example, they can be used in vehicle tracking systems, which can track all the vital parameters of a vehicle and send those details to a centralized data store for analysis. Smart city initiatives can also make use of various sensors to track pollution levels, temperature, and street congestion. IoT has also found its way into agriculture-related activities, such as measuring soil fertility, humidity, and more. You can visit Microsoft Technical Case Studies for IoT, at <https://microsoft.github.io/techcasestudies/#technology=IoT&sortBy=featured>, for real-life examples of how organizations leverage Azure IoT.

Before we explore the tools and services related to IoT, we will first cover IoT architecture in detail.

IoT architecture

Before getting into Azure and its features and services regarding IoT, it is important to understand the various components that are needed to create end-to-end IoT solutions.

Consider that IoT devices across the globe are sending millions of messages every second to a centralized database. Why is this data being collected? Well, the answer is to extract rich information about events, anomalies, and outliers that are to do with whatever those devices are monitoring.

Let's understand this in more detail.

IoT architecture can be divided into distinct phases, as follows:

1. **Connectivity:** This phase involves a connection being made between a device and the IoT service.
2. **Identity:** After connecting to the IoT service, the first thing that happens is the identification of the device and ensuring that it is allowed to send device telemetry to the IoT service. This is done using an authentication process.
3. **Capture:** During this phase, the device telemetry is captured and received by the IoT service.
4. **Ingestion:** In this phase, the IoT service ingests the device telemetry.
5. **Storage:** The device telemetry is stored. It could be a temporary or permanent store.
6. **Transformation:** The telemetry data is transformed for further processing. This includes augmenting existing data and inferring data.
7. **Analytics:** The transformed data is used to find patterns, anomalies, and insights.
8. **Presentation:** The insights are shown as dashboards and reports. Additionally, new alerts can be generated that could invoke automation scripts and processes.

Figure 17.1 shows a generic IoT-based architecture. Data is generated or collected by devices and sent over to the cloud gateway. The cloud gateway, in turn, sends the data to multiple back-end services for processing. Cloud gateways are optional components; they should be used when the devices themselves are not capable of sending requests to back-end services, either because of resource constraints or the lack of a reliable network. These cloud gateways can collate data from multiple devices and send it to back-end services. The data can then be processed by back-end services and shown as insights or dashboards to users:

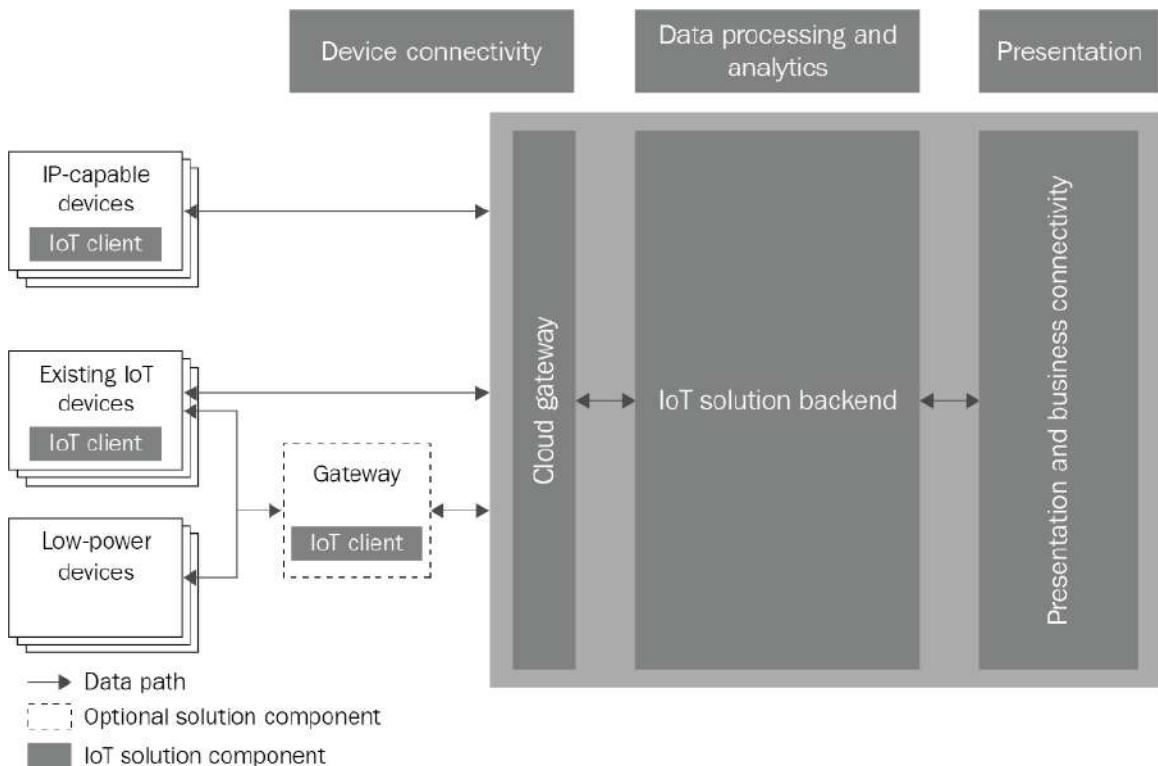


Figure 17.1: A generic IoT application architecture

Now that we are clear about the architecture, let's go ahead and understand how IoT devices communicate with other devices.

Connectivity

IoT devices need to communicate to connect to other devices. There are various connectivity types; for example, connectivity could exist between devices in a region, between devices and a centralized gateway, and between devices and an IoT platform.

In all such cases, IoT devices need connectivity capability. This capability could be in the form of internet connectivity, Bluetooth, infrared, or any other near-device communication.

However, some IoT devices might not have the capability to connect to the internet. In these cases, they can connect to a gateway that in turn has connectivity to the internet.

IoT devices use protocols to send messages. The major protocols are the **Advanced Message Queuing Protocol (AMQP)** and the **Message Queue Telemetry Transport (MQTT)** protocol.

Device data should be sent to an IT infrastructure. The MQTT protocol is a device-to-server protocol that devices can use to send telemetry data and other information to servers. Once the server receives a message through the MQTT protocol, it needs to transport the message to other servers using a reliable technology that is based on messages and queues. AMQP is the preferred protocol for moving messages between servers in an IT infrastructure in a reliable and predictable manner:

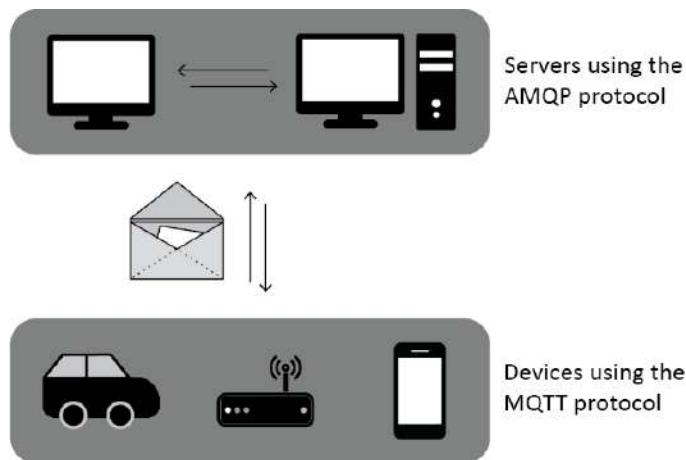


Figure 17.2: Workings of the MQTT and AMQP protocols

Servers receiving initial messages from IoT devices should send those messages to other servers for whatever processing is necessary, such as saving to logs, evaluation, analytics, and presentation.

Some devices do not have the capability to connect to the internet or do not support protocols that are compatible with other server technologies. To enable these devices to work with an IoT platform and the cloud, intermediate gateways can be used. Gateways help in onboarding devices whose connectivity and networking capability is slow and not consistent; such devices may use protocols that are not standard, or their capabilities may be limited in terms of resources and power.

In such circumstances, when devices need additional infrastructure to connect to back-end services, client gateways can be deployed. These gateways receive messages from near devices, and forward (or push) them to the IT infrastructure and the IoT platform for further consumption. These gateways are capable of protocol translation if required.

In this section, you learned about how communication is implemented with other devices and about the role that gateways play in terms of communication. In the next section, we're going to talk about identity.

Identity

IoT devices should be registered with a cloud platform. Devices that are not registered should not be allowed to connect to a cloud platform. The devices should be registered and be assigned an identity. A device should send its identity information when connecting to the cloud. If the device fails to send this identity information, the connectivity should fail. You will see, later in this chapter, how to generate an identity for a device using a simulated application. As you already know, IoT devices are deployed to capture information, and in the next section, we will briefly talk about the capture process.

Capture

IoT devices should be able to capture information. They should have the capability, for example, to read or monitor the moisture content in the air or in soil. This information can be captured based on frequency—maybe even once per second. Once the information is captured, the device should be able to send it across to the IoT platform for processing. If a device can't connect to the IoT platform directly, it can connect to an intermediary cloud gateway instead and have that push the captured information.

The size of captured data and the frequency of capture are the most important things for the device. Whether a device should have local storage to be able to temporarily store captured data is another important aspect that should be considered. For instance, a device can work in offline mode if there is enough local storage available. Even mobile devices sometimes act as IoT devices connected to various instruments and have the capability to store data. Once we have captured the data, we need to ingest it to an IoT platform for further analysis, and in the next section, we will explore ingestion.

Ingestion

Data captured and generated by devices should be sent to an IoT platform that is capable of ingesting and consuming this data to extract meaningful information and insights out of it. The ingestion service is a crucial service because its availability and scalability affect the throughput of incoming data. If data starts getting throttled due to scalability issues, or if data is not able to be ingested due to availability issues, then it will be lost and the dataset might get biased or skewed. We have the data captured and we need a place to store this data. In the next section, you'll learn about storage.

Storage

IoT solutions generally deal with millions or even billions of records, spanning terabytes or even petabytes of data. This is valuable data that can provide insights on operations and their health. This data needs to be stored in such a way that analytics can be performed on it. Storage should be readily available for analytics, applications, and services to consume it. Storage solutions should provide adequate throughput and latency from a performance perspective, and be highly available, scalable, and secure. The next section deals with data transformation, which is needed to store and analyze data.

Transformation

IoT solutions are generally data-driven and have considerably high volumes of data to deal with. Imagine that every car has a device and each one sends messages every five seconds. If there were a million cars sending messages, this would be equal to 288 million messages per day and 8 billion messages per month. Together, this data has lots of hidden information and insights; however, making sense of this kind of data just by looking at it is difficult.

The data that is captured and stored by IoT devices can be consumed to solve business problems, but not all data that is captured is of importance. Just a subset of data might be needed to solve a problem. Additionally, the data that the IoT devices gather might not be consistent either. To ensure that the data is consistent and not biased or skewed, appropriate transformations should be executed upon it to make it ready for analysis. During transformation, data is filtered, sorted, removed, enriched, and transformed into a structure, such that the data can be consumed by components and applications further downstream. We need to perform some analytics with the transformed data before presenting it. As the next step in the workflow, we will discuss analytics.

Analytics

The data transformed in the previous step becomes the input for the analytics step. Depending on the problem at hand, there are different types of analytics that can be performed on transformed data.

The following are the different types of analytics that can be performed:

- **Descriptive analytics:** This type of analytics helps in finding patterns and details about the status of the IoT devices and their overall health. This stage identifies and summarizes the data for further consumption by more advanced stages of analytics. It will help in summarization, finding statistics related to probability, identifying deviation, and other statistical tasks.

- **Diagnostic analytics:** This type of analytics is more advanced than descriptive analytics. It builds on descriptive analytics and tries to answer queries about why certain things have happened. That is, it tries to find the root causes of events. It tries to find answers using advanced concepts, such as hypothesis and correlation.
- **Predictive analytics:** This type of analytics tries to predict things that have a high probability of happening in the future. It generates predictions that are based on past data; regression is one of the examples that is based on past data. Predictive analytics could, for example, predict the price of a car, the behavior of stock on the stock market, when a car tire will burst, and more.
- **Prescriptive analytics:** This type of analytics is the most advanced. This stage helps in identifying the action that should be performed to ensure that the health of devices and solutions does not degrade, and identifying proactive measures to undertake. The results of this stage of analytics can help in avoiding future issues and eliminating problems at their root causes.

In the final stage, the output from analytics is presented in a human-readable manner for a wider audience to understand and interpret. In the next part, we will discuss presentation.

Presentation

Analytics help in identifying answers, patterns, and insights based on data. These insights also need to be available to all stakeholders in formats that they can understand. To this end, dashboards and reports can be generated, statistically or dynamically, and then be presented to stakeholders. Stakeholders can consume these reports for further action and improve their solutions continuously.

As a quick recap of all the preceding steps, we started off this section by looking at connectivity, where we introduced gateways for sending the data from devices that don't support the standard protocols. Then, we talked about identity and how data is captured. The captured data is then ingested and stored for further transformation. After transformation, analytics is done on the data before it's presented to all stakeholders. As we are working on Azure, in the next section, we will cover what Azure IoT is and consider the basics concepts that we have learned about so far from an Azure standpoint.

Azure IoT

Now you've learned about the various stages of end-to-end IoT solutions; each of these stages is crucial and their proper implementation is a must for any solution's success. Azure provides lots of services for each of these stages. Apart from these services, Azure provides Azure IoT Hub, which is Azure's core IoT service and platform. It is capable of hosting complex, highly available, and scalable IoT solutions. We will dive deep into IoT Hub after going through some other services:

Devices	Device Connectivity	Storage	Analytics	Presentation & Action
	Events	SQL Database	Machine Learning	App Service
	Service Bus	Table/Blob Storage	Stream Analytics	Power BI
	External Data Sources	Cosmos DB	HDIInsight	Notification Hubs
		External Data Sources	Data Factory	Mobile Services
		Time Series Insights	Databricks	Logic Apps

Figure 17.3: List of devices and services for IoT solutions

In the next section, we will follow a similar pattern as we did for our coverage of IoT architecture to learn about communication, identity, capture, ingestion, storage, transformation, analytics, and presentation with Azure IoT.

Connectivity

IoT Hub provides all the important protocol suites for devices to connect to IoT hubs. It offers:

- **HTTPS:** The HyperText Transport Protocol Secure method uses certificates consisting of a pair of keys, known as private-public keys, that are used to encrypt and decrypt data between a device and IoT Hub. It provides one-way communication from a device to the cloud.
- **AMQP:** AMQP is an industry standard for sending and receiving messages between applications. It provides a rich infrastructure for the security and reliability of messages, and that is one of the reasons why it is quite widely used within the IoT space. It provides both device-to-Hub as well as Hub-to-device capabilities, and devices can use it to authenticate using **Claims-Based Security (CBS)** or **Simple Authentication and Security Layer (SASL)**. It is used primarily in scenarios where there are field gateways, and a single identity related to multiple devices can transmit telemetry data to the cloud.

- **MQTT:** MQTT is an industry standard for sending and receiving messages between applications. It provides both device-to-Hub as well as Hub-to-device capabilities. It is used primarily in scenarios where each device has its own identity and authenticates directly with the cloud.

In the next section, we will discuss identity and how devices are authenticated.

Identity

IoT Hub provides services for authenticating devices. It offers an interface for generating unique identity hashes for each device. When devices send messages containing a hash, IoT Hub can authenticate them, after checking in its own database for the existence of such a hash. Now let's see how data is captured.

Capture

Azure provides IoT gateways, which enable devices that do not comply with IoT Hub to be adapted and to push data. Local or intermediary gateways can be deployed near devices in such a way that multiple devices can connect to a single gateway to capture and send their information. Similarly, multiple clusters of devices with local gateways can also be deployed. There can be a cloud gateway deployed on the cloud itself, which is capable of capturing and accepting data from multiple sources and ingesting it for IoT Hub. As discussed previously, we need to ingest the data that we capture. In the next section, you will learn about ingestion with IoT Hub.

Ingestion

An IoT hub can be a single point of contact for devices and other applications. In other words, the ingestion of IoT messages is the responsibility of the IoT Hub service. There are other services, such as Event Hubs and the Service Bus messaging infrastructure, that can ingest incoming messages; however, the benefits and advantages of using IoT Hub to ingest IoT data far outweigh those of using Event Hubs and Service Bus messaging. In fact, IoT Hub was made specifically for the purpose of ingesting IoT messages within the Azure ecosystem so that other services and components can act on them. The ingested data is stored to storage. Before we do any kind of transformation or analytics, let's explore the role of storage in the workflow in the next section.

Storage

Azure provides multiple ways of storing messages from IoT devices. These include storing relational data, schema-less NoSQL data, and blobs:

- **SQL Database:** SQL Database provides storage for relational data, JSON, and XML documents. It provides a rich SQL-query language and it uses a full-blown SQL server as a service. Data from devices can be stored in SQL databases if it is well defined and the schema will not need to undergo changes frequently.
- **Azure Storage:** Azure Storage provides Table and Blob storage. Table storage helps in storing data as entities, where the schema is not important. Blobs help in storing files in containers as blobs.
- **Cosmos DB:** Cosmos DB is a full-blown enterprise-scale NoSQL database. It is available as a service that is capable of storing schema-less data. It is a globally distributed database that can span continents, providing high availability and scalability of data.
- **External data sources:** Apart from Azure services, customers can bring or use their own data stores, such as a SQL server on Azure virtual machines, and can use them to store data in a relational format.

The next section is on transformation and analytics.

Transformation and analytics

Azure provides multiple resources to execute jobs and activities on ingested data. Some of them are listed here:

- **Data Factory:** Azure Data Factory is a cloud-based data integration service that allows us to create data-driven workflows in the cloud for orchestrating and automating data movement and data transformation. Azure Data Factory helps to create and schedule data-driven workflows (called pipelines) that can ingest data from disparate data stores; process and transform data by using compute services such as **Azure HDInsight**, **Hadoop**, **Spark**, **Azure Data Lake Analytics**, **Azure Synapse Analytics**, and **Azure Machine Learning**; and publish output data to a data warehouse for **Business Intelligence (BI)** applications rather than a traditional **Extract-Transform-Load (ETL)** platform.
- **Azure Databricks:** Databricks provides a complete, managed, end-to-end Spark environment. It can help in the transformation of data using Scala and Python. It also provides a SQL library to manipulate data using traditional SQL syntax. It is more performant than Hadoop environments.

- **Azure HDInsight:** Microsoft and Hortonworks have come together to help companies by offering a big data analytics platform with Azure. HDInsight is a high-powered, fully managed cloud service environment powered by Apache Hadoop and Apache Spark using Microsoft Azure HDInsight. It helps in accelerating workloads seamlessly using Microsoft and Hortonworks' industry-leading big data cloud service.
- **Azure Stream Analytics:** This is a fully managed, real-time, data analytics service that helps in performing computation and transformation on streaming data. Stream Analytics can examine high volumes of data flowing from devices or processes, extract information from the data stream, and look for patterns, trends, and relationships.
- **Machine Learning:** Machine learning is a data science technique that allows computers to use existing data to forecast future behaviors, outcomes, and trends. Using machine learning, computers learn behaviors based on the model we create. Azure Machine Learning is a cloud-based predictive analytics service that makes it possible to quickly create and deploy predictive models as analytics solutions. It provides a ready-to-use library of algorithms to create models on an internet-connected PC and deploy predictive solutions quickly.
- **Azure Synapse Analytics:** Formerly known as Azure Data Warehouse. Azure Synapse Analytics provides analytics services that are ideal for enterprise data warehousing and big data analytics. It supports direct streaming ingestion, which can be integrated with Azure IoT Hub.

Now that you are familiar with the transformation and analytics tools used in Azure for the data ingested by IoT devices, let's go ahead and learn about how this data can be presented.

Presentation

After appropriate analytics have been conducted on data, the data should be presented to stakeholders in a format that is consumable by them. There are numerous ways in which insights from data can be presented. These include presenting data through web applications that are deployed using Azure App Service, sending data to notification hubs that can then notify mobile applications, and more. However, the ideal approach for presenting and consuming insights is using **Power BI** reports and dashboards. Power BI is a Microsoft visualization tool that is used to render dynamic reports and dashboards on the internet.

To conclude, Azure IoT is closely aligned with the basic concepts of IoT architecture. It follows the same process; however, Azure gives us the freedom to choose different services and dependencies based on our requirements. In the next section, we will focus on Azure IoT Hub, a service hosted in the cloud and completely managed by Azure.

Azure IoT Hub

IoT projects are generally complex in nature. The complexity arises because of the high volume of devices and data. Devices are embedded across the world, for example, monitoring and auditing devices that are used to store data, transform and analyze petabytes of data, and finally take actions based on insights. Moreover, these projects have long gestation periods, and their requirements keep changing because of timelines.

If an enterprise wants to embark on a journey with an IoT project sooner rather than later, then it will quickly realize that the problems we have mentioned are not easily solved. These projects require enough hardware in terms of computing and storage to cope, and services that can work with high volumes of data.

IoT Hub is a platform that is built to enable the faster, better, and easier delivery of IoT projects. It provides all the necessary features and services, including the following:

- Device registration
- Device connectivity
- Field gateways
- Cloud gateways
- Implementation of industry protocols, such as AMQP and the MQTT protocol
- A hub for storing incoming messages
- The routing of messages based on message properties and content
- Multiple endpoints for different types of processing
- Connectivity to other services on Azure for real-time analytics and more

We have covered an overview of Azure IoT Hub, so let's take a deep dive to understand more about the protocols and how the devices are registered with Azure IoT Hub.

Protocols

Azure IoT Hub natively supports communication over the MQTT, AMQP, and HTTP protocols. In some cases, devices or field gateways might not be able to use one of these standard protocols and will require protocol adaptation. In such cases, custom gateways can be deployed. The Azure IoT protocol gateway can enable protocol adaptation for IoT Hub endpoints by bridging the traffic to and from IoT Hub. In the next section, we will discuss how devices are registered with Azure IoT Hub.

Device registration

Devices should be registered before they can send messages to IoT Hub. The registration of devices can be done manually using the Azure portal or it can be automated using the IoT Hub SDK. Azure also provides sample simulation applications, through which it becomes easy to register virtual devices for development and testing purposes. There is also a Raspberry Pi online simulator that can be used as a virtual device, and then, obviously, there are other physical devices that can be configured to connect to IoT Hub.

If you want to simulate a device from a local PC that is generally used for development and testing purposes, then there are tutorials available in the Azure documentation in multiple languages. These are available at <https://docs.microsoft.com/azure/iot-hub/iot-hub-get-started-simulated>.

The Raspberry Pi online simulator is available at <https://docs.microsoft.com/azure/iot-hub/iot-hub-raspberry-pi-web-simulator-get-started>, and for physical devices that need to be registered with IoT Hub, the steps given at <https://docs.microsoft.com/azure/iot-hub/iot-hub-get-started-physical> should be used.

To manually add a device using the Azure portal, IoT Hub provides the **IoT devices** menu, which can be used to configure a new device. Selecting the **New** option will let you create a new device as shown in Figure 17.4:

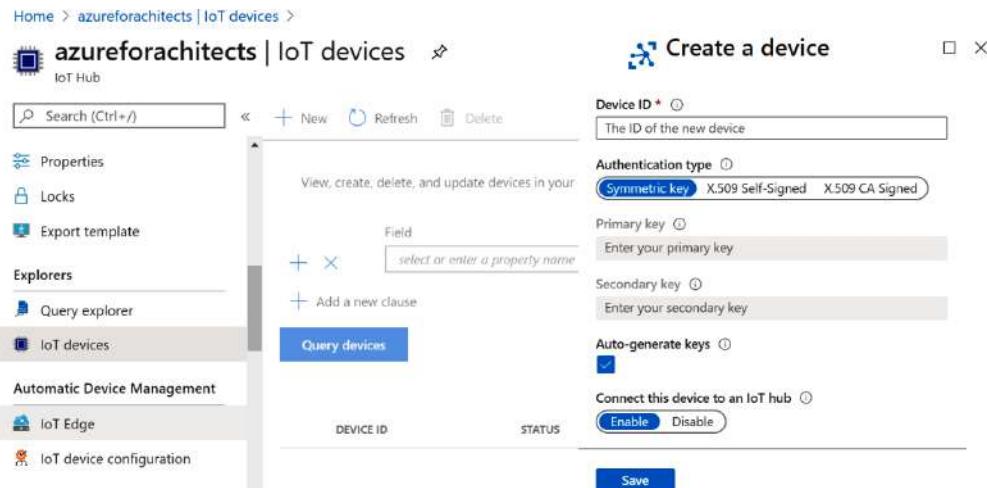


Figure 17.4: Adding a device via the Azure portal

After the device identity is created, a primary key connection string for IoT Hub should be used in each device to connect to it:

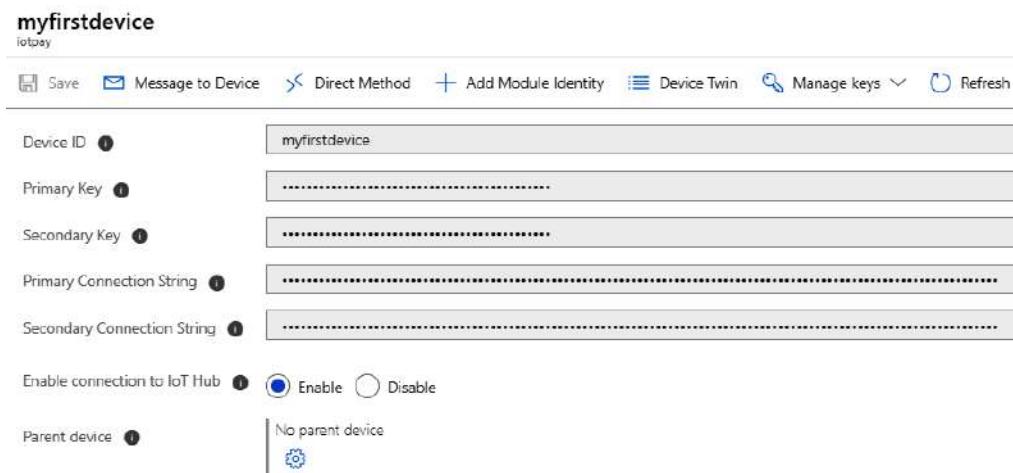


Figure 17.5: Creating connecting strings for each device

At this stage, the device has been registered with IoT Hub, and our next mission is to make communication happen between the device and IoT Hub. The next section will give you a good understanding of how message management is done.

Message management

After devices are registered with IoT Hub, they can start interacting with it. Message management refers to how the communication or interaction between the IoT device and IoT Hub is done. This interaction could be from the device to the cloud, or from the cloud to the device.

Device-to-cloud messaging

One of the best practices that must be followed in this communication is that although the device might be capturing a lot of information, only data that is of any importance should be transmitted to the cloud. The size of the message is very important in IoT solutions since IoT solutions generally have very high volumes of data. Even 1 KB of extra data flowing in can result in a GB of storage and processing wasted. Each message has properties and payloads; properties define the metadata for the message. This metadata contains data about the device, identification, tags, and other properties that are helpful in routing and identifying messages.

Devices or cloud gateways should connect to IoT Hub to transfer data. IoT Hub provides public endpoints that can be utilized by devices to connect and send data. IoT Hub should be considered as the first point of contact for back-end processing. IoT Hub is capable of further transmission and routing of these messages to multiple services. By default, the messages are stored in event hubs. Multiple event hubs can be created for different kinds of messages. The built-in endpoints used by the devices to send and receive data can be seen in the **Built-in endpoints** blade in IoT Hub. Figure 17.6 shows how you can find the built-in endpoints:

The screenshot shows the 'iotpay | Built-in endpoints' blade in the Azure IoT Hub interface. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (Shared access policies, Pricing and scale, IP Filter, Private endpoint connections, Certificates), and Built-in endpoints (which is selected). The main area displays configuration for an event hub named 'iotpay'. It includes fields for Partitions (set to 4), Event Hub-compatible name (set to 'iotpay'), Event Hub-compatible endpoint (set to 'Endpoint=sb://iothub-ns-iotpay-3451087-d32ba0084d.servicebus.windows.net/;SharedAccessKeyName=iothubowner;SharedAccessKey=...'), Retain for (set to 1 Days), Consumer Groups (Consumer Groups), and \$Default. There is also a 'Create new consumer group' button. Below this, the 'Cloud to device messaging' section is shown with fields for Default TTL (1 Hours), Feedback retention time (1 Hours), and Maximum delivery count (10 Attempts).

Figure 17.6: Creating multiple event hubs

Messages can be routed to different endpoints based on the message header and body properties, as shown in *Figure 17.7*:

The screenshot shows the 'Add a route' interface in the Azure portal. At the top, there's a breadcrumb navigation: All services > iotpay-516221011 | Overview > iotpay | Message routing > Add a route. Below the title 'Add a route' is a 'Name' input field with a placeholder 'name'. The 'Endpoint' field is expanded, showing a dropdown menu with 'Event hubs', 'Service bus queue', 'Service bus topic', and 'Storage'. A yellow box highlights the '+ Add endpoint' button. The 'Data source' field is set to 'Device Telemetry Messages'. Under 'Enable route', the 'Enable' button is selected. A note below says 'Create a query to filter messages before data is routed to an endpoint.' with a 'Learn more' link. The 'Routing query' field contains the value '1 true'.

Figure 17.7: Adding a new route to different endpoints

Messages in an IoT hub stay there for seven days by default, and their size can go up to 256 KB.

There is a sample simulator provided by Microsoft for simulating sending messages to the cloud. It is available in multiple languages; the C# version can be viewed at <https://docs.microsoft.com/azure/iot-hub/iot-hub-csharp-csharp-c2d>.

Cloud-to-device messaging

IoT Hub is a managed service providing a bi-directional messaging infrastructure. Messages can be sent from the cloud to devices, and then based on the message, the devices can act on them.

There are three types of cloud-to-device messaging patterns:

- Direct methods require immediate confirmation of results. Direct methods are often used for the interactive control of devices, such as opening and closing garage shutters. They follow the request-response pattern.
- Setting up device properties using Azure IoT provides **device twin** properties. For example, you can set the telemetry-sending interval to 30 minutes. Device twins are JSON documents that store device state information (such as metadata, configurations, and conditions). An IoT hub persists a device twin for each device in the IoT hub.

- Cloud-to-device messages are used for one-way notifications to the device app. This follows the fire-and-forget pattern.

In every organization, security is a big concern, and even in the case of IoT devices and data, this concern is still there. We will be discussing security in the next section.

Security

Security is an important aspect of IoT-based applications. IoT-based applications comprise devices that use the public internet for connectivity to back-end applications. Securing devices, back-end applications, and connectivity from malicious users and hackers should be considered a top priority for the success of these applications.

Security in IoT

IoT applications are primarily built around the internet, and security should play a major role in ensuring that a solution is not compromised. Some of the most important security decisions affecting IoT architecture are listed here:

- Regarding devices using HTTP versus HTTPS REST endpoints, REST endpoints secured by certificates ensure that messages transferred from a device to the cloud and vice versa are well encrypted and signed. The messages should make no sense to an intruder and should be extremely difficult to crack.
- If devices are connected to a local gateway, the local gateway should connect to the cloud using a secure HTTP protocol.
- Devices should be registered to IoT Hub before they can send any messages.
- The information passed to the cloud should be persisted into storage that is well protected and secure. Appropriate SAS tokens or connection strings that are stored in Azure Key Vault should be used for connection.
- Azure Key Vault should be used to store all secrets, passwords, and credentials, including certificates.
- Azure Security Center for IoT provides threat prevention and analysis for every device, IoT Edge, and IoT Hub across your IoT assets. We can build our own dashboards in Azure Security Center based on the security assessments. Some key features include central management from Azure Security Center, adaptive threat protection, and intelligent threat detection. It's a best practice to consider Azure Security Center while implementing secured IoT solutions.

Next, we are going to look at the scalability aspect of IoT Hub.

Scalability

Scalability for IoT Hub is a bit different than for other services. In IoT Hub, there are two types of messages:

- **Incoming:** Device-to-cloud messages
- **Outgoing:** Cloud-to-device messages

Both need to be accounted for in terms of scalability.

IoT Hub provides a couple of configuration options during provision time to configure scalability. These options are also available post-provisioning and can be updated to better suit the solution requirements in terms of scalability.

The scalability options that are available for IoT Hub are as follows:

- The **Stock Keeping Unit (SKU)** edition, which is the size of IoT Hub
- The number of units

We will first look into the SKU edition option.

The SKU edition

The SKU in IoT Hub determines the number of messages a hub can handle per unit per day, and this includes both incoming and outgoing messages. There are four tiers, as follows:

- **Free:** This allows 8,000 messages per unit per day and allows both incoming and outgoing messages. A maximum of 1 unit can be provisioned. This edition is suitable for gaining familiarity and testing out the capabilities of the IoT Hub service.
- **Standard (S1):** This allows 400,000 messages per unit per day and allows both incoming and outgoing messages. A maximum of 200 units can be provisioned. This edition is suitable for a small number of messages.
- **Standard (S2):** This allows 6 million messages per unit per day and allows both incoming and outgoing messages. A maximum of 200 units can be provisioned. This edition is suitable for a large number of messages.
- **Standard (S3):** This allows 300 million messages per unit per day and allows both incoming and outgoing messages. A maximum of 10 units can be provisioned. This edition is suitable for a very large number of messages.

The upgrade and scaling options are available in the Azure portal, under the **Pricing and scale** blade of IoT Hub. The options will be presented to you as shown in Figure 17.8:

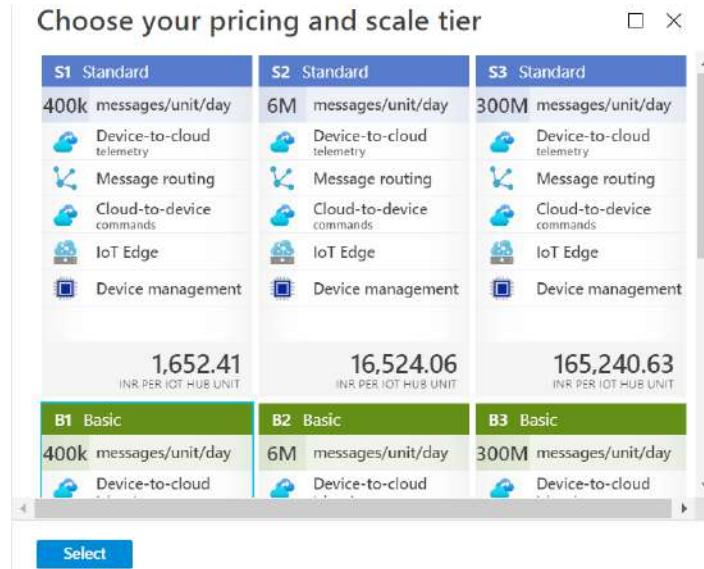


Figure 17.8: Choosing a pricing and scale tier

You may notice that the **Standard S3** tier allows a maximum of only **10 units**, compared to other standard units that allow **200 units**. This is directly related to the size of the compute resources that are provisioned to run IoT services. The size and capability of virtual machines for **Standard S3** are significantly higher compared to other tiers, where the size remains the same.

Units

Units define the number of instances of each SKU running behind the service. For example, 2 units of the **Standard S1** SKU tier will mean that the IoT hub is capable of handling $400K * 2 = 800K$ messages per day.

More units will increase the scalability of the application. Figure 17.9 is from the **Pricing and scale** blade of IoT Hub, where you can see the current pricing tier and number of units:

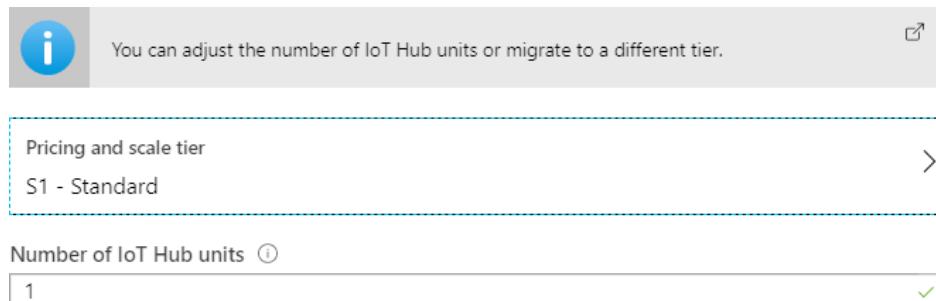


Figure 17.9: Options to adjust or migrate IoT Hub units

One of the booming services in Azure IoT Hub right now is Azure IoT Edge, which is a completely managed service built on Azure IoT Hub. We will be exploring what Azure IoT Edge is in the next section.

Azure IoT Edge

Microsoft Azure IoT Edge leverages edge compute to implement IoT solutions. Edge compute refers to the compute resources that are available on your on-premises network, right at the end of your network, where the public internet starts. This can be deployed on your main network or a guest network with firewall isolation.

Azure IoT Edge comprises the IoT Edge runtime, which needs to be installed on a computer or a device. Docker will be installed on the computer; the computer can run either Windows or Linux. The role of Docker is to run the IoT Edge modules.

Azure IoT Edge relies on the hybrid cloud concept, where you can deploy and manage IoT solutions on on-premises hardware and easily integrate them with Microsoft Azure.

Microsoft provides comprehensive documentation for Azure IoT Edge, with quick-start templates and guidance on how to install the modules. The link to the documentation is <https://docs.microsoft.com/azure/iot-edge>.

In the next section, we will look at how infrastructure is managed in the case of Azure IoT Hub and how high availability is provided to customers.

High availability

IoT Hub is a **platform as a service (PaaS)** offering from Azure. Customers and users do not directly interact with the underlying number and size of virtual machines on which the IoT Hub service runs. Users decide on the region, the SKU of the IoT hub, and the number of units for their application. The rest of the configuration is determined and executed by Azure behind the scenes. Azure ensures that every PaaS service is highly available by default. It does so by ensuring that multiple virtual machines provisioned behind the service are on separate racks in the datacenter. It does this by placing those virtual machines on an availability set and on a separate fault and update domain. This helps ensure high availability for both planned and unplanned maintenance. Availability sets take care of high availability at the datacenter level.

In the next section, we will discuss Azure IoT Central.

Azure IoT Central

Azure IoT Central provides a platform to build enterprise-grade IoT applications to meet your business requirements in a secure, reliable, and scalable fashion. IoT Central eliminates the cost of developing, maintaining, and managing IoT solutions.

IoT Central provides centralized management by which you can manage and monitor devices, device conditions, rule creation, and device data. In *Figure 17.10*, you can see some of the templates that are available in the Azure portal during the creation of IoT Central applications:

Home > IoT Central Applications >

- Custom application
- Custom application (legacy)
- In-store Analytics – Condition Monitor...
- Water Consumption Monitoring
- Digital Distribution Center
- Smart Inventory Management
- Connected Logistics
- Smart Meter Analytics
- Micro-fulfillment Center
- Continuous Patient Monitoring
- Solar Power Monitoring

Select a template

Learn more about application templates

Location *

Create Automation options

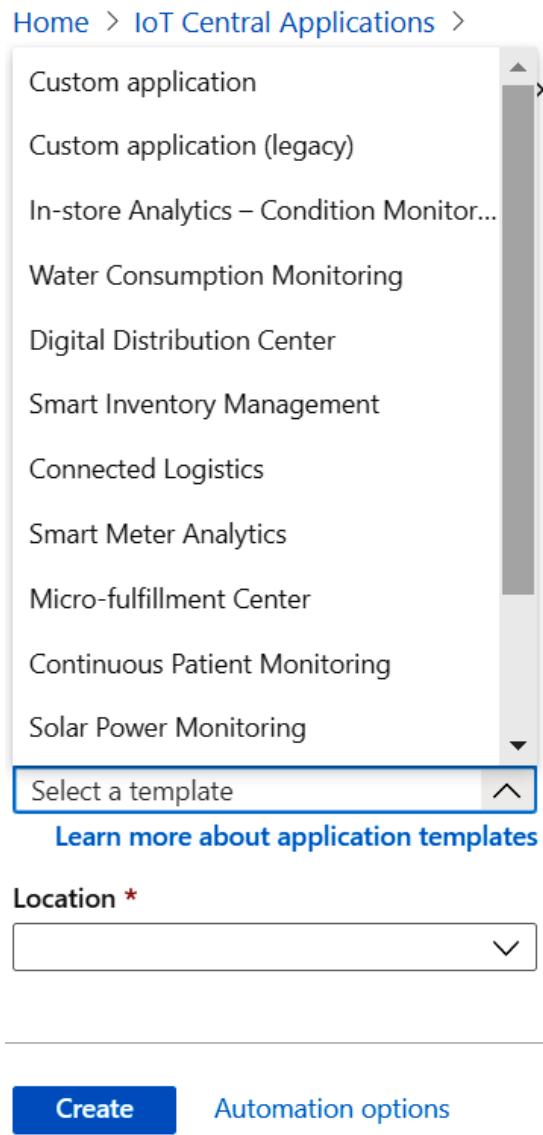
The screenshot shows the 'Create Application' interface in the Azure IoT Central portal. At the top, there's a breadcrumb navigation: 'Home > IoT Central Applications >'. Below this is a list of ten pre-defined application templates. The first two items in the list ('Custom application' and 'Custom application (legacy)') have a small 'X' icon to their right, indicating they are the most recent or recommended. The other eight items are listed below them. At the bottom of the template list is a blue rectangular button labeled 'Select a template' with a dropdown arrow icon to its right. Below this button is a link 'Learn more about application templates'. Further down, there's a section labeled 'Location *' with a dropdown menu. At the very bottom of the screen are two large, prominent buttons: a blue 'Create' button on the left and a light blue 'Automation options' button on the right.

Figure 17.10 Creating an Azure IoT Central application

Templates will give you a head start and you can customize them as per your requirements. This will save you a lot of time during the development phase.

IoT Central offers a seven-day trial at the time of writing, and you can see the pricing for this service here: <https://azure.microsoft.com/pricing/details/iot-central/?rtc=1>.

Azure IoT Central is a boon for every organization that is developing IoT applications.

Summary

IoT is one of the biggest emerging technologies of this decade and it is already disrupting industries. Things that sounded impossible before are now suddenly possible.

In this chapter, we explored IoT Hub and discussed the delivery of IoT solutions to the customer in a faster, better, and cheaper way than alternative solutions. We also covered how IoT can fast-track the entire development life cycle and help expedite time-to-market for companies. Finally, you learned about Azure IoT Edge and Azure IoT Central.

To help you effectively analyze ever-growing volumes of data, we will discuss Azure Synapse Analytics in the next chapter.

18

Azure Synapse Analytics for architects

Azure Synapse Analytics is a groundbreaking evolution of Azure SQL Data Warehouse. Azure Synapse is a fully managed, integrated data analytics service that blends data warehousing, data integration, and big data processing with accelerating time to insight to form a single service.

In this chapter, we will explore Azure Synapse Analytics by covering the following topics:

- An overview of Azure Synapse Analytics
- Introduction to Synapse workspaces and Synapse Studio
- Migrating from existing legacy systems to Azure Synapse Analytics
- Migrating existing data warehouse schemas and data to Azure Synapse Analytics
- Re-developing scalable ETL processes using Azure Data Factory
- Common migration issues and resolutions
- Security considerations
- Tools to help migrate to Azure Synapse Analytics

Azure Synapse Analytics

Nowadays, with inexpensive storage and high elastic storage capacities, organizations are amassing more data than ever before. Architecting a solution to analyze such massive volumes of data to deliver meaningful insights about a business can be a challenge. One obstacle that many businesses face is the need to manage and maintain two types of analytics systems:

- **Data warehouses:** These provide critical insights about the business.
- **Data lakes:** These provide meaningful insights about customers, products, employees, and processes through various analytics methodologies.

Both of these analytics systems are critical to businesses, yet they operate independently of one another. Meanwhile, businesses need to gain insights from all their organizational data in order to stay competitive and to innovate processes to obtain better results.

For architects who need to build their own end-to-end data pipelines, the following steps must be taken:

1. Ingest data from various data sources.
2. Load all these data sources into a data lake for further processing.
3. Perform data cleaning over a range of different data structures and types.
4. Prepare, transform, and model the data.
5. Serve the cleansed data to thousands of users through BI tools and applications.

Until now, each of these steps has required a different tool. Needless to say, with so many different services, applications, and tools available on the market, choosing the best-suited ones can be a daunting task.

There are numerous services available for ingesting, loading, preparing, and serving data. There are countless services for data cleansing based on the developer's language of choice. Furthermore, some developers might prefer to use SQL, some might want to use Spark, while others might prefer to use code-free environments to transform data.

Even after the seemingly proper collection of tools has been selected, there is often a steep learning curve for these tools. Additionally, architects could encounter unexpected logistical challenges in maintaining a data pipeline over dissimilar platforms and languages due to incompatibilities. With such a range of issues, implementing and maintaining a cloud-based analytics platform can be a difficult task.

Azure Synapse Analytics solves these problems and more. It simplifies the entire modern data warehouse pattern, allowing architects to focus on building end-to-end analytics solutions within a unified environment.

A common scenario for architects

One of the most common scenarios that an architect faces is having to conjure up a plan for migrating existing legacy data warehouse solutions to a modern enterprise analytics solution. With its limitless scalability and unified experience, Azure Synapse has become one of the top choices for many architects to consider. Later in this chapter, we will also discuss common architectural considerations for migrating from an existing legacy data warehouse solution to Azure Synapse Analytics.

In the next section, we will provide a technical overview of the key features of Azure Synapse Analytics. Architects who are new to the Azure Synapse ecosystem will gain the necessary knowledge about Synapse after reading this chapter.

An overview of Azure Synapse Analytics

Azure Synapse Analytics enables data professionals to build end-to-end analytics solutions while leveraging a unified experience. It delivers rich functionalities for SQL developers, serverless on-demand querying, machine learning support, the ability to embed Spark natively, collaborative notebooks, and data integration within a single service. Developers can choose from a variety of supported languages (for example, C#, SQL, Scala, and Python) through different engines.

Some of the main capabilities of Azure Synapse Analytics include:

- SQL Analytics with pools (fully provisioned) and on-demand (serverless).
- Spark with full support for Python, Scala, C#, and SQL.
- Data Flow with code-free big data transformation experience.
- Data integration and orchestration to integrate data and operationalize code development.
- A cloud-native version of **Hybrid Transactional/Analytical Processing (HTAP)**, delivered by Azure Synapse Link.

To access all of the aforementioned capabilities, Azure Synapse Studio provides a single unified web UI.

This single integrated data service is advantageous to enterprises as it accelerates the delivery of BI, AI, machine learning, Internet of Things, and intelligent applications.

Azure Synapse Analytics can derive and deliver insights from all your data residing in the data warehouse and big data analytics systems at lightning-fast speeds. It enables data professionals to use familiar languages, such as SQL, to query both relational and non-relational databases at petabyte scale. In addition, advanced features such as limitless concurrency, intelligent workload management, and workload isolation help optimize the performance of all queries for mission-critical workloads.

What is workload isolation?

One of the key features of running enterprise data warehouses at scale is workload isolation. This is the ability to guarantee resource reservations within a compute cluster so that multiple teams can work on the data without getting in each other's way, as illustrated in Figure 18.1:

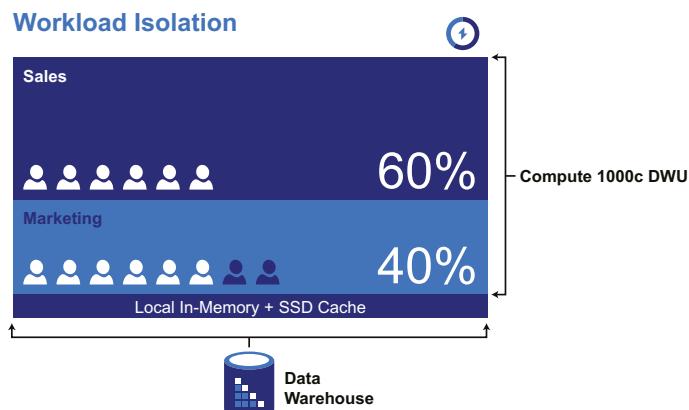


Figure 18.1: Example of workload isolation

You can create workload groups within a cluster by setting a couple of simple thresholds. These are automatically adjusted depending on the workload and the cluster, but they always guarantee a quality experience for users running the workloads. Refer to <https://techcommunity.microsoft.com/t5/data-architecture-blog/configuring-workload-isolation-in-azure-synapse-analytics/ba-p/1201739> to read more about configuring workload isolation in Azure Synapse Analytics.

To fully appreciate the benefits of Azure Synapse, we will first take a look at Synapse workspaces and Synapse Studio.

Introduction to Synapse workspaces and Synapse Studio

At the heart of Azure Synapse is the workspace. The workspace is the top-level resource that comprises your analytics solution in a data warehouse. The Synapse workspace supports both relational and big data processing.

Azure Synapse provides a unified web UI experience for data preparation, data management, data warehousing, big data analytics, BI, and AI tasks known as Synapse Studio. Together with Synapse workspaces, Synapse Studio is an ideal environment for data engineers and data scientists to share and collaborate their analytics solutions, as shown in Figure 18.2:

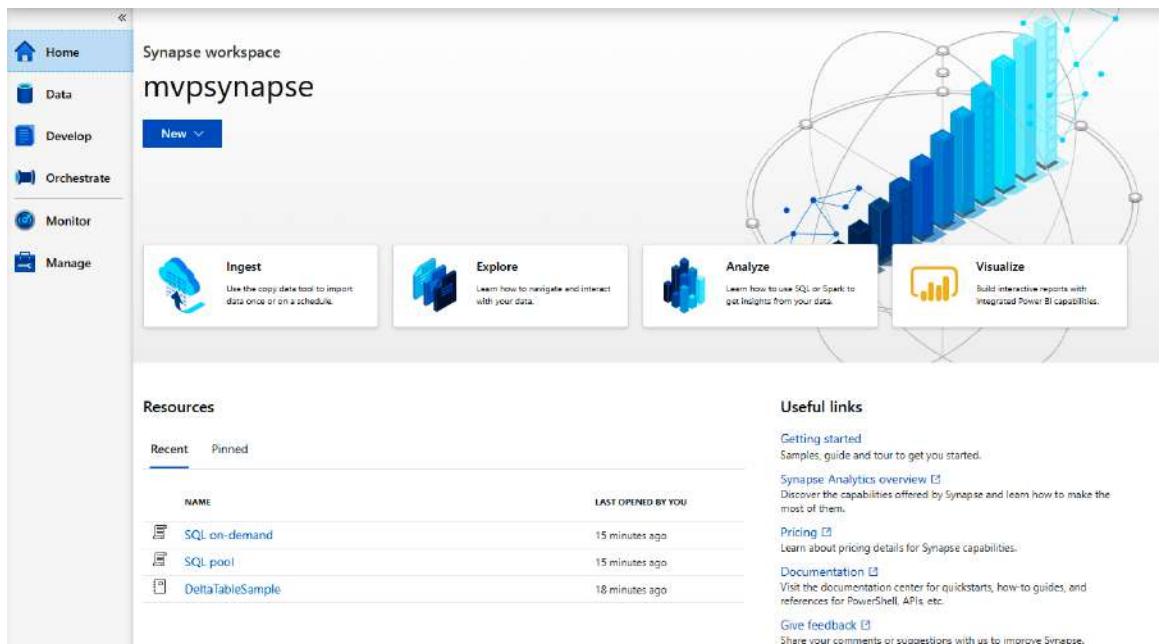


Figure 18.2: A Synapse workspace in Azure Synapse Studio

The following sections highlight the capabilities, key features, platform details, and end user services of Synapse workspaces and Synapse Studio:

Capabilities:

- A fast, highly elastic, and secure data warehouse with industry-leading performance and security
- The ability to explore Azure Data Lake Storage and data warehouses using familiar T-SQL syntax using SQL on-demand (serverless) and SQL queries
- Apache Spark integrated with Azure Machine Learning
- Hybrid data integration to accelerate data ingestion and the operationalization of the analytics process (ingest, prepare, transform, and serve)
- Business report generation and serving with Power BI integration

Key features:

- Create and operationalize pipelines for data ingestion and orchestration.
- Directly explore data in your Azure Data Lake Storage or data warehouse, as well as any external connections to the workspace, using Synapse Studio.
- Write code using notebooks and T-SQL query editors.
- Code-free data transformation tool, if you prefer not to write your own code.
- Monitor, secure, and manage your workspaces without leaving the environment.
- Web-based development experience for the entire analytics solution.
- The backup and restore feature in the Azure Synapse SQL pool allows restore points to be created to make it easy to recover or copy a data warehouse to a previous state.
- The ability to run concurrent T-SQL queries through SQL pools across petabytes of data to serve BI tools and applications.
- SQL on-demand provides serverless SQL queries for ease of exploration and data analysis in Azure Data Lake Storage without any setup or maintenance of infrastructure.
- Meets the full range of analytics needs, from data engineering to data science, using a variety of languages, such as Python, Scala, C#, and Spark SQL.
- Spark pools, which alleviate the complex setup and maintenance of clusters and simplify the development of Spark applications and usage of Spark notebooks.

- Offers deep integration between Spark and SQL, allowing data engineers to prepare data in Spark, write the processed results in SQL Pool, and use any combination of Spark with SQL for data engineering and analysis, with built-in support for Azure Machine Learning.
- Highly scalable, hybrid data integration capability that accelerates data ingestion and operationalization through automated data pipelines.
- Provides a friction-free integrated service with unified security, deployment, monitoring, and billing.

Platform

- Supports both provisioned and serverless compute. Examples of provisioned compute include SQL compute and Spark compute.
- Provisioned compute allows teams to segment their compute resources so that they can control cost and usage to better align with their organizational structure.
- Serverless compute, on the other hand, allows teams to use the service on-demand without provisioning or managing any underlying infrastructure.
- Deep integration between Spark and SQL engines.

In the following section, we will cover the other features of Azure Synapse, including Apache Spark for Synapse, Synapse SQL, SQL on-demand, Synapse pipelines, and Azure Synapse Link for Cosmos DB.

Apache Spark for Synapse

For customers who want Apache Spark, Azure Synapse has first-party support through Azure Databricks and is fully managed by Azure. The latest version of Apache Spark will automatically be made available to users, along with all security patches. You can quickly create notebooks with your choice of language, such as Python, Scala, Spark SQL, and .NET for Spark.

If you use Spark within Azure Synapse Analytics, it is provided as a Software as a Service offering. For example, you can use Spark without setting up or managing your own services, such as a virtual network. Azure Synapse Analytics will take care of the underlying infrastructure for you. This allows you to use Spark immediately in your Azure Synapse Analytics environment.

In the next section, we will explore Synapse SQL.

Synapse SQL

Synapse SQL allows the use of T-SQL to query and analyze data. There are two models to choose from:

1. Fully provisioned model
2. SQL on-demand (serverless) model

SQL on-demand

SQL on-demand provides serverless SQL queries. This allows easier exploration and data analysis in Azure Data Lake Storage without any setup or infrastructure maintenance:

Traditional IT	IaaS	PaaS	Serverless	SaaS
Application	Application	Application	Application	Application
Data	Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware	Middleware
OS	OS	OS	OS	OS
Virtualization	Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking	Networking



You Manage

Table 18.1: Comparison between different infrastructures

Key Features:

- Analysts can focus on analyzing the data without worrying about managing any infrastructure.
- Customers can benefit from a simple and flexible pricing model, as they only pay for what they use.

- It uses the familiar T-SQL language syntax and the best SQL Query Optimizer on the market. The SQL Query Optimizer is the brain behind the query engine.
- You can easily scale your compute and storage, independently of one another, as your needs grow.
- Seamlessly integrate with SQL Analytics Pool and Spark via metadata sync and native connectors.

Synapse pipelines

Synapse pipelines allow developers to build end-to-end workflows for data movement and data processing scenarios. Azure Synapse Analytics uses the **Azure Data Factory (ADF)** technology to provide data integration features. The key features of ADF that are essential to the modern data warehouse pipeline are available in Azure Synapse Analytics. All these features are wrapped with a common security model, **Role-Based Access Control (RBAC)**, in the Azure Synapse Analytics workspace.

Figure 18.3 shows an example of a data pipeline and the activities from ADF that are directly integrated inside the Azure Synapse Analytics environment:

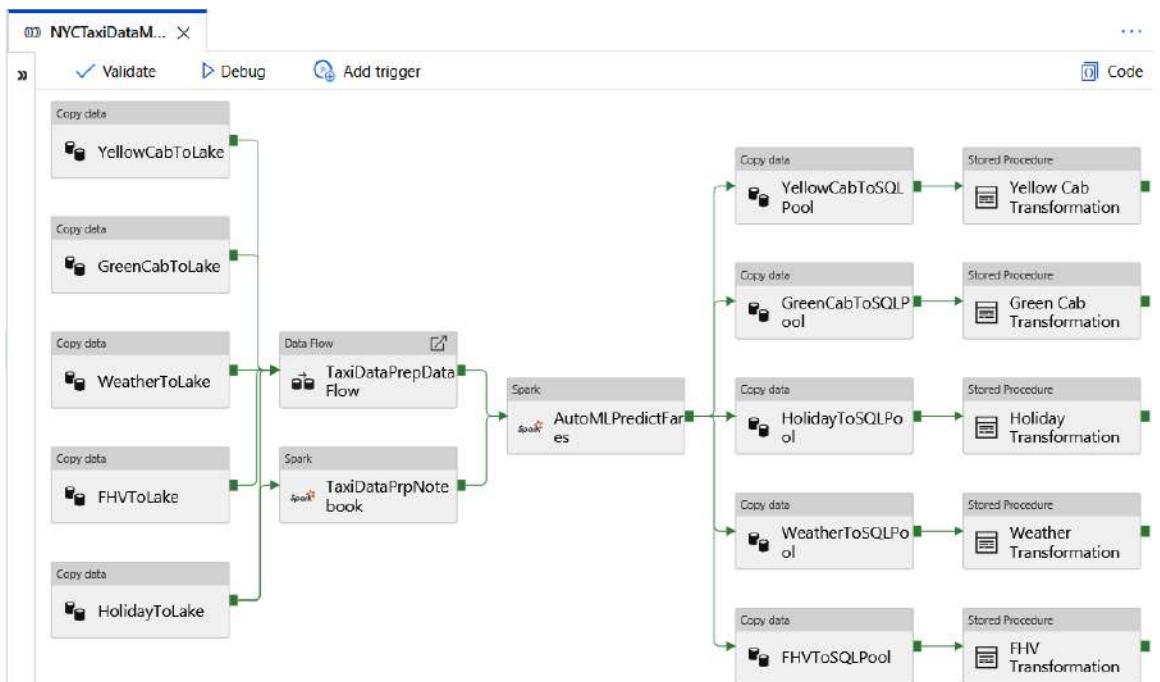


Figure 18.3: Data pipelines in Azure Synapse Analytics

Key Features:

- Integrated platform services for management, security, monitoring, and metadata management.
- Native integration between Spark and SQL. Use a single line of code to read and write with Spark from/into SQL analytics.
- The ability to create a Spark table and query it instantaneously with SQL Analytics without defining a schema.
- "Key-free" environment. With Single Sign-On and Azure Active Directory pass-through, no key or login is needed to interact with **Azure Data Lake Storage (ADLS)**/databases.

In the next section, we will cover Azure Synapse Link for Cosmos DB.

Azure Synapse Link for Cosmos DB

Azure Synapse Link is a cloud-native version of HTAP. It is an extension of Azure Synapse. As we have learned earlier, Azure Synapse is a single managed service for performing analytics over data lakes and data warehouses, using both serverless and provisioned compute. With Azure Synapse Link, this reach can be extended to operational data sources as well.

Azure Synapse Link eliminates the bottleneck that is found in traditional operational and analytical systems. Azure Synapse makes this possible by separating compute from storage across all of its data services. On the transactional side, Cosmos DB is a high-performance, geo-replicated, multi-model database service. On the analytics side, Azure Synapse provides limitless scalability. You can scale the resources for transactions and for analytics independently. Together, this makes cloud-native HTAP a reality. As soon as the user indicates what data in Cosmos DB they wish to make available for analytics, the data becomes available in Synapse. It takes the operational data you wish to analyze and automatically maintains an analytics-oriented columnar version of it. As a result, any changes to the operational data in Cosmos DB are continuously updated to the Link data and Synapse.

The biggest benefit of using Azure Synapse Link is that it alleviates the need for scheduled batch processing or having to build and maintain operational pipelines.

As mentioned previously, Azure Synapse is the most chosen platform by architects for migrating existing legacy data warehouse solutions to a modern enterprise analytics solution. In the next section, we will discuss common architectural considerations for migrating from an existing legacy data warehouse solution to Azure Synapse Analytics.

Migrating from existing legacy systems to Azure Synapse Analytics

Today, many organizations are migrating their legacy data warehouse solutions to Azure Synapse Analytics to gain the benefits of the high availability, security, speed, scalability, cost savings, and performance of Azure Synapse.

For companies running legacy data warehouse systems such as Netezza, the situation is even more dire because IBM has announced the end of support for Netezza (<https://www.ibm.com/support/pages/end-support-dates-netezza-5200-netezza-8x50z-series-and-netezza-10000-series-appliances>).

Many decades ago, some companies chose Netezza to manage and analyze large volumes of data. Today, as technologies evolve, the benefits of having a cloud-based data warehouse solution far outweigh the on-premises counterparts. Azure Synapse is a limitless cloud-based analytics service with unmatched time to insight that accelerates the delivery of BI, AI, and intelligent applications for enterprises. With its multi-cluster and separate compute and storage architecture, Azure Synapse can be scaled instantly in ways not possible with legacy systems such as Netezza.

This section covers the architectural considerations and high-level methodology for planning, preparing, and executing a successful migration of an existing legacy data warehouse system to Azure Synapse Analytics. Whenever appropriate, specific examples and references to Netezza will be given. This chapter is not intended to be a comprehensive step-by-step manual for migration, but rather a practical overview to help with your migration planning and project scoping.

This chapter also identifies some of the common migration issues and possible resolutions. It also provides technical details on the differences between Netezza and Azure Synapse Analytics. They should be taken into consideration as part of your migration plan.

Why you should migrate your legacy data warehouse to Azure Synapse Analytics

By migrating to Azure Synapse Analytics, companies with legacy data warehouse systems can take advantage of the latest innovations in cloud technologies and delegate tasks such as infrastructure maintenance and platform upgrading to Azure.

Customers who have migrated to Azure Synapse are already reaping many of its benefits, including the following.

Performance

Azure Synapse Analytics offers best-of-breed relational database performance by using techniques such as **Massively Parallel Processing (MPP)** and automatic in-memory caching. For more information, please review the Azure Synapse Analytics architecture (<https://docs.microsoft.com/azure/synapse-analytics/sql-data-warehouse/massively-parallel-processing-mpp-architecture>).

Speed

Data warehousing is process intensive. It involves data ingestion, transforming data, cleansing data, aggregating data, integrating data, and producing data visualization and reports. The many processes involved in moving data from original sources to a data warehouse are complex and interdependent. A single bottleneck can slow the entire pipeline and an unexpected spike in data volume amplifies the need for speed. When timeliness of data matters, Azure Synapse Analytics meets the demand for fast processing.

Improved security and compliance

Azure is a globally available, highly scalable, secure cloud platform. It offers many security features, including Azure Active Directory, RBAC, managed identities, and managed private endpoints. Azure Synapse Analytics, which resides inside the Azure ecosystem, inherits all of the aforementioned benefits.

Elasticity and cost efficiencies

In a data warehouse, the demands for workload processing can fluctuate. At times, these fluctuations can vary drastically between peaks and valleys. For example, sudden spikes in sales data volumes can occur during holiday seasons. Cloud elasticity allows Azure Synapse to quickly increase and decrease its capacity according to demand with no impact upon infrastructure availability, stability, performance, and security. Best of all, you only pay for your actual usage.

Managed infrastructure

Eliminating the overhead of data center management and operations for the data warehouse allows companies to reallocate valuable resources to where value is produced and focus on using the data warehouse to deliver the best information and insight. This lowers the overall total cost of ownership and provides better cost control over your operating expenses.

Scalability

The volume of data in a data warehouse typically grows as time passes and as history is collected. Azure Synapse Analytics can scale to match this growth by incrementally adding resources as data and workload increase.

Cost savings

Running an on-premises legacy datacenter is expensive (considering the costs of servers and hardware, networking, physical room space, electricity, cooling, and staffing). These expenses can be substantially minimized with Azure Synapse Analytics. With the separation of the compute and storage layers, Azure Synapse offers a very lucrative price-performance ratio.

Azure Synapse Analytics provides you with true pay-as-you-go cloud scalability without the need for complicated reconfiguration as your data or workloads grow.

Now that you have learned why it is beneficial to migrate to Azure Synapse Analytics, we will begin our discussion of the migration process.

The three-step migration process

A successful data migration project starts with a well-designed plan. An effective plan accounts for the many components that need to be considered, paying particular attention to architecture and data preparation. The following is the three-step migration process plan.

Preparation

- Define the scope of what is to be migrated.
- Build an inventory of data and processes for migration.
- Define the data model changes (if any).
- Define the source data extraction mechanism.
- Identify suitable Azure (and third-party) tools and services to be used.
- Train staff early on the new platform.
- Set up the Azure target platform.

Migration

- Start small and simple.
- Automate wherever possible.
- Leverage Azure built-in tools and features to reduce migration effort.
- Migrate metadata for tables and views.
- Migrate historical data to be maintained.
- Migrate or refactor stored procedures and business processes.
- Migrate or refactor ETL/ELT incremental load processes.

Post-migration

- Monitor and document all stages of the process.
- Use the experience gained to build a template for future migrations.
- Re-engineer the data model if required.
- Test applications and query tools.
- Benchmark and optimize query performance.

Next, we will talk about the two types of migration strategies.

The two types of migration strategies

Architects should begin migration planning by assessing the existing data warehouse to determine which migration strategy works best for their situation. There are two types of migration strategies to consider.

Lift and Shift strategy

For the lift and shift strategy, the existing data model is migrated unchanged to the new Azure Synapse Analytics platform. This is done to minimize the risk and the time required for migration by reducing the scope of changes to the minimum.

Lift and shift is a good strategy for legacy data warehouse environments such as Netezza where any one of the following conditions applies:

- A single data mart is to be migrated.
- The data is already in a well-designed star or snowflake schema.
- There are immediate time and cost pressures to move to a modern cloud environment.

Redesign strategy

In scenarios where the legacy data warehouse has evolved over time, it might be essential to re-engineer it to maintain the optimum performance levels or support new types of data. This could include a change in the underlying data model.

To minimize risk, it is recommended to migrate first using the lift and shift strategy and then gradually modernize the data warehouse data model on Azure Synapse Analytics using the redesign strategy. A complete change in data model will increase risks because it will impact source-to-data warehouse ETL jobs and downstream data marts.

In the next section, we will offer some recommendations on how to reduce the complexity of your existing legacy data warehouse before migrating.

Reducing the complexity of your existing legacy data warehouse before migrating

In the previous section, we presented the two migration strategies. As a best practice, during the initial assessment step, be cognizant of any ways to simplify your existing data warehouse and document them. The goal is to reduce the complexity of your existing legacy data warehouse system before the migration to make the migration process easier.

Here are some recommendations on how to reduce the complexity of your existing legacy data warehouse:

- **Remove and archive unused tables before migrating:** Avoid migrating data that is no longer in use. This will help reduce the overall data volume to migrate.
- **Convert physical data marts to virtual data marts:** Minimize what you have to migrate, reduce the total cost of ownership, and improve agility.

In the next section, we will take a closer look at why you should consider converting a physical data mart to a virtual data mart.

Converting physical data marts to virtual data marts

Prior to migrating your legacy data warehouse, consider converting your current physical data marts to virtual data marts. By using virtual data marts, you can eliminate physical data stores and ETL jobs for data marts without losing any functionality prior to migration. The goal here is to reduce the number of data stores to migrate, reduce copies of data, reduce the total cost of ownership, and improve agility. To achieve this, you will need to switch from physical to virtual data marts before migrating your data warehouse. We can consider this as a data warehouse modernization step prior to migration.

Disadvantages of physical data marts

- Multiple copies of the same data
- Higher total cost of ownership
- Difficult to change as ETL jobs are impacted

Advantages of virtual data marts

- Simplifies data warehouse architecture
- No need to store copies of data
- More agility
- Lower total cost of ownership
- Uses pushdown optimization to leverage the power of Azure Synapse Analytics
- Easy to change
- Easy to hide sensitive data

In the next section, we will talk about how to migrate existing data warehouse schemas to Azure Synapse Analytics.

Migrating existing data warehouse schemas to Azure Synapse Analytics

Migrating the schemas of an existing legacy data warehouse involves the migration of existing staging tables, legacy data warehouse, and dependent data mart schemas.

To help you understand the magnitude and scope of your schema migration, we recommend that you create an inventory of your existing legacy data warehouse and data mart.

Here is a checklist to help you collect the necessary information:

- Row counts
- Staging, data warehouse, and data mart data size: tables and indexes
- Data compression ratios
- Current hardware configuration
- Tables (including partitions): identify small dimension tables
- Data types
- Views
- Indexes
- Object dependencies
- Object usage

- Functions: both out-of-the-box functions and **User-Defined Functions (UDFs)**
- Stored procedures
- Scalability requirements
- Growth projections
- Workload requirements: Concurrent users

With your inventory completed, you can now make decisions on scoping what schema you want to migrate. Essentially, there are four options for scoping your legacy data warehouse schema migration:

1. Migrate one data mart at a time:

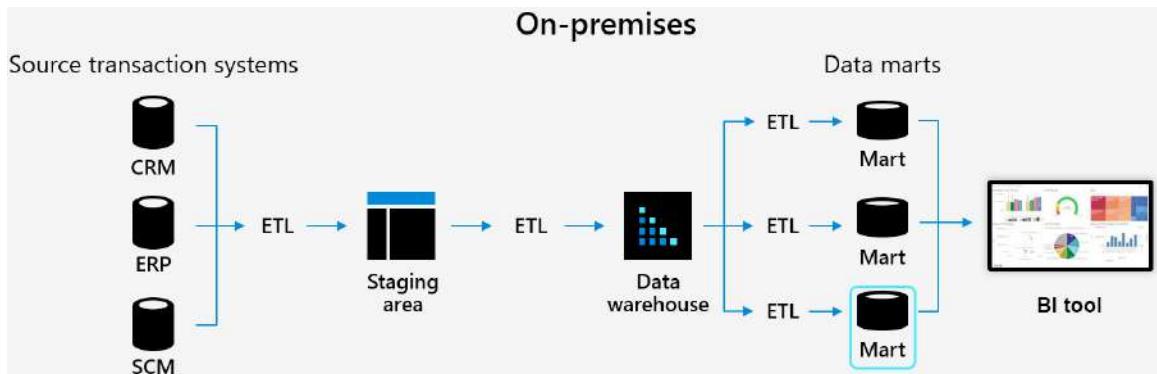


Figure 18.4: Migrating one data mart at a time

2. Migrate all data marts at once, then the data warehouse:

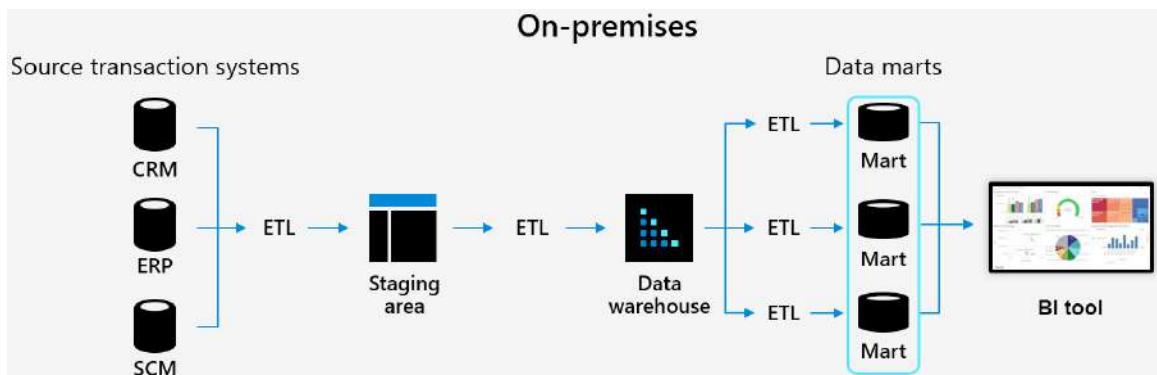


Figure 18.5: Migrating all data marts at once, then the data warehouse

3. Migrate both the data warehouse and the staging area:

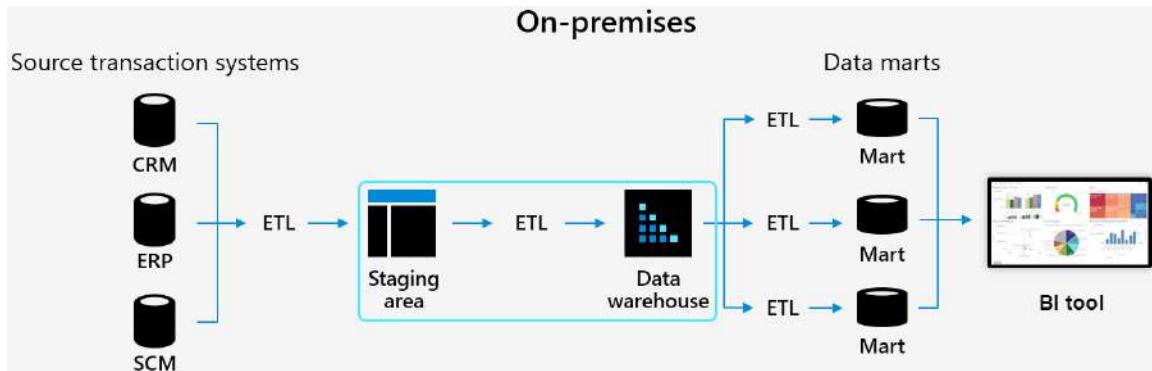


Figure 18.6: Migrating both the data warehouse and the staging area

4. Migrate everything at once:

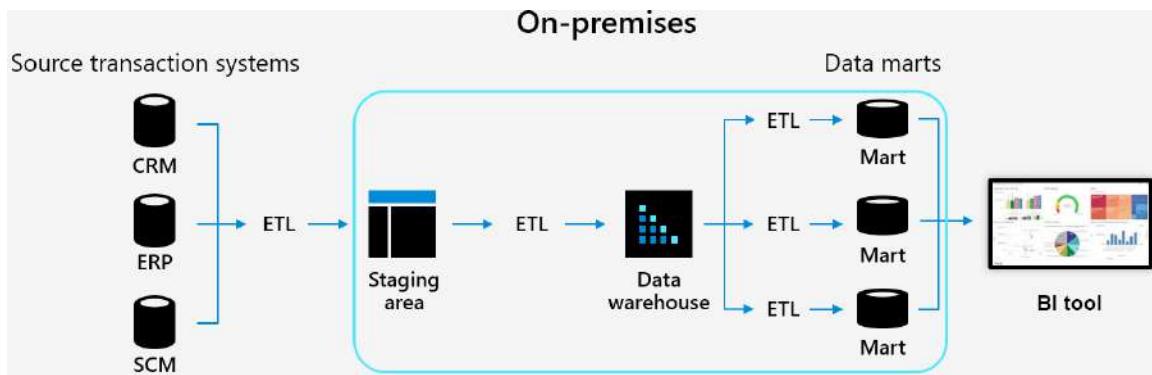


Figure 18.7: Migrating everything at once

Keep in mind when choosing your option that the goal is to achieve a physical database design that will match or exceed your current legacy data warehouse system in performance and preferably at a lower cost.

To recap, here are some of the recommendations for the schema migration:

- Avoid migrating unnecessary objects or processes.
- Consider using virtual data marts to reduce or eliminate the number of physical data marts.
- Automate whenever possible. Implementation of DataOps should be considered alongside the migration to Azure Synapse.
- Use metadata from system catalog tables in the legacy data warehouse system to generate **Data Definition Language (DDL)** for Azure Synapse Analytics.
- Perform any required data model changes or data mapping optimizations on Azure Synapse Analytics.

In the next section, we will talk about how to migrate historical data from a legacy data warehouse to Azure Synapse Analytics.

Migrating historical data from your legacy data warehouse to Azure Synapse Analytics

Once the schema migration scope has been determined, we are now ready to make decisions on how to migrate the historical data.

The steps for migrating historical data are as follows:

1. Create target tables on Azure Synapse Analytics.
2. Migrate existing historical data.
3. Migrate functions and stored procedures as required.
4. Migrate incremental load (ETL/ELT) staging and processes for incoming data.
5. Apply any performance tuning options that are required.

Table 18.2 outlines the four data migration options and their pros and cons:

Data migration option	Pros	Cons
Migrate data mart data first, followed by data warehouse data	<ul style="list-style-type: none"> Migrating data one data mart at a time is an incremental, low-risk approach. It will also provide faster proof of business case to departmental analytics end users. Subsequent ETL migration is limited to only the data in the dependent data marts migrated. 	<ul style="list-style-type: none"> Until your migration is complete, you will have some data that exists on-premises and on Azure. ETL processing from the data warehouse to data marts would need to bridge the firewall and be changed to target Azure Synapse.
Migrate data warehouse data first, followed by data marts	<ul style="list-style-type: none"> All data warehouse historical data is migrated. 	<ul style="list-style-type: none"> Leaving dependent data marts on-premises is not ideal as ETLs need to flow data back into the datacenter. No real opportunity for incremental data migration.
Migrate data warehouse and data marts together	<ul style="list-style-type: none"> All data is migrated in one go. 	<ul style="list-style-type: none"> Potentially higher risk. ETLs will most likely all have to be migrated together.
Convert physical marts to virtual marts and only migrate the data warehouse	<ul style="list-style-type: none"> No data mart data stores to migrate. No ETLs from the data warehouse to the marts to migrate. Only the data warehouse data to migrate. Fewer copies of data. No loss in functionality. Lower total cost of ownership. More agility. Simpler overall data architecture. May be possible with views in Azure Synapse. 	<ul style="list-style-type: none"> If nested views are not capable of supporting virtual data marts, then third-party data virtualization software on Azure will likely be needed. All marts would need to be converted before data warehouse data is migrated. Virtual marts and data warehouse-to-virtual mart mappings will need to be ported to the data virtualization server on Azure and redirected to Azure Synapse.

Table 18.2: Data migration options with their pros and cons

In the next section, we will talk about how to migrate existing ETL processes to Azure Synapse Analytics.

Migrating existing ETL processes to Azure Synapse Analytics

There are a number of options available for migrating your existing ETL processes to Azure Synapse Analytics. Table 18.3 outlines some of the ETL migration options based on how the existing ETL jobs are built:

How are existing ETL jobs built?	Migration options	Why migrate and what to look out for
Custom 3GL code and scripts	<ul style="list-style-type: none"> Plan to re-develop these using ADF. 	<ul style="list-style-type: none"> Code provides no metadata lineage. Hard to maintain if authors have gone. If staging tables are in the legacy data warehouse and SQL is used to transform data, then resolve differences with T-SQL.
Stored procedures that run in your legacy data warehouse DBMS	<ul style="list-style-type: none"> Plan to re-develop these using ADF. 	<ul style="list-style-type: none"> Likely to be significant differences between the legacy data warehouse and Azure Synapse. No metadata lineage. This needs careful evaluation, but the key advantage could be the Pipeline as Code approach, which is possible with ADF.
Graphical ETL tool (such as Informatica or Talend)	<ul style="list-style-type: none"> Continue using your existing ETL tool and switch the target to Azure Synapse. Possibly move to an Azure version of your existing ETL tool and port the metadata to run ELT jobs on Azure making sure you enable access to on-premises data sources. Control the execution of ETL services using ADF. 	<ul style="list-style-type: none"> Avoids re-development. Minimizes risk and quicker to migrate.
Data warehouse automation software	<ul style="list-style-type: none"> Continue using your existing ETL tool, switching the target and staging to Azure Synapse. 	<ul style="list-style-type: none"> Avoids re-development. Minimizes risk and quicker to migrate.

Table 18.3: ETL migration options

In the next section, we will talk about how to re-develop scalable ETL processes using ADF.

Re-developing scalable ETL processes using ADF

Another option for handling your existing legacy ETL processes is by re-developing them using ADF. ADF is an Azure data integration service for creating data-driven workflows (known as pipelines) to orchestrate and automate data movement and data transformation. You can use ADF to create and schedule pipelines to ingest data from different data stores. ADF can process and transform data by using compute services such as Spark, Azure Machine Learning, Azure HDInsight Hadoop, and Azure Data Lake Analytics:

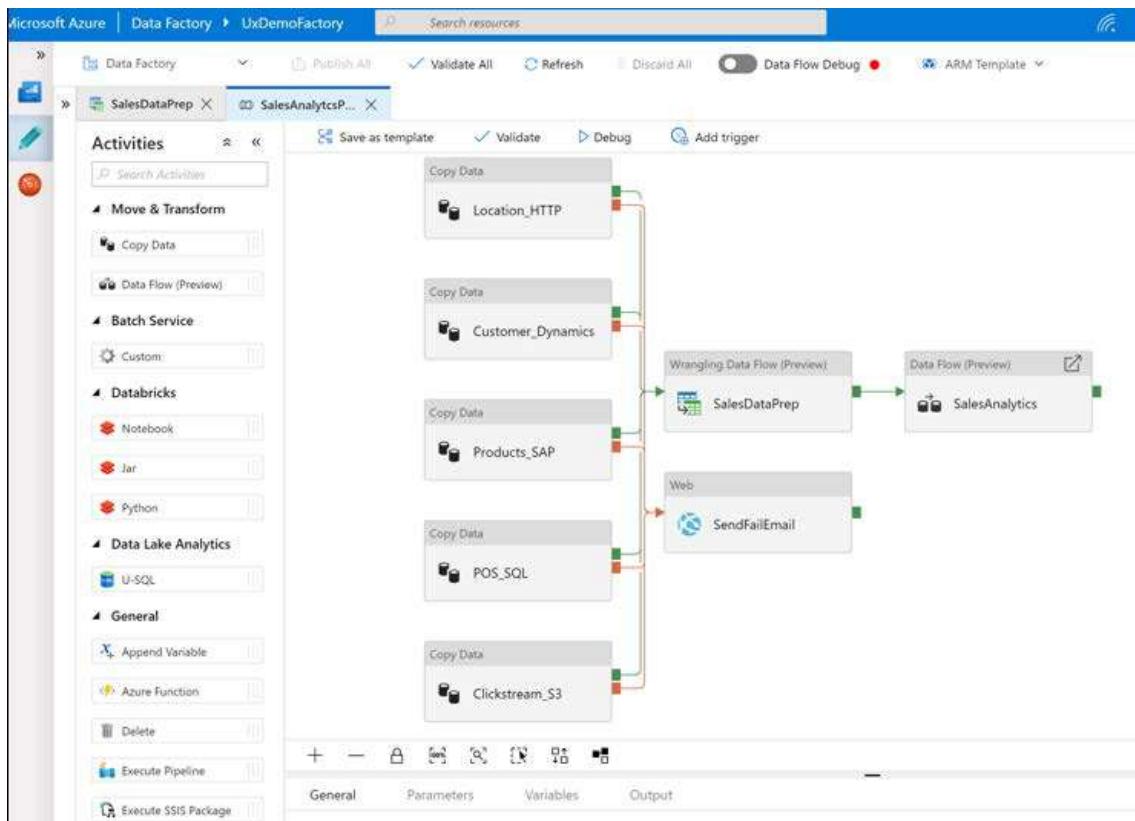


Figure 18.8: Re-developing scalable ETL processes using ADF

The next section will offer some recommendations for migrating queries, BI reports, dashboards, and other visualizations.

Recommendations for migrating queries, BI reports, dashboards, and other visualizations

Migrating queries, BI reports, dashboards, and other visualizations from your legacy data warehouse to Azure Synapse Analytics is straightforward if the legacy system uses standard SQL.

However, often, this is not the case. In this situation, a different strategy must be taken:

- Identify the high-priority reports to migrate first.
- Use usage statistics to identify the reports that are never used.
- Avoid migrating anything that is no longer in use.
- Once you have produced the list of reports to migrate, their priorities, and the unused reports to be bypassed, confirm this list with the stakeholders.
- For reports that you are migrating, identify incompatibilities early to gauge the migration effort.
- Consider data virtualization to protect BI tools and applications from structural changes to the data warehouse and/or data mart data model that might occur during the migration.

Common migration issues and resolutions

During the migration process, you might encounter certain issues that you need to overcome. In this section, we will highlight some of the common issues and provide you with resolutions that you can implement.

Issue #1: Unsupported data types and workarounds

Table 18.4 shows the data types from legacy data warehouse systems that are unsupported, as well as the suitable workarounds for Azure Synapse Analytics:

Unsupported data type	Workaround for Azure Synapse Analytics
geometry	varbinary
geography	varbinary
hierarchyid	nvarchar(4000)
image	varbinary
text	varchar
ntext	nvarchar
sql_variant	Split column into several strongly typed columns
table	Convert to temporary tables
timestamp	Rework code to use datetime2 and the CURRENT_TIMESTAMP function
xml	varchar
user-defined type	Convert back to the native data type when possible

Table 18.4: Unsupported data types and suitable workarounds in Azure Synapse Analytics

Issue #2: Data type differences between Netezza and Azure Synapse

Table 18.5 maps the Netezza data types to their Azure Synapse equivalent data types:

Netezza data type	Azure Synapse data type
BIGINT	BIGINT
BINARY VARYING(n)	VARBINARY(n)
BOOLEAN	BIT
BYTEINT	TINYINT
CHARACTER VARYING(n)	VARCHAR(n)
CHARACTER(n)	CHAR(n)
DATE	DATE
DECIMAL(p,s)	DECIMAL(p,s)
DOUBLE PRECISION	FLOAT
FLOAT(n)	FLOAT(n)
INTEGER	INT
INTERVAL	INTERVAL data types are not currently directly supported in Azure Synapse but can be calculated using temporal functions, such as DATEDIFF
MONEY	MONEY
NATIONAL CHARACTER VARYING(n)	NVARCHAR(n)
NATIONAL CHARACTER(n)	NCHAR(n)
NUMERIC(p,s)	NUMERIC(p,s)
REAL	REAL
SMALLINT	SMALLINT
ST_GeOMETRY(n)	Spatial data types such as ST_GeOMETRY are not currently supported in Azure Synapse, but the data could be stored as VARCHAR or VARBINARY
TIME	TIME
TIME WITH TIME ZONE	DATETIMEOFFSET
TIMESTAMP	DATETIME

Table 18.5: Netezza data types and their Azure Synapse equivalents

Issue #3: Integrity constraint differences

Pay close attention to the integrity constraint differences between your legacy data warehouse or data mart and Azure Synapse Analytics. In Figure 18.9, the left side represents the old legacy data warehouse system with primary key and foreign key constraints, and on the right side is the new Azure Synapse Analytics environment:

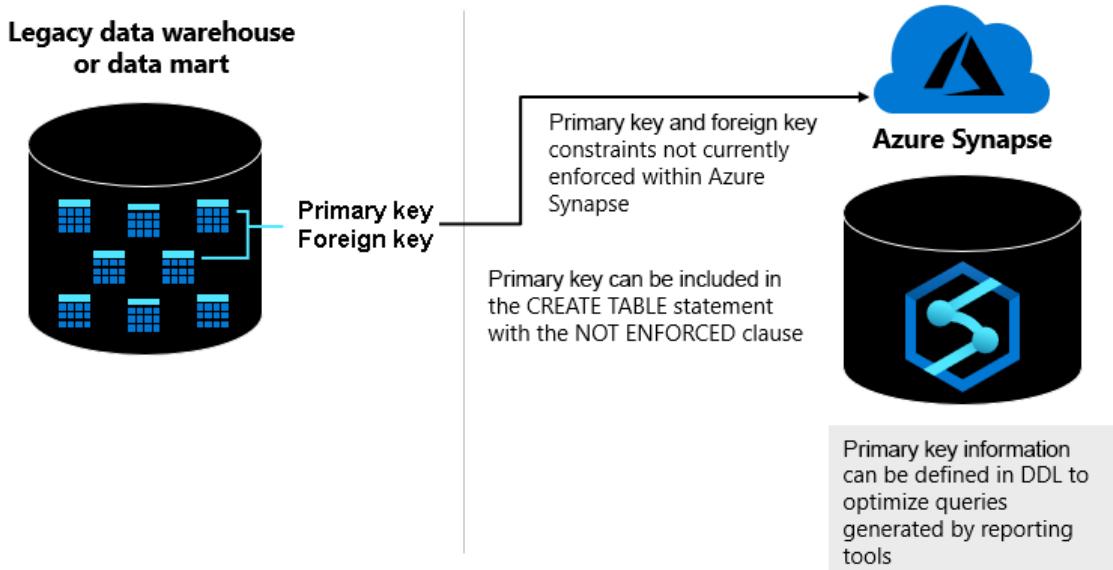


Figure 18.9: Integrity constraint differences

The next sections will provide comprehensive coverage on how to resolve other common SQL incompatibilities during the migration from a legacy data warehouse to Azure Synapse Analytics.

Common SQL incompatibilities and resolutions

This section will provide technical details regarding common SQL incompatibilities and resolutions between legacy data warehouse systems and Azure Synapse Analytics. The section will explain and compare the differences and provide resolutions using a quick-reference table that you can refer to later on as you embark on your migration project.

The topics that we will cover are as follows:

- **SQL Data Definition Language (DDL)** differences and resolutions
- **SQL Data Manipulation Language (DML)** differences and resolutions
- **SQL Data Control Language (DCL)** differences and resolutions
- Extended SQL differences and workarounds

SQL DDL differences and resolutions

In this section, we will discuss the differences and resolutions for SQL DDL between legacy data warehouse systems and Azure Synapse Analytics.

Issues	Legacy data warehouse system	Resolutions
Proprietary table types	<ul style="list-style-type: none">On the legacy system, identify any use of proprietary table types.	<ul style="list-style-type: none">Migrate to standard tables within Azure Synapse Analytics.For time series, index or partition on the date/time column.Additional filtering will need to be added into the relevant temporal queries.
Views	<ul style="list-style-type: none">Identify views from catalog tables and DDL scripts.	<ul style="list-style-type: none">Views with proprietary SQL extensions or functions will have to be re-written.Azure Synapse Analytics also supports materialized views and will automatically maintain and refresh these.
Nulls	<ul style="list-style-type: none">NULL values can be handled differently in legacy SQL databases. For example, in Oracle, an empty string is equivalent to a NULL value.Some DBMSes have proprietary SQL functions for handling NULLs; for example, NVL in Oracle.	<ul style="list-style-type: none">Generate SQL queries to test for NULL values.Test reports that include nullable columns.

Table 18.6: SQL DDL differences between legacy systems and Azure Synapse

SQL DML differences and resolutions

In this section, we will discuss the differences and resolutions for SQL DML between legacy data warehouse systems and Azure Synapse Analytics:

Function	Netezza	Azure Synapse equivalent
STRPOS	SELECT STRPOS('ABCDEFG', 'BCD') ...	SELECT CHARINDEX('BCD', 'ABCDEFG') ...
AGE	SELECT AGE('25-12-1940', '25-12-2020') FROM ...	SELECT DATEDIFF(day, '1940-12-25', '2020-12-25') FROM ...
NOW()	NOW()	CURRENT_TIMESTAMP
SEQUENCE	CREATE SEQUENCE ...	Rewrite using IDENTITY columns on Azure Synapse Analytics.
UDF	Netezza UDFs are written in nzLua or C++.	Rewrite in T-SQL on Azure Synapse Analytics.
Stored Procedures	Netezza stored procedures are written in NZPLSQL (based on Postgres PL/pgSQL).	Rewrite in T-SQL on Azure Synapse Analytics.

Table 18.7: SQL DML differences between Netezza and Azure Synapse

Next, we will talk about the differences and resolutions of SQL DCL between legacy data warehouse systems and Azure Synapse Analytics.

SQL DCL differences and resolutions

In this section, we will discuss the differences and resolutions for SQL DCL between legacy data warehouse systems and Azure Synapse Analytics. Netezza supports two classes of access rights: admin and object. Table 18.8 map the Netezza access rights and their corresponding Azure Synapse equivalents for quick reference.

Mapping Netezza admin privileges to the Azure Synapse equivalents

Table 18.8 maps the Netezza admin privileges to the Azure Synapse equivalents:

Admin privilege	Description	Azure Synapse equivalent
Backup	Allows users to create backups and to run the nzbackup command.	Backup and restore feature in Azure Synapse SQL pool
[Create] Aggregate	Allows the user to create User-Defined Aggregates (UDAs). Permission to operate on existing UDAs is controlled by object privileges.	Azure Synapse's CREATE FUNCTION feature incorporates Netezza aggregate functionality
[Create] Database	Allows the user to create databases. Permission to operate on existing databases is controlled by object privileges.	CREATE DATABASE
[Create] External Table	Allows the user to create external tables. Permission to operate on existing tables is controlled by object privileges.	CREATE TABLE
[Create] Function	Allows the user to create UDFs. Permission to operate on existing UDFs is controlled by object privileges.	CREATE FUNCTION
[Create] Group	Allows the user to create groups. Permission to operate on existing groups is controlled by object privileges.	CREATE ROLE
[Create] Index	For system use only. Users cannot create indexes.	CREATE INDEX
[Create] Library	Allows the user to create shared libraries. Permission to operate on existing shared libraries is controlled by object privileges.	N/A
[Create] Materialized View	Allows the user to create materialized views.	CREATE VIEW
[Create] Procedure	Allows the user to create stored procedures. Permission to operate on existing stored procedures is controlled by object privileges.	CREATE PROCEDURE
[Create] Schema	Allows the user to create schemas. Permission to operate on existing schemas is controlled by object privileges.	CREATE SCHEMA

Admin privilege	Description	Azure Synapse equivalent
[Create] Sequence	Allows the user to create database sequences.	N/A
[Create] Synonym	Allows the user to create synonyms.	CREATE SYNONYM
[Create] Table	Allows the user to create tables. Permission to operate on existing tables is controlled by object privileges.	CREATE TABLE
[Create] Temp Table	Allows the user to create temporary tables. Permission to operate on existing tables is controlled by object privileges.	CREATE TABLE
[Create] User	Allows the user to create users. Permission to operate on existing users is controlled by object privileges.	CREATE USER
[Create] View	Allows the user to create views. Permission to operate on existing views is controlled by object privileges.	CREATE VIEW
[Manage] Hardware	Allows the user to view hardware status, manage SPUs, manage topology and mirroring, and run diagnostic tests.	Automatically managed via the Azure portal in Azure Synapse
[Manage] Security	Allows the user to run commands and operations related to managing and configuring history databases; managing multi-level security objects, including specifying security for users and groups; and managing database keys for the digital signing of audit data.	Automatically managed via the Azure portal in Azure Synapse
[Manage] System	Allows the user to do the following management operations: start/stop/pause/resume the system, abort sessions, and view the distribution map, system statistics, and logs. The user can use these commands: nzsystem, nzstate, nzstats, and nzsession.	Automatically managed via the Azure portal in Azure Synapse
Restore	Allows the user to restore the system. The user can run the nzrestore command.	Automatically handled in Azure Synapse
Unfence	Allows the user to create or alter a UDF or aggregate to run in unfenced mode.	N/A

Table 18.8: Netezza admin privileges and their Azure Synapse equivalents

Mapping Netezza object privileges to their Azure Synapse equivalent

Table 18.9 maps the Netezza object privileges to the Azure Synapse equivalents for quick reference:

Object privilege	Description	Azure Synapse equivalent
Abort	Allows the user to abort sessions. Applies to groups and users.	KILL DATABASE CONNECTION
Alter	Allows the user to modify object attributes. Applies to all objects.	ALTER
Delete	Allows the user to delete table rows. Applies only to tables.	DELETE
Drop	Allows the user to drop objects. Applies to all object types.	DROP
Execute	Allows the user to run UDFs, UDAs, or stored procedures.	EXECUTE
GenStats	Allows the user to generate statistics on tables or databases. The user can run the GENERATE STATISTICS command.	Automatically handled in Azure Synapse
Groom	Allows the user to reclaim disk space for deleted or outdated rows, and reorganize a table by the organizing keys, or to migrate data for tables that have multiple stored versions.	Automatically handled in Azure Synapse
Insert	Allows the user to insert rows into a table. Applies only to tables.	INSERT
List	Allows the user to display an object name, either in a list or in another manner. Applies to all objects.	LIST
Select	Allows the user to select (or query) rows within a table. Applies to tables and views.	SELECT
Truncate	Allows the user to delete all rows from a table. Applies only to tables.	TRUNCATE
Update	Allows the user to modify table rows. Applies only to tables.	UPDATE

Table 18.9: Netezza object privileges and their Azure Synapse equivalents

Extended SQL differences and workarounds

Table 18.10 describes the extended SQL differences and possible workarounds when migrating to Azure Synapse Analytics:

SQL extension	Description	How to migrate
UDFs	<ul style="list-style-type: none"> Can contain arbitrary code Can be coded in various languages (such as Lua and Java) Can be called within a SQL SELECT statement in the same way that built-in functions such as SUM() and AVG() are used 	<ul style="list-style-type: none"> Use CREATE FUNCTION and re-code in T-SQL.
Stored procedures	<ul style="list-style-type: none"> Can contain one or more SQL statements as well as procedural logic around those SQL statements Implemented in a standard language (such as Lua) or in a proprietary language (such as Oracle PL/SQL) 	<ul style="list-style-type: none"> Recode in T-SQL. Some third-party tools that can help with migration: Datometry WhereScape
Triggers	<ul style="list-style-type: none"> Not supported by Azure Synapse 	<ul style="list-style-type: none"> Equivalent functionality can be achieved by using other parts of the Azure ecosystem. For example, for streamed input data, use Azure Stream Analytics.
In-database analytics	<ul style="list-style-type: none"> Not supported by Azure Synapse 	<ul style="list-style-type: none"> Run advanced analytics such as machine learning models at scale to use Azure Databricks. Azure Synapse opens the possibility of performing machine learning functions with Spark MLlib. Alternatively, migrate to Azure SQL Database and use the PREDICT function.
Geospatial data types	<ul style="list-style-type: none"> Not supported by Azure Synapse 	<ul style="list-style-type: none"> Store geospatial data, such as latitude/longitude, and popular formats, such as Well-KnownText (WKT) and Well-Known Binary (WKB), in VARCHAR or VARBINARY columns and access them directly by using geospatial client tools.

Table 18.10: Extended SQL differences and workarounds

In this section, we talked about common migration issues that architects might encounter during a migration project and possible solutions. In the next section, we will take a look at security considerations that an architect should be mindful of.

Security considerations

Protecting and securing your data assets is paramount in any data warehouse system. When planning a data warehouse migration project, security, user access management, backup, and restore must also be taken into consideration. For instance, data encryption may be mandatory for industry and government regulations, such as HIPAA, PCI, and FedRAMP, as well as in non-regulated industries.

Azure includes many features and functions as standard that would traditionally have to be custom-built in legacy data warehouse products. Azure Synapse supports data encryption at rest and data in motion as standard.

Data encryption at rest

- **Transparent Data Encryption (TDE)** can be enabled to dynamically encrypt and decrypt Azure Synapse data, logs, and associated backups.
- Azure Data Storage can also automatically encrypt non-database data.

Data in motion

All connections to Azure Synapse Analytics are encrypted by default, using industry-standard protocols such as TLS and SSH.

In addition, **Dynamic Data Masking (DDM)** can be used to obfuscate data for given classes of users based on data masking rules.

As a best practice, if your legacy data warehouse contains a complex hierarchy of permissions, users and roles, consider using automation techniques in your migration process. You can use existing metadata from your legacy system to generate the necessary SQL to migrate users, groups, and privileges on Azure Synapse Analytics.

In the final section of this chapter, we will review some of the tools that architects can choose to help migrate from legacy data warehouse systems to Azure Synapse Analytics.

Tools to help migrate to Azure Synapse Analytics

Now that we have covered the planning and preparation and an overview of the migration process, let's have a look at the tools that you can use for migrating your legacy data warehouse to Azure Synapse Analytics. The tools that we will discuss are:

- ADF
- Azure Data Warehouse Migration Utility
- Microsoft Services for Physical Data Transfer
- Microsoft Services for Data Ingestion

Let's get started.

ADF

ADF is a fully managed, pay-as-you-use, hybrid data integration service for cloud-scale ETL processing. It offers the following features:

- Processes and analyzes data in memory and in parallel to scale and maximize throughput
- Creates data warehouse migration pipelines that orchestrate and automate data movement, data transformation, and data loading into Azure Synapse Analytics
- Can also be used to modernize your data warehouse by ingesting data into Azure Data Lake Storage, processing and analyzing data at scale, and loading data into a data warehouse
- Supports role-based user interfaces for mapping data flows for IT professionals and self-service data wrangling for business users
- Can connect to multiple data stores spanning datacenters, clouds, and SaaS applications
- Over 90 natively built and maintenance-free connectors available (<https://azure.microsoft.com/services/data-factory>)
- Can mix and match wrangling and mapping data flows in the same pipeline to prepare data at scale
- ADF orchestration can control data warehouse migration to Azure Synapse Analytics
- Can execute SSIS ETL packages

Azure Data Warehouse Migration Utility

Azure Data Warehouse Migration Utility can migrate data from an on-premises SQL Server-based data warehouse to Azure Synapse. It offers the following features:

- Uses a wizard-like approach to perform a lift and shift migration of schema and data from an on-premises, SQL Server-based data warehouse.
- You can select the on-premises database containing the table(s) that you want to export to Azure Synapse. Then, you can select the tables that you want to migrate and migrate the schema.
- Automatically generates T-SQL code needed to create an equivalent empty database and tables on Azure Synapse. Once you provide connection details to Azure Synapse you can run the generated T-SQL to migrate the schema.
- Following schema creation, you can use the utility to migrate the data. This exports the data from your on-premises SQL Server-based data warehouse and generates **Bulk Copy Program (BCP)** commands to load that data into Azure Synapse.

Microsoft Services for Physical Data Transfer

In this section, we will look at common Microsoft services that can be used for physical data transfer, including Azure ExpressRoute, AzCopy, and Azure Databox.

Azure ExpressRoute

Azure ExpressRoute allows you to make private connections between your datacenters and Azure without going over the public Internet. It offers the following features:

- Bandwidth of up to 100 Gbps
- Low latency
- Connects directly to your **Wide-Area Network (WAN)**
- Private connections to Azure
- Increased speed and reliability

AzCopy

AzCopy is a command-line tool for copying files and blobs to/from storage accounts. It offers the following features:

- Ability to copy data to/from Azure via the Internet.
- A combination of AzCopy with the necessary ExpressRoute bandwidth could be an optimal solution for data transfer to Azure Synapse.

Azure Data Box

Azure Data Box allows you to transfer large volumes of data to Azure quickly, reliably, and cost-effectively. It offers the following features:

- Capable of transferring large volumes of data (tens of terabytes to hundreds to terabytes)
- No network connectivity restrictions
- Great for one-time migration and initial bulk transfer

Microsoft Services for data ingestion

In this section, we will look at common Microsoft services that can be used for data ingestion, including:

- PolyBase
- BCP
- SqlBulkCopy API
- Standard SQL

PolyBase (recommended method)

PolyBase provides the fastest and most scalable bulk data loading into Azure Synapse Analytics. It offers the following features:

- Uses parallel loading to give the fastest throughput
- Can read from flat files in Azure Blob storage or from external data sources via connectors
- Tightly integrated with ADF
- CREATE TABLE AS or INSERT ... SELECT
- Can define a staging table as type HEAP for fast load
- Support rows up to 1 MB in length

BCP

BCP can be used to import and export data from any SQL Server environment, including Azure Synapse Analytics. It offers the following features:

- Supports rows larger than 1 MB in length
- Originally developed for earlier versions of Microsoft SQL Server

Refer to <https://docs.microsoft.com/sql/tools/bcp-utility> to read more about the BCP utility.

SqlBulkCopy API

SqlBulkCopy API is the API equivalent of the BCP functionality. It offers the following features:

- Allows the implementation of load processes programmatically
- Ability to bulk load SQL Server tables with data from selected sources

Refer to <https://docs.microsoft.com/dotnet/api/system.data.sqlclient.sqlbulkcopy> to read more about this API.

Standard SQL Support

Azure Synapse Analytics supports standard SQL, including the ability to:

- Load individual rows or results of **SELECT** statements into data warehouse tables.
- Bulk insert data from extracted data via external data sources into data warehouse tables using **INSERT ... SELECT** statements within PolyBase.

This section provided the architectural considerations and high-level methodology for planning, preparing, and executing a successful migration of an existing legacy data warehouse system to Azure Synapse Analytics. It contains a wealth of information that you can refer to later on as you embark on your migration project to Azure Synapse Analytics.

Summary

Azure Synapse Analytics is a limitless analytics service with unmatched time to insight that accelerates the delivery of BI, AI, and intelligent applications for enterprises. You will gain a lot of benefits by migrating your legacy data warehouse to Azure Synapse Analytics, including performance, speed, improved security and compliance, elasticity, managed infrastructure, scalability, and cost savings.

With Azure Synapse, data professionals of varying skillsets can collaborate, manage, and analyze their most important data with ease—all within the same service. From Apache Spark integration with the powerful and trusted SQL engine, to code-free data integration and management, Azure Synapse is built for every data professional.

This chapter provided the architectural considerations and high-level methodology needed to prepare for and execute the migration of an existing legacy data warehouse system to Azure Synapse Analytics.

Successful data migration projects start with a well-designed plan. An effective plan accounts for the many components that need to be considered, paying particular attention to architecture and data preparation.

After you have successfully migrated to Azure Synapse, you can explore additional Microsoft technologies in the rich Azure analytical ecosystem to further modernize your data warehouse architecture.

Here are some ideas to ponder:

- Offload your staging areas and ELT processing to Azure Data Lake Storage and ADF.
- Build trusted data products once in common data model format and consume everywhere—not just in your data warehouse.
- Enable collaborative development of data preparation pipelines by business and IT using ADF mapping and wrangling data flows.
- Build analytical pipelines in ADF to analyze data in batch and real time.
- Build and deploy machine learning models to add additional insights to what you already know.
- Integrate your data warehouse with live streaming data.
- Simplify access to data and insights in multiple Azure analytical data stores by creating a logical data warehouse using PolyBase.

In the next chapter, you will learn in detail about Azure Cognitive Services, with a focus on architecting solutions that include intelligence as their core engine.

19

Architecting intelligent solutions

Cloud technology has changed a lot of things, including the creation of intelligent applications in an agile, scalable, and pay-as-you-go way. Applications prior to the rise of cloud technology generally did not incorporate intelligence within themselves, primarily because:

- It was time-consuming and error-prone.
- It was difficult to write, test, and experiment with algorithms on an ongoing basis.
- There was a lack of sufficient data.
- It was immensely costly.

Over the last decade, two things have changed that have led to the creation of significantly more intelligent applications than in the past. These two things are the cost-effective, on-demand unlimited scalability of the cloud along with the availability of data in terms of volume, variety, and velocity.

In this chapter, we will look at architectures that can help build intelligent applications with Azure. Some of the topics covered in this chapter are:

- The evolution of AI
- Azure AI processes
- Azure Cognitive Services
- Building an optical character recognition service
- Building a visual features service using the Cognitive Search .NET SDK

The evolution of AI

AI is not a new field of knowledge. In fact, the technology is a result of decades of innovation and research. However, its implementation in previous decades was a challenge for the following reasons:

1. **Cost:** AI experiments were costly in nature and there was no cloud technology. All the infrastructure was either purchased or hired from a third party. Experiments were also time-consuming to set up and immense skills were needed to get started. A large amount of storage and compute power was also required, which was generally missing in the community at large and held in the hands of just a few.
2. **Lack of data:** There were hardly any smart handheld devices and sensors available generating data. Data was limited in nature and had to be procured, which again made AI applications costly. Data was also less reliable and there was a general lack of confidence in the data itself.
3. **Difficulty:** AI algorithms were not documented enough and were primarily in the realms of mathematicians and statisticians. They were difficult to create and utilize within applications. Just imagine the creation of an **optical character recognition (OCR)** system 15 years ago. There were hardly any libraries, data, processing power, or the necessary skills to develop applications using OCR.

Although the influx of data increased with time, there was still a lack of tools for making sense of the data in a way that added business value. In addition, good AI models are based on sufficiently accurate data and trained with algorithms to be capable of resolving real-life problems. Both cloud technology and the large number of sensors and handheld devices have redefined this landscape.

With cloud technology, it is possible to provision on-demand storage and compute resources for AI-based applications. Cloud infrastructure provides lots of resources for data migration, storage, processing, and computation, as well as generating insights and eventually providing reports and dashboards. It does all this at a minimal cost in a faster way since there is nothing physical involved. Let's dive into understanding what goes on behind building an AI-based application.

Azure AI processes

Every AI-based project is required to go through a certain set of steps before being operational. Let's explore these seven phases:

Data ingestion

In this phase, data is captured from various sources and stored such that it can be consumed in the next phase. The data is cleaned before being stored and any deviations from the norm are disregarded. This is part of the preparation of data. The data could have different velocity, variety, and volume. It can be structured similarly to relational databases, semi-structured like JSON documents, or unstructured like images, Word documents, and so on.

Data transformation

The data ingested is transformed into another format as it might not be consumable in its current format. The data transformation typically includes the cleaning and filtering of data, removing bias from the data, augmenting data by joining it with other datasets, creating additional data from existing data, and more. This is also part of the preparation of the data.

Analysis

The data from the last phase is reused for analysis. The analysis phase contains activities related to finding patterns within data, conducting exploratory data analysis, and generating further insights from it. These insights are then stored along with existing data for consumption in the next phase. This is part of the model packaging process.

Data modeling

Once the data is augmented and cleaned, appropriate and necessary data is made available to the AI algorithms to generate a model that is conducive to achieving the overall aim. It is an iterative process known as experimentation by using various combinations of data (feature engineering) to ensure that the data model is robust. This is also part of the model packaging process.

The data is fed into learning algorithms to identify patterns. This process is known as training the model. Later, test data is used on the model to check its effectiveness and efficiency.

Validating the model

Once the model is created, a set of test data is used to find its effectiveness. If the analysis obtained from the test data is reflective of reality, then the model is sound and usable. Testing is an important aspect of the AI process.

Deployment

The model is deployed to production so that real-time data can be fed into it to get the predicted output. This output can then be used within applications.

Monitoring

The model deployed to production is monitored on an ongoing basis for the future analysis of all incoming data and to retrain and improve the effectiveness models.

The AI stages and processes, by nature, are time-consuming and iterative. Thus, applications based on them have an inherent risk of being long-running, experimental, and resource-intensive, along with getting delayed with cost overruns and having low chances of success.

Keeping these things in mind, there should be out-of-the-box AI-based solutions that developers can use in their applications to make them intelligent. These AI solutions should be easily consumable from applications and should have the following features:

- **Cross-platform:** Developers using any platform should be able to consume these services. They should be deployed and consumed on Linux, Windows, or Mac without any compatibility problems.
- **Cross-language:** Developers should be able to use any language to consume these solutions. Not only will the developers encounter a shorter learning curve but they also won't need to change their preferred choice of language to consume these solutions.

These solutions should be deployed as services using industry standards and protocols. Generally, these services are available as HTTP REST endpoints that can be invoked using any programming language and platform.

There are many such types of service that can be modeled and deployed for developer consumption. Some examples include:

- **Language translation:** In such services, the user provides text in one language and gets corresponding text in a different language as output.
- **Character recognition:** These services accept images and return the text present in them.
- **Speech-to-text conversion:** These services can convert input speech to text.

Now that we have gone through the details of building an AI/ML-based project, let's dive into the applications of various cognitive services offered by Azure.

Azure Cognitive Services

Azure provides an umbrella service known as Azure Cognitive Services. Azure Cognitive Services is a set of services that developers can consume within their applications to turn them into intelligent applications.

Vision	Web Search	Language	Speech	Decision
<ul style="list-style-type: none"> • Computer Vision • Face • Video Indexer • Custom Vision • Form Recognizer (Preview) • Ink Recognizer (Preview) 	<ul style="list-style-type: none"> • Bing Autosuggest • Bing Custom Search • Bing Entity Search • Bing Image Search • Bing News Search • Bing Spell Check • Bing Video Search • Bing Web Search 	<ul style="list-style-type: none"> • Immersive Reader (Preview) • Language Understanding • QnA Maker • Text Analytics • Translator 	<ul style="list-style-type: none"> • Speech to Text • Text to Speech • Speech Translation • Speaker Recognition (Preview) 	<ul style="list-style-type: none"> • Anomaly Detector (Preview) • Content Moderator • Personalizer

Table 19.1: Azure Cognitive Services

The services have been divided into five main categories depending on their nature. These five categories are as follows:

Vision

This API provides algorithms for image classification and helps in image processing by providing meaningful information. Computer vision can provide a variety of information from images on different objects, people, characters, emotions, and more.

Search

These APIs help in search-related applications. They help with search based on text, images, video, and providing custom search options.

Language

These APIs are based on natural language processing and help extract information about the intent of user-submitted text along with entity detection. They also help in text analytics and translation to different languages.

Speech

These APIs help in translating speech to text, text to speech, and in speech translation. They can be used to ingest audio files and take actions based on the content on behalf of users. Cortana is an example that uses similar services to take actions for users based on speech.

Decision

These APIs help in anomaly detection and content moderation. They can check for content within images, videos, and text and find out patterns that should be highlighted. An example of such an application is displaying a warning about adult content.

Now that you have an understanding of the core of Cognitive Services, let's discuss how they work in detail.

Understanding Cognitive Services

Azure Cognitive Services consists of HTTP endpoints that accept requests and send responses back to the caller. Almost all requests are HTTP POST requests and consist of both a header and a body.

The provisioning of Cognitive Services generates two important artifacts that help a caller invoke an endpoint successfully. It generates an endpoint URL and a unique key.

The format of the URL is `https://[azure location].api.cognitive.microsoft.com/[cognitive type]/[version]/[sub type of service]?[query parameters]`. An example URL is:

```
https://eastus.api.cognitive.microsoft.com/vision/v2.0/
ocr?language=en&detectOrientation=true
```

Cognitive Service is provisioned in the East US Azure region. The type of service is computer vision using version 2 and the subtype is OCR. There are generally a few subtypes for each top-level category. Lastly, there are a few query string parameters, such as `language` and `detectOrientation`. These query parameters are different for each service category and subcategory.

Either the header or the query parameters should provide the key value for the endpoint invocation to be successful.

The key value should be assigned to the `Ocp-Apim-Subscription-Key` header key with the request.

The content of the request body can be a simple string, a binary, or a combination of both. Depending on the value, the appropriate content-type header should be set in the request.

The possible header values are:

- `Application/octet-stream`
- `multipart/form-data`
- `application/json`

Use `octet-stream` when sending binary data and `json` for sending string values. `form-data` can be used for sending multiple combination values of binary and text.

The key is a unique string used to validate whether the caller has been given permission to invoke the URL. This key must be protected such that others who should not be able to invoke the endpoints do not get access to it. Later in the chapter, you will see ways to safeguard these keys.

Consuming Cognitive Services

There are two ways to consume Cognitive Services:

- **Using an HTTP endpoint directly:** In this case, the endpoint is invoked directly by crafting both the header and body with appropriate values. The return value is then parsed and data is extracted out of it. All the AI services in Cognitive Services are REST APIs. They accept HTTP requests in JSON, as well as other formats, and replies in JSON format.
- **Using an SDK:** Azure provides multiple **software development kits (SDKs)**. There are SDKs available for the .NET, Python, Node.js, Java, and Go languages.

In the following section, we will look into the utilization of one of the Cognitive Services using both ways. Let's explore this by building some AI services using HTTP endpoints.

Building an OCR service

In this section, we will be using some of the AI services using C# as well as PowerShell to show their usage using the HTTP endpoint directly. The next section will concentrate on doing the same using a .NET SDK.

Before getting into building a project using Cognitive Services, the first step is to provision the API itself.

Optical character recognition is available as a Vision API and can be provisioned using the Azure portal, as shown next. Create a vision API by navigating to **Cognitive Services > Compute Vision > Create**, as shown in *Figure 19.1*:

The screenshot shows the 'Create' page for a Computer Vision API in the Azure portal. The steps in the breadcrumb navigation are Home > New > Computer Vision > Create. The main title is 'Create' under 'Computer Vision'. The form fields are as follows:

- Name ***: azureforarchitects
- Subscription ***: RiteshSubscription
- Location ***: (US) East US
- Pricing tier (View full pricing details) ***: F0 (20 Calls per minute, 5K Calls per month)
- Resource group ***: (New) testrg

A 'Create new' button is located at the bottom of the form.

Figure 19.1: Create a Vision API

Once the API is provisioned, the overview page provides all the details for consuming the API. It provides the base URL and the key information. Make a note of the key as it will be used later:

- 1 Get the API Key and endpoint to authenticate your applications and start sending calls to the service.

Key1 [?](#)
ff0cd61f27d8452bbadad36942570c48 [Copy](#)

Endpoint [?](#)
<https://azureforarchitects.cognitiveservices.azure.com/> [Copy](#)

All Computer Vision calls and Docker container activations require a key. Specify the key either in the request header (Web API), the Computer Vision client (SDK) or through the command-line (Docker container).
- 2 Try the service in the API console - requires an API Key and selecting your location: eastus

Use the API Console to quickly try the API without writing code. Be sure to select the correct location for this resource, as listed above.

[API Console](#)
- 3 Make a web API call - requires your API Key and endpoint

Use the sample code in these quickstarts to begin integrating Computer Vision into your applications to analyze images. You can unlock insights such as detecting objects, brands, faces and more.

[C# Quickstart](#)
[Python Quickstart](#)
[Java Quickstart](#)
- 4 Try the Computer Vision Containers

The Recognize Text portion of the Computer Vision Cognitive Service is also available as a Docker container. This enables you to run the API on-premises if you don't want your data to leave your machine or environment. These containers have the same interfaces and capabilities as the operation in the hosted API.

[Installing the Computer Vision Containers](#)
[Container Support in Azure Cognitive Services](#)
[Container Samples](#)

Figure 19.2: Overview page

It also provides an API console to quickly test the API. Clicking on it opens a new window that has all the endpoints related to this service available. Clicking on **OCR** will provide a form that can be filled in with appropriate data and execute the service endpoints. It also provides a complete response. This is shown in Figure 19.3. The URL is available as a request URL, and the request is a typical HTTP request with a **POST** method. The URL points to the endpoint in the East US Azure region. It is also related to the Vision group of APIs, version 2, and the OCR endpoint.

The subscription key is passed in the header with the name **ocp-apim-subscription-key**. The header also contains the content-type key with **application/json** as a value. This is because the body of the request contains a JSON string. The body is in the form of JSON with the URL of the image from which text should be extracted:

Request URL

```
https://eastus.api.cognitive.microsoft.com/vision/v2.0/ocr?language=en&detectOrientation=true
```

HTTP request

```
POST https://eastus.api.cognitive.microsoft.com/vision/v2.0/ocr?language=en&detectOrientation=true HTTP/1.1
Host: eastus.api.cognitive.microsoft.com
Content-Type: application/json
Ocp-Apim-Subscription-Key: ****

{"url": "https://ichef.bbci.co.uk/news/320/cpsprodpb/F944/production/_109321836_oomzonlz.jpg"}
```

Send

Figure 19.3: Request URL

The request can be sent to the endpoint by clicking on the **Send** button. It will result in an HTTP response 200 OK, as shown next, if everything goes right. If there is an error in the request values, the response will be an error HTTP code:

Response content

```
csp-billing-usage: CognitiveServices.ComputerVision.OCR=1
apim-request-id: 6b08bdc8-edac-41e0-9361-f9679373b7e1
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
x-content-type-options: nosniff
Date: Sat, 04 Apr 2020 14:02:35 GMT
Content-Length: 257
Content-Type: application/json; charset=utf-8

{
  "language": "en",
  "textAngle": -0.029670597283902981,
  "orientation": "Up",
  "regions": [
    {
      "boundingBox": "159,136,80,18",
      "lines": [
        {
          "boundingBox": "159,136,80,18",
          "words": [
            {
              "boundingBox": "159,137,44,17",
              "text": "NY02"
            },
            {
              "boundingBox": "210,136,29,15",
              "text": "ZRO"
            }
          ]
        }
      ]
    }
]
```

Figure 19.4: HTTP response 200 OK

The response consists of details related to billing usage, an internal request ID generated by the endpoint, the content length, the response content type (being JSON), and the data and time of the response. The content of the response consists of a JSON payload with the coordinates of the text and the actual text itself.

Using PowerShell

The same request can be created using PowerShell. The following PowerShell code can be executed using the PowerShell ISE.

The code uses the **Invoke-WebRequest** cmdlet to invoke the Cognitive Services endpoint by passing the URL to the **Uri** parameter using the **POST** method, and adds both the appropriate headers as discussed in the last section, and finally, the body consisting of data in JSON format. The data is converted into JSON using the **ConvertTo-Json** cmdlet:

```
$ret = Invoke-WebRequest -Uri "https://eastus.api.cognitive.microsoft.com/vision/v2.0/ocr?language=en&detectOrientation=true" -Method Post  
-Headers @{"Ocp-Apim-Subscription-Key"="ff0cd61f27d8452bbadad36942570c48"; "Content-type"="application/json"} -Body $(ConvertTo-Json -InputObject @>{"url"="https://ichef.bbci.co.uk/news/320/cpsprodpb/F944/production/_109321836_oomzonlz.jpg"})  
  
$val = Convertfrom-Json $ret.content  
  
foreach ($region in $val.regions) {  
    foreach($line in $region.lines) {  
        foreach($word in $line.words) {  
            $word.text  
        }  
    }  
}
```

The response from the cmdlet is saved in a variable that also consists of data in JSON format. The data is converted into a PowerShell object using the **Convertfrom-Json** cmdlet and looped over to find the words in the text.

Using C#

In this section, we will build a service that should accept requests from users, extract the URL of the image, construct the HTTP request, and send it to the Cognitive Services endpoint. The Cognitive Services endpoint returns a JSON response. The appropriate text content is extracted from the response and returned to the user.

Architecture and design

An intelligent application is an ASP.NET Core MVC application. An MVC application is built by a developer on a developer machine, goes through the continuous integration and delivery pipeline, generates a Docker image, and uploads the Docker image to Azure Container Registry. Here, the major components of the application are explained, along with their usage:

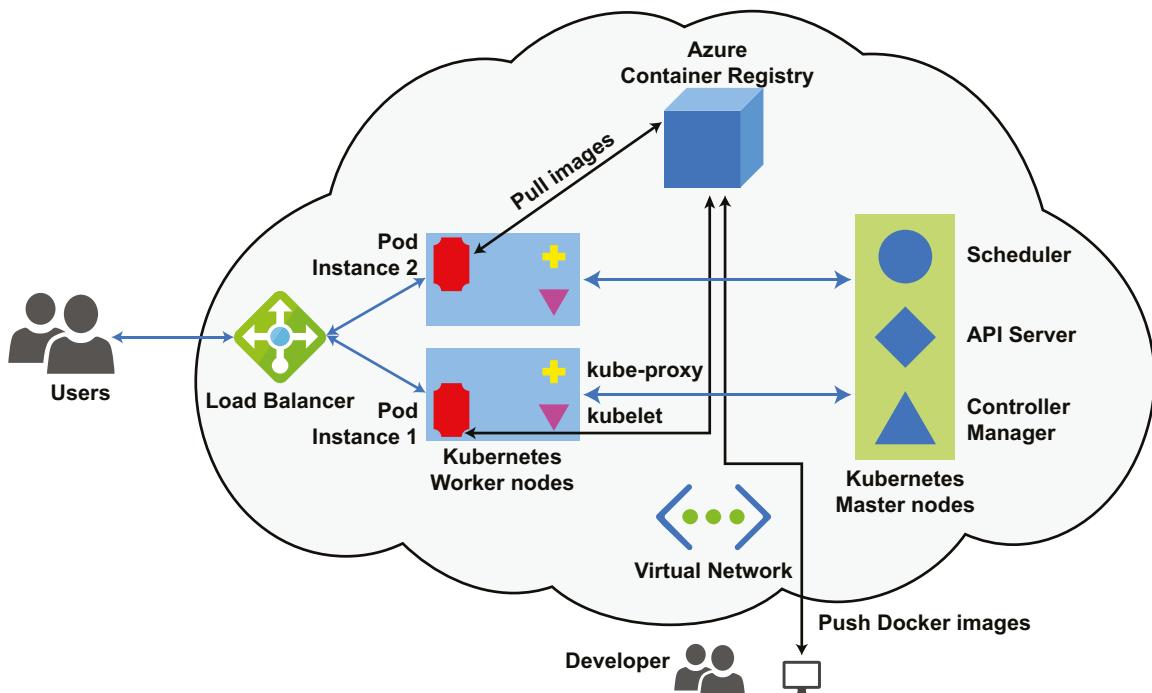


Figure 19.5: Workflow of an intelligent application

Docker

Docker is one of the major players within container technologies and is available cross-platform, including Linux, Windows, and Mac. Developing applications and services with containerization in mind provides the flexibility to deploy them across clouds and locations, as well as on-premises. It also removes any dependencies on the host platform, which again allows less reliance on platform as a service. Docker helps with the creation of custom images, and containers can be created out of these images. The images contain all the dependencies, binaries, and frameworks needed to make the application or service work, and they are completely self-reliant. This makes them a great deployment target for services such as microservices.

Azure Container Registry

Azure Container Registry is a registry that's similar to Docker Hub for the storage of container images in a repository. It is possible to create multiple repositories and upload multiple images in them. An image has a name and a version number, together forming a fully qualified name used to refer to them in a Kubernetes Pod definition. These images can be accessed and downloaded by any Kubernetes ecosystem. A prerequisite of this is that appropriate secrets for pulling the image should already be created beforehand. It need not be on the same network as Kubernetes nodes and, in fact, there is no need for a network to create and use Azure Container Registry.

Azure Kubernetes Service

The intelligent application that accepts the URL of an image to retrieve the text in it can be hosted on vanilla virtual machines or even within Azure App Service. However, deploying in Azure Kubernetes Service offers lots of advantages, which was covered in *Chapter 8, Architecting Azure Kubernetes Solutions*. For now, it is important to know that these applications are self-healing in nature and a minimum number of instances is automatically maintained by the Kubernetes master along with providing the flexibility to update them in a multitude of ways, including blue-green deployments and canary updates.

Pods, replica sets, and deployments

The developer also creates a Kubernetes deployment-related YAML file that references the images within the Pod specification and also provides a specification for the replica set. It provides its own specification related to the update strategy.

Runtime design

The architecture and design remain the same as in the previous section; however, when the application or service is already live and up and running, it has already downloaded the images from Azure Container Registry and created Pods running containers in them. When a user provides an image URL for decoding the text it contains, the application in the Pod invokes the Azure Cognitive Services Computer Vision API and passes the URL to it and waits for a response from the service:

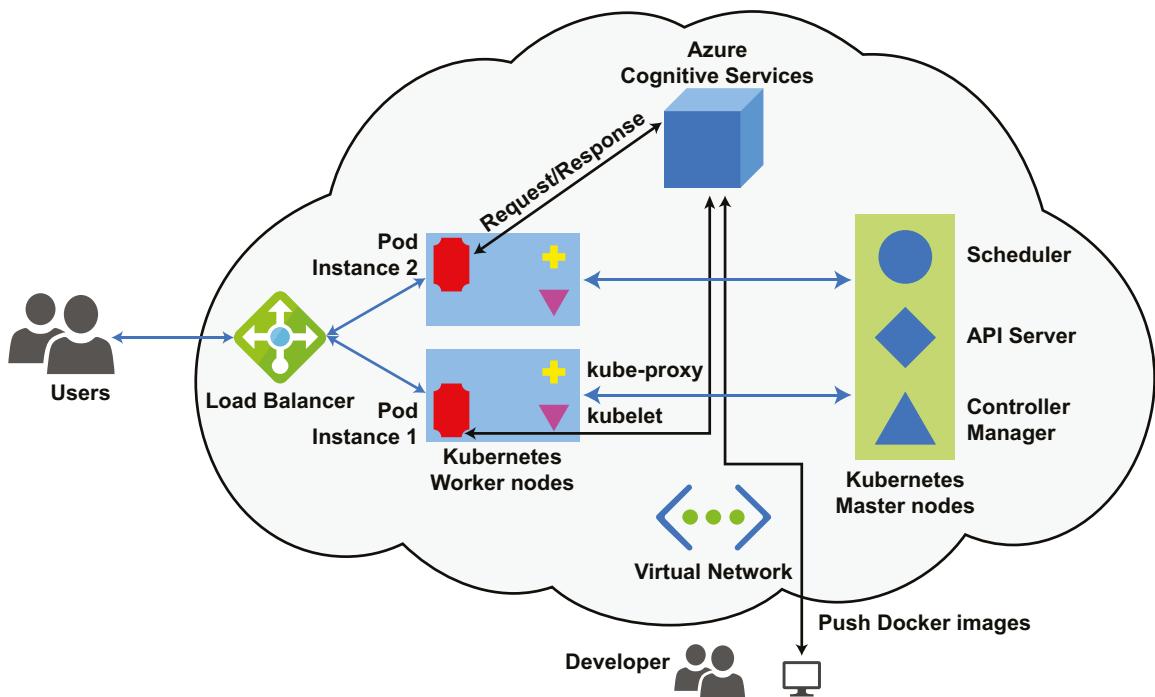


Figure 19.6 Workflow of an intelligent application

Once it receives the JSON response from the services, it can retrieve the information and return it to the user.

The development process

The development environment can be Windows or Linux. It will work with both Windows 10 and the Windows 2016/19 server. When using Windows, it is useful to deploy Docker for Windows so that it will create both a Linux and a Windows Docker environment.

When creating an ASP.NET Core web application project using Visual Studio 2019, the **Docker support** option should be selected with either **Windows** or **Linux** as values. Depending on the chosen value, appropriate content will be generated in **Dockerfile**. The main difference in **Dockerfile** is the base image names. It uses different images for Linux compared to Windows.

When installing Docker for Windows, it also installs a Linux virtual machine, and so it is important to turn on the Hyper-V hypervisor.

In this example, instead of sending the data as a JSON string, the image is downloaded, and binary data is sent to the Cognitive Services endpoint.

It has a function that accepts a string input for URL values. It then invokes Cognitive Services with appropriate header values and a body containing the URL. The header values should contain the key provided by Cognitive Services while provisioning the service. The value in the body can contain vanilla string values in the form of JSON or it can contain binary image data itself. The content-type header property should be set accordingly.

The code declares the URL and the key related to the Cognitive Services. This is shown for demonstration purposes only. The URL and key should be placed in configuration files.

Using the **HttpClient** object, the image corresponding to the URL supplied by the user is downloaded and stored within the **responseMessage** variable. Another **HttpClient** object is instantiated and its headers are filled with **Ocp-Apim-Subscription-Key** and **content-type keys**. The value of the content-type header is **application/octet-stream** since binary data is being passed to the endpoint.

A post request is made after extracting the content from the **responseMessage** variable and passing it as the body of a request to the cognitive service endpoint.

The code for the controller action is shown next:

```
[HttpPost]
public async Task<string> Post([FromBody] string value)
{
    string myurl = " https://eastus.api.cognitive.microsoft.com/
vision/v2.0/ocr?language=en&detectOrientation=true
    string token = ".....";
    using (HttpClient httpClient = new HttpClient())
    {
        var responseMessage = await httpClient.GetAsync(value);
```

```
        using (var httpClient1 = new HttpClient())
        {
            httpClient1.BaseAddress = new Uri(myurl);
            httpClient1.DefaultRequestHeaders.Add("Ocp-Apim-
Subscription-Key", token);

            HttpResponseMessage content = responseMessage.Content;
            content.Headers.ContentType = new
MediaTypeWithQualityHeaderValue("application/octet-stream");
            var response = await httpClient1.PostAsync(myurl,
content);

            var responseContent = await response.Content.
ReadAsByteArrayAsync();
            string ret = Encoding.ASCII.GetString(responseContent,
0, responseContent.Length);
            dynamic image = JsonConvert.
DeserializeObject<object>(ret);
            string temp = "";
            foreach (var regs in image.regions)
            {
                foreach (var lns in regs.lines)
                {
                    foreach (var wds in lns.words)
                    {
                        temp += wds.text + " ";
                    }
                }
            }
            return temp;
        }
    }
}
```

After the endpoint finishes its processing, it returns the response with a JSON payload. The context is extracted and deserialized into .NET objects. Multiple loops are coded to extract the text from the response.

In this section, we created a simple application that uses Cognitive Services to provide word extractions from features using the OCR API and deployed it within Kubernetes Pods. This process and architecture can be used within any application that wants to consume Cognitive Services APIs. Next, we will take a look at another Cognitive Services API, known as visual features.

Building a visual features service using the Cognitive Search .NET SDK

The last section was about creating a service that uses an OCR cognitive endpoint to return text within images. In this section, a new service will be created that will return visual features within an image, such as descriptions, tags, and objects.

Using PowerShell

The code in PowerShell is similar to the previous OCR example, so it is not repeated here. The URL is different from the previous code example:

The screenshot shows a PowerShell window with the following content:

Request URL

```
https://eastus.api.cognitive.microsoft.com/vision/v2.0/analyze?visualFeatures=Description&language=en
```

HTTP request

```
POST https://eastus.api.cognitive.microsoft.com/vision/v2.0/analyze?visualFeatures=Description&language=en HTTP/1.1
1
Host: eastus.api.cognitive.microsoft.com
Content-Type: application/json
Ocp-Apim-Subscription-Key: .....
{"url": "https://thefinanser.com/wp-content/uploads/2016/06/People.jpg"}
```

Send

Figure 19.7: Request URL

The request is made using a **POST** method, and the URL points to the endpoint in the East US Azure region. It also uses version 2 and consumes the Vision API.

The Cognitive Services access key is part of the HTTP header named **ocp-apim-subscription-key**. The header also contains the header content-type with **application/json** as the value. This is because the body of the request contains a JSON value. The body has the URL of the image from which text should be extracted.

The response will be in JSON format containing the image content and a description.

Using .NET

This example is again an ASP.NET Core MVC application and has the **Microsoft.Azure.CognitiveServices.Vision.ComputerVision** NuGet package installed in it:

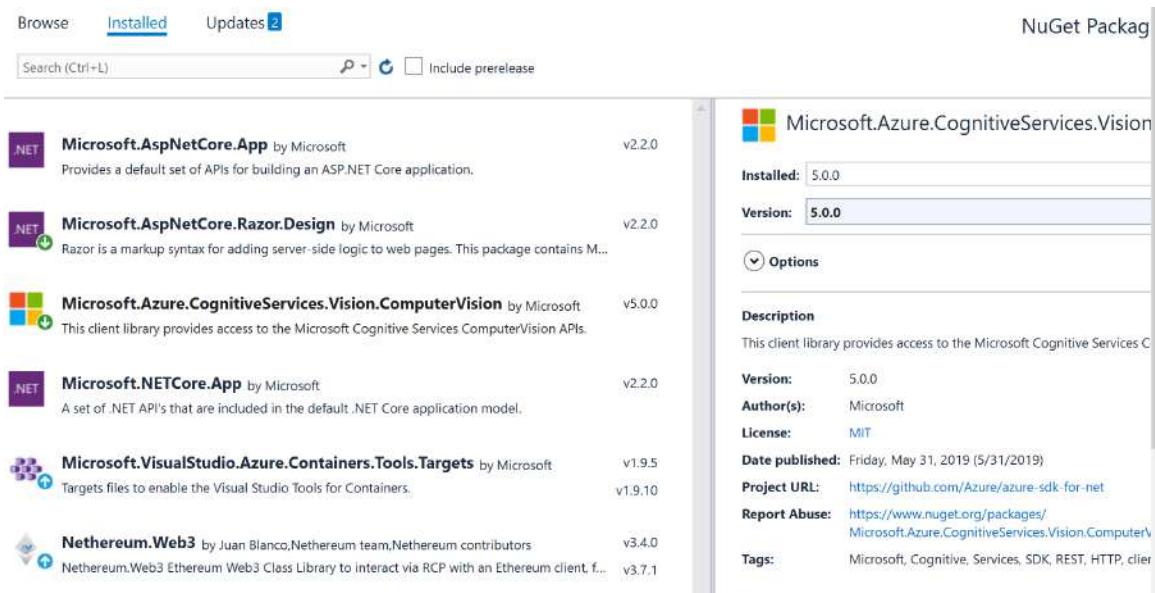


Figure 19.8: ASP.NET Core MVC application with the Microsoft.Azure.CognitiveServices.Vision.ComputerVision NuGet package

The code for the controller action is shown next. In this code, the cognitive service and key are declared. It also declares variables for the **ComputerVisionClient** and **VisionType** objects. It creates an instance of the **ComputerVisionClient** type, providing it the URL and the key.

The **VisionTypes** list consists of multiple types of data sought from the image—tags, descriptions, and objects are added. Only these parameters will be extracted from the image.

An **HttpClient** object is instantiated to download the image using the URL provided by the user and sends this binary data to the Cognitive Services endpoint using the **AnalyzeImageInStreamAsync** function of type **ComputerVisionClient**:

```
[HttpPost]
    public string Post([FromBody] string value)
    {
        private string visionapiurl = " https://
eastus.api.cognitive.microsoft.com/vision/v2.0/
analyze?visualFeature=tags,description,objects&language=en";
        private string apikey = "e55d36ac228f4d718d365f1fcddc0851";
        private ComputerVisionClient client;
        private List<VisualFeatureTypes> visionType = new
List<VisualFeatureTypes>();

client = new ComputerVisionClient(new ApiKeyServiceClientCredentials(apikey))
{
    Endpoint = visionapiurl
};

visionType.Add(VisualFeatureTypes.Description);
visionType.Add(VisualFeatureTypes.Tags);
visionType.Add(VisualFeatureTypes.Objects);

string tags = "";
string descrip = "";
string objprop = "";
using (HttpClient hc = new HttpClient())
{
    var responseMessage = hc.GetAsync(value).GetAwaiter().
GetResult();
    Stream streamData = responseMessage.Content.
ReadAsStreamAsync().GetAwaiter().GetResult();
    var result = client.AnalyzeImageInStreamAsync(streamData,
visionType).GetAwaiter().GetResult();
    foreach (var tag in result.Tags) {
        tags += tag.Name + " ";
    }
    foreach (var caption in result.Description.Captions)
```

```
{  
    descrip += caption.Text + " ";  
}  
  
foreach (var obj in result.Objects)  
{  
    objprop += obj.ObjectProperty + " ";  
}  
  
}  
return tags;  
// return descrip or objprop  
  
}
```

The results are looped through and tags are returned to the user. Similarly, descriptions and object properties can also be returned to the user. Now let's check out the ways we can safeguard the exposure of service keys.

Safeguarding the Cognitive Services key

There are multiple ways to safeguard the exposure of keys to other actors. This can be done using the API Management resource in Azure. It can also be done using Azure Functions Proxies.

Using Azure Functions Proxies

Azure Functions Proxies can refer to any URL, whether internal or external. When a request reaches Azure Functions Proxies, it will use the URL of the cognitive service along with the key to invoke the cognitive endpoint, and it will also override the request parameters and add the incoming image URL and append it to the cognitive endpoint URL as POST data. When a response comes back from the service, it will override the response, remove the headers, and pass JSON data back to the user.

Consuming Cognitive Services

Consuming Cognitive Services follows a consistent pattern. Each cognitive service is available as a REST API, with each API expecting different sets of parameters to work on. Clients invoking these URLs should check out the documentation for associate parameters and provide values for them. Consuming URLs is a relatively raw method of using Cognitive Services. Azure provides SDKs for each service and for multiple languages. Clients can use these SDKs to work with Cognitive Services.

The **LUIS (Language Understanding Intelligence Service)** authoring API is available at [https://\[luis resource name\]-authoring.cognitiveservices.azure.com/](https://[luis resource name]-authoring.cognitiveservices.azure.com/) and the production API is available at

```
https://[azure region].api.cognitive.microsoft.com/luis/prediction/v3.0/apps/
{application id}/slots/production/predict?subscription-key={cognitive key}
&verbose=true&show-all-intents=true&log=true&query=YOUR_QUERY_HERE.
```

Similarly, the Face API is available at [https://\[endpoint\]/face/v1.0/detect\[?returnFaceId\]\[&returnFaceLandmarks\]\[&returnFaceAttributes\]\[&recognitionModel\]\[&returnRecognitionModel\]\[&detectionModel\]](https://[endpoint]/face/v1.0/detect[?returnFaceId][&returnFaceLandmarks][&returnFaceAttributes][&recognitionModel][&returnRecognitionModel][&detectionModel]).

There are many Cognitive Services APIs, with each having multiple flavors in terms of URLs, and the best way to know about these URLs is to use the Azure documentation.

Summary

In this chapter, you gained an understanding of the deployment architecture and application architecture for creating intelligent applications in Azure. Azure provides Cognitive Services with numerous endpoints—each endpoint is responsible for executing an AI-related algorithm and providing outputs. Almost all Cognitive Services endpoints work in a similar manner with regard to HTTP requests and responses. These endpoints can also be invoked using SDKs provided by Azure for different languages, and you saw an example of obtaining visual features using them. There are more than 50 different endpoints, and you are advised to get an understanding of the nature of endpoints using the API console feature provided by Azure.



Microsoft.Source Newsletter - Inbox

Message

Microsoft

Microsoft.Source Newsletter | Issue 7

You're reading Microsoft.Source, the developer community newsletter featuring ideas and projects from your peers down the street—and around the world. If someone forwarded you this newsletter and you want to receive future editions, [sign up >](#)

[Give feedback](#) Get more of what you want in each edition.

Featured Story

Vanilla JS and HTML –No frameworks, no libraries, no problem >
Do you know what it takes to render HTML elements without the complexity of AngularJS, React, Svelte, or Vue.js? See how to create a simple web page with pure HTML, CSS, and JS.
Web, JavaScript, HTML

What's New

Build a web experience to send GIFs to MXChip >
IoT, project

The Making of Azure Mystery Mansion >
Game, Twine, PlayFab

Trying to make FETCH happen >
Serverless, IoT, Azure Functions

Events [See all events](#)

Cosmos DB Live Webcast / Online >
Expert-led, containers, .Net

OpenHack Serverless / Los Angeles >
In-person event, serverless, hack

Learning

Microsoft Ignite – Watch videos on demand >
Watch all keynotes, announcements, and sessions on demand

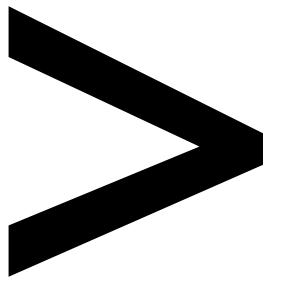
By developers, for developers

Microsoft.Source newsletter

Get technical articles, sample code, and information on upcoming events in Microsoft.Source, the curated monthly developer community newsletter.

- Keep up on the latest technologies
- Connect with your peers at community events
- Learn with hands-on resources





Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.

A

access: 3, 8, 10, 12-13, 19, 71, 73-74, 76, 80, 84-85, 87, 110, 113, 115, 145-148, 156, 158-159, 163-165, 168, 170-171, 177-178, 184, 189-190, 204-206, 208, 210, 213, 221, 226-227, 230-232, 234, 237-243, 245, 247-252, 256, 265, 267, 275, 287, 294, 297-298, 319, 322, 358-359, 361-363, 371, 395-396, 399, 402, 442, 446, 458, 479, 486, 503-504, 517, 522, 545, 553, 565-566, 569, 576, 604, 609, 627, 632, 637, 645, 655 account: 38, 52, 60, 64, 87, 105-107, 109-111, 113, 116-117, 119, 121, 132, 134, 136-137, 139, 141, 151, 157, 172, 174, 176-177, 179, 184, 191, 194, 220-221, 233, 248-252, 257, 274, 279-284, 286-289, 293, 296-297, 309-312, 318-320, 329, 332, 335-339, 347-348, 358, 361-364, 366, 369, 371-372, 375, 378, 399-400, 405, 408, 410-413, 415, 417, 443, 458, 467-468, 483, 504, 511, 516-519, 522, 524, 526-529, 532, 535, 538, 554, 559, 565-569 adappname: 361 addcontent: 374

address: 29, 33, 35, 37, 73, 75, 77, 93, 226, 233, 236, 246, 250, 256, 268, 446, 458-459, 471, 480, 485-486, 501-503, 526, 528-529, 560-561, 570, 572 aggregates: 405, 407 alerts: 27, 58, 61, 63-66, 129, 133, 171, 198, 229, 237, 247-248, 254, 266, 333, 578 algorithm: 50, 659 analytics: 8, 50, 61-64, 66-67, 70, 74, 193, 244, 247-248, 265-266, 269, 273, 276-277, 333, 347, 351, 389-392, 403-408, 411, 413-416, 418, 421, 442-443, 462, 472, 534, 578, 580, 582-588, 598, 601-607, 609-614, 616, 619-623, 625-627, 631-633, 635-637, 644 android: 222, 317 ansible: 428, 430, 442 apache: 7, 275-276, 298, 392, 587, 606-607, 637 api-based: 116 apikey: 657 apiversion: 151, 157, 485, 488-490, 496, 498-499, 511-512, 514, 520, 524, 527-528, 530, 533-539, 542-544, 551-554, 568, 571-572 appname: 485, 488-491, 496 architect: 20, 23-24, 67, 71, 101, 162, 179, 184, 190-191, 269, 343, 355, 386, 603, 632 artifacts: 106-107, 112, 162-163, 230, 285-286, 395, 429, 432, 434-436, 441, 444, 459-460, 644 assemblies: 265, 432, 448, 460 attack: 228, 237, 248, 265 audits: 187 automate: 17-18, 20, 101, 105, 109, 184, 428, 475, 613, 619, 622, 633 autoscale: 48, 53, 315 azcopy: 634 azureappid: 363, 367 azureblob: 536 azurecr: 485, 490-491 azurerm: 119, 122-123 azure-sql: 199, 213, 217

B

backend: 163, 310, 315, 321, 505 backup: 46, 201, 606, 632 bacpac: 429 balancer: 29, 32-35, 38-39, 45, 50, 189, 245, 458-459, 487-488, 498, 500, 526 bandwidth: 42, 70, 73, 76, 79, 87, 215, 634 binary: 272, 406, 439, 645, 653, 656 binding: 307-311, 326, 330-332 blobadded: 335, 340 blobsource: 539 browser: 17, 263, 295, 470, 500, 576 bundle: 518, 520

C

caching: 58, 190, 612
callback: 261
canary: 651
cassandra: 7, 219–221, 298
certkey: 115
cgroups: 476
client: 16–17, 36–37,
 57–58, 85, 238, 241,
 245, 248, 254, 257,
 259–260, 276, 321, 380,
 463, 481, 580, 657
clouds: 7, 67, 72, 136, 139,
 141, 145, 173, 633, 651
cluster: 29, 200, 275–276,
 295, 478, 480–481,
 483–484, 486–487,
 490, 492–495,
 500–506, 604–605
cmdlet: 18, 112, 116–118, 121,
 127, 137–139, 150, 159,
 340, 342, 358, 361, 368,
 469–470, 516, 519, 649
cognitive: 8, 317,
 325, 347, 637, 640,
 643–646, 649–650,
 652–653, 655–659
columnar: 199, 298, 610
compile: 108, 430–431, 470
concurrent: 195, 606, 617
configure: 12, 19, 27, 40,
 62, 66, 130–131, 134,
 136–137, 162, 197, 218,
 222, 233, 259–261, 266,
 292, 307, 313, 333, 346,
 348, 350–352, 361, 371,
 412, 414–415, 429–430,
 433, 446, 457, 460,
 467–468, 470, 473, 495,
 513, 530, 567, 589, 594

constraint: 79, 198,
 502, 625
container: 4, 10, 15–16, 32,
 50, 60, 147, 178, 190, 211,
 220, 280, 282, 288–289,
 297, 316, 318–321, 330,
 340, 395, 413–414, 417,
 438, 443, 462–463, 473,
 475, 477–479, 484–485,
 493, 496, 500–501,
 505–507, 522, 558,
 565–566, 569, 650–652
cosmosdb: 330

D

dacpac: 448
daemon: 16–17, 463
dashboard: 61, 107,
 172–173, 176, 180,
 247, 373, 443
database: 6–7, 16, 39–40,
 106, 111, 163, 178, 190,
 193–199, 201–202, 204,
 206, 208–215, 217–220,
 222, 231, 242–243,
 252–253, 255–256, 269,
 330–331, 405, 432, 442,
 445–446, 448, 534–536,
 557, 569, 577, 585–586,
 610, 612, 618, 634
databricks: 272, 274, 276,
 293–295, 297, 302–303,
 510, 534, 586, 607
datacenter: 7, 26–27, 29,
 31, 36, 38, 61, 70, 76,
 80–82, 86, 197, 230,
 251, 253, 392, 545,
 556–557, 565, 596, 613
dataops: 619

dataset: 179, 282,
 287–289, 292, 298, 406,
 536–538, 541, 581
decryption: 33,
 227, 239–240
deployment: 4–8, 10–12,
 14, 17, 19–20, 24–25,
 39, 42, 45, 49, 58–59,
 72–75, 78, 152, 163–164,
 179–180, 184, 188, 190,
 194–200, 211, 213,
 218–219, 222, 225–226,
 228, 230–231, 233,
 235–236, 239, 244–245,
 247, 256, 268, 306,
 308, 314–315, 343, 358,
 425–427, 429, 431–437,
 440, 443, 445–446,
 460, 463, 466, 472–473,
 475, 478–479, 483, 485,
 488–490, 492, 498–500,
 506, 509, 512–513,
 516–521, 525–526,
 534, 545, 547–548,
 550–551, 554, 556,
 567, 569–570, 572–573,
 607, 642, 651, 659
devops: 6, 11, 226–227,
 286, 303, 418, 421–427,
 430, 435–444, 457–458,
 461–462, 464–466,
 471–473, 479, 493
dnsserver: 544
downstream: 14, 274,
 410, 582, 614

E

ecosystem: 1, 6–8, 44, 303,
 390, 424, 443, 477, 484,
 585, 603, 612, 637, 651
emulator: 327

encryption: 33, 70, 75, 79, 85, 208-209, 227, 231, 239, 251, 253-254, 267-268, 358, 632
endpoints: 4, 19, 29, 35-38, 40, 125, 256-257, 265, 268, 316, 321, 325, 334, 486-487, 588-589, 591-593, 612, 643-647, 659
enrolment: 168
entities: 84, 87, 220, 227, 586
eventgrid: 336, 342, 365, 381
eventtype: 335, 341, 368
extension: 36, 50, 240, 307, 325, 336, 377-378, 429, 463, 558, 564, 610

F
failover: 40, 78, 86, 89, 221
filename: 115, 137, 297, 497, 537
filesystem: 208, 237, 360-361
firewall: 29, 33, 38, 40, 74, 189, 204-205, 233-237, 245-247, 252-253, 265, 269, 446, 458, 483, 493, 557, 570, 596
formats: 256, 272-273, 277, 298, 317, 405-406, 583, 646
framework: 9, 18, 46, 222, 275, 307-308, 423, 429, 468, 501
frequency: 84, 184, 348, 412, 537-538, 540, 581

G
github: 54, 293, 303, 535, 545, 577
gremlin: 219-221
H
hadoop: 7, 269, 272, 274-275, 303, 389, 586-587, 622
header: 125-127, 252, 297-298, 341-342, 368-369, 406, 495, 592, 644-646, 648, 653, 655
horizontal: 51, 212
hosting: 1, 3, 14, 30, 33, 38, 46, 74, 131, 133, 189, 197-198, 226, 229, 238, 285, 315, 343, 407, 414, 445, 476, 493, 503, 506, 526, 532, 584
hostname: 485
httpclient: 653-654, 656-657
httppost: 653, 657
httpstart: 325-326
I
identifier: 117, 128, 241, 289, 335, 375, 569
images: 14, 17, 58, 84, 272, 316, 443, 463, 479, 641, 643-644, 651-653, 655
import: 112, 122-123, 137, 300, 362, 364-365, 469, 636
ingestion: 273, 277, 391-392, 399, 403-404, 578, 581, 584-585, 587, 606-607, 612, 633, 635, 641
installer: 18
instance: 12, 26, 30, 32-33, 36, 46-47, 49, 54, 56-57, 64, 70, 148, 152, 182, 185, 194, 198-199, 210, 213-214, 222, 245, 272, 277, 279-280, 282, 289, 346, 357-358, 390-391, 393, 487, 492-493, 501-503, 519-520, 522-524, 550, 581, 632, 656
integrity: 76, 227, 625
invoicing: 168, 176-177
isolation: 4, 10, 15-16, 73, 76, 83, 90, 195, 220, 247, 432, 497, 549, 596, 604-605
J
javascript: 19, 58, 265, 307, 405, 407, 430
jenkins: 6, 418, 422, 464-466
K
kernel: 15-16, 462, 476
keyvault: 241, 365, 521, 525, 569
kubectl: 495-497, 499-500, 504
kubelet: 483, 493, 504-507
L
lambda: 307
libraries: 17, 58, 265, 432, 463, 640
license: 188

localhost: 137, 139-140, 471
locations: 14, 18, 58,
81, 86, 153, 164, 221,
519, 556, 651
logging: 59-60, 74, 115,
141, 238, 263, 265,
358, 477, 479, 532
lookup: 405

M

machine: 7, 25, 30, 57,
65, 70-72, 74, 84, 99,
127, 130, 145, 161, 180,
185-186, 188-190,
197-198, 213, 231-233,
235, 238, 256, 268, 274,
276, 279, 295, 315, 356,
360, 389-390, 405,
407, 429-430, 442-443,
460, 462, 467, 471, 473,
478-479, 526, 528-530,
532, 541-545, 551, 556,
558-560, 562-565,
570, 572, 586-587,
603-604, 606-607,
622, 637, 650, 653

mainframe: 1
mariadb: 196
master: 169, 213, 275-276,
294, 343, 448, 478-484,
492, 500, 535, 545,
555, 557, 565-567,
571-572, 651

memory: 14, 44, 46-48,
51, 186, 199, 204,
217-218, 361, 408, 633

metadata: 12-13, 51, 152,
167, 309, 485, 488-491,
496, 498-499, 564,
591-592, 609-610,
613, 619, 632

migration: 11, 279-280,
602, 611, 613-617,
619-621, 623, 625,
632-637, 641
modeling: 228-229, 642
monitoring: 11, 23, 32, 42,
52, 58-59, 61-63, 66-67,
74, 187, 198, 210-211,
222, 229, 231, 244, 265,
267-269, 279, 313, 316,
319, 389, 435, 443, 462,
471-472, 477, 483, 577,
588, 607, 610, 642

N

namespace: 15, 279,
283, 393-396, 400,
403, 409-410, 497,
504, 512, 551
native: 178, 213, 257,
479, 501, 609-610
nginx-lb: 499-500
notebook: 295-296

O

offload: 33, 343, 637
on-demand: 2-3, 6, 14,
240, 306, 318, 603-604,
606-608, 639, 641
openid: 227, 257
oracle: 7
os-level: 30

P

package: 18, 243, 255, 324,
429, 435, 446, 564, 656
partition: 275, 398,
400-402
password: 112-116, 210,

240-241, 359-361,
364, 380, 386, 495,
502, 536, 544
pattern: 69, 86, 90-101,
218, 265, 333, 415,
466, 555, 584,
592-593, 603, 659
payload: 35, 66, 184, 227,
308-309, 335, 341, 368,
382, 495, 649, 655
petabytes: 8, 85, 274,
582, 588, 606
pipeline: 278, 280,
282-284, 286-287, 291,
293, 431, 433-434,
443, 446, 448, 452,
457, 459-461, 463-464,
472, 538, 541, 603,
609, 612, 633, 650
playbook: 247
policy: 8, 13, 56, 145-148,
152-153, 155-156,
162-164, 174, 278,
361, 395-396, 399,
522, 540, 565
postgresql: 6, 196
postman: 127, 328
powershell: 8, 10, 17-20,
50, 72, 108-109, 111-112,
114-115, 117-118, 120-121,
125, 127, 134, 148-150,
152, 157-159, 163, 237,
266, 316, 340, 356-357,
362, 364, 366, 386,
429, 440, 443, 458,
466-468, 470, 472,
493, 510, 512-513, 515,
519, 521, 543-544, 564,
567, 646, 649, 655
premium: 30, 47-48,
84, 87, 190, 200,
212, 294, 315, 568

pricing: 141, 168, 178, 184-185, 191, 214-215, 217-220, 222, 247, 315, 347, 370, 393-394, 409, 594-595, 598, 608
principal: 59, 107, 111, 113, 115-118, 121, 161, 210, 241, 252, 287, 358-362, 367, 380-381, 516, 569
protected: 66, 75, 230-231, 236, 239, 247, 251, 256-257, 265, 360, 522, 553, 593, 645
protocol: 29, 33, 35, 84-85, 88, 227, 232-233, 251, 257, 341, 370, 392, 485, 488, 490-491, 497, 579-580, 584, 588-589, 593
psgallery: 564
publish: 333-334, 341-342, 357, 368-369, 385, 586
pyspark: 300
python: 6, 109, 112, 307, 316, 356, 466, 586, 603-604, 606-607, 646

R

raspberry: 589
rawdata: 282
readable: 19, 202, 405, 430
reboot: 27, 30, 564
rebuild: 503
recovery: 4, 7, 197-198, 201, 221-222, 513, 545
redundancy: 25-27, 58, 86, 189, 200

regions: 7, 12-14, 27-29, 36-41, 67, 69-73, 75, 79-80, 101, 106, 163-164, 189, 191, 197, 202-203, 219, 221, 230, 246, 333, 512, 515, 519, 545, 556, 649, 654
registry: 15, 237, 316, 479, 485, 650-652
reliable: 82, 84, 88-89, 101, 103, 275-276, 426, 578, 580, 597, 640
remote: 81, 98, 238, 246, 459
replicas: 86, 221, 489, 491, 498-499
replicaset: 482-483, 488, 490-493
report: 176, 180, 182, 184, 436, 442, 577, 606
resilient: 29, 38, 98, 274, 298, 436
response: 33, 43, 99, 127-128, 131, 182, 184, 221-222, 233, 244, 247, 315, 321, 493, 647-650, 652, 654-656, 658
retrieve: 19, 150, 159, 182, 241, 277, 330, 332, 341, 368, 651-652
revoke: 250
rgname: 569
role-based: 8, 115, 145, 242, 294, 504, 609, 633
runbook: 105-106, 109, 111, 118-121, 123-129, 131-132, 134-136, 141, 333, 356-357, 366-367, 369, 379, 382, 385, 466

S

scaling: 3-4, 44-49, 51, 53-54, 56-57, 197, 315, 475, 477, 576, 594
scenarios: 29, 53-54, 76, 78-79, 96, 107, 109, 136, 180, 184, 222, 277, 301, 323, 392, 457, 584-585, 603, 609, 614
schedule: 53-54, 98, 237, 276, 307, 346, 369, 385, 425, 483, 493, 505, 586, 622
scopes: 170, 172, 237
script: 107, 111, 115, 118-121, 134, 136-138, 152, 237, 251, 356, 512, 516-518, 521, 525, 543, 558, 564-565
secret: 239-241, 261-262, 357, 359-360, 368-369, 519-521, 523-524, 569
sendgrid: 355, 357, 370-373, 375, 377-379, 386
sentinel: 243-245, 247
servicebus: 325, 395, 400
servicenow: 66
session: 15-16, 29, 32-33, 35, 38, 40, 263, 302, 392, 459
setting: 49, 57, 85, 107, 131, 136, 161, 211, 217, 266, 285, 315-316, 366, 371-373, 375, 378-379, 522, 566, 592, 605, 607
sharepoint: 233
signature: 85, 113, 249, 287, 395, 399, 522, 553
skuname: 533
slaves: 275

socket: 239, 392
sqlazure: 534
sqlclient: 636
sqldataset: 538-540
sql-query: 586
standalone: 236, 242, 244
stateless: 57, 142, 392
static: 58, 65, 75, 190,
 312, 320, 373, 576
streaming: 266, 391-392,
 403, 407-408, 414,
 418, 587, 637
structured: 84,
 272-273, 641
subnet: 37, 50, 73, 75, 207,
 232-233, 235, 238-239,
 501-503, 506-507,
 527-529, 562, 567
subscribe: 333-334,
 338, 379
subscribed: 333
switch: 27, 172, 262,
 478, 495, 615
synapse: 405, 586-587,
 598, 601-614,
 616, 619-637
sysadmin: 567

T

target: 29, 33, 87, 113,
 242-243, 277-278, 289,
 292-293, 310, 315, 319,
 390, 407, 426, 443,
 468, 613, 619, 651
template: 19-20, 56, 157,
 336, 348, 358, 446,
 489, 491, 498, 510-519,
 521-522, 524-527, 529,
 532, 534-535, 538,
 541, 547-561, 563,
 565-568, 570-573, 614

tenant: 77, 96, 117,
 257, 358, 361, 363,
 367, 380, 569
terraform: 428
tokens: 85, 87, 182, 245,
 249-250, 252, 569, 593
tracking: 63, 76, 178,
 247, 255, 437, 577
traffic: 29, 33-38, 40,
 45-47, 72, 74, 77, 79,
 82, 106, 142, 188-189,
 234, 237-238, 245,
 248, 390, 501, 589
twilio: 355, 357, 371-373,
 375, 378-379, 386
twitter: 257, 265, 269,
 307, 406, 411-413, 415

U

ubuntu: 235
upgrade: 54-56, 489, 594

V

validation: 277, 431,
 433-434, 437,
 460, 471, 481
vaults: 241, 268, 358, 366,
 368, 379, 525, 569
version: 11-12, 19, 51, 56,
 274, 279, 284-285, 293,
 295, 308, 324, 430,
 438-439, 441, 444, 478,
 488-490, 510-513, 516,
 531, 592, 604, 607, 610,
 645, 647, 651, 655
virtualize: 15, 462

W

warehouse: 277,
 586-587, 601-606,
 609-612, 614-619,
 622-623, 625-627,
 632-634, 636-637
web-based: 33, 606
webdeploy: 429, 432
webhook: 66, 119, 125-127,
 129, 131, 333, 342
webserver: 468, 470, 496
webserver-: 496
windows: 1, 6, 14-18, 50,
 63, 108, 188, 210, 234,
 236, 243, 297-298, 316,
 395, 400, 429-430,
 442, 462, 466, 476-477,
 503, 536, 552-553,
 558, 564-565, 596,
 642, 651-653

workbooks: 247

worker: 105-106, 111,
 119, 127, 134-136, 276,
 294-295, 478-481,
 483, 492, 500

workflows: 108-109, 278,
 321-322, 346, 350, 355,
 466, 586, 609, 622

workloads: 26, 58, 211,
 226, 233, 235, 237, 317,
 587, 604-605, 613

