Serverless Computing Using Azure Functions

Build, Deploy, Automate, and Secure Serverless Application Development with Azure Functions

VARUN KUMAR
KETAN AGNIHOTRI

bpb

Serverless Computing Using Azure Functions

Build, Deploy, Automate, and Secure Serverless Application Development with Azure Functions

---

Varun Kumar

Ketan Agnihotri

www.bpbonline.com

FIRST EDITION 2021

Copyright © BPB Publications, India

ISBN: 978-93-90684-946

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

Lt. Sh. Vinod Kumar
My Father, who is no more with us.

– Varun Kumar

Agnihotri Family

– Ketan Agnihotri

About the Authors

Varun Kumar works with Microsoft as an architect helping various enterprise customers and start-ups in their cloud transformations, conducting technical reviews of the existing architecture, defining technology roadmaps, and architecting multi-million-dollar solutions. He has more than 12 years of experience in leading solutions delivery in .Net tech stack, Microsoft Azure, and AWS. He also has vast experience in direct engagement with stakeholders at all levels, including CxOs, MDs, business owners, architects, SMEs, vendors, consultants, and designers, etc.

Varun is a lifelong learner and passionate about technology. He is Microsoft Certified Azure Solutions Architect Expert and The Open Group Certified: TOGAF® 9. In his career, he has contributed to multiple events as a speaker and has authored a few blogs. Within Microsoft, he is part of multiple initiatives for learning and shares his knowledge with developers inside and outside of Microsoft and students of various colleges.

Varun volunteers his spare time helping, coaching, and mentoring young people in taking up careers in technology. He also loves to play table tennis and watch movies.

Blog: https://cloudavenue.in/

LinkedIn: https://www.linkedin.com/in/varun-kumar-64576825

Ketan Agnihotri is a proud Indian working as an Azure Consultant at Microsoft and loves making new professional acquaintances. He is a Senior Full Stack developer having team-leading experience and is a Microsoft Certified Solutions Expert (MCSE). He is a seasoned team-oriented and an ambitious software engineer with a successful track record of delivering high-performance applications. He is also a speaker, blogger, and trainer working with people across the globe.

He likes to solve data structure, system programming problems, and try to learn something new every day. He has comprehensive work experience with industries like real estate, travel, automobile, healthcare, insurance, media marketing, and public sector projects. He helps people build software, make customers happy and engage people with technology. He also takes social work seriously and works with NGOs.

Please reach out to talk technology, business, cricket, yoga, or India, and he'll do his best to help you.

Blog: https://k10.in/

LinkedIn: https://www.linkedin.com/in/ketanagnihotri/

About the Reviewers

Abhijit Jana is a technology leader, strategist, solution architect, development consultant, and trusted technology advisor with 14 years of experience in the IT industry with expertise in development, architecting, engineering, consulting, services delivery, leadership, and mentoring. He is currently associated with West Pharmaceutical Services as a Director of Software Engineering and responsible for building, managing, and leading the overall application development and software engineering team. Before joining West, he was associated with Microsoft and held different delivery, development, architect, consulting, and pre-sales roles. As a development lead and architect at Microsoft, he managed several complex projects and delivered solutions to enterprise customers worldwide.

He is a former Microsoft MVP, Code Project MVP, speaker at various technology conferences and several IITs across India, and an avid technology evangelist. He is the author of the book "Kinect for Windows SDK Programming Guide" and co-authored the book "HoloLens Blueprint." He is also the founder of a very well-known community website for developers, architects, and consultants working on the Microsoft Technology Platform.

He shares his thoughts on his blog at You can follow him on Twitter at

Abhijit lives in Bangalore, India, with his wife, Ananya, and daughter Nilova.

Abhishek Mishra is an architect with a leading software multinational company and has deep expertise in designing and building enterprise-grade intelligent Azure and .NET-based architectures. He is an expert in .NET Full-stack, Azure (PaaS, IaaS, Serverless), Infrastructure as Code, Azure Machine Learning, Intelligent Azure (Azure Bot Services and Cognitive Services), and Robotics Process Automation. He has a rich 15+ years of experience working across top organizations in the industry. He loves blogging and is an active blogger in the C# Corner developer community. He has been awarded C# Corner Most Valuable Professional (MVP) - December 2018, December 2019, and December 2020 three years in a row for his contributions to the developer community.

He is an active speaker and delivers sessions on Microsoft Azure. He has spoken in leading conferences like C# Corner Azure Conference 2020, nopCommerce Days 2019 Mumbai, C# Corner Developer Conference 2019 Pune, Global Power Platform Bootcamp 2020 Pune, and many more.

Certifications to his credit – TOGAF Certified, Microsoft Certified Solutions Associate in Machine Learning, Microsoft Certified Azure Developer Associate, and many more.

Acknowledgements

I must first express my gratitude to my parents for all their sacrifices to make me the person I am today and my wife for putting up with me while I spent many weekends and evenings writing this book.

I want to thank my family, friends, and colleagues for supporting, encouraging, and tolerating me during the various phases of life.

I would also like to thank Suresh Vishwanath, my previous manager at Microsoft, for his continuous support and motivation. Thanks to Nik Gupta and Abhishek Arun, my managers in my previous organization, for believing in me and introducing me to Azure. They continuously mentored and guided me during my early days of learning cloud technologies. Thanks to Ashirvadam, my first manager; I worked with him in my first job as a software engineer. Reaching this position would not have been possible without all your help and guidance.

A sincere thanks to Abhijit Jana & Abhishek Mishra for his valuable suggestions and feedback in the technical review, which has helped improve the book's quality.

Thanks, BPB Publications, for giving me this opportunity to write my first book for them.

Finally, I would like to thank all the readers of this book and request them to share their valuable feedback and suggestions.

– *Varun Kumar*

Writing a book is harder than I thought and more fulfilling than I may have ever imagined. None of this would have been possible without great support from my family, colleagues, and friends.

A very special thanks to my wife Pranali Dahale for all the support throughout my career, for being an important first draft reviewer and helping me during the practical implementation of use cases added in this book.

A sincere thanks to Abhijit Jana, Abhishek Mishra for his valuable suggestions/feedback and BPB Publications for giving me this opportunity to write this book.

Finally, thanks to all those who have been a part of this Journey: Abhay Pandey, Suresh Vishwanath, Milind Agnihotri, Chitra Agnihotri, Rashmi Agnihotri, teachers, well-wishers, and readers.

– *Ketan Agnihotri*

Preface

The last decade had been a cloud computing revolution that has given a new shape to the world's digital infrastructure. The invention of Serverless has defined even a newer way of hosting applications without worrying about the underlying infrastructure and how it scales.

The book starts by explaining the concepts of cloud and various offerings provided by Microsoft Azure and progresses slowly to more advanced topics to give you an end-to-end understanding of Azure functions. It covers the fundamentals of serverless technology and Azure Functions by explaining the structure of the function app, development tools, how to create your first Azure function with the help of various tools, and how to integrate functions with other Azure services.

After covering the basic concepts, the book will take you to more advanced topics such as durable functions and serverless orchestration, and different design patterns for orchestration. You will also learn to create a durable function with the help of use cases.

After that, the book will help you understand the various security mechanisms to secure Azure functions in production and deploy them with the help of continuous deployment pipelines and manual deployments.

In the end, the book will cover the various ways to monitor and troubleshoot Azure functions running on production and connecting to various monitoring and analytics options on Azure such as Application Insights, Azure Monitoring Logs, etc. Finally, it will guide you how to integrate the Application Insights to Power BI to get even more insightful dashboards of the Azure functions.

By the end of this book, you would have all the necessary skills to work with Azure Functions, including creating functions and orchestration, securing the functions on production, deploying and integrating with CD pipelines, and making functions production-ready with the help of telemetry and insights.

To cover all these, the book is written in the following eight chapters:

Chapter 1 gives the basic understanding of cloud computing and Microsoft Azure, including Azure portal, regions, resource groups, Azure resource manager, availability zones, and its offerings for compute, storage, databases, security, etc. Then chapter talks about the evolution of cloud service models and how we have reached to options that we have today. Finally, it talks about what serverless computing is, its benefits, and offerings from various cloud providers.

Chapter 2 covers the Azure Functions in detail, and various hosting plan and pricing models' Azure functions provide. It also talks about how functions are different from WebJobs. Finally, the chapters explain the Azure Function App structure, its supported

languages and runtimes, and various development tools it provides to increase development productivity.

Chapter 3 will help you create your first Azure function using multiple options Microsoft provides, including-Portal, Visual studio, CLI, and ARM. It also helps to explore other options the Azure Functions provide.

Chapter 4 will cover various triggers and bindings available for Azure Functions and the implementation of commonly used triggers and bindings with the help of use cases. It will also cover how to convert a monolithic architecture into microservices using Azure Functions.

Chapter 5 will provide an overview of the durable functions and serverless orchestration. You will be able to understand behind the scenes working of durable functions and design patterns and bindings. The chapter will also discuss a common use case for durable functions.

Chapter 6 covers the security aspect of functions from a comprehensive view, including Azure Active Directory Authentication, role-based access control, network security, and enabling CORS. You will learn how to use managed identity to enable access to other Azure services. You will also learn how to enable and use SSL/TLS for functions and enable network security with Azure API management.

Chapter 7 talks about the deployment aspect of functions with the help of common deployment tools like Azure DevOps and GitHub action. It also explains the various deployment options for Azure Functions such as deployment slots, etc.

Chapter 8 will help you understand various built-in capabilities for monitoring and troubleshooting Azure Functions. You will learn how to integrate Azure Functions to Application Insights and create alerts. You will also learn to perform advanced-level diagnostics for Azure Functions and learn to integrate the Application Insights to Power BI to create rich visualizations for better analytics of Azure Functions.

Downloading the coloured images:

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/f380d4

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB IS SEARCHING FOR AUTHORS LIKE YOU

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

IF YOU ARE INTERESTED IN BECOMING AN AUTHOR

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

1. Overview of Azure and Serverless Computing
Structure
Objective
The evolution of cloud service platforms
On-Premises
Infrastructure as a service (IaaS)
Platform as a Service (PaaS)
Serverless
Getting to know Azure Cloud
Infrastructure as a Service
Platform as a Service
Software as a Service
Overview of Azure products and services
Introduction to Azure portal and regions
Creating an Azure account
Azure regions
Resource group
Azure Resource Manager
Azure availability zones
Azure pricing and cost analysis
Azure cost analysis
Review costs in cost analysis
View forecasted costs for a service
Overview of serverless computing
Benefits of serverless
Conclusion
Points to remember

2. Introduction to Azure Functions
Structure
Objective
Getting started with functions
Planning
Building
Testing
Deploying
Monitoring and operating
Benefits of Azure Functions
Azure Functions hosting plans
Consumption plan
Premium plan
App service plan
Difference between Azure Functions and Azure WebJobs
Introducing triggers and binding
Triggers
Bindings
Working with Azure functions app
Functions code
Functions settings
Folder structure
Runtime versions
Supported languages
Development tools
App Service Editor
Advanced tools (Kudu)
Console
Azure functions behind the scenes

Functions cold start
Cold start process
Pre-warmed instances
Preventing cold start
Conclusion
Points to remember

3. Creating Your First Azure Function
Structure
Objective
Creating a function in the Azure portal
Creating your first function app
Creating your first function in the function app
Testing and running the function in the portal
Creating a function using Visual Studio Code
Creating an Azure Functions project
Publishing a function application to Azure
Creating a function application using Visual Studio 2019
Azure CLI
Creating a function using Azure CLI
Create resource group
Create an Azure storage account
Create a serverless function app
Deploy the function app to Azure
Creating a function using an ARM template
Deleting the resource group
Conclusion
Points to remember

4. Azure Function Triggers and Bindings

Structure
Objective
Overview of triggers and bindings
Understanding triggers with a sample use case
Using Azure Functions to convert a monolithic architecture into microservices
Azure Functions Integrations in the portal
HTTP triggers and bindings
Configuring the custom HTTP endpoint for your function
Azure Blob storage triggers and bindings
Azure Functions Blob input and output bindings
Sending an email notification using SendGrid
Configuring SendGrid using the Azure portal
Configure Function App
Implementing 'AddMessage' function
Implementing 'SendEmail' function
Timer trigger and bindings
Conclusion
Points to remember

5. Durable Functions and Orchestration
Structure
Objective
Overview of Durable Functions
Orchestrator functions
Client functions
Activity functions
Entity functions
Understanding the working of Durable Orchestrations
Handling race condition

Sub-orchestration
Underlying Azure Storage provider for Durable Functions
History table
Instance table
Work-item queue
Control queue
Durable Functions patterns
Function chaining pattern
Fanning out and fanning in pattern
Async long-running APIs pattern
Monitor pattern
Human interaction pattern
Aggregator pattern
Working with Durable Functions
Activity function 1: Getting records from a SQL table to export
Activity function 2: Exporting records.txt file to Azure blob
Azure Function to send a file as an email attachment using SendGrid and Blob triggers
Running and testing Azure Durable Function locally
Publishing Azure Durable Function from Visual Studio
Handling exceptions in Durable Functions
Conclusion
Points to remember

6. Configuring Security for Azure Functions Security
Structure
Objective
Function access keys
Accessing host, master, and system keys
Accessing function keys

Creating a new key
Renew an existing key
Azure active directory authentication
Configuring Azure active directory authentication advanced settings
TLS/SSL certificate for function apps
Configuring protocol settings and TLS/SSL bindings
Managing private key certificates
Uploading public key certificates
Configuring auto-renewal settings
Using managed identities for functions
Enabling the system-assigned identity
Creating the user-assigned identity
Network security
Use Case 1: Integrating Azure Functions with an Azure virtual network
Creating a function app in the premium plan
Create a virtual machine in a virtual network without a public IP address
Connecting the function app to the virtual network
Creating a function proxy to access resources of the virtual network from inside the virtual network
Enabling CORS for Azure Function
Enabling access restrictions
Private endpoint connections
Network security using API Management
Conclusion
Points to remember

7. Continuous Deployment for Azure Functions

Structure
Objective

Deployment methods and technologies
Continuous integration and continuous deployment using Azure DevOps
Source control integration for deployments:
Setting up Azure DevOps account
Create Azure DevOps project in an organization
Pushing Azure Functions source code in Azure Repos
Build automation (CI) - creating a build pipeline
Deployment automation (CD) – Creating release pipeline
Building and deploying Azure Function using GitHub actions
Azure Functions deployment slots
Creating deployment slots
Swap slots
Conclusion
Points to remember

8. Troubleshooting and Monitoring Azure Functions
Structure
Objective
Integrating Azure Functions with Application Insights
Enabling Application Insights
Configuring log level
Configuring the aggregator
Query telemetry data using the Kusto Query language
Troubleshooting Azure Functions
Viewing exceptions in Application Insights
Troubleshooting guides
Diagnosing and solving problems

Starting Azure Functions diagnostics
Monitoring Azure Functions
Live streaming

Azure Monitor Logs
Azure activity log
Monitoring alerts
Creating an alert for your function
Performance monitoring of your function app
AppDynamics
Prometheus for Azure
Application maps
Integrating Application Insights with Power BI
API capabilities
Creating the API Key in the Azure Portal
Creating the API key using PowerShell and ARM
Connecting to Power BI
Conclusion
Points to remember

Index

CHAPTER 1

Overview of Azure and Serverless Computing

The software industry wants to adopt a modern application development method by building and deploying software on the cloud. The cloud industry is continuously evolving around software development, deployment, security, reliability, and performance. How can we solve business problems with the cloud? How can we focus on adding business value by developing great software and delivering a great customer experience? You will get answers to these questions in this and subsequent chapters, so let's get started!

Structure

In this chapter, we will cover the following topics:

The evolution of cloud service platform

Getting to know Azure Cloud

Overview of Azure products and services

Introduction to Azure portal and regions

Overview of serverless computing

Objective

After studying this chapter, you should be able to:

Understand Azure, Azure Services, and Azure regions

Understand the serverless computing components

Set up your Azure subscriptions

The evolution of cloud service platforms

The cloud service platforms have evolved through various stages to the options, which we have today. The industry shifted from on-premises to cloud - IaaS and PaaS to meet their business requirements. Serverless is the latest and most promising option in this evolution.

On-Premises

Before the advent of the cloud, everything was on-premises. Organizations had to buy/rent their hardware and software licenses and run them in their own or rented datacentres. They had to deal with a multitude of questions - the number of servers, physical and network security handling, hardware failures, software patching, backups, managing the underlined infrastructures, and more. Essentially a lot of questions, not all of which might be directly related to running the business. For example, insurance would like to focus on their business, selling policies rather than managing the IT infrastructure. Similarly, a pharma company would like to focus on selling medicines.

Figure 1.1 shows the things organizations may have to worry about with on-premises systems:

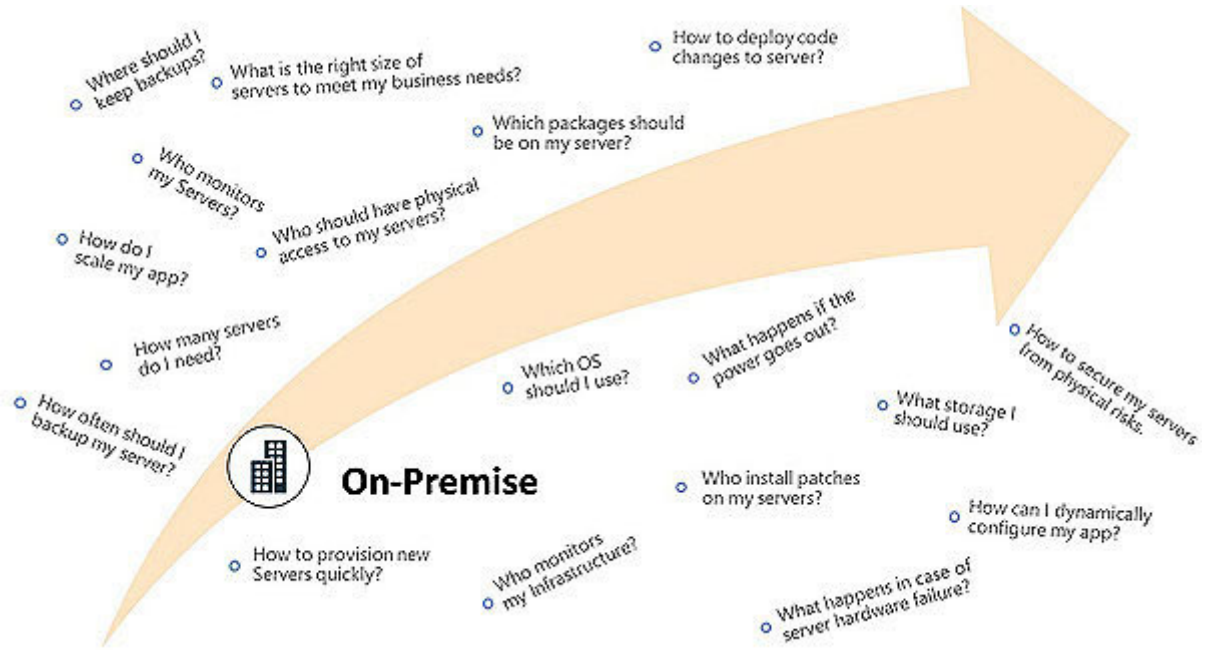


Figure 1.1: *On-Premise activities*

Then came the cloud, and it provided relief from some of the challenges with the on-premises model.

Infrastructure as a service (IaaS)

IaaS allowed utilizing servers hosted by the cloud service providers instead of hosting it yourself. While the worries around managing physical infrastructure were gone and it provides complete control to organizations, several questions still needed to be thought through:

How many virtual machines are needed currently?

How many virtual machines would be needed in the future when the usage increases?

How to monitors infrastructure and applications?

How to manage code deployment and various application dependencies, such as runtimes, and so on?

How to keep the latest security patches updated on servers?

This means that organizations cannot focus purely on their core business and not worry about IT infrastructure.

Figure 1.2 shows the reduced number of concerns organizations must worry about with IaaS:

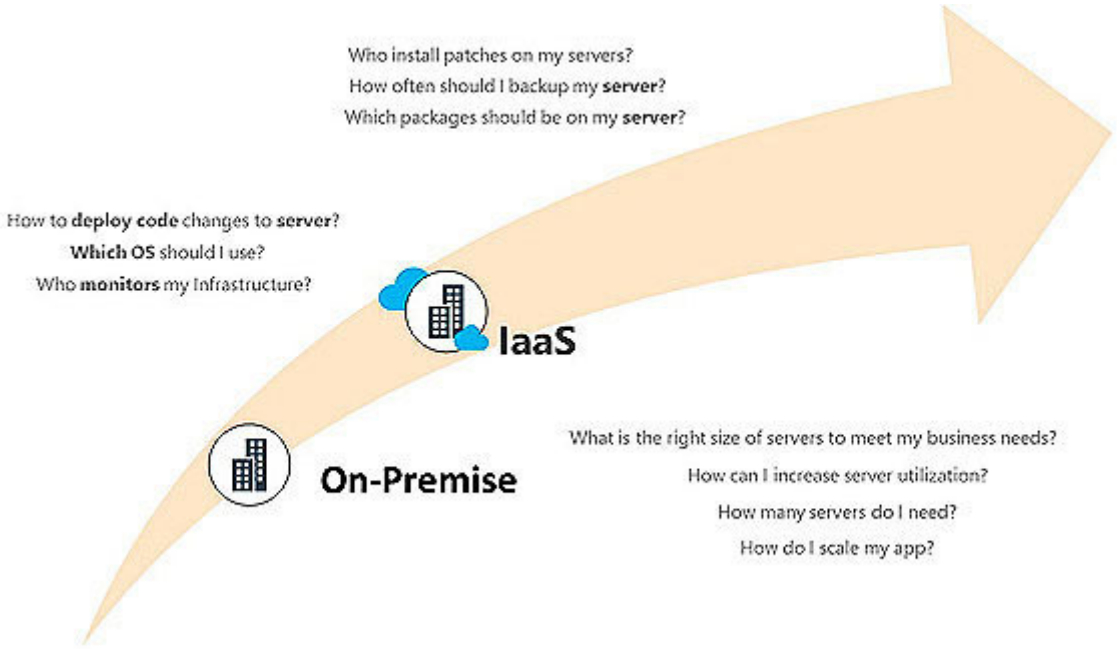


Figure 1.2: *Infrastructure as a Service activities*

Platform as a Service (PaaS)

The cloud continued to grow and helped with these problems. With PaaS, there were even more simplifications. PaaS allows organizations to just worry about their business applications and let the cloud provider take care of the physical hardware, ongoing management of the environment like operating systems, virtual machines or containers, as well as the various application runtimes, whether it is Java Runtime, .NET, Node.js, PHP, Python or others. The cloud provides configurable options and tools to monitor the resources on the portal.

Figure 1.3 shows even lesser concerns with PaaS:

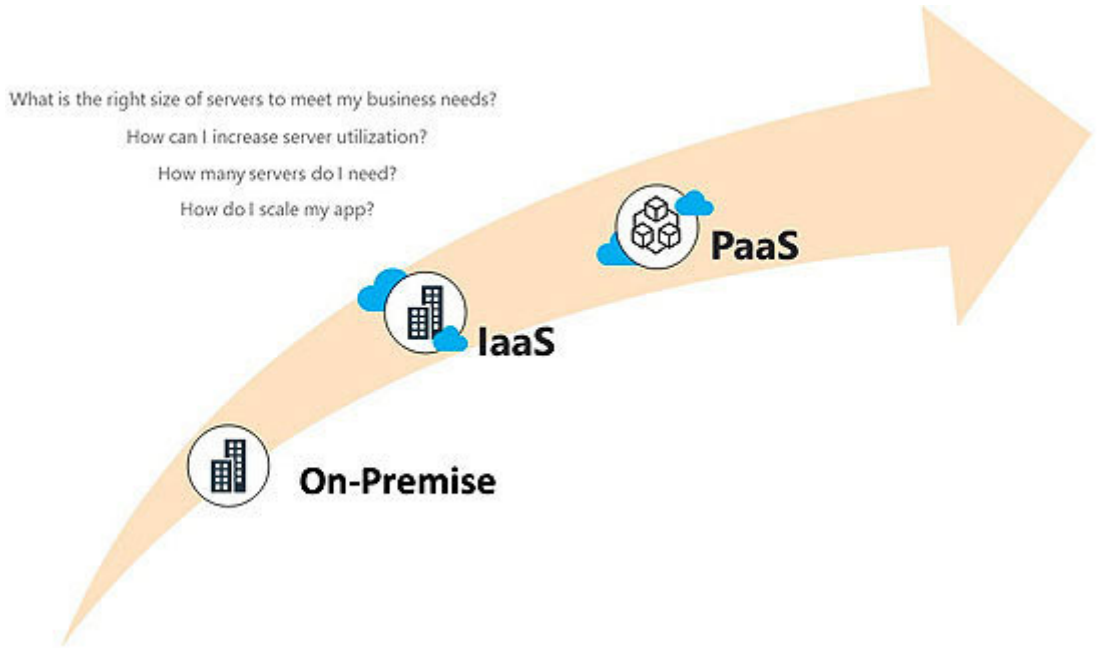


Figure 1.3: *Platform as a Service activities*

PaaS has become very popular because of the ease of deployment, monitoring, and troubleshooting. Moreover, cloud providers guarantee SLAs on uptime with PaaS offerings. However, they are still not completely free from thinking about how to scale their applications with business growth and more customers. This is limited to applications, but capacity planning must also be done to specify the number and size of the compute resources for the applications. The cost and volume have to be estimated.

Serverless

The latest evolution series is serverless, which provides even further abstraction from the underlying infrastructure. It allows organizations to focus on building the applications and making optimum use of them to achieve the best results for the business.

Serverless allows you to focus on your code and forget about the infrastructure.

Figure 1.4 suggests that all the questions are answered serverless:

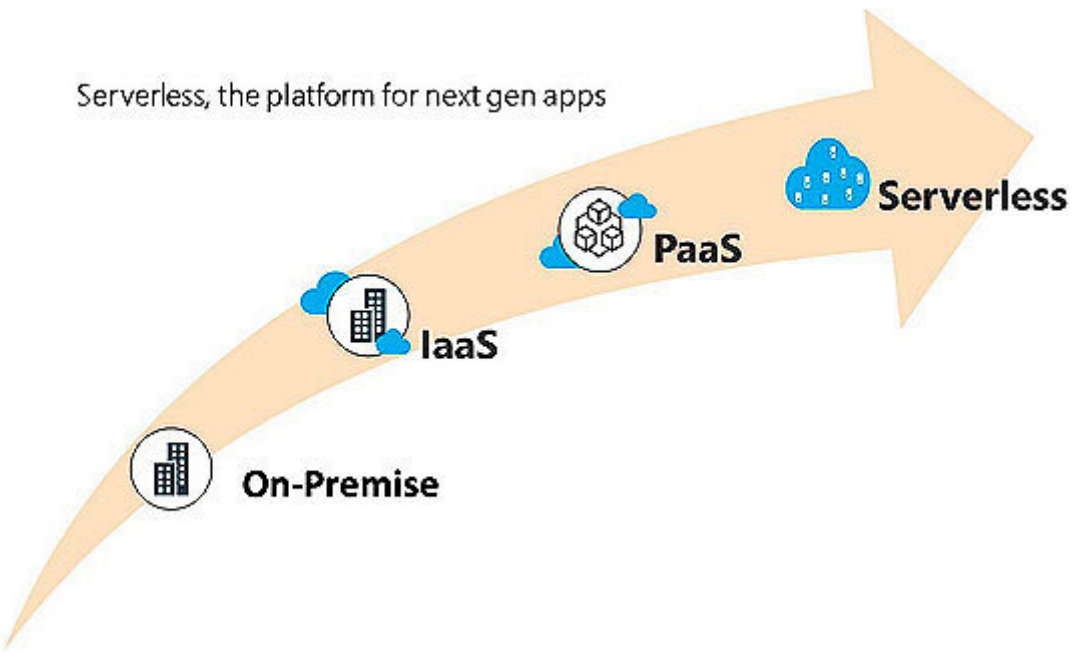


Figure 1.4: *Serverless Computing*

Getting to know Azure Cloud

Microsoft Azure, commonly referred to as Azure, is a public, private, and hybrid cloud computing platform founded in the mid-2000s by Microsoft. Azure helps build, test, deploy, and manage software applications and services through Microsoft-managed data centers located across the globe.

When you think about migrating an application to the cloud, you may have many questions, a lot of concerns about data management, data security, or how you will create and manage solutions. Azure takes care of these challenges.

While using Azure services like compute and data storage, you don't have to manage any hardware. You get rapid capabilities to expand data storage, data processing power, computation power, and any other kind of scale you need without investing capital in hardware.

In cloud computing, there is a different way that resources or cloud services can be provided in a cloud environment. Those ways are **Infrastructure as a Service Platform as a Service** and **Software as a Service** that can be used for services such as analytics, virtual computing, storage, networking, and much more.

The following *figure 1.5* gives a glimpse of different cloud services:

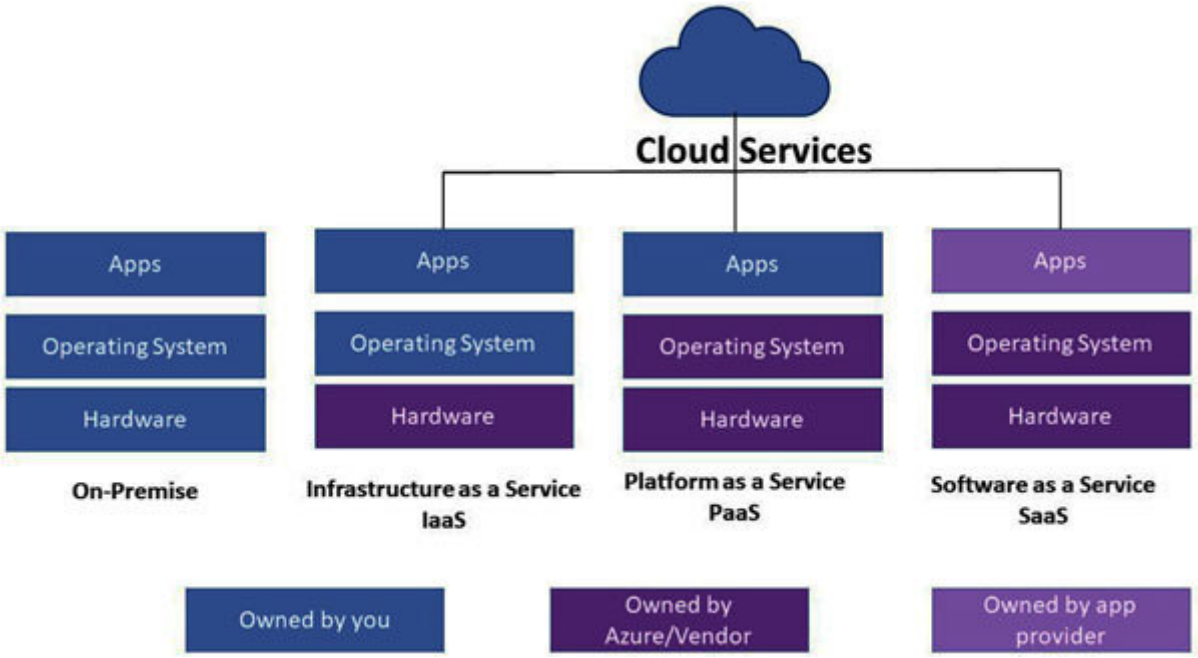


Figure 1.5: *Cloud Computing services*

Infrastructure as a Service

Cloud infrastructure services are known as IaaS, where you get provided with highly scalable and automated compute resources to create your computing environment like your own data centers. You can create compute resources like virtual machines, operating systems, hard drives, storage space, networking, and IP addresses, and so on through virtualization technology.

Platform as a Service

PaaS provides an abstract virtualization platform with infrastructure resources to run your applications on top of it, and you are not managing any infrastructure components like virtual machines, storage, hardware. PaaS helps rapid software development, testing, and deployment.

Software as a Service

SaaS is an even higher level of abstraction. You need to subscribe to applications available over the internet and do not require client-side installation. SaaS applications are hosted on a remote server by third-party vendors.

Now, the question that comes to mind is who is using Azure? How is Azure helping their business to grow? Most of the Fortune 500 companies trust their business to run on Azure and enriching customer experience. We are going to understand the evolution of cloud service platforms in the next sections of this chapter.

The following are the key benefits of cloud computing:

Reduced time and cost to provision

Increased flexibility in scale and deployment

Hybrid solutions

Data security and control over your data

Mobility

Overview of Azure products and services

Azure offers a range of services based on use cases with great benefits to build and manage powerful applications using Azure products and services. Azure services are divided into categories like compute, networking, storage, data analytics, databases, and identity, as shown in *figure*

Figure 1.6: *Azure Services*

Let's understand some of the important Azure services one by one.

Azure compute has many available services that are publicly available and can be accessed via the internet or a private connection. Choosing the right compute service, Azure compute, is closely related to the operating system, CPU, memory, and cost consumed by your application code while running on servers. You can create an Azure virtual machine with various parameters like operating system, location, compute power, memory, and so on. There is support for containers, an application packaging format, and a set of tools for running your applications. Azure compute helps manage and scale resources to meet demand and supports various languages and frameworks.

We have computing options like a virtual machine, VM scale set, Azure Kubernetes services, container instances, Azure app service, service fabric, Azure functions, and Logic Apps.

Let's understand commonly used Azure compute options:

Azure provides virtual machines in various sizes - CPU, storage, memory, graphics options with different capabilities and place in a virtual network.

This is a smartphone-based hosting service for web applications, REST APIs, and mobile backend, written in popular languages and frameworks like ASP.NET, ASP.NET Core, Java, Ruby, Node.js, PHP, or Python. Automatic scale up or out is possible to promise high availability. It has attributes like region, size, SKU, and scale count.

This serverless compute services hide underlying infrastructure, utilize compute based on required execution, and charge accordingly. We can provide input/output bindings to third-party services. The execution can be triggered and scheduled. As we will move forward with this book, we will be exploring this service in depth.

The idea of running an application code without thinking about instances, virtual machines, or any underlying infrastructure is supported by Azure compute, known as serverless computing. Azure serverless compute has services like

It is crucial to choose a secure, highly available, scalable, and durable data storage option in the modern era. Data can be structure/unstructured, images, videos, text files, or binary files, and Azure storage supports a wide range of data storage options. Let's understand the fundamentals of Azure storage and its types. Knowing all storage types available helps to choose the right to manage and improve the performance of your application.

Azure storage is a distributed cloud service that allows you to store and manage different data types that can be accessed anytime from anywhere. Azure storage services can handle burst loads and large amounts of data, traffic, or interactions with the database.

Let's understand the Azure storage types:

Azure blobs provide a facility to store unstructured data types, which can be any file, images, videos, etc., up to 500 TB per storage account. This can be stored by creating one or more containers, and each file is stored with some unique identifier. There are three types of blobs:

This storage type is recommended to store documents and media file block blobs in an organized and secure way.

If you want to store some piece of information in which you want to keep on adding some stuff at the end, for example, storing activity log information in a text file that continuously updates as you perform an activity, appends blob is the answer.

The main purpose of page blobs is infrastructure as a service. You can store VHD/disk in page blobs, and you can access it using REST API, PowerShell, and so on.

In this type of storage, you can store many structured data accessed inside and outside the Azure cloud. You will see data as tables with rows and columns in the Azure portal, but internally, it is stored as key-value pairs in entities. This type of implementation is called a

Queue storage is ideal for storing messages which can be used for asynchronous communication. Each message stored in a

queue can be up to 64 KB in size. You can also set an expiry for these messages. Crucial functionalities like load-levelling with messaging and workflow can be achieved with queues.

When you need storage of shared data that can be used using SMB file share, azure files can be used.

Why should one move databases to the cloud? Once you get familiar with the features of moving data to the cloud, you will get an answer to this question. So, there are multiple reasons to migrate databases to the Azure cloud. One of the main advantages is the reliability of the data. If one data center is down or fails to start, you can have another data center as a failover center. Other reasons are the capacity to scale up databases, security of the data, and features you get as a platform as a service. Microsoft Azure provides database migration services using which you can migrate all on-premises databases to the cloud.

In Azure, you can have a hybrid architecture of your database because Azure supports different types of database as listed as follows:

Azure supports open-source databases like PostgreSQL and MySQL.

You can have row and column-oriented database structures with the relational and static schema. This supports

vertical scaling, and it's the same as our traditional SQL database. For example, Azure SQL database managed instance.

With NoSQL databases, you don't need to predefine database schema, and unstructured data can be stored. Hence it supports dynamic schema and horizontal scaling. This is a great option to store big data.

We have various data storage options for applications like cloud-managed databases (Azure Cosmos DB), data storage for analytics (Azure Data Lakes), data processing tools (Ingestion Event Hubs), real-time data analysis, and high volume message processing. You can have all these types of databases as infrastructure as a service: database engine hosted on virtual machines and platform as a service that is a managed offering by Azure.

The Azure virtual network is a building block to create a secure private network that connects Azure resources such as virtual machines, databases, and on-premises servers.

The virtual network provides the following connection options:

Virtual network peering allows you to connect resources within the same region

Global Virtual network peering allows you to peer virtual networks in different Azure regions to build a global private network in Azure

Point to site connectivity helps a client to connect to a virtual network from his computer

Site to site connectivity type of connection helps the on-premises network to connect Azure compute services, IaaS, PaaS

VPN gateway is used to send encrypted traffic between an Azure virtual network and on-premises servers over the public internet

Express route, a private connection from on-premise to Azure services, is expensive than gateways but can be used when there are compliance requirements

Please refer to the sample Azure network with networking component in *figure*

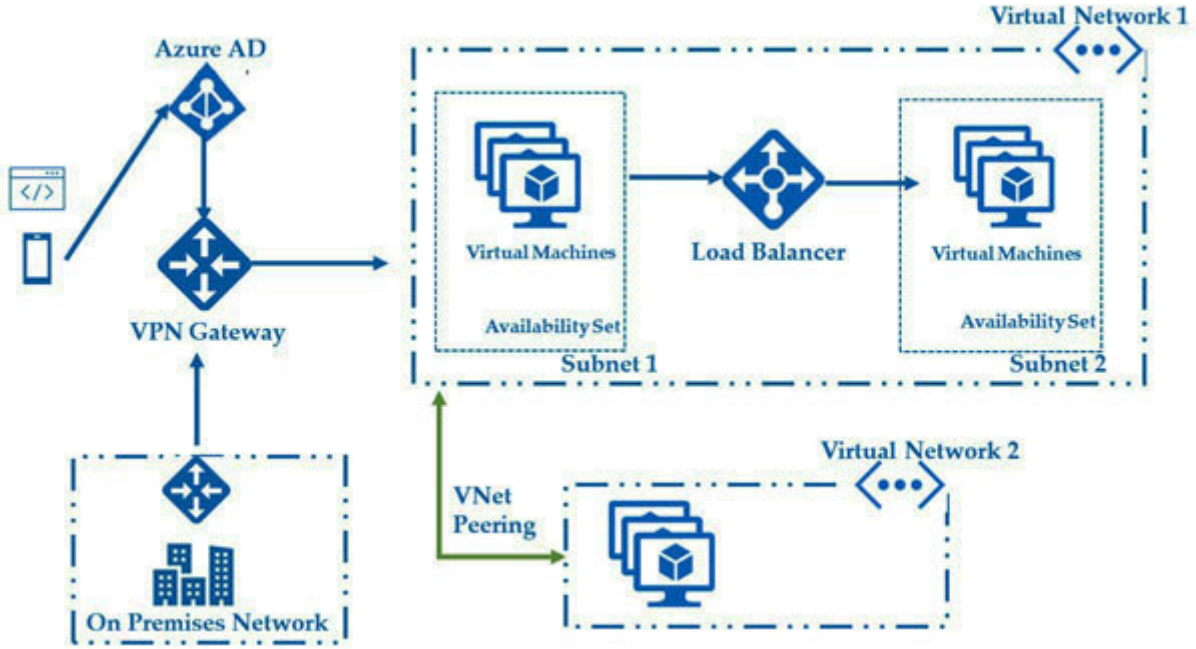


Figure 1.7: *Sample Azure Network*

Azure has the following load balancing services:

Azure application gateway provides OSI layer-7 web traffic load-balancing capabilities for your applications and supports features like web application firewall, cookie-based session, autoscaling, SSL offloading, and web application firewall

Azure traffic manager is DNS-based load balancing that distributes traffic optimally to services across all Azure regions

Azure load balancer balances traffic at layer four and allows you to scale your application to achieve high availability and resiliency

To ensure security, identity, and access management, Azure has services like Azure Azure Active Directory B2C, Azure Active Directory Domain Services. These services work well with cloud as well as hybrid environments.

Azure AD has features like directory synchronization, conditional access, single sign-on, and multi-factor authentication to ensure advanced data and identity protection and prevent cybersecurity attacks.

Azure Security services provide data protection tools like *Information Key* and *Hardware Security*

Azure Key Vault is a key-value pair-based cloud service for securely storing and accessing secrets like API keys, passwords, certificates, or cryptographic keys. It is always recommended to have an Azure Key Vault per environment with the least privilege access principal to grant access. You can enable role-based access control and can take regular backups of the Azure Key Vault.

Introduction to Azure portal and regions

To use Azure, first, you need to understand core concepts like Azure subscription, Azure regions, resource groups, and pricing.

Creating an Azure account

To get started with Azure, you need to create an Azure account by signing up on You can get started with 12 months of free services. You can get more details by visiting

Azure regions

Azure services are offered in regions throughout the world. The Azure region consists of one or more data centers, also referred to as availability zones. Azure is available in 60+ regions globally. New regions and data centers get added every year. The Azure region offers flexibility to deploy applications closer to customers around the globe. While choosing the region, we need to consider factors like pricing, service availability, compliance, and data residency.

The following image shows the Azure regions available across the world:

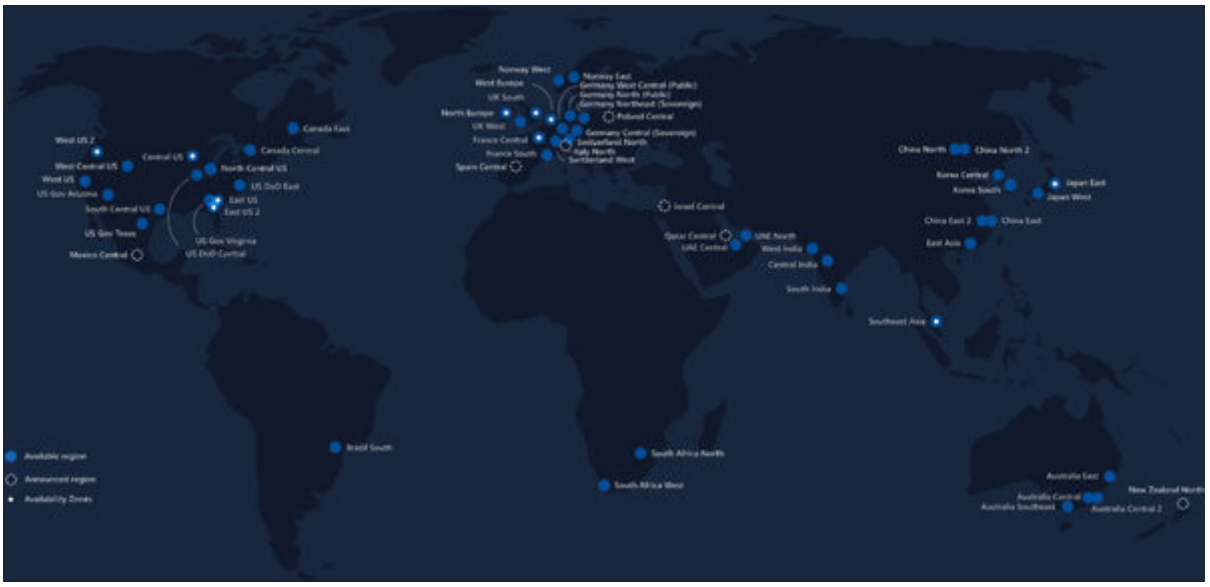


Figure 1.8: *Azure Regions*

Note: You can get the latest Azure region details by visiting the following link:

https://azure.microsoft.com/en-in/global-infrastructure/regions/

Resource group

Azure has four levels of management scope: management groups, subscriptions, resource groups, and resources.

Management group helps you to manage access, policy, and compliance for all the subscriptions.

Subscriptions relate to user accounts and resources. You can define a limit or quota on the number of resources you can create and use.

A resource group is a collection of resources that share the same lifecycle, permissions, and policies. The resource group can include the resources for a solution or department or any other criteria that you want to manage as a group based on the organizational needs. It makes sense to add resources that have the same lifecycle to the same resource group to be deployed, configured and deleted easily. Also, the resource group stores metadata about these resources in the same location where the resource group is created. Therefore, the resource group should be created in regions as per the organization's compliance policy. Please refer to *figure*

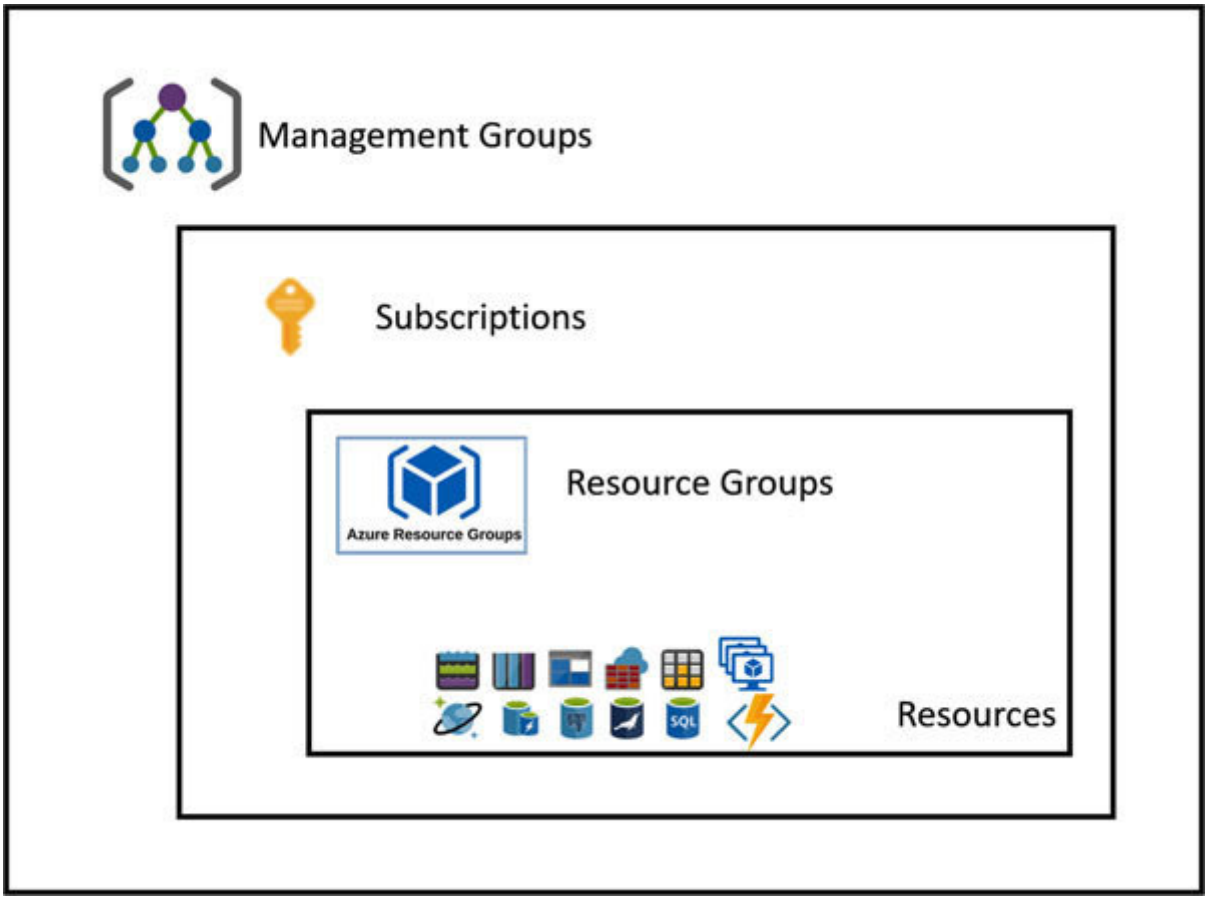


Figure 1.9: *Management scope levels*

Resources can be moved between the resource groups. A resource group is a logical grouping of resources and not a boundary of access. **Role-based access control** can also be managed using resource groups and applied on the entire subscription or user level. We can add tags to Azure resources which helps to categorize, manage billing, search and group the Azure resources.

Creating a resource group in the Azure portal

In the Azure portal, select **Resource** Please refer to *figure*

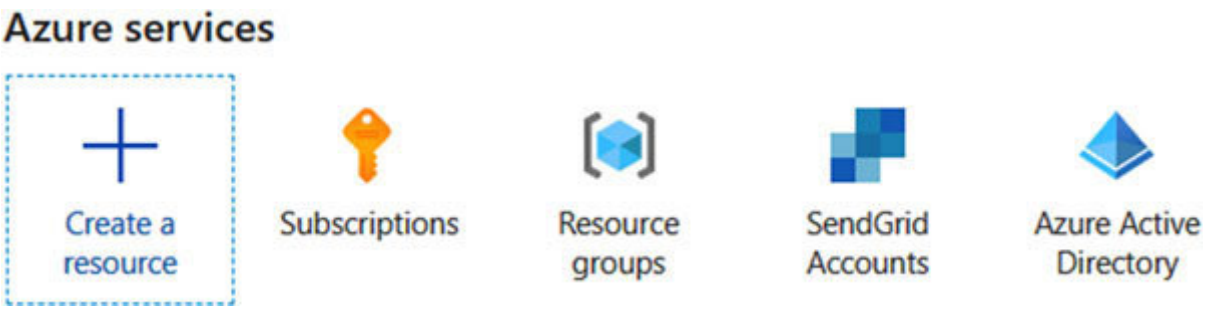


Figure 1.10: *Create Resource Group*

It can also be found in the left pane under **Settings** in **Subscriptions,** as shown in *figure*

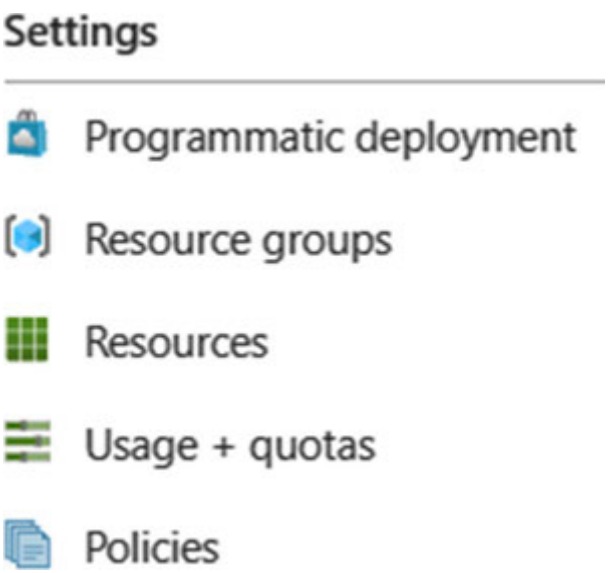


Figure 1.11: *Create a Resource*

Click on **Add,** as shown in *figure*

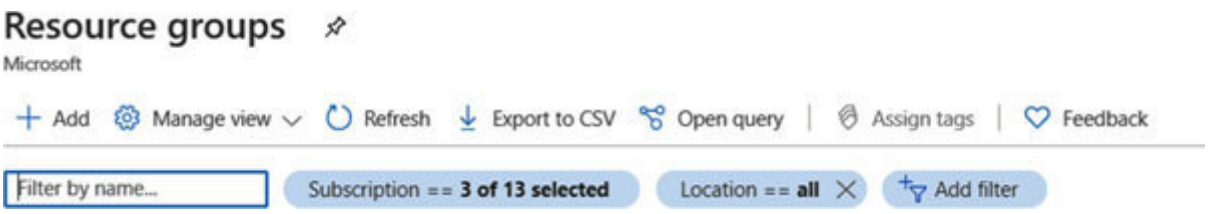


Figure 1.12: *Add Azure Resource*

Enter the following values:

Select your Azure subscription in which you want your resource group and resources to be created in. A single subscription can utilize multiple Azure regions.

Resource Enter a new unique resource group name that you want to give to your group.

Select an Azure location (based on an organization or compliance needs). Please refer to *figure 1.13* for the mentioned fields:

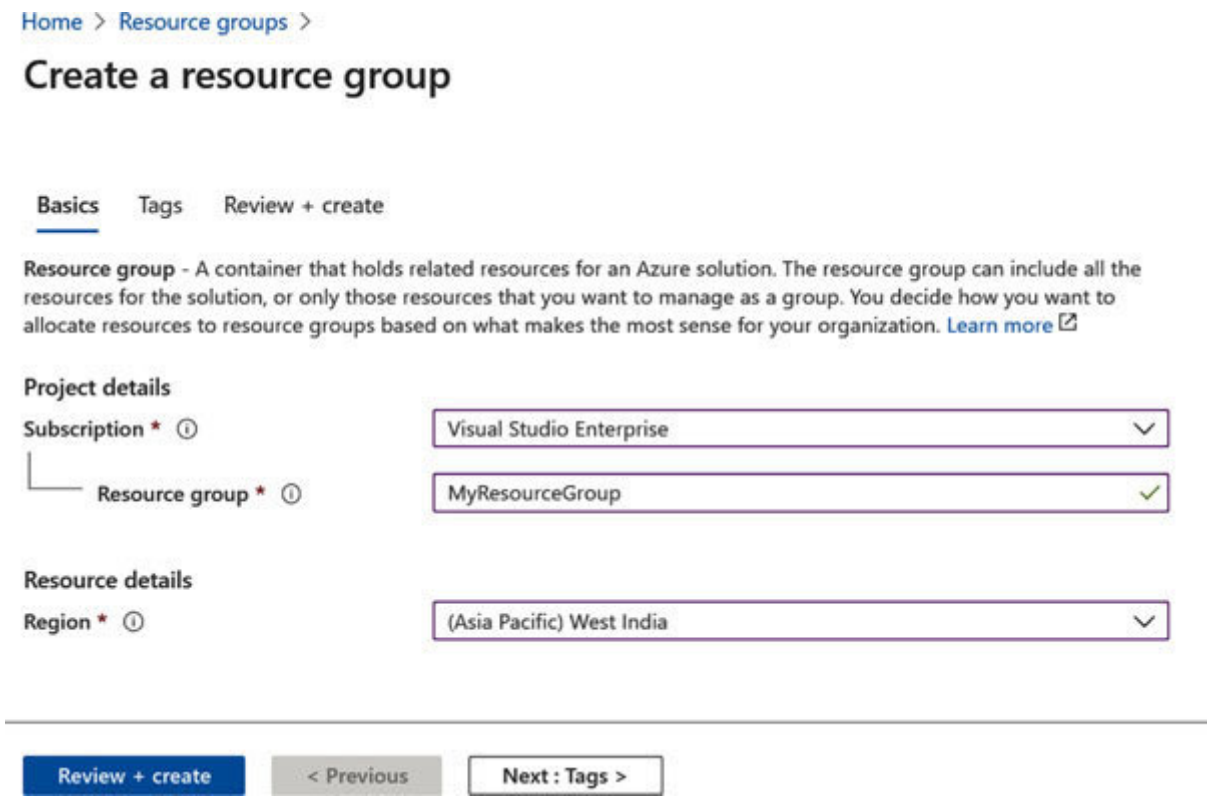


Figure 1.13: *Create Resource Group*

Click on **Review +**

You will see the **Validation Passed** message.

Click on It may take a few seconds to get the resource group created.

Once created, click on **Refresh** for the list of resource groups to be refreshed and select the resource group you have created to open it. Alternatively, you can also click on **Notification** from the top and then go to resource group to open the same.

Azure Resource Manager

Azure Resource Manager templates are a key component for JSON-based deployment, supporting declarative parametrized templates. We get JSON templates for resource groups which provide a scalable and repeatable method for deploying Azure resources.

ARM templates ensure parallelism and can be used with Azure CLI, Azure portal, or PowerShell to accept parameters like type of resource and resource location.

This is a widely used approach enabling **Infrastructure-as-Code** deployment automation, and standardization. We can download, export, and save JSON-based ARM templates. We can do nesting of JSON templates. Please refer to *figure*

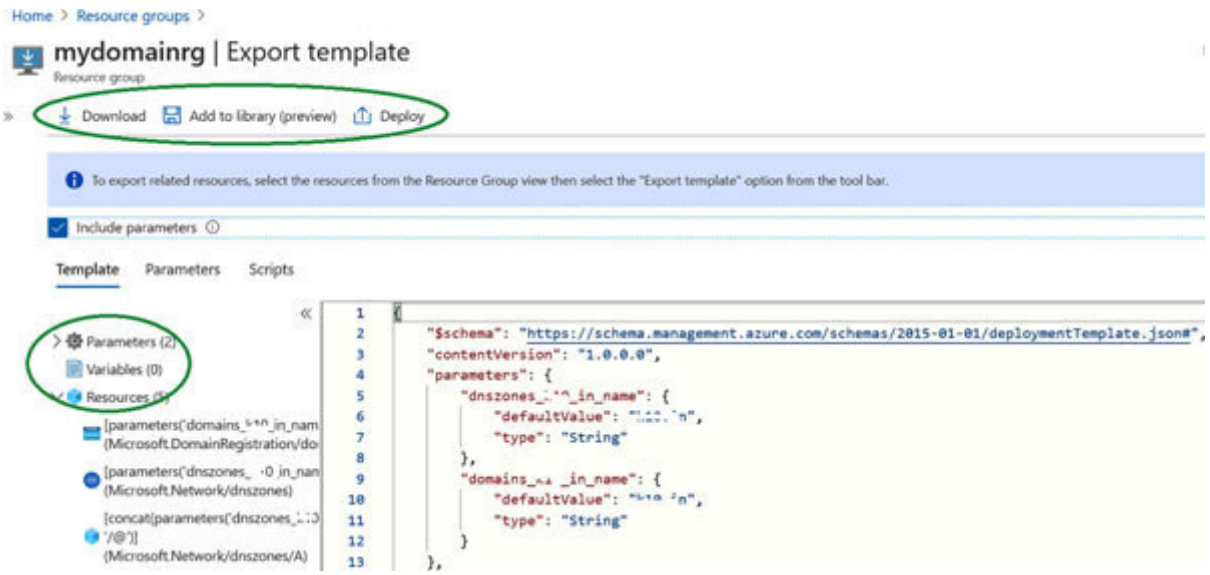
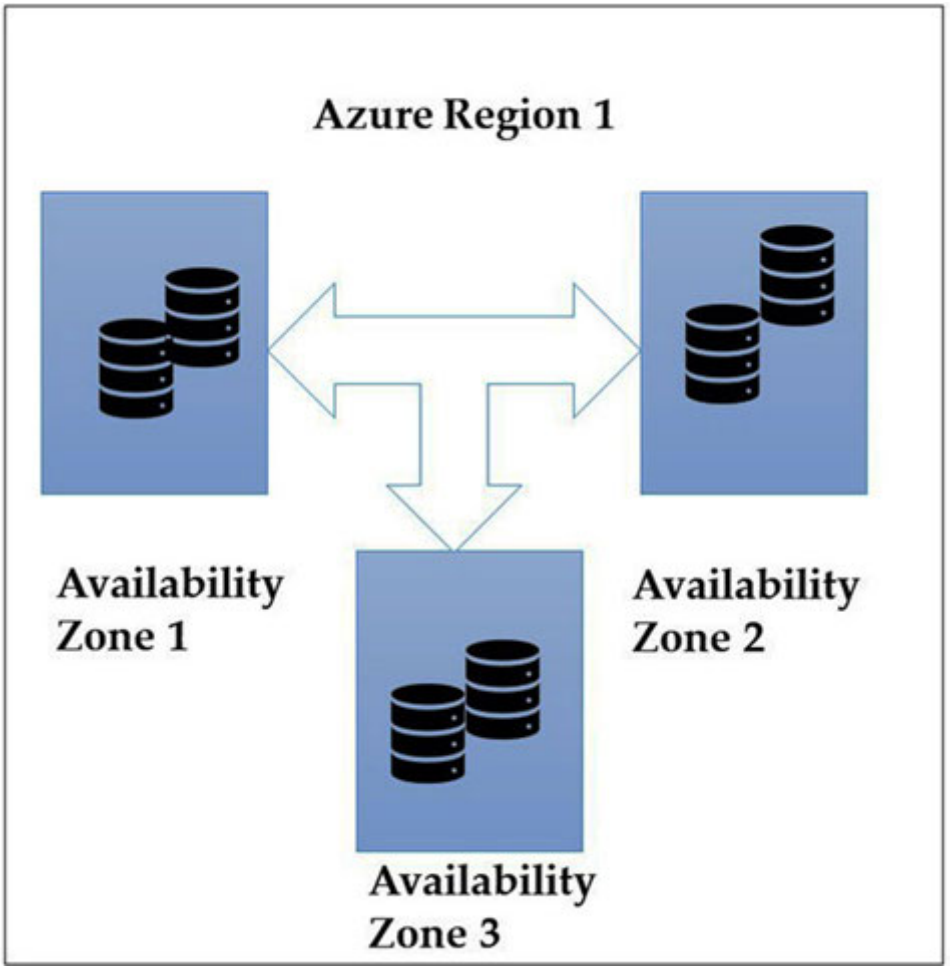


Figure 1.14: *Export Template*


Azure Region 1
Availability Zone 1
Availability Zone 2
Availability Zone 3

Azure pricing and cost analysis

Azure cost analysis

Azure cost analysis provides an option to analyze the organizational costs of various Azure services. The aggregate costs view shows the spending trends and understand where costs occur more. Also, you can view accumulated costs overtime to estimate monthly, quarterly, or yearly cost trends against the defined budget. You can also download the report's data for further analysis.

Figure 1.16 shows a sample cost analysis view with accumulated costs for a particular subscription:

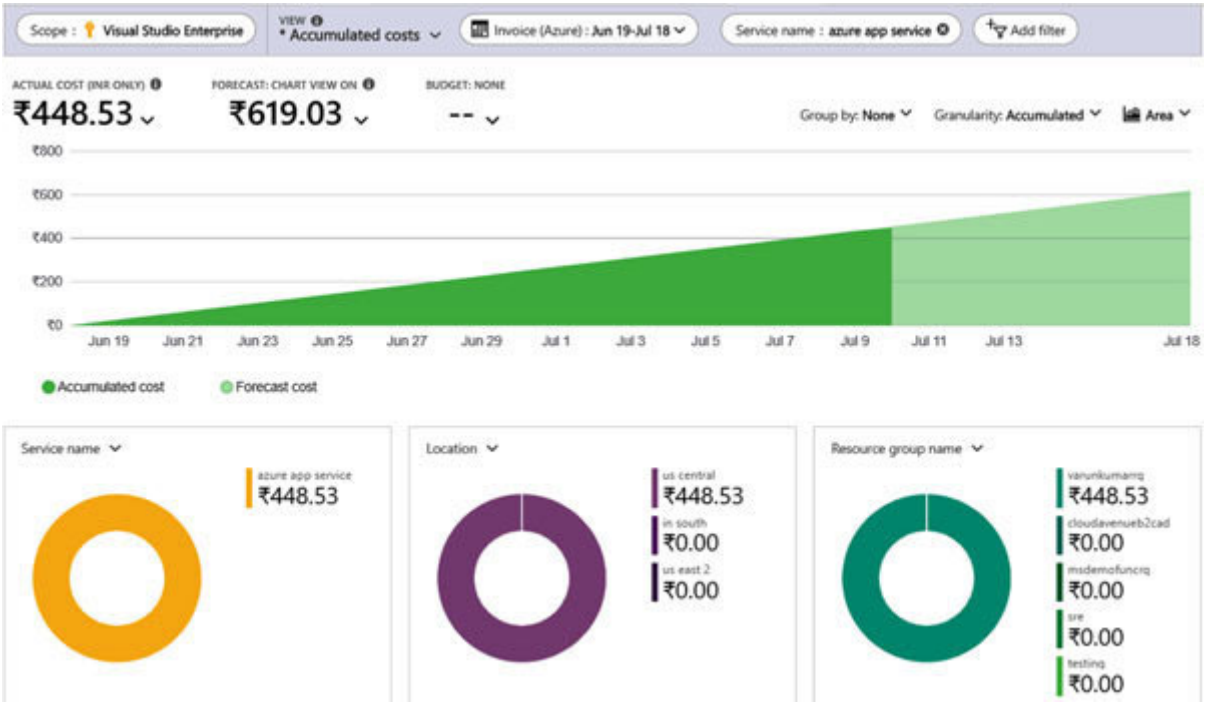


Figure 1.16: *Azure Cost Analysis*

Review costs in cost analysis

To access cost analysis, go to select the subscription from the list you want to review cost for, and then select **Cost analysis** in the menu. The scope can be changed to a different subscription, resource group, directory, etc., in cost analysis.

The cost analysis view provides the following options:

Cost You can choose from multiple available built-in views– accumulated costs, daily costs, cost by service, or resources. Each view provides an option to select the date range, granularity, group by, and filter settings.

Actual It shows the usage and purchases for Azure and third-party marketplace offerings as shown on the invoice. It also gives an option to select the currency in which you want to view cost.

It shows the estimated costs for the selected time. This is based on the time series regression model and takes at least ten days to forecast costs accurately.

It shows the spending limit that you have planned for the selected scope.

Accumulated It shows the aggregated daily costs from the start of the billing period.

It provides dynamic charts to visualize cost trends in different representations and can be changed to the area, line, column, or table.

View forecasted costs for a service

Azure cost analysis can also be used to view forecasted costs for a single service. For example, the Azure function app:

In the Azure portal, go to **Cost Management + Billing → Cost Management → Cost**

Click on

Select **Service name** to see the cost.

From the list, select the service, for example, the Azure app service.

Please refer to *figure 1.17* for a sample cost analysis:

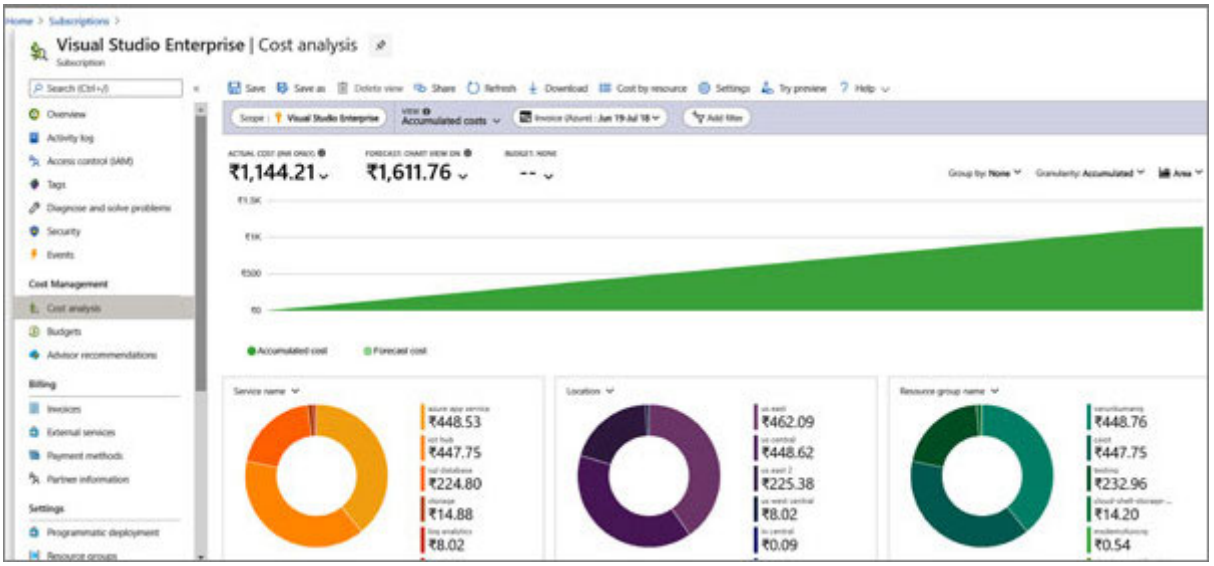


Figure 1.17: *Forecast of Azure Service Cost*

You can even add more customizations to the view by adding a second filter.

Overview of serverless computing

Serverless is an event-driven, stateless code execution environment in which you write code and consume services. It doesn't mean that there are no servers, of course. It just means that consumers don't have to manage servers. Cloud providers will manage it for the consumer. In simple words, it's a system where you don't have to think about servers at all, so for you, they are kind of invisible.

When we talk about serverless, the first thing that comes into the picture is **Functions as a Service** which can execute custom code on demand without worrying about where the code is executing or the scaling when the number of requests grows. The user only pays for the time in which the code gets executed. Different cloud providers have their offerings for the serverless–AWS has Lambda and Microsoft Azure has Azure Functions.

Serverless computing has brought in a paradigm shift in computing, with deployment taking few seconds as opposed to hours or days with physical machines.

The next question is: What kind of apps and scenarios can be built using serverless technologies?

Virtually anything that needs some code or logic to run in response to an event. Some examples could be as follows:

Real-time stream processing, where IoT devices send data that needs to be analyzed, transformed, aggregated, and stored in storage or databases.

Every organization needs to perform routine tasks, such as archival of database entries that are no more required for current transactions at the end of every day or truncating logs at the end of every week. Such tasks that need to be run only at specific times and may require few seconds to a couple of minutes to complete don't require dedicated infrastructure.

Backends for mobile or web apps can also be a good use case for serverless platforms. For example, a mobile app sending an image to a function to store into an Azure Blob Storage. This storage can also trigger another function to process an image or extract text from an image.

Building intelligent bots for automated response to a request received by a customer using serverless technology is again a great example where the focus needs to be on logic instead of infrastructure.

There can be an undefined number of other use cases that can fit with the serverless.

Benefits of serverless

The following are the few benefits of serverless:

No need to manage the infrastructure and related challenges

Pay for what you use

Seamless integration with other cloud services

Easy to create, deploy, and execute with support of popular frameworks and programming languages

Faster time to market

No manual scaling or need to defining scale rules

Supports continuous deployment through popular DevOps platforms

Conclusion

Azure is the cloud computing platform from Microsoft that overcomes the challenges with classic data centers and provides a new innovative way of managing and accessing the IT infrastructure required for an organization. With cloud evolving from on-premises to IaaS, PaaS, and now serverless has brought in a positive shift for application management. Functions as a Service provides the ability to execute code without having to worry about the underlying infrastructure.

In the next chapter, we will discuss more Azure Serverless computing with Azure Functions overview.

Points to remember

The cloud provides services in different categories – IaaS, PaaS, SaaS, and serverless and not just the virtual machines

Azure offers various services for all kinds of organizational IT needs related to compute, storage, databases, network, cognitive services, and many more

Azure services are offered in 60+ regions globally, with new regions getting added every year

Each Azure region consists of one or more data centers, also referred to as availability zones

A resource group is a collection of resources that share the same lifecycle, permissions, and policies

Azure Provides a pay-as-you-go and reserved instances pricing model with different pricing units for each service

You can track and analyse the organizational costs on various Azure services with the help of Azure cost analysis

Serverless is an event-driven, stateless code execution environment in which you write code and consume services

Serverless does not mean the absence of servers, but here, the consumer doesn't have to manage servers

Serverless supports continuous deployment through popular DevOps platforms

CHAPTER 2

Introduction to Azure Functions

Nothing has been more progressive to the software industry than the rise of serverless computing. We understood foundational knowledge of Azure services and serverless computing, which allows you to build and run applications without having to manage infrastructure. In this chapter, you will understand Azure Functions, and why it's important, and what problems it can solve.

Structure

In this chapter, we will cover the following topics:

Getting started with functions

Azure Functions hosting plans

Difference between Azure Functions and Azure WebJobs

Introducing triggers and binding

Working with Azure functions app

Azure Functions behind the scenes

Objective

After studying this chapter, you should understand the basics of Azure Functions, supported languages, and runtime. You will learn the various available hosting plans, the pricing models for the functions, and the use of triggers and bindings for Azure Functions. Later in the chapter, you will understand the structure of a function app and various available development tools in the Azure portal. You will be able to differentiate between the functions and Azure WebJobs in app services. Finally, at the end of the chapter, you will learn the behind-the-scenes working of Azure Functions and optimize execution based on your application requirements.

Getting started with functions

Azure Functions is a shiny tool available in the serverless computing offering in the market that you can leverage to architect, build, and deploy modern applications. This offering is also known as a **Function as a Service** While building an application with Azure Functions, you need to keep the business requirements and the end customer in mind. Architecting Azure building functions is a technical choice and an ability to adapt technology to meet business needs and provide agility when needed.

To get started, you need to create a function. First, that will contain the source code of your functionality written in your favourite language like .Net, Java, Python, and so on, that is, serverless functions. You can trigger these functions using HTTP. While developing and deploying the functions apps, you need to test this implemented function with test data.

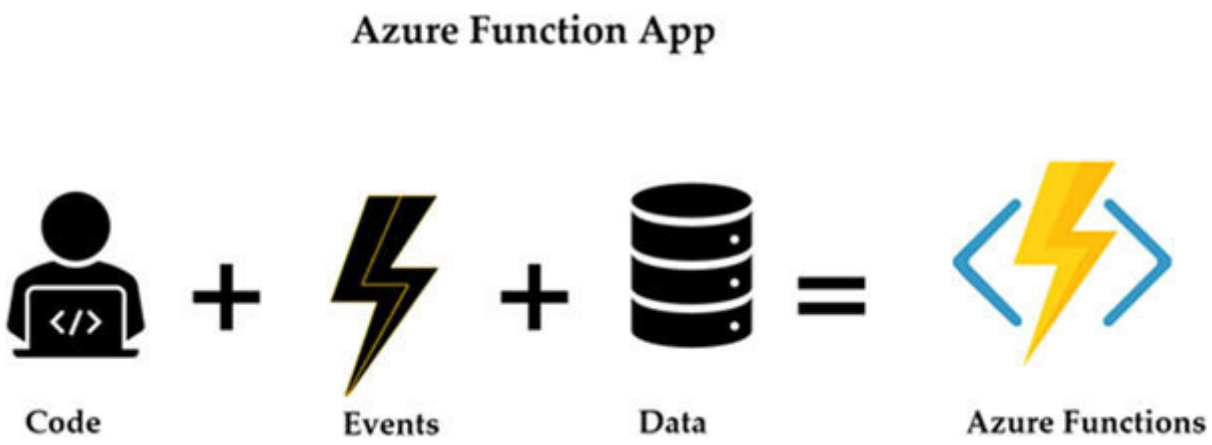


Figure 2.1: *Azure functions app*

To build a functions app, you need to adapt the phases like planning, building, testing, deploying, and operations. Let's understand these phases one by one.

Planning

You need to identify the functionalities needed to build requested functionalities, events, and triggers to execute the functions app. With the functions app, we can build a distributed system, where the system is partitioned into many different functions app. While the application is running, those functions are communicating with each other through events, triggers. It's called **event-driven**

Building

You need to write functions code with best coding practices to build and deliver mentioned functionalities in the planning phase.

Testing

After building the functions app, you must consider the testing cycle to ensure performance, security, scalability, and other aspects of the application.

Deploying

When an application is built and tested, it is ready to get deployed into the production environment.

Monitoring and operating

After deploying the application into a production environment, you continuously need to improve on operations. You can check the function execution logs, which will help in monitoring the functions app. Functions may run without an error. However, there may be concerning things in the log which you need to address to keep Functions up and running all the time. You can introduce automation in getting and reading the Azure functions logs. Please refer to *figure 2.2* to understand the functions app phases:

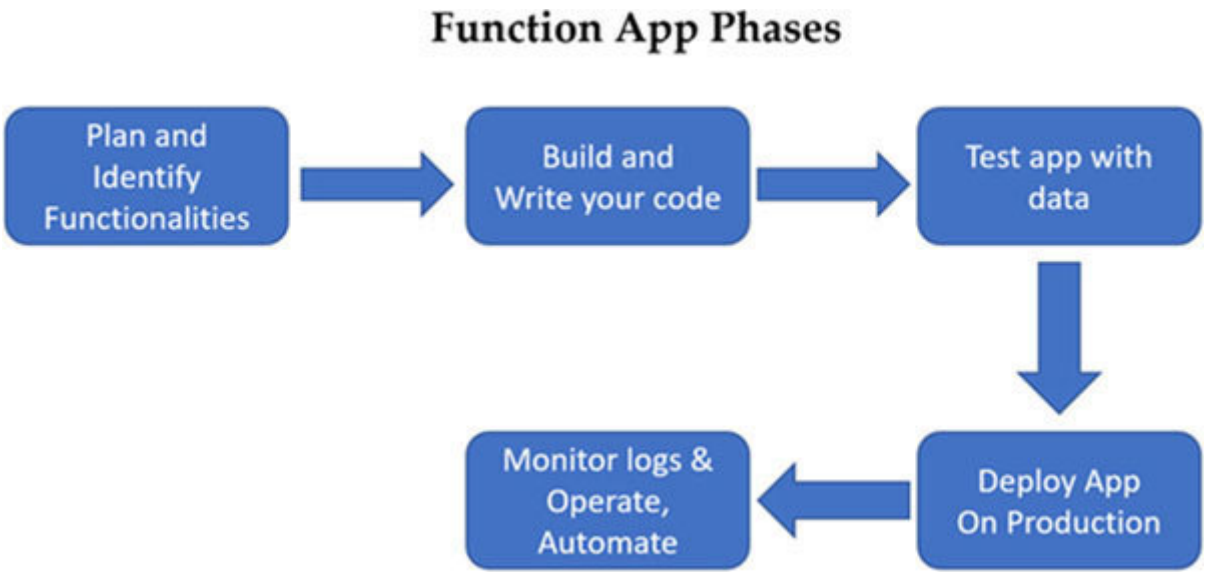


Figure 2.2: *Functions app phases*

Even though Azure Functions manage infrastructure resources for you, you must be mindful about using the resources. You don't have to deal with the notion of resources but must understand how we're interacting with resources like databases, storage

systems, and a message queue. Hence, you must clean up used resources when not needed.

These phases help you innovate production-ready applications as quickly as possible and look like a DevOps chain system separated into application development and operations. functions app is about continuous planning, continuous building, continuous testing, continuous deploying, and continuously operating. So, ultimately, these phase makes it a continuously improving environment; you can always improve with the capabilities to automate, deploy, test, and integrate.

You need to understand the ins and outs of the entire distributed system you are building with Azure Functions here regarding how all these things work and assemble to meet the requirements.

Benefits of Azure Functions

Let's continue to understand how Azure Functions can benefit you while designing, architecting, and implementing serverless applications to solve business problems:

The rapid development of distributed systems

Cost-effective' pay-as-you-go' pricing model

Write code in your favourite programming language within the Azure portal

Set up continuous integration and deploy

Ability to manage authentication, certificates, custom domains, and settings

Fewer operation tasks

Independent auto-scaling

Azure Functions can be considered an option to integrate systems, integrate to other Azure services, bulk data processing, expose API endpoints, and implement micro-services. Many templates are available and supported in Azure Functions to deal with scenarios

like HTTP, timers, Azure cosmos DB, blob storage, queues, event grid, event hub, service bus queue, and server bus topic.

Azure Functions hosting plans

While creating the functions app, you must select a hosting plan that decides app scalability, available features, and pricing. There are three types of hosting plans available in Azure Functions. These hosting plans are consumption plan, premium plan, app service plan, which are available for Windows and Linux operating systems. Let's understand the differences between hosting plans.

Consumption plan

In this plan, you need to pay for what you use (pay-as-you-go). It is based on the number of executions, execution time (CPU time), memory usage (RAM) of all functions available in a functions app. This hosting plan is the default one that can scale automatically in case of high requests per second. You can run multiple functions apps in the same consumption plan. You don't have to take care of Azure resource management.

Premium plan

The premium plan is the same hosting plan as the consumption plan but with more features like improved performance, virtual network connectivity, premium instances with high computational power, and predictable pricing. If your functions app is idle for several minutes, the functions apps may take some to scale up to serve a request. You can have pre-warmed functions app instances; hence you can have continuously running Functions apps. This plan is the right fit when your code executes longer than the time allowed in the consumption plan.

App service plan

This plan helps you run your functions as a web, API, or mobile app; if you have an underutilized virtual machine instance, you can select an existing app service plan and pay for a dedicated server. You get several pricing tiers when choosing this hosting plan for your Functions app and predicting monthly costs. In this plan, there is no function duration constraint.

This app service plan is a good fit for long-running applications or jobs, which need scaling.

Please refer to *Table 2.1* to get a summary of available hosting models:

models: models: models:
models: models:
models: models: models:

Table 2.1: *Summary of available hosting models*

Difference between Azure Functions and Azure WebJobs

WebJobs is an Azure service used for running long-running or short-running background tasks, and it is a cheaper option than Azure IaaS offerings. You can share existing app service plans or have separate app service plans for WebJobs. WebJobs can be executed manually or based on a schedule.

Nowadays, Azure Functions is the preferred and coolest choice to run background tasks. Azure Functions are built on top of Azure WebJobs with some extra functionalities. Azure Functions are WebJobs that can be executed manually, based on schedule, or triggered by Azure or external services.

Let's understand the difference in *Table*

Table 2.2: *Difference between Azure WebJobs and Functions*

Introducing triggers and binding

Let's understand the important building block of serverless computing using Azure Functions. A high-level component of Azure Functions is triggers and bindings. Triggers are responsible for running functions, and binding helps connect with other services. You will come across these building blocks in further chapters.

Triggers

Azure Functions can be triggered by any event in Azure services, third-party services, or on-premises systems. Triggers are events that invoke Azure Functions with data as a payload, that is, input parameter to the Functions. There are multiple types of ways to invoke the Functions of the app.

Azure Functions triggers types are as follows:

Timer trigger is invoked based on a pre-defined schedule

Blob trigger gets invoked based on updates in a blob

Event Hub triggers are related to events in the Event Hub

HTTP trigger is related to an HTTP request

Queue trigger listens for messages to get added to a queue

A generic trigger is related to the HTTP webhook

Service Bus trigger is executed when a new message is received in a service bus queue or topic

GitHub trigger is related to GitHub events on a branch, builds, and so on

Bindings

Bindings are closely related to input data, and output data passed to the Functions, which establishes connectivity with external service. These bindings can be input binding, output bindings, or both as per your need.

Please find a sample JavaScript declaration to read data from an HTTP request.

```
{
"datatype": "string",
"type": "httpTrigger",
"name": "req",
"direction": "in"
}
```

Direction parameters tell you how external resources are connected and can be defined as in, out, or out. Please refer to *figure*

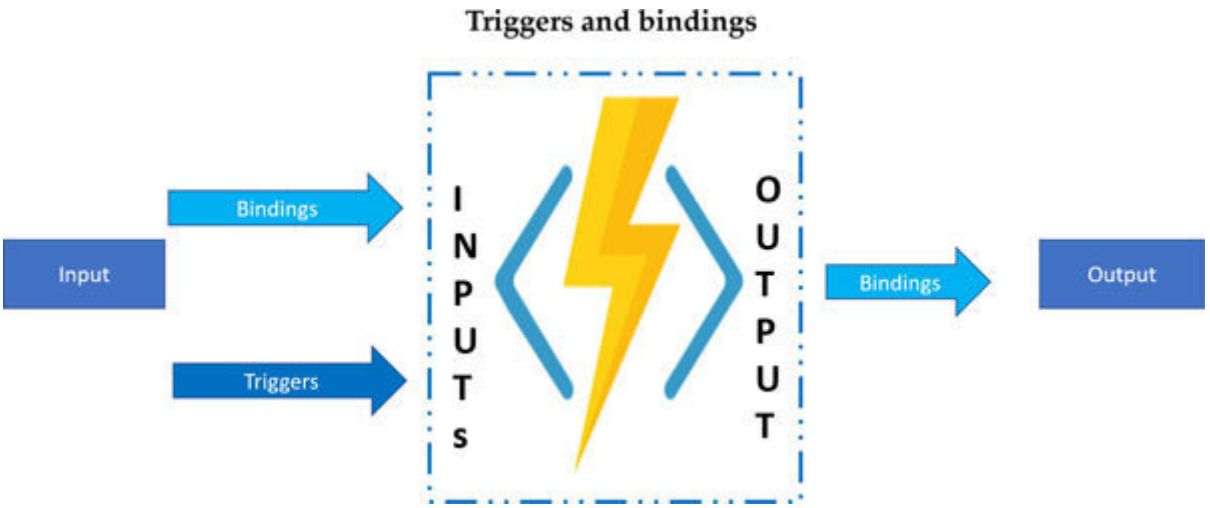


Figure 2.3: *Triggers and bindings*

Azure functions can have multiple inputs and output bindings but only one trigger binding and can be used in combination as per your need.

Please refer to the following sample C# code snippet with HTTP trigger:

```
using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Azure.WebJobs;

using Microsoft.Azure.WebJobs.Extensions.Http;

using Microsoft.AspNetCore.Http;
```

using Microsoft.Extensions.Logging;

{

{

Task

[HttpTrigger(AuthorizationLevel.Function, Route =

HttpRequest req,

Ilogger log)

{

string name =

trigger function processed." + name);

}

```
    }

}
```

Some of the trigger types like Blob storage, Queue storage, Service Bus (queue/topic) have inbuilt retry support on failure, and for other types of triggers or bindings types, you need to implement manual retry logic.

One sample use case can be an Azure function that gets triggered when a new message is added to the Service Bus queue with the email details of the customer. There can be an input binding for Azure Blob storage that has a document to be sent. The function would process that message, get the customer's email from the message, and send the document over the email to the customer through SendGrid. Figure 2.4 shows the triggers and bindings for the above use case:

Figure 2.4: *Triggers and bindings for the use case*

Working with Azure functions app

An Azure functions app is a collection of one or more Azure Functions that are managed together. All the functions in a Functions app share the same behaviors and pricing plan and are written in the same language.

The function in a functions app is managed, scaled, and deployed together, whether through Visual studio or using the build and release pipelines in Azure DevOps.

The application settings and connection strings are also functions app-specific. Any data or configurations that need to be shared between two or more functions apps can be stored in a persisted store externally, such as Azure storage, key vault, and so on.

Functions code

Functions code and the config, the **function.json** file, are two important parts of Azure Functions. The code can be written in various languages that Azure Functions support and discussed later. The config file is generally generated automatically for the complied languages.

The **function.json** file contains the trigger and bindings as well as other configuration settings used by the functions. We will discuss the triggers and bindings in the later section and detail them in a separate chapter. The config file is an important file used by the function's runtime to determine the input data and output data flows.

A typical **function.json** file looks like the following *figure*

TestFunctionAppCA \ HttpTrigger1 \ function.json

```
1  {
2    "bindings": [
3      {
4        "authLevel": "function",
5        "name": "req",
6        "type": "httpTrigger",
7        "direction": "in",
8        "methods": [
9          "get",
10         "post"
11       ]
12     },
13     {
14       "name": "$return",
15       "type": "http",
16       "direction": "out"
17     }
18   ]
19 }
```

Figure 2.5: *An example function.json file*

Functions settings

Application settings and connection strings can be defined in **Settings ➔ Configuration** in a functions app, under the corresponding section, as shown in *figure*

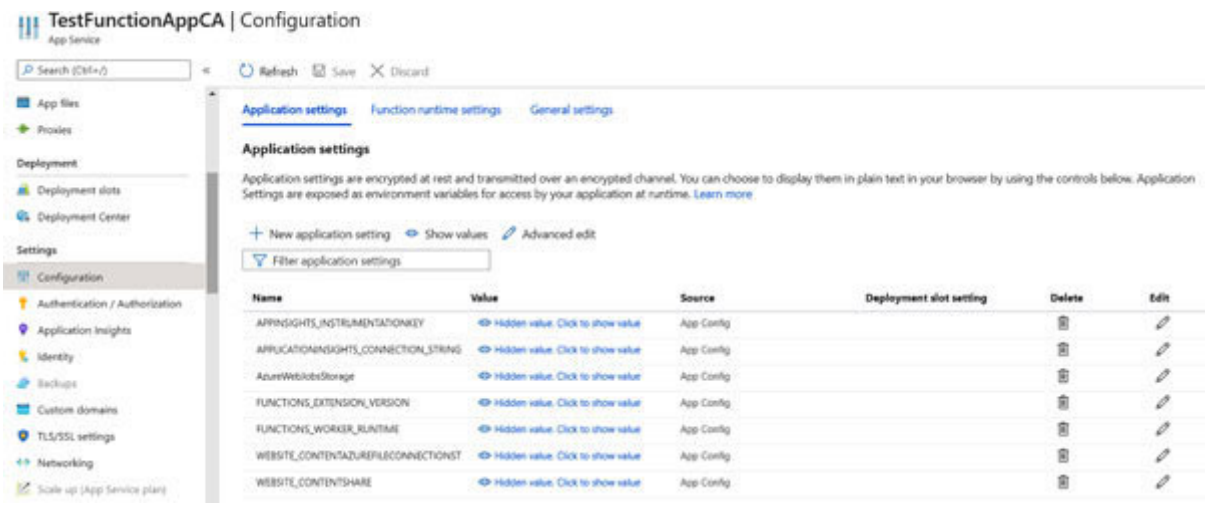


Figure 2.6: *Functions app configurations*

These settings can be read and used in the function's code as environment variables.

Folder structure

The code for all the functions is in the wwwroot project folder inside the functions app. It contains a **host.json** file and one subfolder each for each function. The folder structure is shown in *figure*

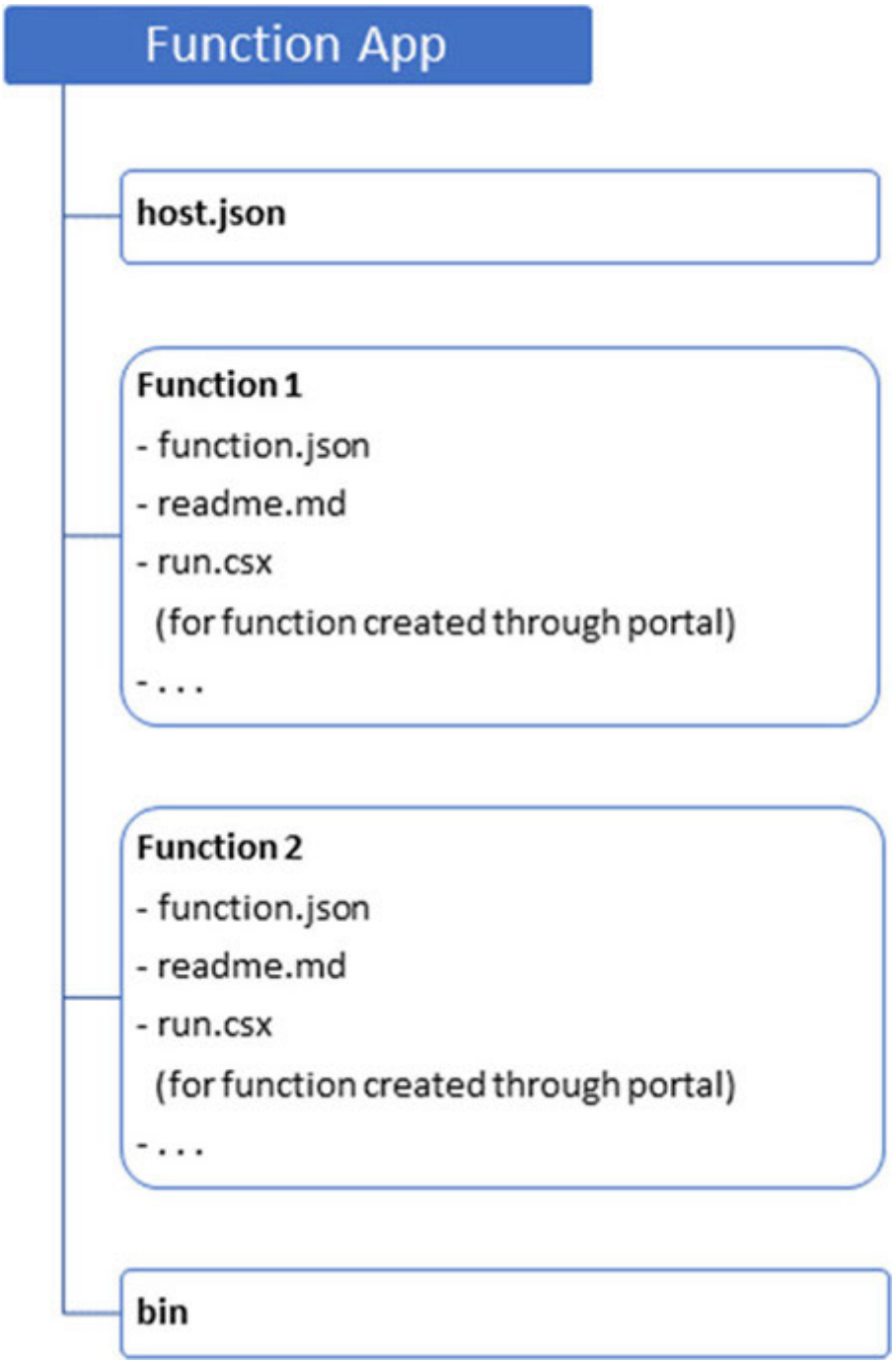


Figure 2.7: *Functions app folder structure*

The **host.json** file shown in *figure 2.7* in the root folder contains runtime-specific configurations. The bin folder contains packages and other library files the functions app requires.

Note: The folder structure shown in *figure 2.7* is a default and recommended structure for a functions app. However, it is possible to change the file location of a function's code by modifying the scriptFile section in the function.json file.

Runtime versions

There are three major versions available while writing this book for a functions app: 1.x, 2.x, and 3.x. All the functions in a functions app need to run on one of these versions. By default, functions apps are configured to run in version 2.x.

Azure Functions manage the runtime version with the help of the FUNCTIONS_EXTENSION_VERSION application setting in a functions app.

The runtime versions can be changed for a functions app in **Settings → Configuration → Functions runtime settings** tab as shown in *figure*

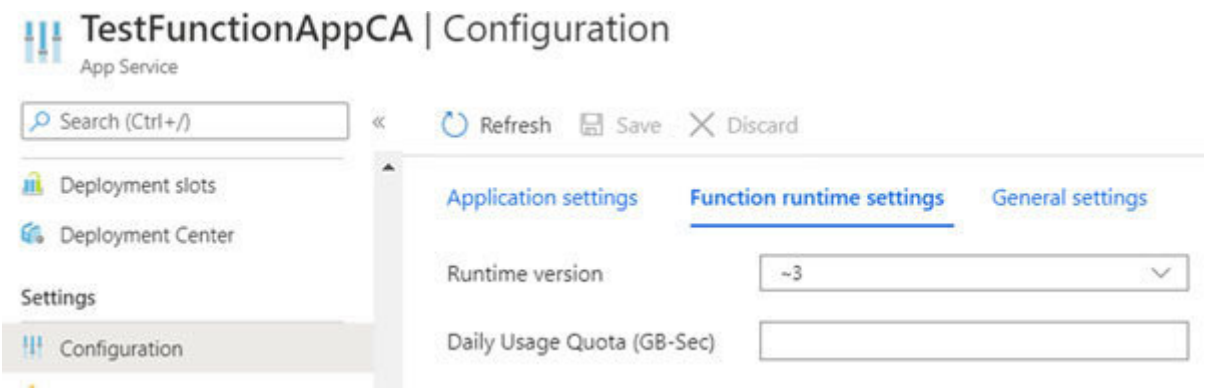


Figure 2.8: *Functions app runtime settings*

The functions app needs to be restarted from the **Overview** tab for the runtime to take effect. The restart option is shown in

figure

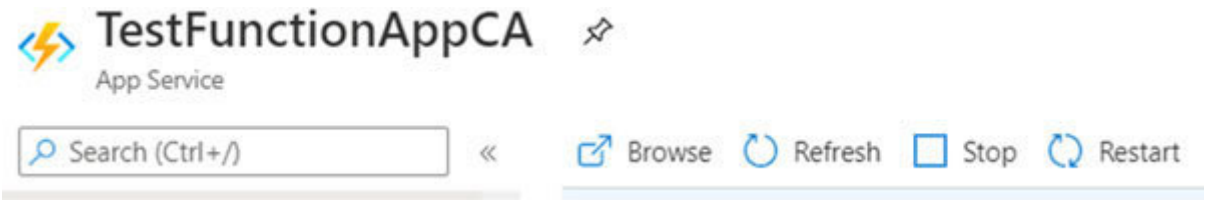


Figure 2.9: *Functions app restart option*

Supported languages

Azure Functions allows you to write the function's code in the language of your choice among the supported languages.

The *table 2.3* defines the languages supported by each of the Azure Functions runtime versions when writing this book. There may come more language support in the future.

future.
future. future. future.
future. future.
future. future. future. future.
future. future. future. future. future.
future. future. future.
future. future. future.
future.

Table 2.3: *Supported languages*

Development tools

Azure provides advanced-level development tools on the portal in the functions app. These tools are discussed in the next section.

App Service Editor

The App Service Editor is an inbuilt editor within the Azure portal that can modify code and JSON configuration files in a functions app. App Service Editor provides integration with the Git repository and allows to run code and view output online. This editor provides an enhanced development environment for the functions app.

Note: The code can only be modified for functions developed using the portal. The functions developed locally and published using the Visual Studio, or other IDE cannot be modified using an App Service Editor.

The App Service Editor can be accessed from **Development Tools ➔ App Service Editor** from the left pane inside your functions app, as shown in *figure*

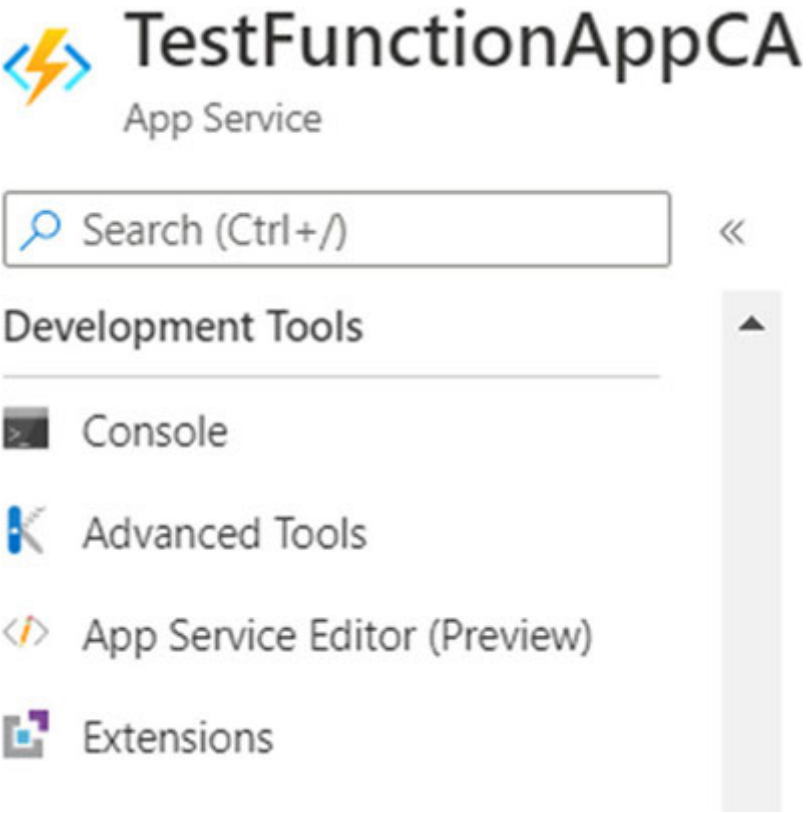


Figure 2.10: *The functions app development tools section*

This opens the App Service Editor in a separate tab in the browser, like *figure*

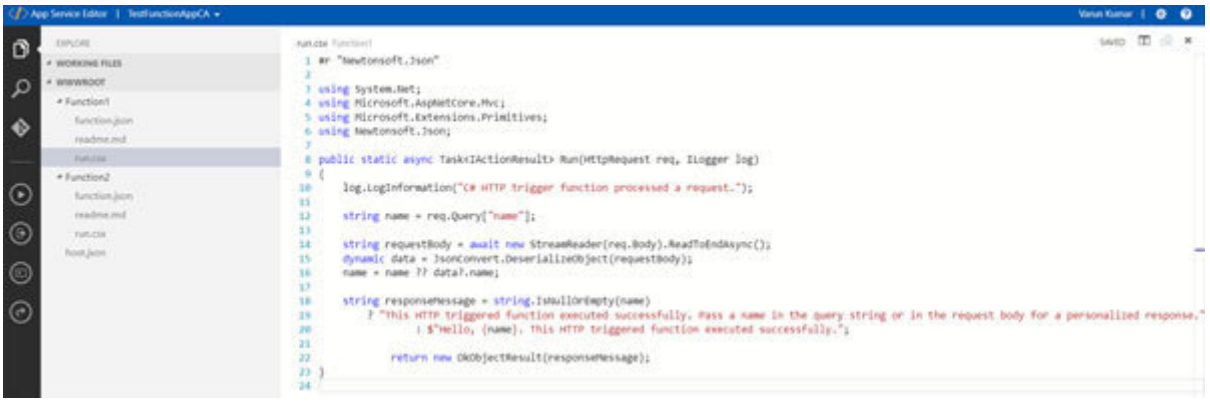


Figure 2.11: *The App Service Editor tab*

The Git integration runs code and shows output options that can be accessed from the left vertical menu, as shown in the preceding figure.

Advanced tools (Kudu)

The advanced tools, also known as provide access to advanced administrative features of the Azure functions app such as:

System information

App settings

Connection strings

Environment variables

Runtime versions

Server variables

HTTP headers

Site extensions

Kudu can be accessed using the SCM endpoint for the functions app - **https://.scm.azurewebsites.net/** or accessing **Development Tools → App Service Editor** from the left pane inside your functions app.

Figure 2.12 shows the Kudu landing page:

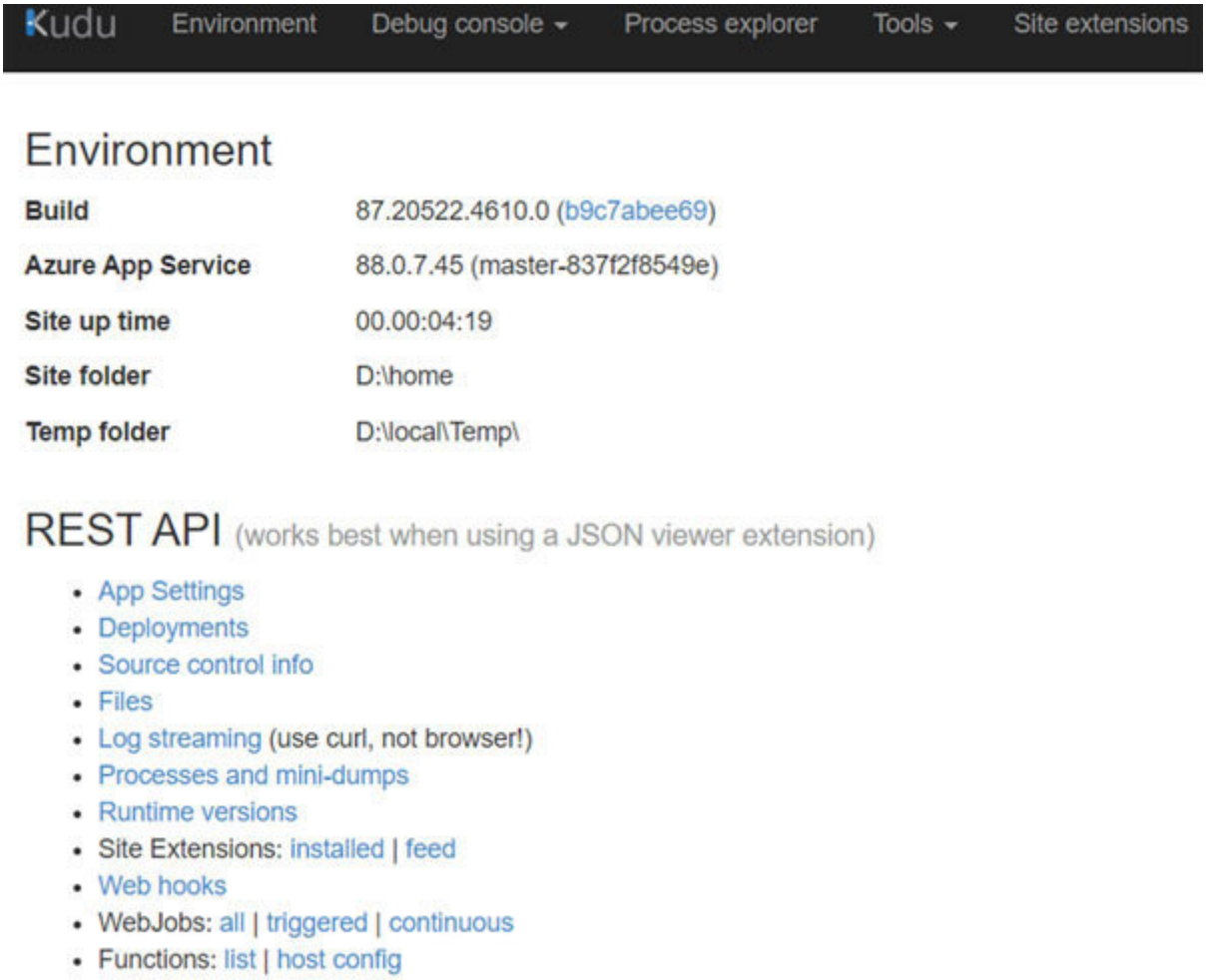


Figure 2.12: *Kudu landing page*

It can be used for troubleshooting and analysis as well because it provides access to required logs and helps monitor the processes that are running in the background, as shown in *figure*

Process Explorer

Find Handle...

Refresh

name	pid	user_name	total_cpu_time	working_set	private_memory	thread_count	properties	profiling	
w3wp.exe	5328	TestFunctionAppCA	9 s	71,308 KB	78,792 KB	42	Properties...	Collect IIS Events	Start Profiling
cmd.exe	6948	TestFunctionAppCA	<1 s	7,244 KB	3,368 KB	1	Properties...	Collect IIS Events	Start Profiling
node.exe	8060	TestFunctionAppCA	4 s	29,056 KB	53,180 KB	9	Properties...	Collect IIS Events	Start Profiling

Figure 2.13: *Kudu process explorer*

It is also one of the fastest and easiest ways of deployment for development and testing; however, not a recommended way. It also helps in the health check of functions by providing memory dumps, and so on.

Figure 2.14 shows the Kusto **Debug Console** ➔ cmd from the top menu that helps access the files in the functions app as well as the Kudu Remote Execution console:

Kudu Environment Debug console Process explorer Tools Site extensions

/ 3 items

	Name	Modified	Size
	data	7/21/2020, 2:44:14 PM	
	LogFiles	7/21/2020, 2:32:47 PM	
	site	7/21/2020, 2:32:47 PM	

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new CMD process.
Type 'cls' to clear the console

Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

D:\home>
```

Figure 2.14: *The Kudu debug console*

Console

The console in the functions app is a great tool to interact with the functions app with the help of the command line. You can create and access directories and files and executing scripts and batch files through the console.

Figure 2.15 shows the default console in the Azure portal:

Figure 2.15: *The Kudu debug console*

Azure functions behind the scenes

By now, you might be wondering where the functions run behind the scenes and how Azure takes care of the availability of the compute resources to execute functions. Azure Functions run on the virtual machines with the specialized environment, that is, the function runtime. We don't have to worry about how these VMs are allocated and de-allocated from an operational perspective. However, it is beneficial to understand the process to reduce latency and save costs.

Functions cold start

Cold start is the process of provisioning resources and initiating the function to start responding to events. The function running for the first time or not executed recently takes longer to execute.

When you create a new function app and write your function, it gets stored in Azure storage since nothing is executed yet. When the function is triggered for the first time, Azure performs the following actions to make your function ready:

Allocation of a virtual machine to execute a function

Getting virtual machine ready with functions runtime

Setting up your function code to execute

Cold start process

Azure takes a sophisticated approach for the cold start process as defined here:

Allocates a preconfigured server from the pool of warm servers to the functions app with already installed functions runtime. However, it is an unspecialized server. Specialization is a process of mounting function app files stored in the storage to the assigned server and applying the app settings.

The server is specialized by configuring the function's runtime to be specific to the function's app.

The functions runtime is reset, and the required extensions are loaded as defined in the **function.json** file of the functions app.

The functions are loaded into memory and initialized by the language providers.

Finally, the function code gets executed and starts handling requests.

The following *figure 2.16* shows the functions cold start process:

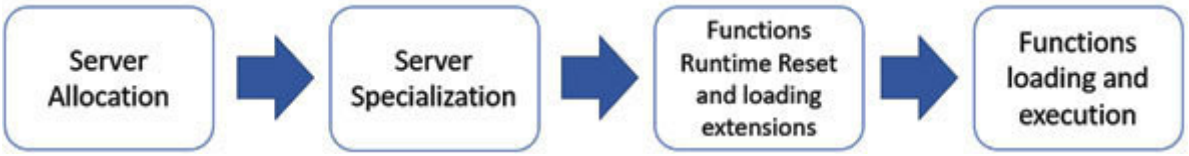


Figure 2.16: *The functions cold start process*

Pre-warmed instances

The Azure function allows configuring a pool of pre-warm instances that are preconfigured with the function's runtime. Azure allocates instances from that pool when more instances are required. This reduces the start time to three to four times. Azure keeps buffering pre-warmed instances till the maximum scale-out limit.

Let's understand this with an example of a premium function app with two always running and one pre-warmed instance configured. When there is no request on the functions, you will be billed for the two always running instances but not for the pre-warmed instance as it's not yet allocated.

Now, when the first requests come in, the pre-warmed instance is allocated. The functions app now will have three allocated instances and two always running instances, and a fourth pre-warmed instance in a buffer. As the load continues to grow, Azure will keep allocating buffered instances and buffering new instances in the pool till the maximum defined limit.

Preventing cold start

You can avoid the cold start time by following one or more of the below:

Choosing a dedicated mode allows you to have always running functions. But then you lose the benefits of serverless. However, for applications that require low latency for each call, this can be an option.

Azure Functions' premium plan allows you to keep some servers always running, but you have to pay 100% of the time for these servers. You can have a maximum of 20 such instances with a premium plan.

Keeping dependencies lightweight also helps in reducing the server's start time.

Creating an availability (ping) test for the HTTP trigger functions on a frequent interval not to be considered for de-allocation.

Conclusion

Azure Functions is the Function as a Service offering from Microsoft Azure that allows you to create highly scalable and available applications. You can write Azure Functions code in the language of your choice among multiple supported languages. Azure provides great flexibility and capability on the portal to develop, deploy, and manage the functions with the help of advanced-level development tools. Azure Functions can be triggered through multiple Azure services with the help of built-in triggers and binding options. To sum up, Azure Functions are the new building blocks to develop modern serverless designs.

In the next chapter, we will discuss how to create your first functions app on Azure through various available options.

Points to remember

An Azure functions app collects one or more Azure Functions that share the same behaviours and pricing plan.

Functions apps help you to focus on implementing the required functionalities and not on resource management.

The functions in a functions app are deployed and managed together.

Azure functions support multiple languages.

Azure functions app has three major runtime versions available. 2.x is the default version.

The functions app provides different development tools for more agility, better troubleshooting, and access to the app resources.

Azure Functions is a key component of serverless computing with benefits like 'pay-as-you-go,' fast go-to-market, and cost-effective service.

You can leverage the various development support tools provided by Azure on the portal to boost productivity.

There is a process of cold start involved for the functions running for the first time or after a long time.

The cold start time can be reduced for critical business cases by choosing the right hosting plan, defining always running and pre-warmed instances, and so on.

CHAPTER 3

Creating Your First Azure Function

As discussed in the previous chapter, an Azure Function consists of function code and function settings. In this chapter, we will understand how to create our first function using multiple available options provided by Azure like Azure Portal, Visual Studio, VS Code, **command-line interface** (**CLI**), and **Azure Resource Manager** (**ARM**) templates. We will also check the implementation and running of your Azure Functions locally and publishing it on Azure.

Structure

In this chapter, we will cover the following topics:

Creating a function in the Azure portal

Creating a function using Visual Studio code

Creating a function using Azure CLI

Creating a function using an ARM template

Objective

After studying this chapter, you should create a sample Azure function app and your first function using your favorite tool. You will also be able to run and debug the newly created Azure Functions.

Creating a function in the Azure portal

To create your function in the portal, you need to log into the Azure portal by accessing **http://portal.azure.com** through your browser and signing in. If you don't have an Azure subscription or are a first-time Azure user, you can create your free Azure account, as discussed in *Chapter 1: Overview of Azure and Serverless*

Once you have successfully signed in to the Azure portal, you will be taken to the Azure home page. Before creating a function, you need to create a function app to host your functions.

Creating your first function app

Follow these steps to create your first Azure function app on the portal:

On the home page, click on **Create a resource,** as shown in *figure*

Figure 3.1: *Create a resource on the Azure home page*

Alternatively, you can also access this from the top-left expandable menu (Azure portal menu), as shown in *figure* Use this option if you are not able to see this on the home page.

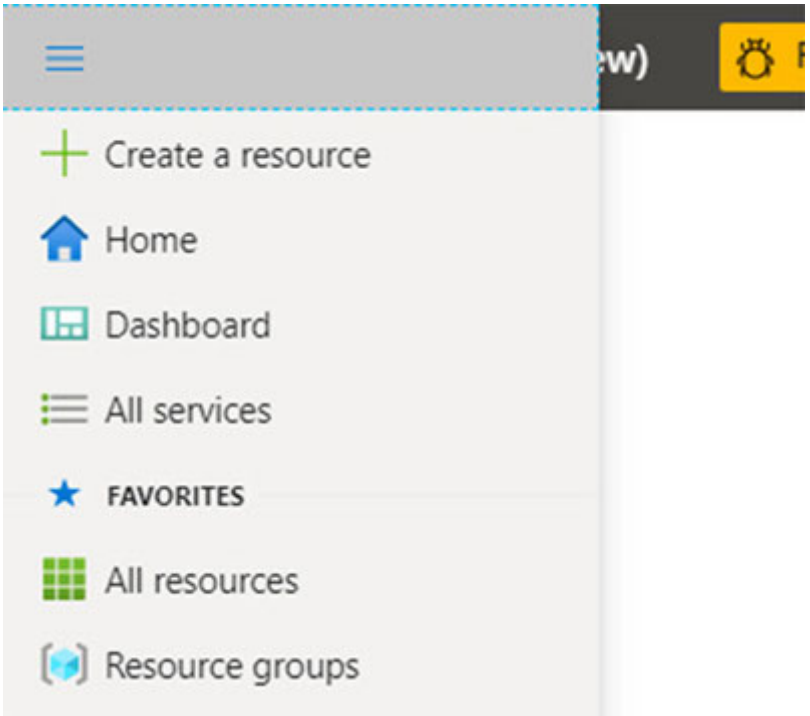


Figure 3.2: *Create a resource on the Azure top left menu*

In this menu, click on **Compute ➔ Function** as shown in *figure*

Figure 3.3: *Create a new resource menu*

Once you have clicked on **Function** you will be taken to the **Create Function App ➔** Basics page.

On the Basics page, you need to fill in the project details such as the subscription, resource group, and instance details such as function app name, runtime, region, and so on. Please refer to the following list to understand these settings:

The subscription in which you want to create your new function app. You can choose a subscription to which you want the execution of the function app to be billed to.

Resource The resource group in which you want to create your function app. You can choose an existing resource group or create a new resource group by clicking If you remember, a resource group is a collection of resources that share the same lifecycle, permissions, and policies. Steps to create a new resource group were discussed in *Chapter 1: Overview of Azure and Serverless*

Function app This will be the name of your function app. The name can consist of characters (a-z), numbers (0-9), and '-.' The name is not case sensitive. Also, the function app name constitutes the URL of your function app, For example, the function app with the name testfunctionapp has the URL as

You can choose your function app to be published as code or a docker container.

Runtime This allows you to choose the runtime supported by the programming language of your choice. You can choose from .NET Core, Node.js, Python, Java, or PowerShell core. In this book, we will select .Net Core for C# functions. But you can choose your favourite programming language.

This drop-down option auto-populates based on the runtime stack selected and allows you to select the specific version of your

runtime environment. For example, we have *version 3.1* available for .Net core, version 3.6, 3.7, and 3.8 for Python, and so on.

This defines the geographical location where your function app will be hosted. You must select the nearest location to your users or other services that would access your Azure Functions.

Please refer to *figure 3.4* for a sample Azure Functions with the mentioned settings.

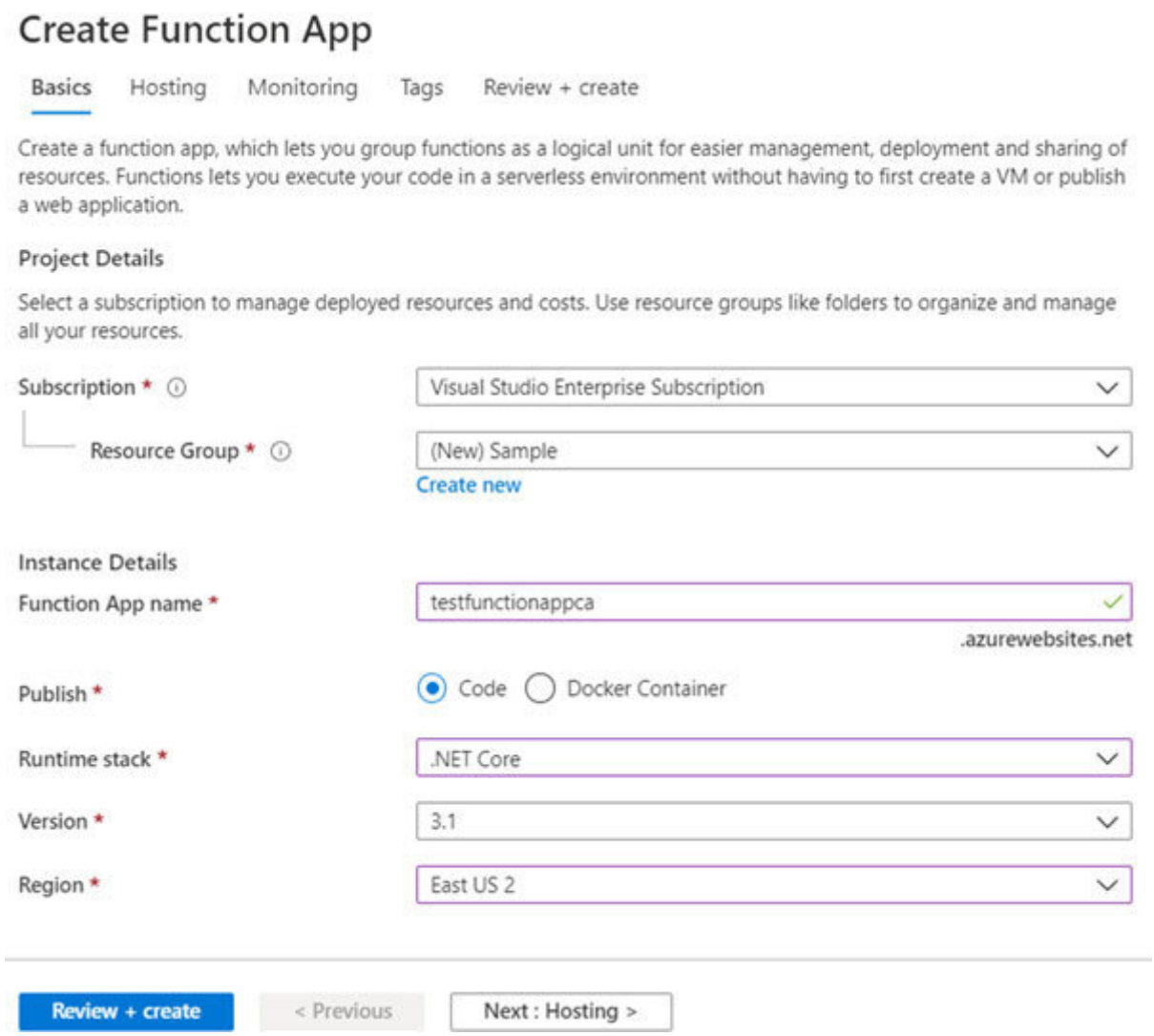


Figure 3.4: *Create function app basics page*

Click on **:** after filling in all the required information, as shown in *figure*

You need to select the storage account, operating system, and hosting plan on the Hosting page.

Please refer to the following list to understand these settings:

Storage The Azure function app uses the storage account. You can select an already existing storage or choose to create a new storage account by clicking

Operating The operating system is pre-selected based on the selected runtime stack on the Basics page. But you can change if required for Linux and Windows.

Plan The plan type defines the hosting plan for your function app. You may choose from the consumption, premium, or app service plan. The details on hosting plans were discussed in the previous chapter.

Please refer to *figure 3.5* for the sample Azure Functions with the mentioned settings on the hosting page.

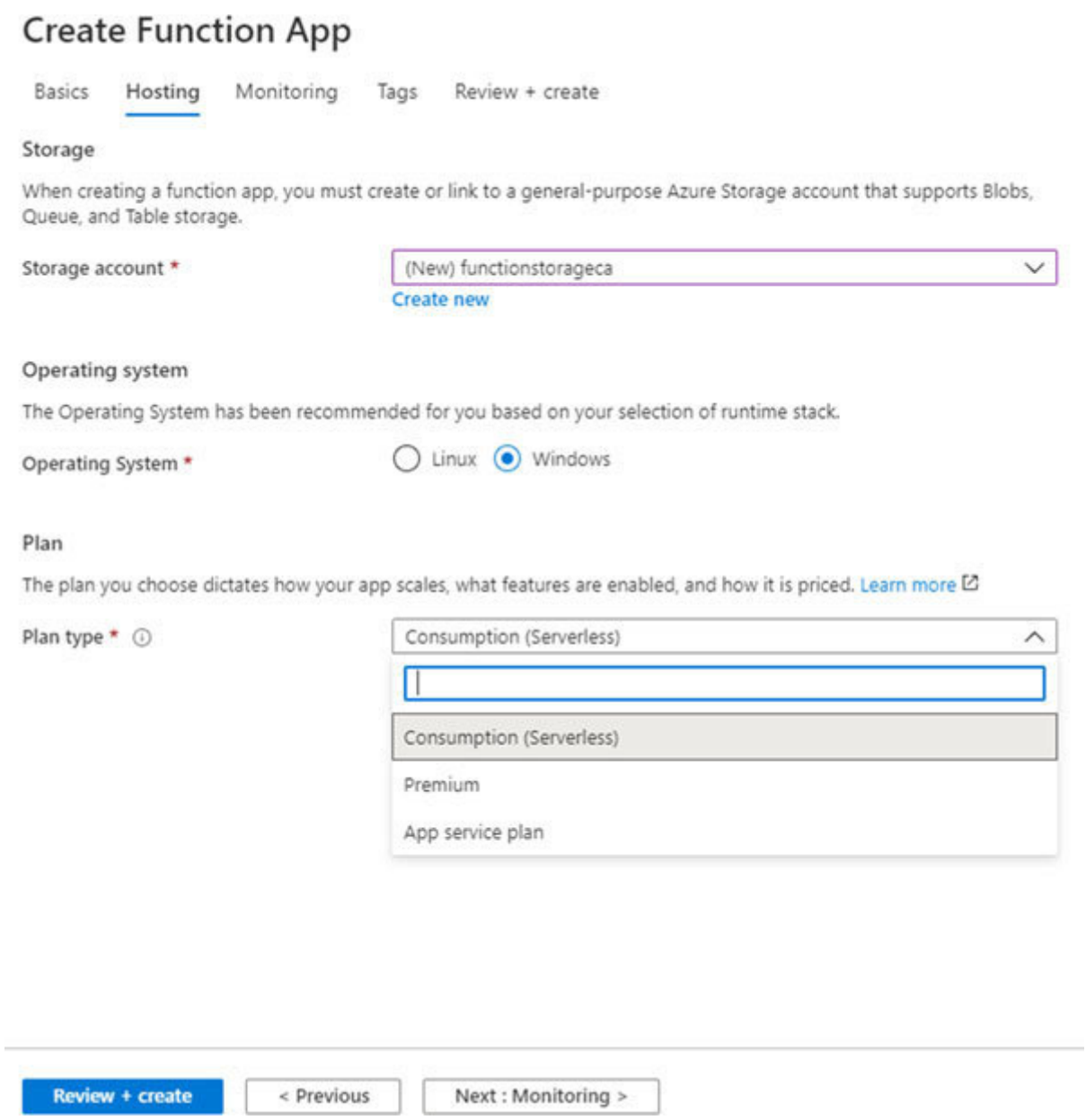


Figure 3.5: *Create function app hosting page*

Click on **:** after filling in all the required information, as shown in *figure*

On the **Monitoring** page, you can choose to enable and select **Application** You can also create a new Application Insights

resource by clicking If you choose to create new, the region selected is the same as the function app, as shown in *figure*

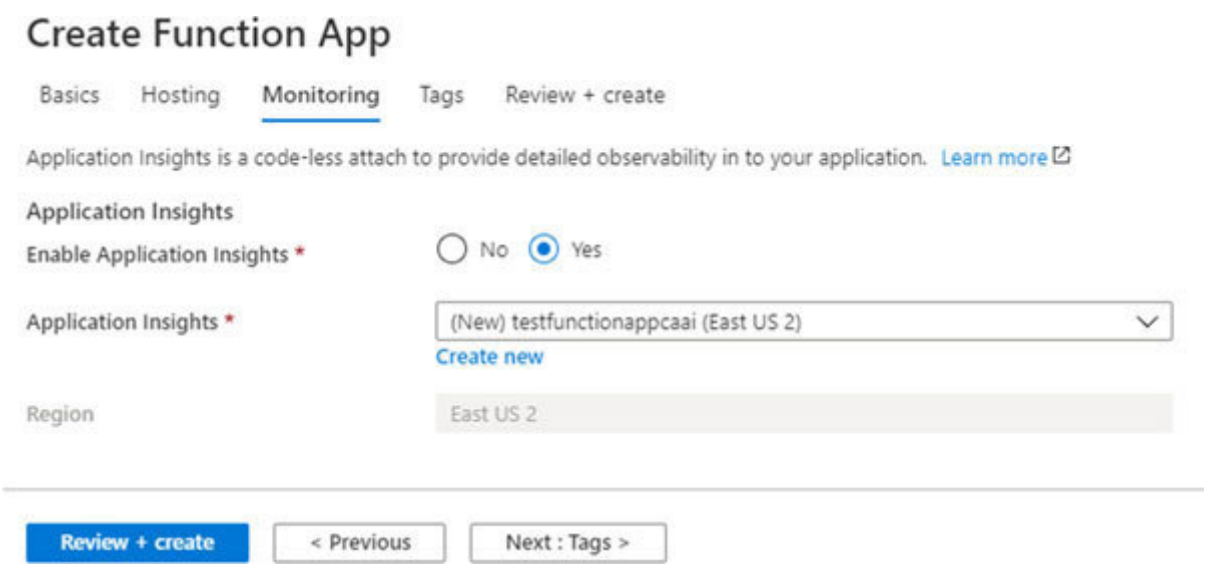


Figure 3.6: *Create function app Monitoring page*

Click on **:** to add tags and then **+** or directly **+** on the **Monitoring** page, as shown in *figure*

Review your settings on the **+** page and click on to provision and deploy your Azure function app. You can also download the ARM template for the function app by clicking **a template for** as shown in *figure*

Create Function App

Basics | Hosting | Monitoring | Tags | Review + create

Summary

Function App
by Microsoft

Details

Subscription	da400927-72d7-4ace-9c06-c5796bc847b0
Resource Group	Sample
Name	testfunctionappca
Runtime stack	.NET Core 3.1
Tags	Type: test

Hosting

Storage (New)

Storage account	functionstorageca
Tags	Type: test

Plan (New)

Plan type	Consumption (Serverless)
Name	ASP-Sample-a09d
Operating System	Windows
Region	East US 2
SKU	Dynamic
Tags	Type: test

Monitoring (New)

Application Insights	Enabled
Name	testfunctionappcaai
Region	East US 2
Tags	Type: test

Create | < Previous | Next > | Download a template for automation

Figure 3.7: *Create function app "Review + Create" page*

Click on notifications (bell icon) in the upper-right corner of the portal to check the deployment succeeded message, as shown in *figure*

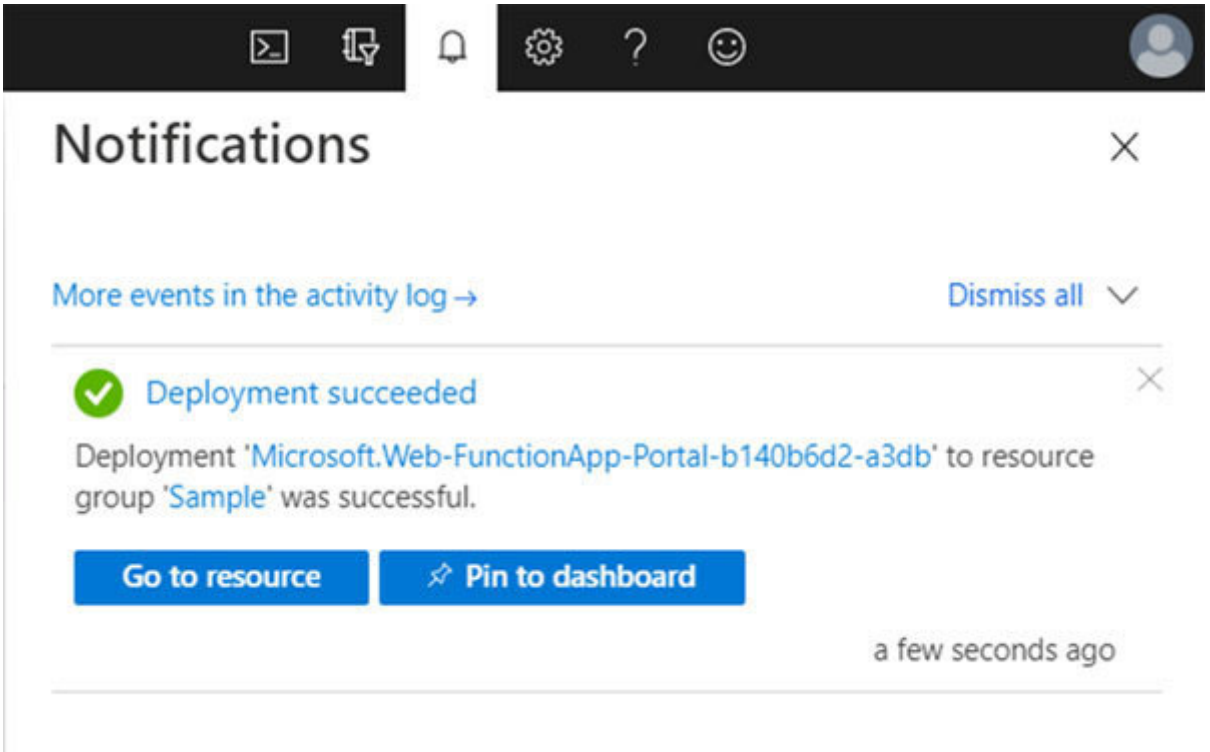


Figure 3.8: *Create function app notification*

Click on **to** to open your function app. You can also open your function app by going to Resource Groups in Home Page ➔ Resource Group Name (in this case Sample) ➔ Clicking on the function app name in the list of resources under your resource group.

Note: When a function app is created with the mentioned options, it creates four resources in the resource group – a function app

(app service), an app service plan, a storage account, and an applications insights resource.

Creating your first function in the function app

Once your function app is created, you can create your first function in the newly created function app by following these steps:

Open your function app and click on **Functions** under the Functions section in the left menu of your function app, as shown in *figure*

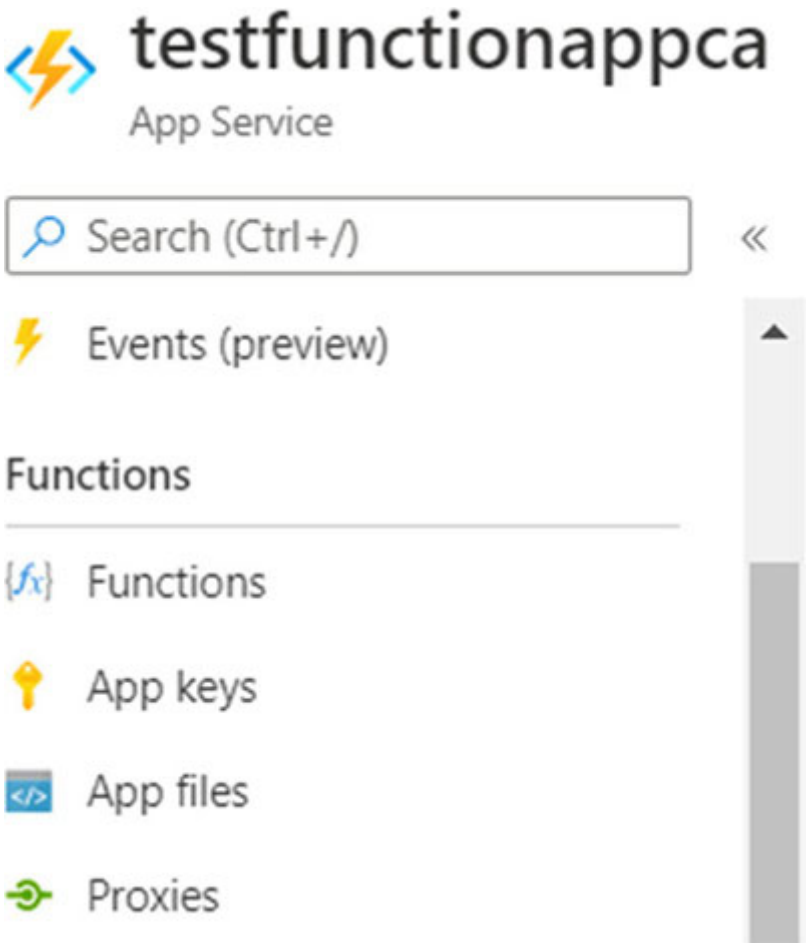


Figure 3.9: *Function app left menu*

This will open the functions that are hosted in the function app. We will not see any function yet here.

Click on at the top, as shown in *figure*

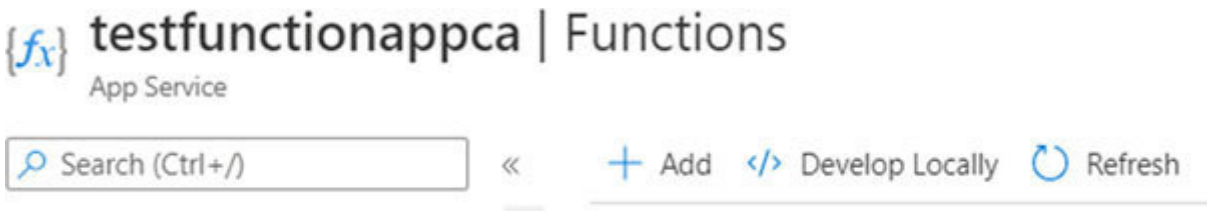


Figure 3.10: *Add new function*

This will open a right menu as shown in *figure 3.11* as follows:

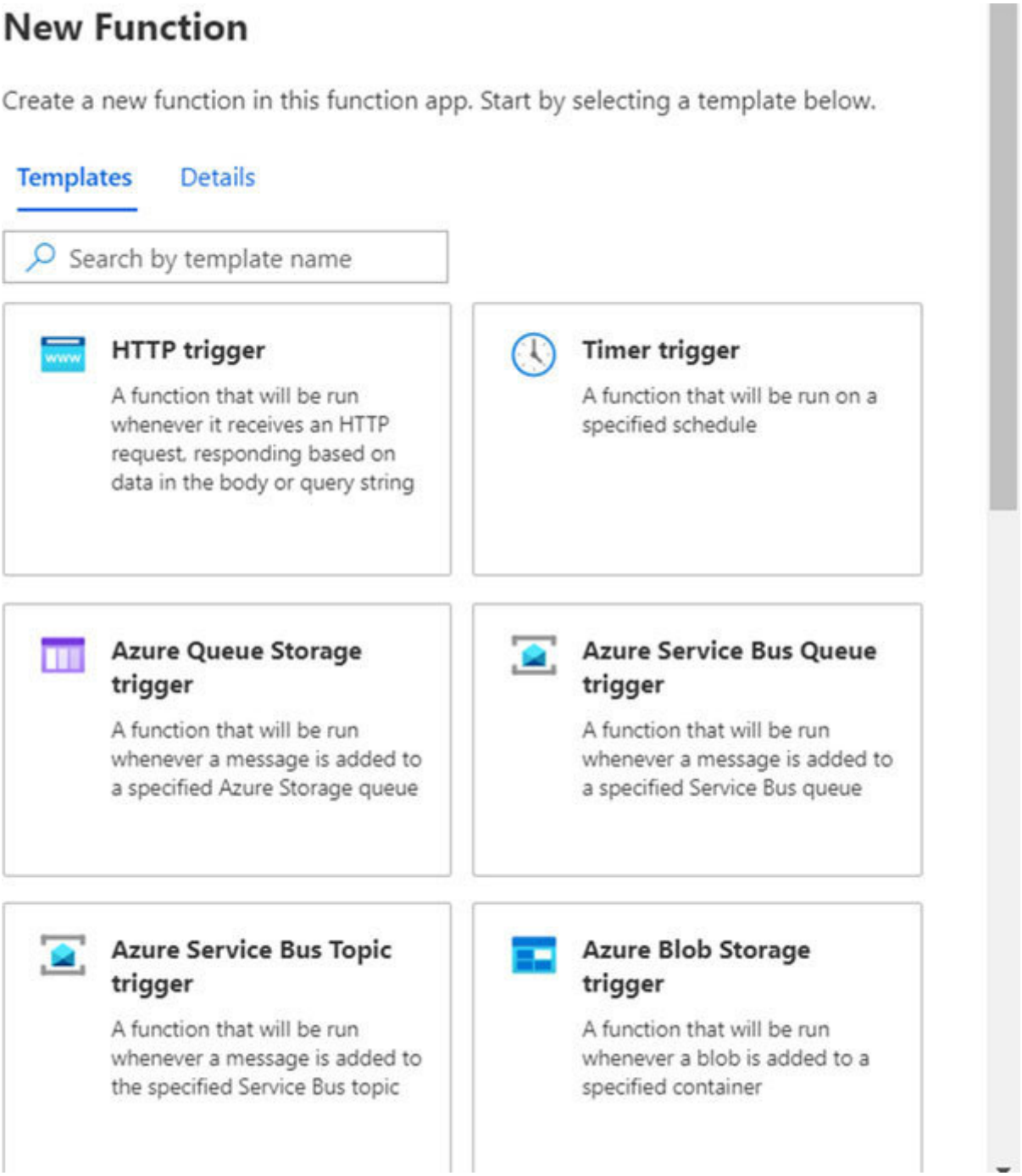


Figure 3.11: *New function menu*

You can create a different type of triggered function from this menu. For our first function, we will select the HTTP trigger function.

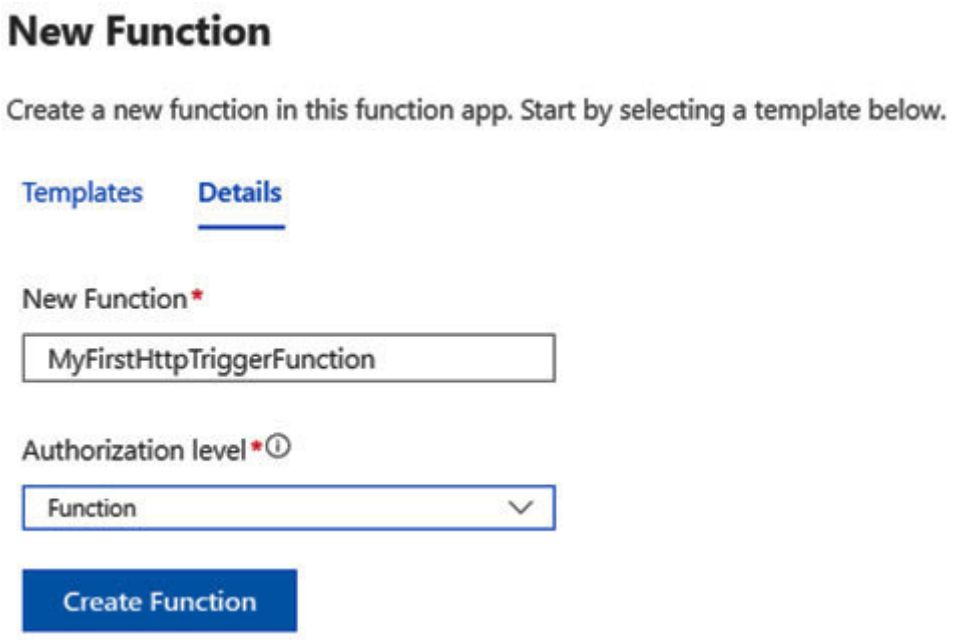


Figure 3.12: *New function menu*

Once we click on the HTTP Trigger template, it asks to provide the name of the function and the authorization level. We will discuss the authorization levels in *Chapter 6: Configuring Security for Azure Functions* For now, we will select **Anonymous** and click as shown in *figure*

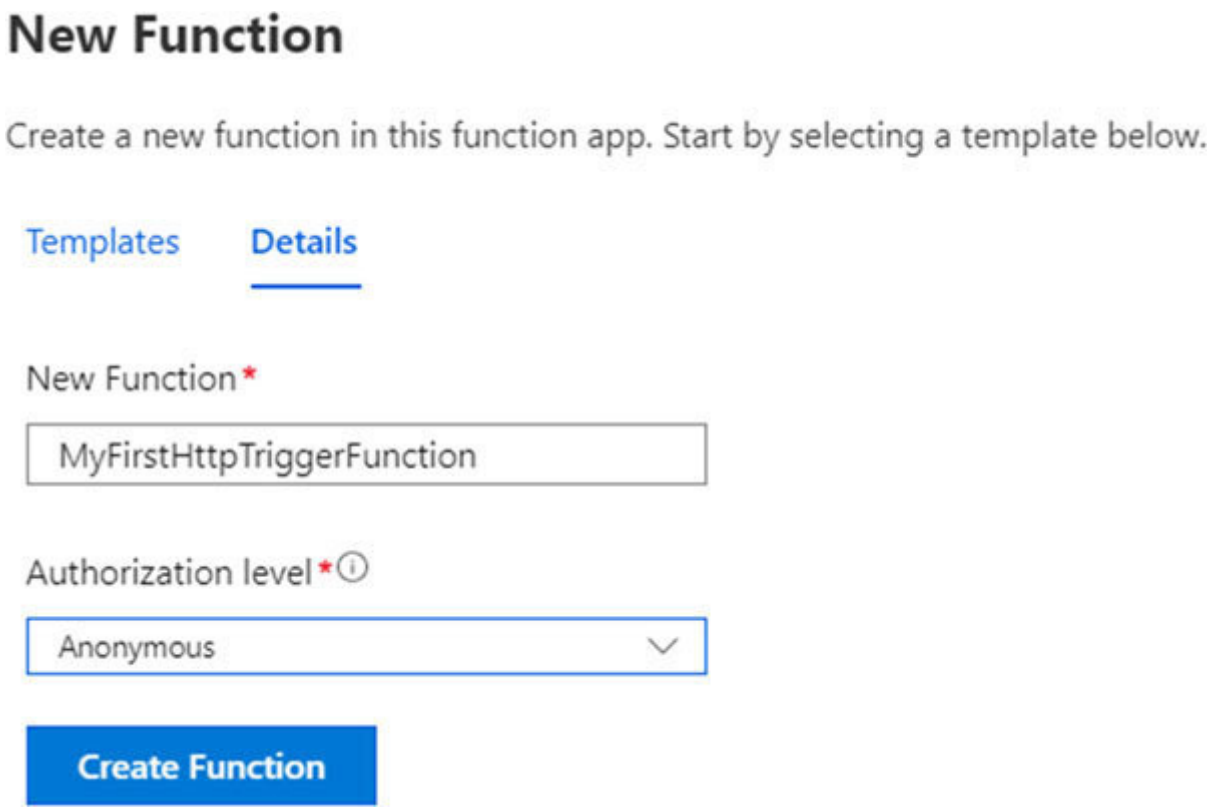


Figure 3.13: *Create a new function*

This will create a new Azure function in our function app that can be triggered by sending an HTTP request to the function URL.

Testing and running the function in the portal

Please follow these steps to test your function in the portal:

Click on + in your newly created HTTP-triggered function. You will find the option in the left menu of your function, as shown in *figure*

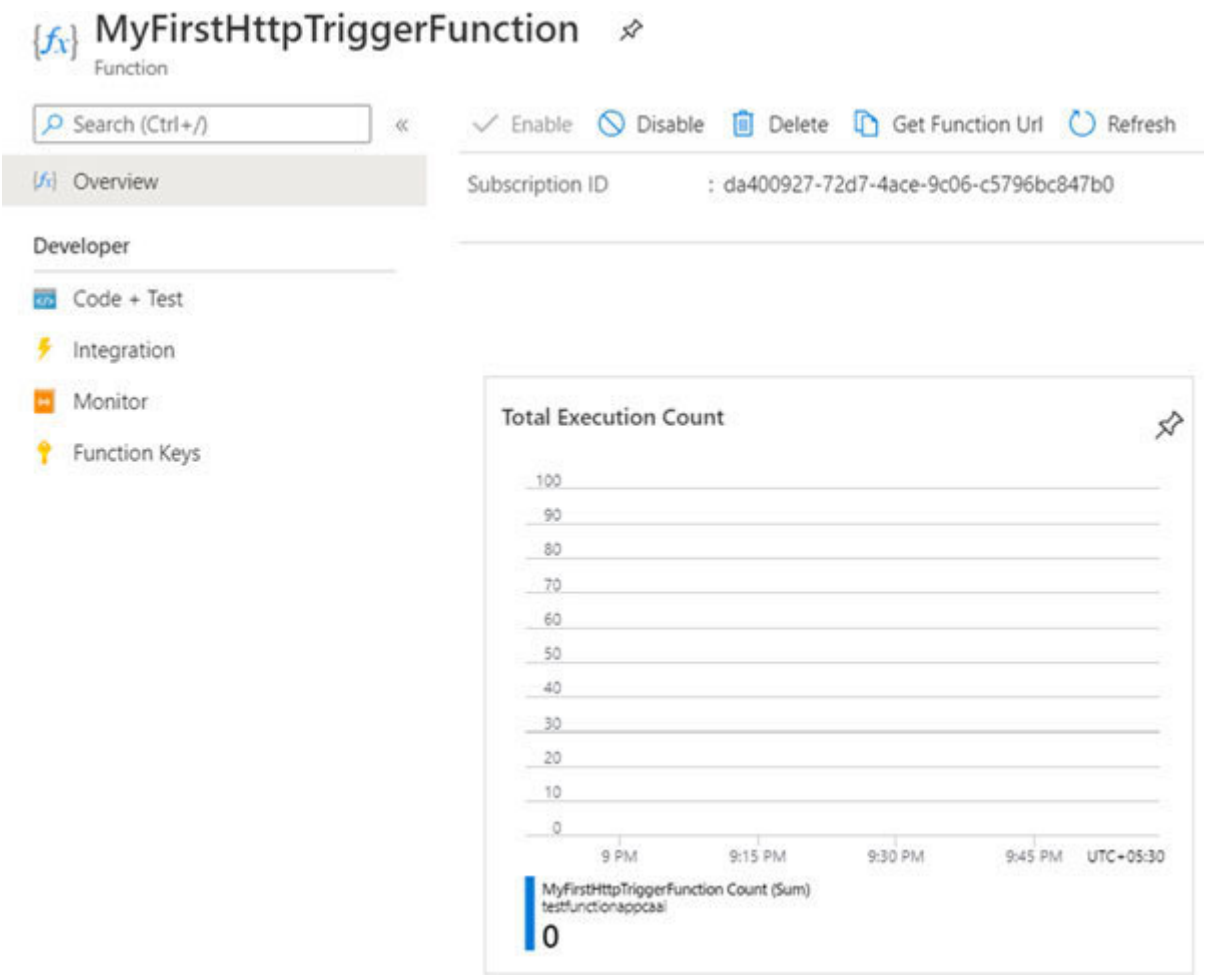


Figure 3.14: *Azure Function left menu*

This will open the function code that is created as the default HTTP-triggered template. You can modify or re-write the code as per your need. For the first function, we will keep it as is.

On the **code + Test** page, click on **Function** from the top menu, as shown in *figure*

Figure 3.15: *Function Code + Test Page.*

A dialog box will appear, select default as key, and click on the **to** icon in the URL box, as shown in *figure*

Figure 3.16: *Get Function UTL dialog box*

Open the browser and paste the function URL in the address bar. Add the query string parameter – at the end of the URL, and hit *Enter* to execute the request. *Figure 3.17* shows the response with an example URL.

Figure 3.17: *HTTP trigger default function response on browser*

You can also view the trace information in the logs when your function runs by expanding the logs at the bottom of the page + page, as shown in *figure*

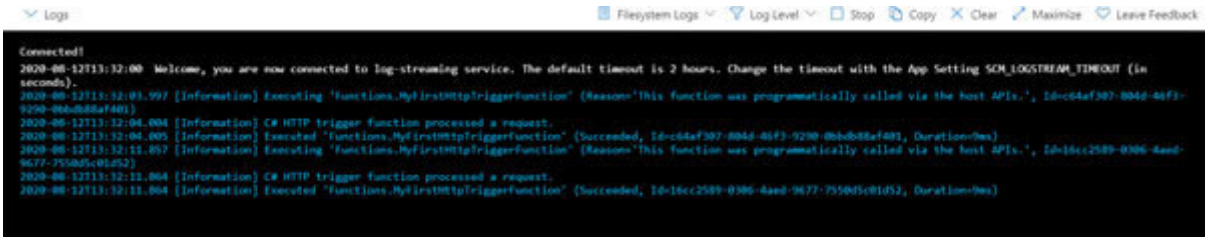


Figure 3.18: *Function logs on the "Code + Test" page*

Creating a function using Visual Studio Code

In the last section, you understood how to create Functions in the Azure portal. Let us leverage Visual Studio Code's built-in templates without worrying about underlying framework connectivity. After creating a sample project using built templates, you can easily edit configurations, source code using a standard editor like Visual Studio, Visual Studio code, and so on. For developers, moving from portal to editor can be quite challenging sometimes. Azure Functions supports multiple extension tools to make this transition easy. Developers can use features like IntelliSense support, line-by-line code debugging, and integration with Azure DevOps.

Before you get started with hands-on activities, you must have the following software, core tools, and extensions installed on your machine or environment:

An Azure account with an active subscription

Visual Studio or **Visual Studio Code**

The C# Extensions for VS Code

The Azure Functions extensions for VS Code

The Azure extensions for VS Code

Node.js for NPM, which is required by Windows only, not by macOS and Linux

Please find all installation URL as follows. At the time of writing this book, all the mentioned links are active and working.

You can install extensions from **https://marketplace.visualstudio.com/vscode** or extensions menu option available in VS Code.

Install .NET Core for Windows from

Install Node.js, which consists of NPM, from

Install Visual Studio Code from

You can install Azure Functions Core Tools from These tools help you to authenticate your Azure account, publish the function to Azure.

You can also use the CLI command **extensions** to install all required extensions.

Use the **az --version** command to ensure the Azure CLI *version is 2.0.76* or above.

In *Chapter 2, Introduction to Azure* you have already read about the supported language runtime by Azure Functions. Please refer to *table 3.1* to Install the language-specific requirements:

requirements:
requirements: requirements: requirements: requirements: requirements: requirements:
requirements: requirements: requirements: requirements: requirements: requirements: requirements: requirements: requirements: requirements:
requirements:
requirements: requirements: requirements: requirements: requirements:

Table 3.1: *Language-specific tool requirements*

The Azure Functions extension enables you to edit, build, and run functions on your local machine, writing functions in your languages of your choice, and publishing functions to Azure directly. The extension can be used with C# (compiled/script*), PowerShell, Python, JavaScript, and Java languages.

You need to search for Azure Functions extension by clicking on **Extensions** from the left menu or select the 'link in VS Code. Please refer to *figure*

Click on **Install** to install the extension for Visual Studio Code. After installation, you can see an Azure:Functions area in the Azure activity bar if you are signed in to your Azure account, or you may sign in with your credentials to see all subscriptions associated with your account.

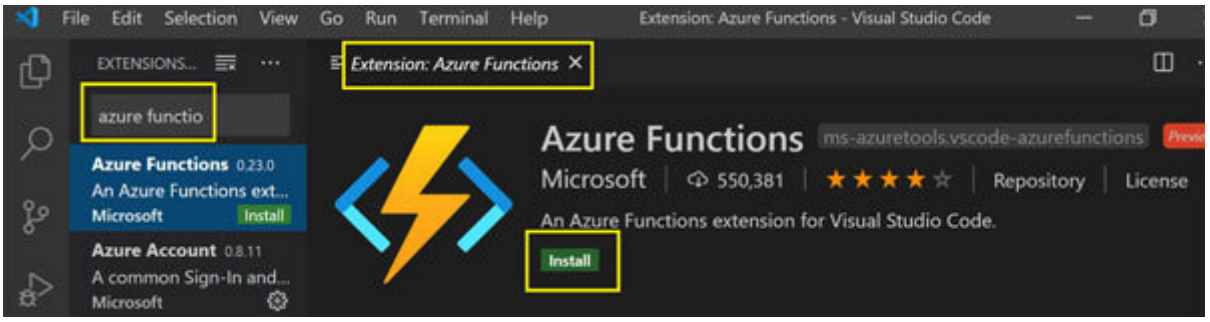


Figure 3.19: *Install Azure extension for Visual Studio Code*

In VS Code, you need to press *F1* to open the command palette, and you are good to get started with the Azure activities you wish to do. Please refer to *figure*

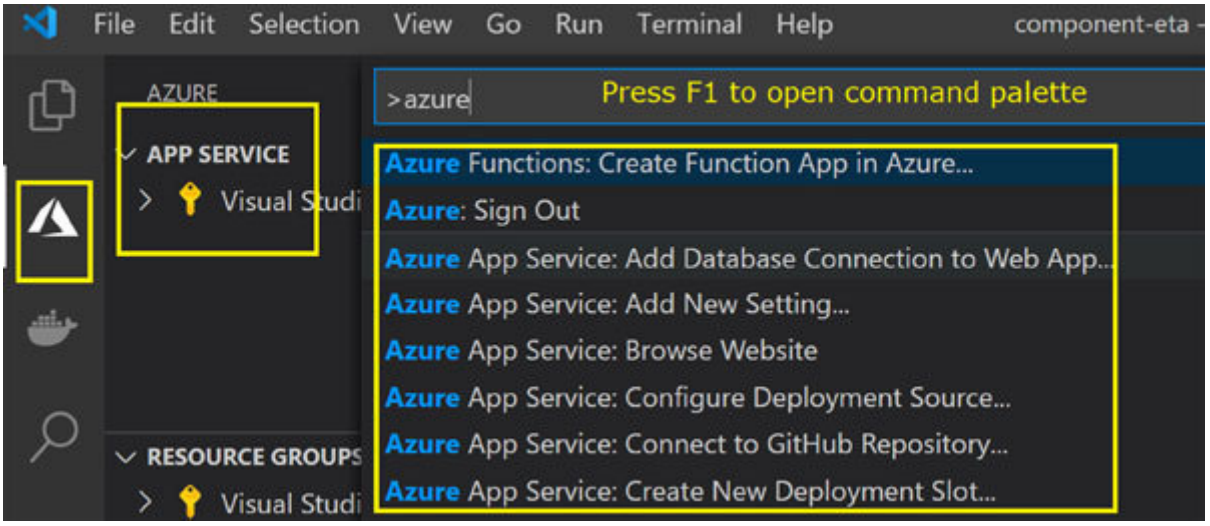


Figure 3.20: *Command palette*

Creating an Azure Functions project

After installing the required extensions, you must sign in with your Azure subscription.

Let's understand project creation steps as follows:

In the Azure Functions section of the Azure activity pane, you can create a sample project and create a function using built-in templates. Please refer to *figure*

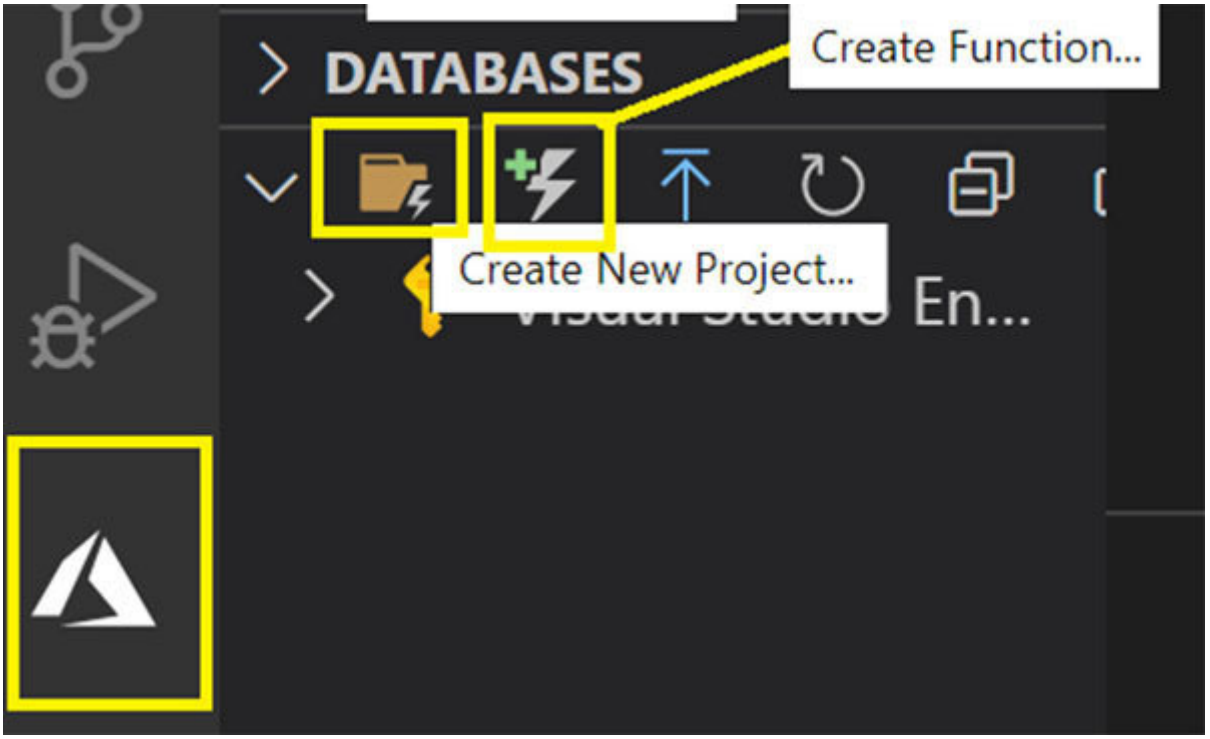


Figure 3.21: *Create a new project and a function*

You need to enter the information in every step when prompted in the command palette. Select a language of the function project, for example, C#, as shown in *figure*

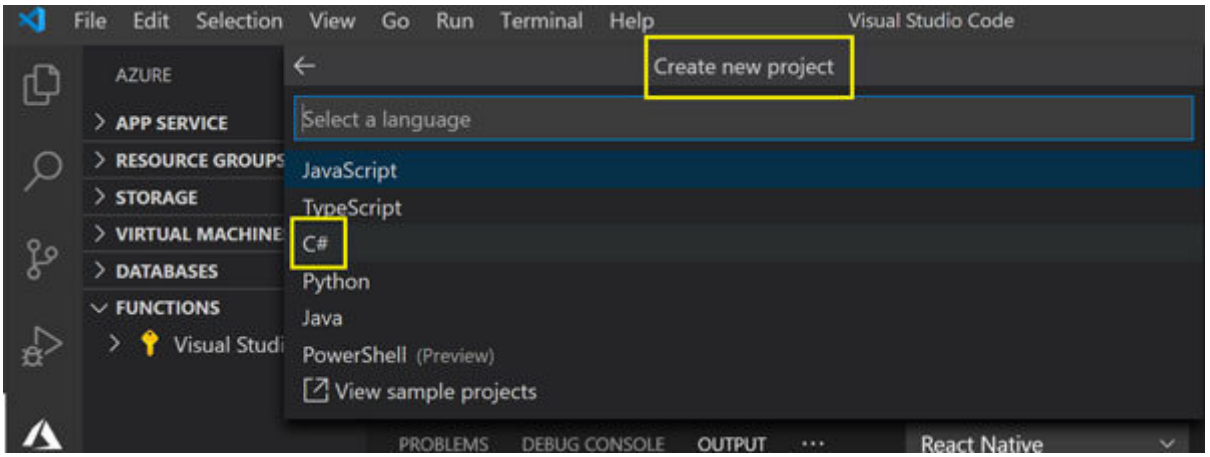


Figure 3.22: *Choose a programming language*

Then, select a built-in template for the Azure Functions project's function, for example, HTTP trigger. You may create a sample function now or can add it later. Please refer to *figure*

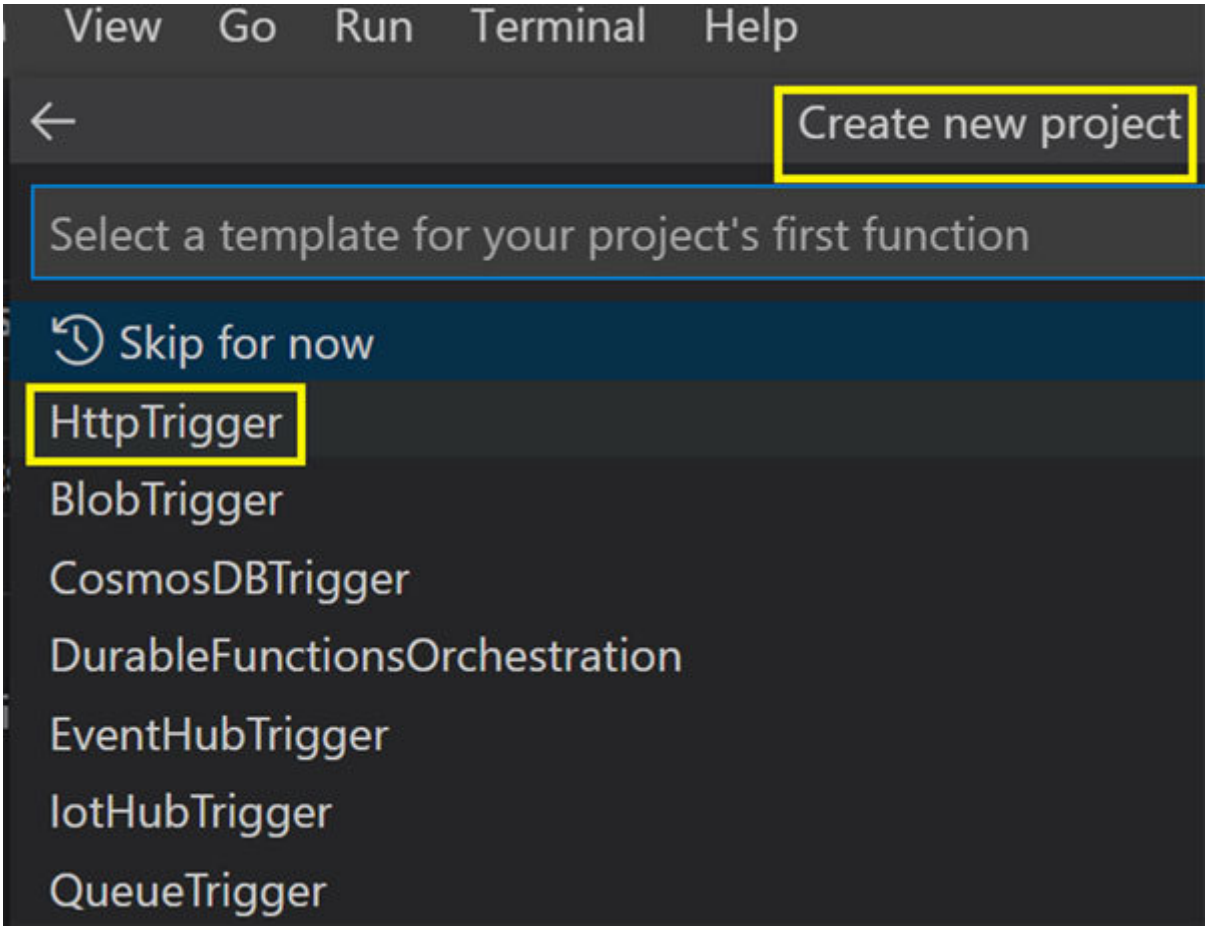


Figure 3.23: *Select from available template*

The next step is to provide function name and namespace, as shown in *figure*

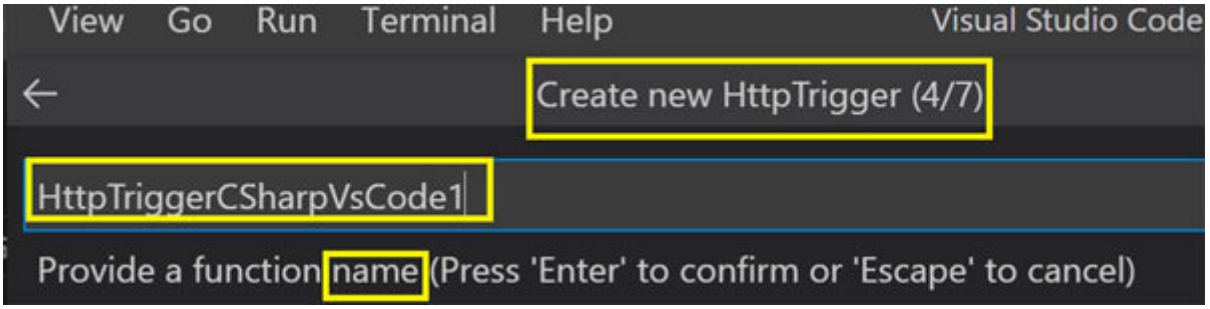


Figure 3.24: *Enter the name of the function*

The next step is to select the function authorization parameter. These authorization levels can be function, anonymous, or admin. Function and admin are key-based access rights. Function access rights need host keys. Anonymous authorization level enables everyone to access your function endpoint, i.e., no authentication required. Please refer to *figure*

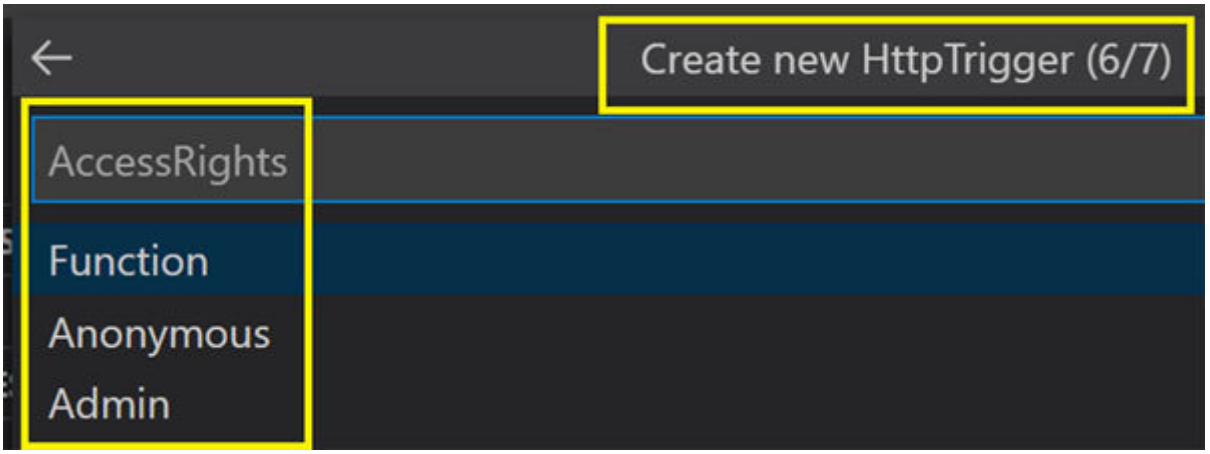


Figure 3.25: *Choose access rights of function*

After providing values in all steps, you can explore the sample project created using a template, as shown in *figure*

host.json helps you configure the functions host and applied while running locally and while running on Azure

local.settings.json has settings that get applied while running functions locally

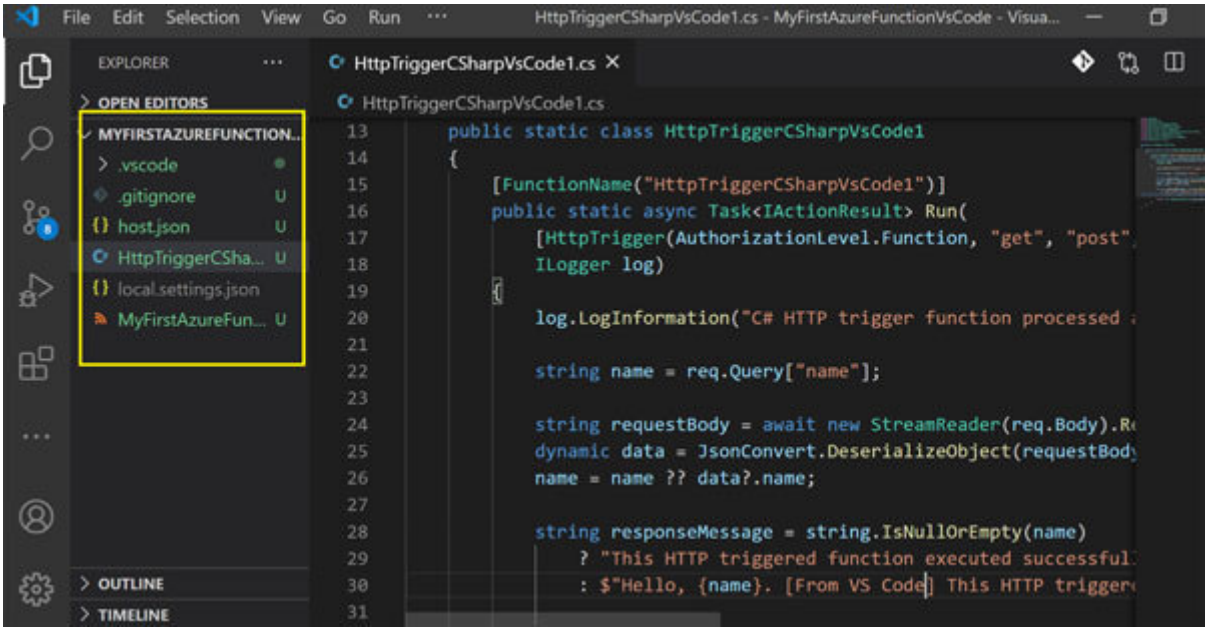


Figure 3.26: *Sample function created*

Publishing a function application to Azure

You can pack and deploy your functions project to Azure by creating a function app and other required resources.

You have two options to publish Azure function app like the quick function app of creating a path and an advanced path, as shown in *figure* Zip deploy technology is used to deploy Azure Function to Azure directly from Visual Studio Code.

If you choose the first option to create a new function app in Azure, values are picked automatically based on your function app. Please refer to *figure* If you want to provide your values, then you must select the advanced create path.

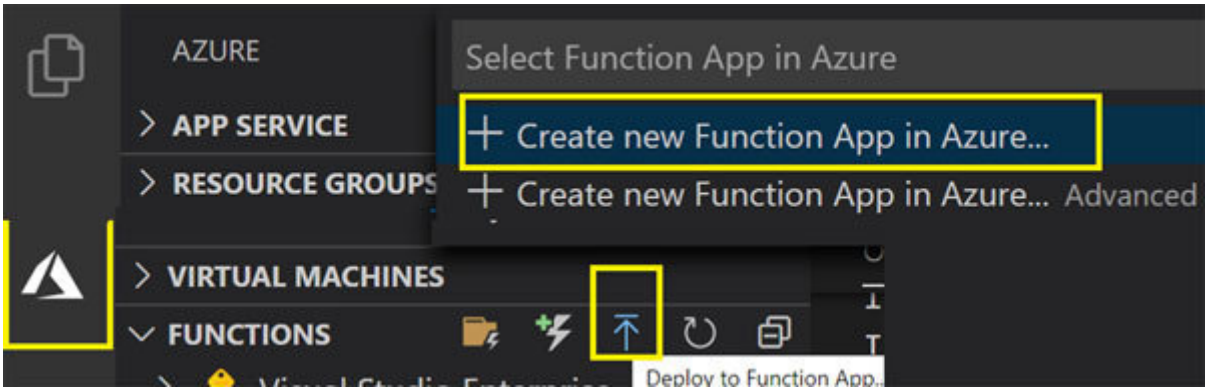


Figure 3.27: *Deploy function to Azure*

In the second option, you need to select OS, hosting plan, runtime, resource group, storage account, and resource locations,

as shown in figure

Then, select runtime for your function app stack, as shown in figure

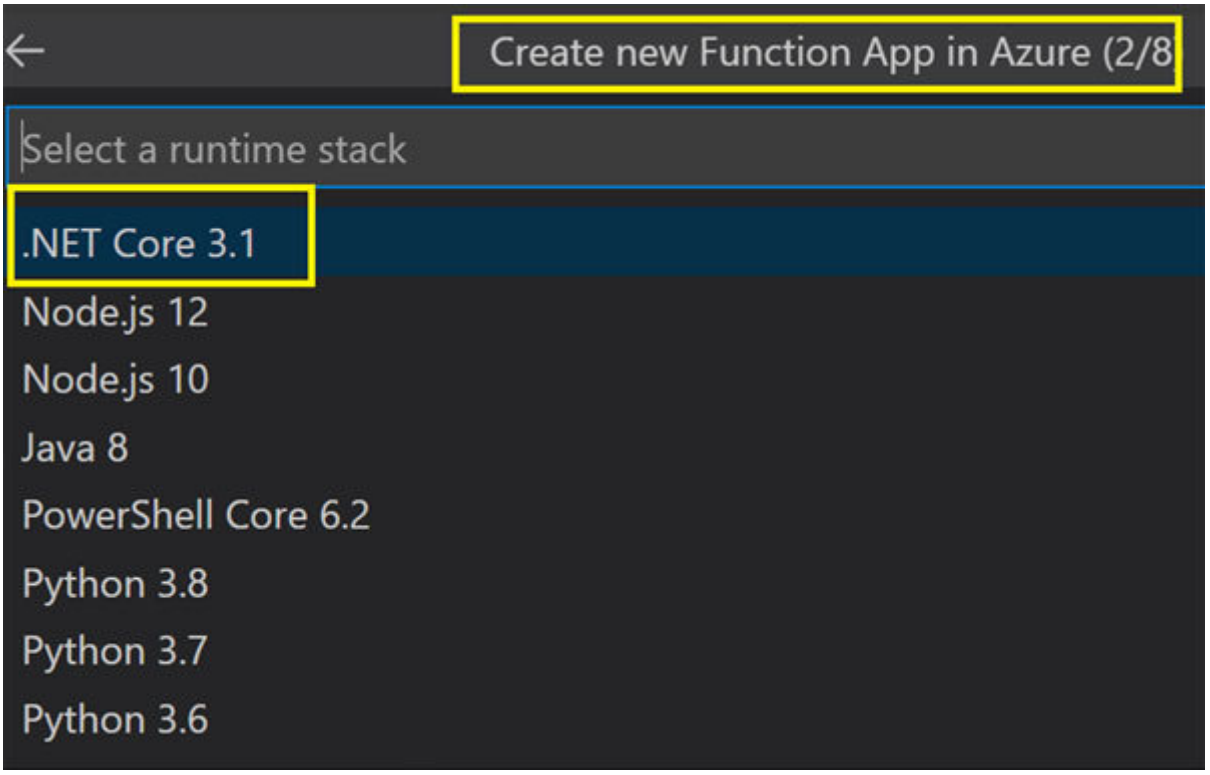


Figure 3.28: *Select runtime stack*

After selecting runtime, now you must select the operating system on which your function app will run. Please refer to figure 3.29.

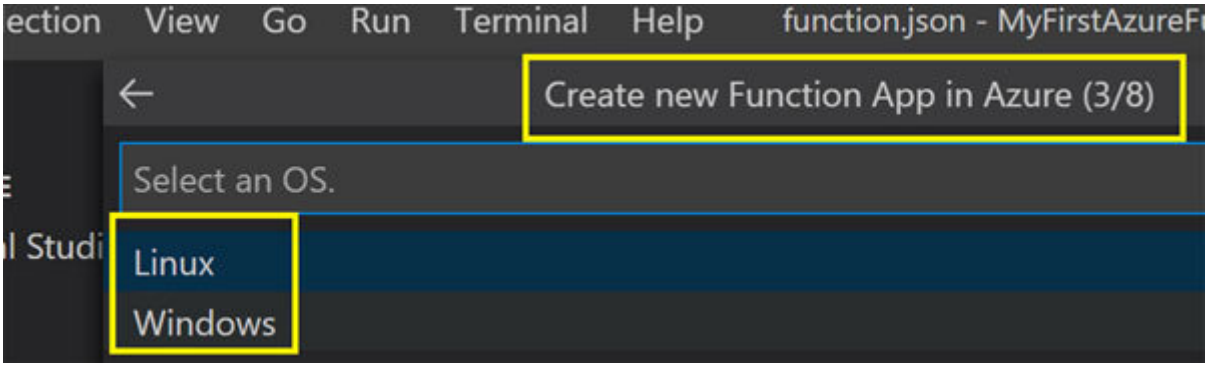


Figure 3.29: *Select operating system*

Now you must select the hosting plan as per the requirement of your function. Please refer to *figure*

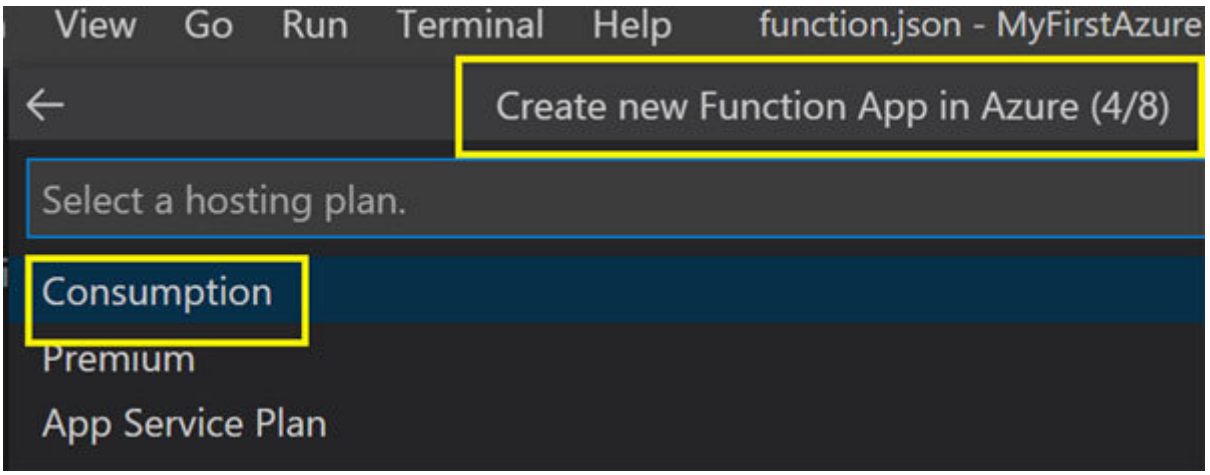


Figure 3.30: *Select hosting plan*

To group the resources logically, you must select a resource group. This resource group will help you later to clean up the resources you don't need. Please refer to *figure*

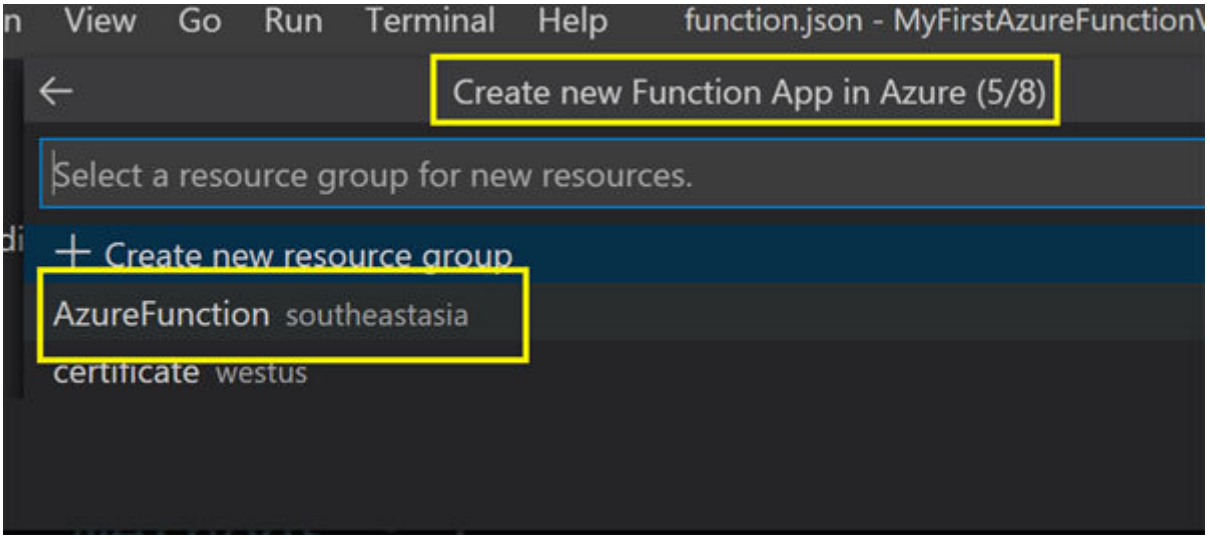


Figure 3.31: *Create a resource group*

The last mandatory step is to select an existing storage account or create a new storage account. Please refer to *figure*

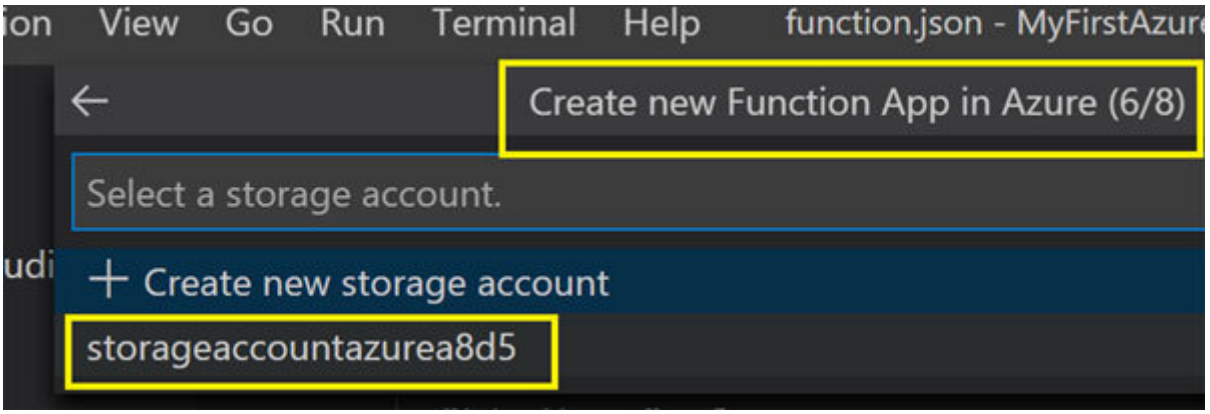


Figure 3.32: *Select or create an Azure storage account*

You may create or select an existing Application Insights resource to analyze the function app's execution logs; if required, otherwise can be skipped. Please refer to *figure*

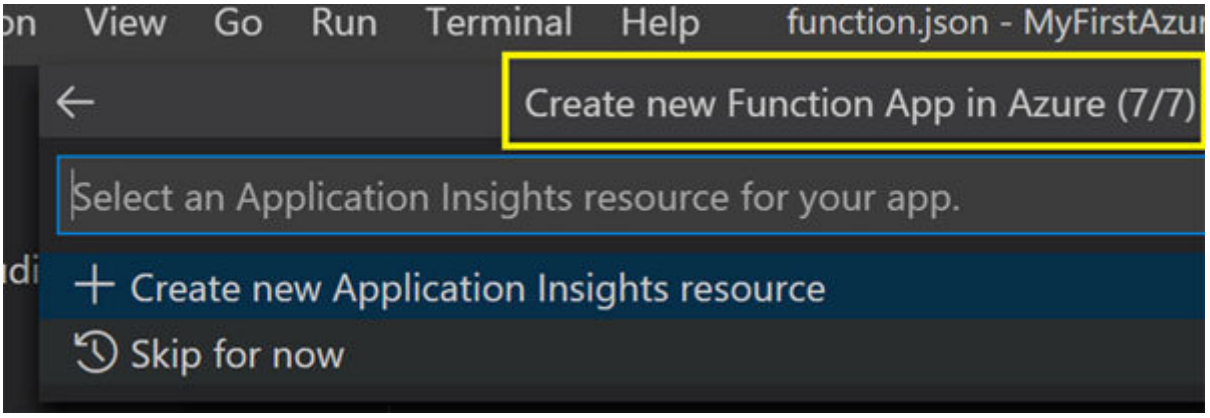


Figure 3.33: *Create application insight or skip*

After providing all possible values in each step, Visual Studio Code will start creating a function app against your selected Azure subscription. Please refer to *figure*

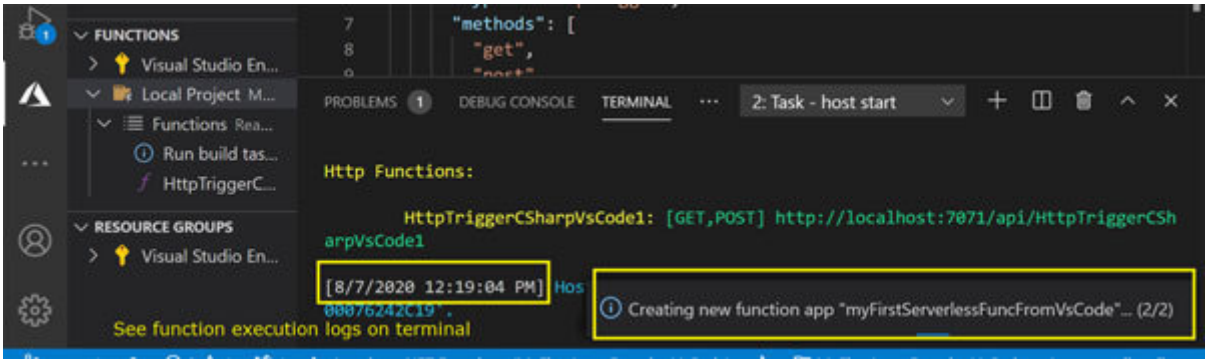


Figure 3.34: *Function creation in progress*

You need the URL to invoke your HTTP-triggered function after deployment—Press *F1* to run Azure Functions: Copy Function URL command. You may check your deployments, as shown in *figure*

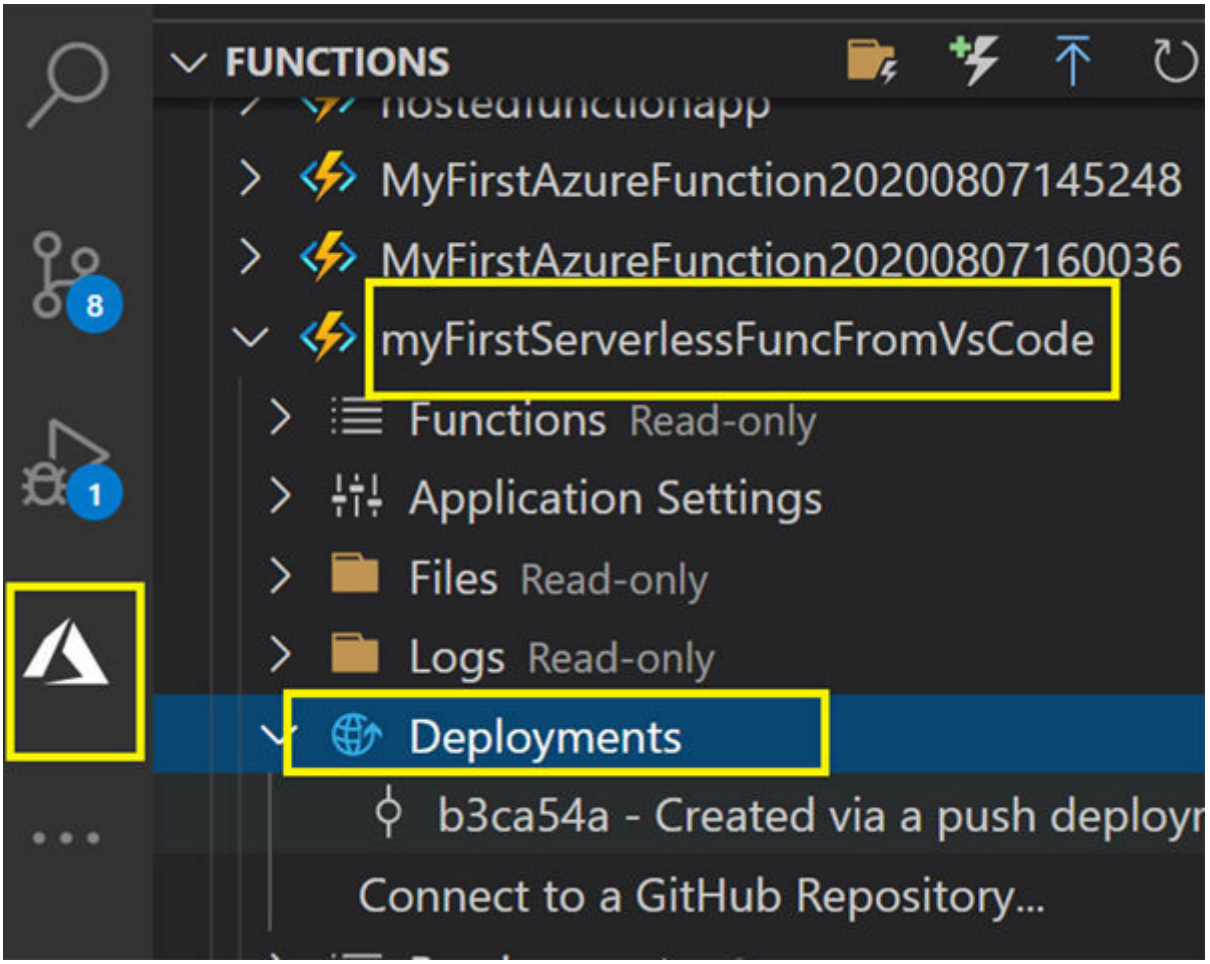


Figure 3.35: *Azure Functions deployment details*

By installing core tools, you can run a functions project by pressing *F5* on your local machine. The **local.setting.json** file configurations are applied while running it locally.

After running it, you can see the output on the terminal; you can see the endpoint URL on a terminal or browse using another option, as shown in *figure* This function URL can be used to invoke your function using a client tool like browser, postman. To stop the execution, you need to press *CTRL +*

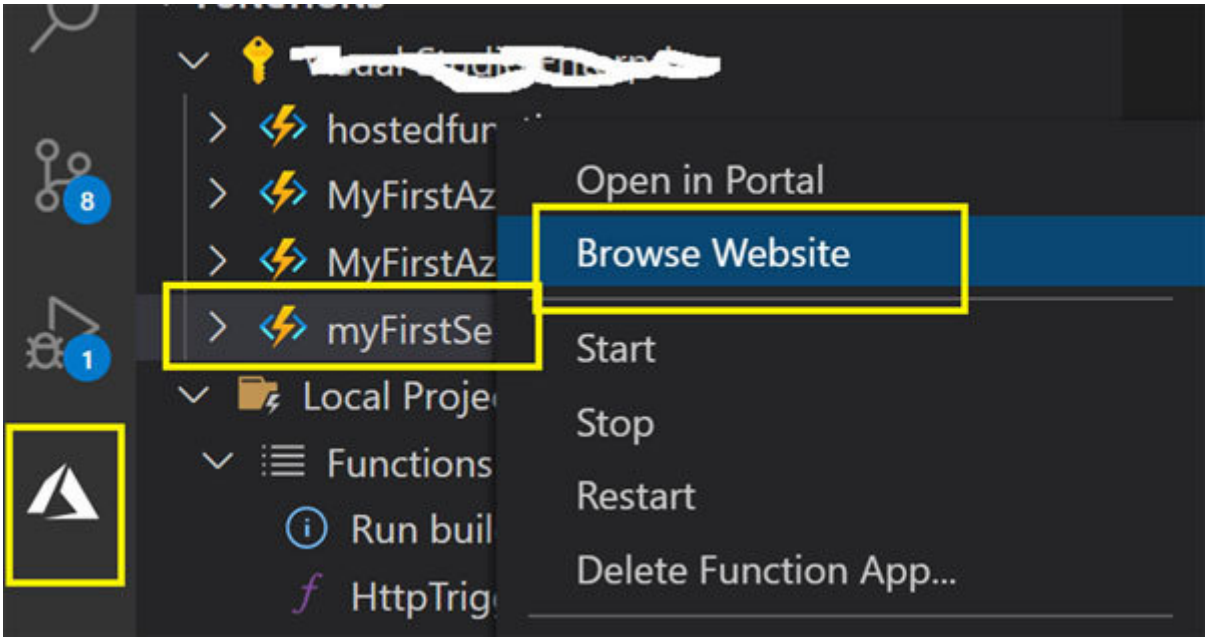


Figure 3.36: *Browse Azure Functions*

You can browse your local function and deployed function by hitting URL in the browser, as shown in *figure* These are GET request which includes as a query string parameter.

A sample local URL can be
http://localhost:7071/api/HttpTriggerCSharp

Figure 3.37: *Azure Functions output after running local and hosted copy*

You can also verify whether your function is created in the Azure portal or not. Please refer to *figure*

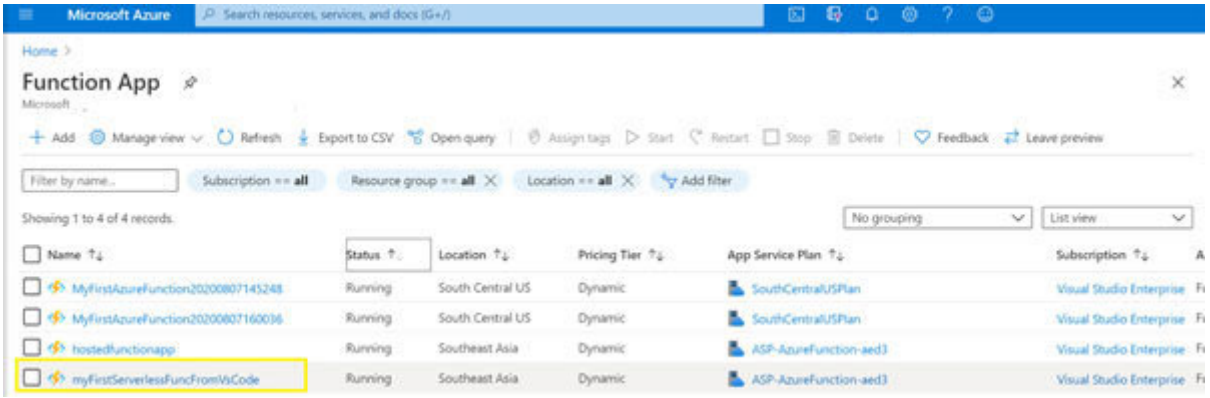


Figure 3.38: *Azure Function app created in the Azure portal*

Creating a function application using Visual Studio 2019

Azure Functions provides options to choose the developer's favorite IDE tool to write code. You can quickly create and deploy Azure functions locally using the same templates provided on the Azure portal. You can easily integrate these functions with other cloud services. The steps to create and deploy Azure Functions are nearly the same in Visual Studio.

Without Azure live subscription, you can code, run, debug, and test your Azure Functions in your local machine using Visual Studio. Most of the time in your development career, you might have faced a situation where your code works perfectly fine in your local environment but not in the production environment. But this is not the case with Azure Functions because the local runtime and Azure cloud runtime are the same.

Download the latest version of Visual Studio 2019 from

While installing Visual Studio for development, you must select the **Azure development** option in the **Workloads** section and then click on the **Install** button. Please refer to *figure*

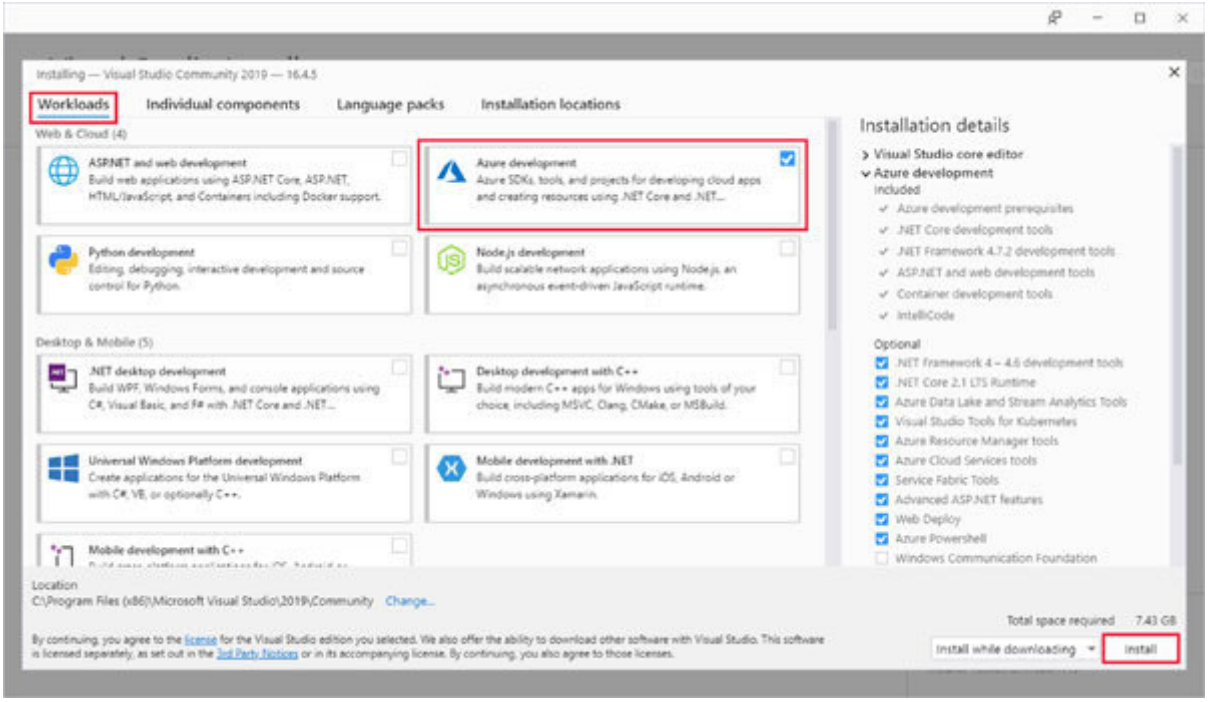


Figure 3.39: *Install Visual Studio with Azure development components*

Let's learn how to create an Azure function in Visual Studio using the latest Azure Functions runtime.

In Visual Studio, Select the **File** > **New** > **Project** option. Then, to create a new project, search for Azure Functions, as shown in *figure*

Next, you need to configure your project with fields like project name, directory and click on

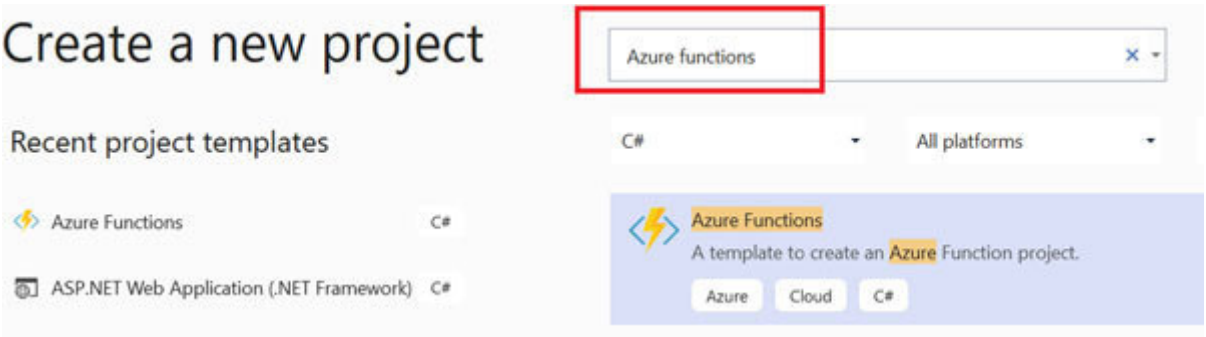


Figure 3.40: *Create Azure Functions project*

To create a new Azure Function application, select functions runtime, function template, storage account, and authorization level.

Azure Functions needs a storage account to publish your function app to Azure.

Except for the HTTP trigger, all other trigger type needs valid storage account connection string.

You can provide access to anonymous users or restrict access by setting up the authorization level. Please refer to *figure*

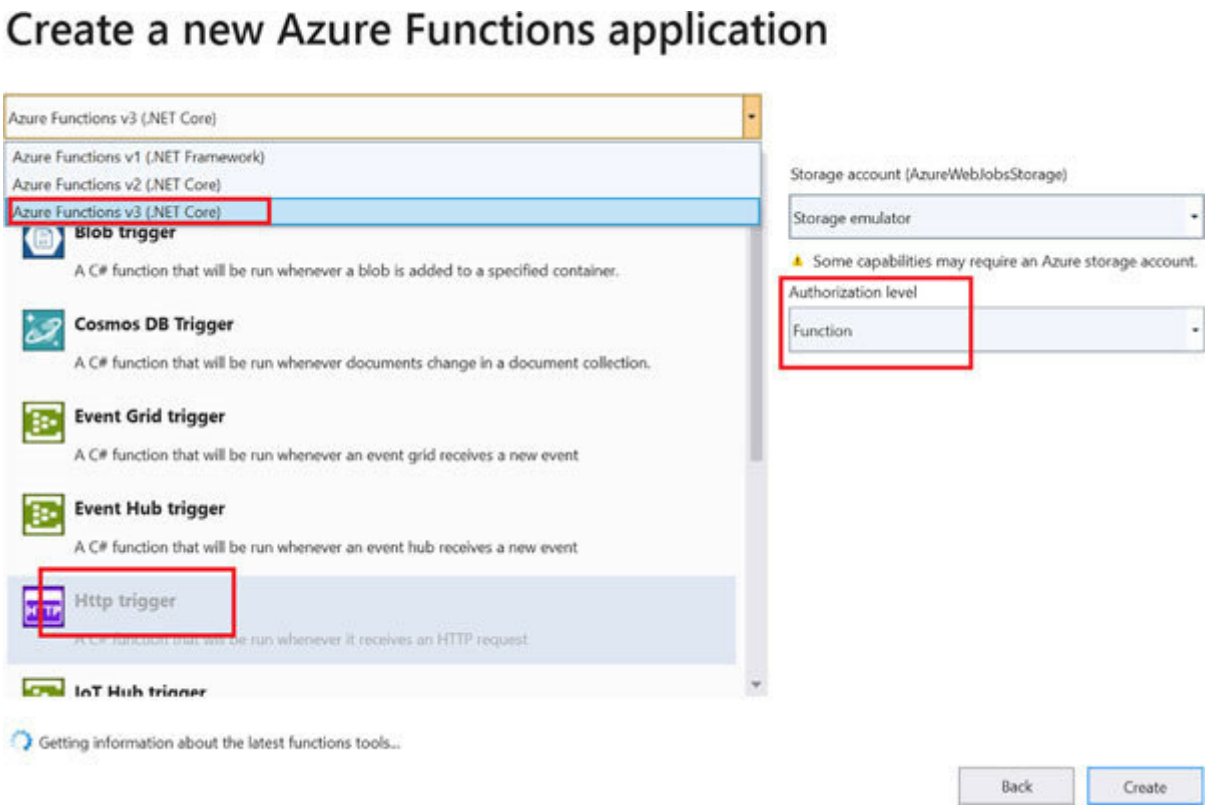


Figure 3.41: *Choose an existing template*

By pressing *CTRL* + you can run your function app locally, as shown in *figure* If Azure CLI tools are not yet installed on your machine, they will automatically be downloaded and installed.

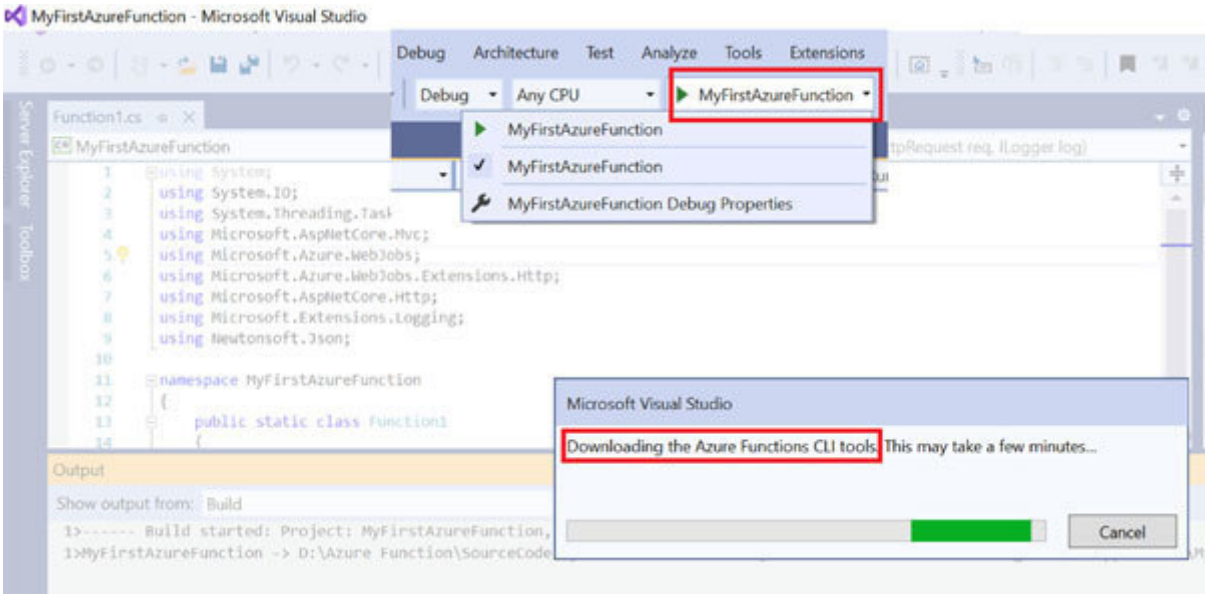


Figure 3.42: *Run Azure Functions locally to test*

While developing Azure Functions locally, you can add debuggers to understand execution and troubleshoot problems. You can see function execution logs as well on the terminal, as shown in *figure* Once the function app starts successfully, you can check by accessing the localhost URL with a port number, for example, You can do debugging of Azure Functions running on the cloud as well. We will check how to debug the function running on the production cloud in the next chapters.

```
C:\Users\keagniho\AppData\Local\AzureFunctionsTools\Releases\3.3.1\cli_x64\func.exe
8/7/2020 9:16:31 AM] Starting JobHost
8/7/2020 9:16:31 AM] Starting Host (HostId=keagniho-237103906, InstanceId=def44e19-e51c-419d-9f21-4f8bbac28a86, Versi
3.0.13107, ProcessId=26952, AppDomainId=1, InDebugMode=False, InDiagnosticMode=False, FunctionsExtensionVersion=(null
8/7/2020 9:16:31 AM] Loading functions metadata
8/7/2020 9:16:31 AM] 1 functions loaded
8/7/2020 9:16:31 AM] Generating 1 job function(s)
8/7/2020 9:16:32 AM] Found the following functions:
8/7/2020 9:16:32 AM] MyFirstAzureFunction.Function1.Run
8/7/2020 9:16:32 AM]
8/7/2020 9:16:32 AM] Initializing function HTTP routes
8/7/2020 9:16:32 AM] Mapped function route 'api/Function1' [get,post] to 'Function1'
8/7/2020 9:16:32 AM]
8/7/2020 9:16:32 AM] Host initialized (722ms)
8/7/2020 9:16:32 AM] Host started (736ms)
8/7/2020 9:16:32 AM] Job host started
Hosting environment: Production
Content root path: D:\Azure Function\SourceCode\MyFirstAzureFunction\MyFirstAzureFunction\bin\Debug\netcoreapp3.0
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.

Http Functions:

        Function1: [GET,POST] http://localhost:7071/api/Function1

8/7/2020 9:16:39 AM] Host lock lease acquired by instance ID '000000000000000000000000076242C19'.
```

Figure 3.43: *Function execution logs on terminal*

You can run your HTTP trigger function by passing parameter values and can verify response on a browser window.

Please refer to *figure*

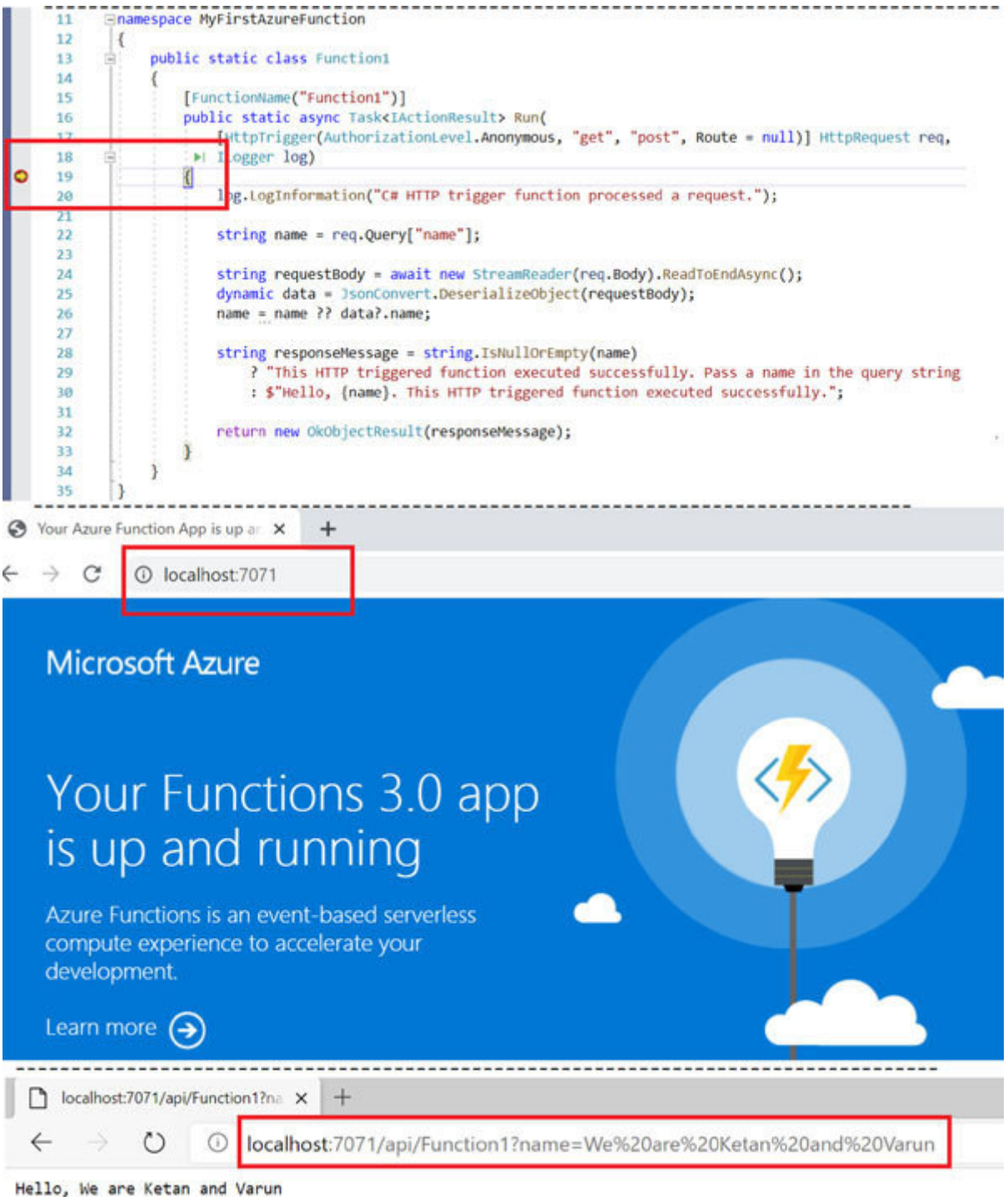


Figure 3.44: *Debugging function locally*

To publish your function app from Visual Studio, you need to right-click on the **project** and select the **Publish** option. Please

refer to *figure*

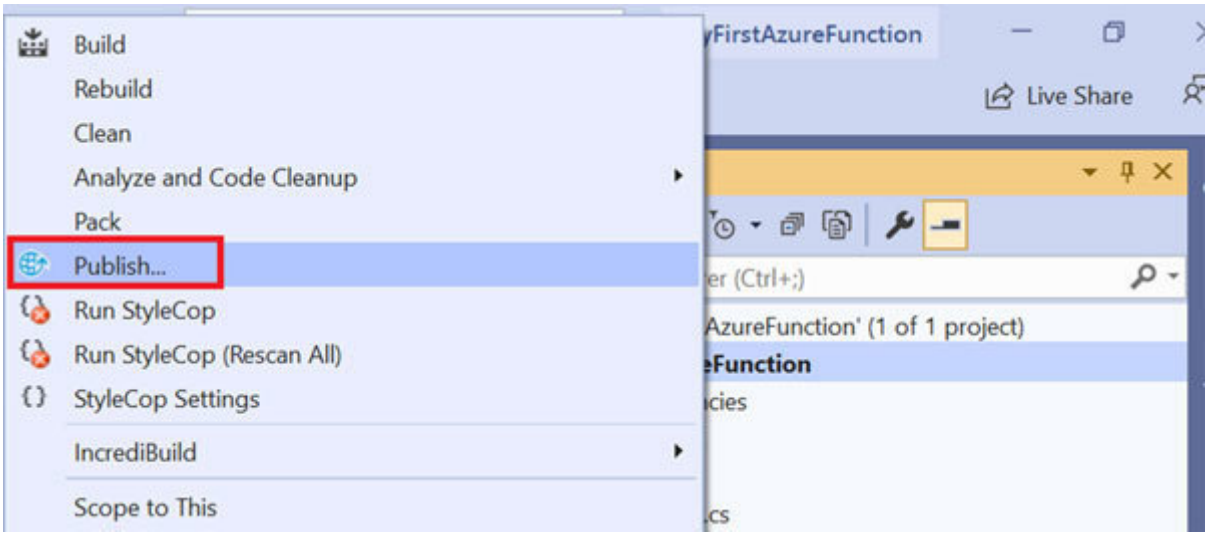


Figure 3.45: *Publish Azure Functions on Azure*

You need to select the target where you want to publish the Azure function app. It can be Azure, Docker, Folder, or your custom publish settings. Let's select Azure. Please refer to *figure*

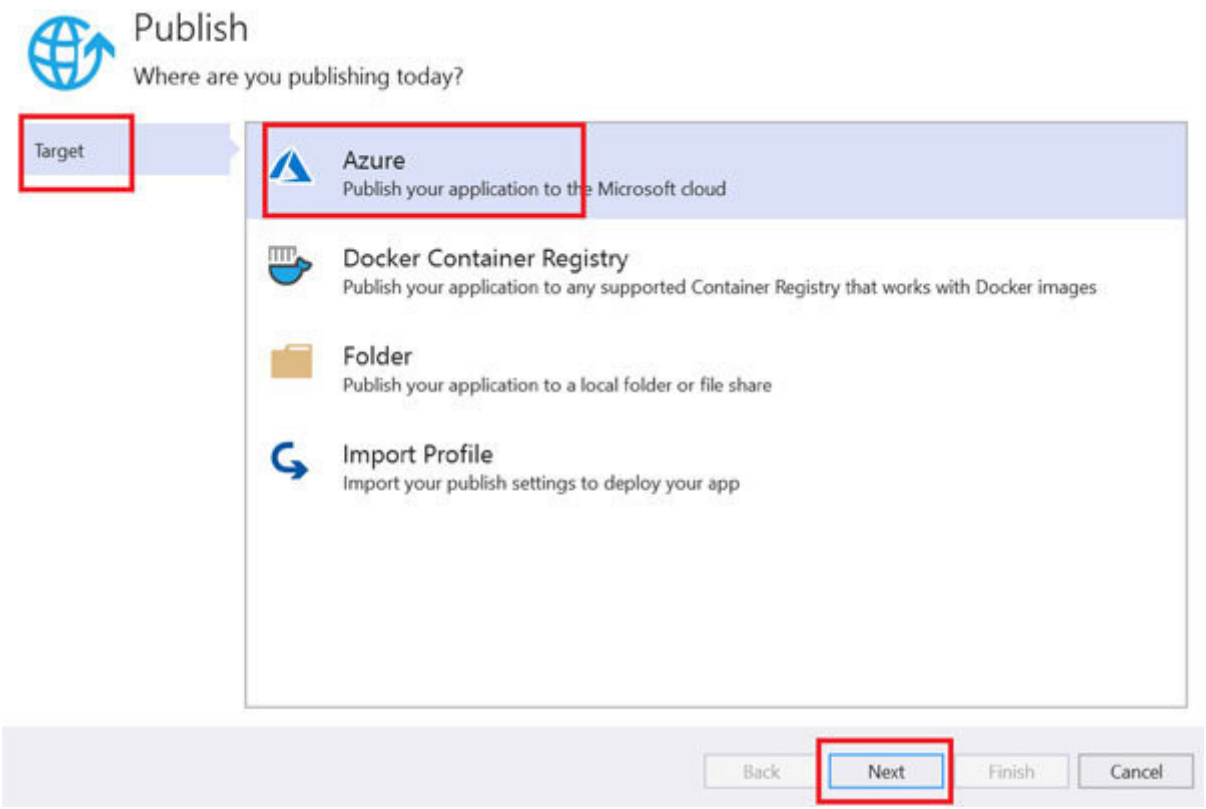


Figure 3.46: *Select target to publish*

You need to select a specific target in Azure, like a function app running on Windows, Linux, Function app container, or Azure container registry. Please refer to *figure*

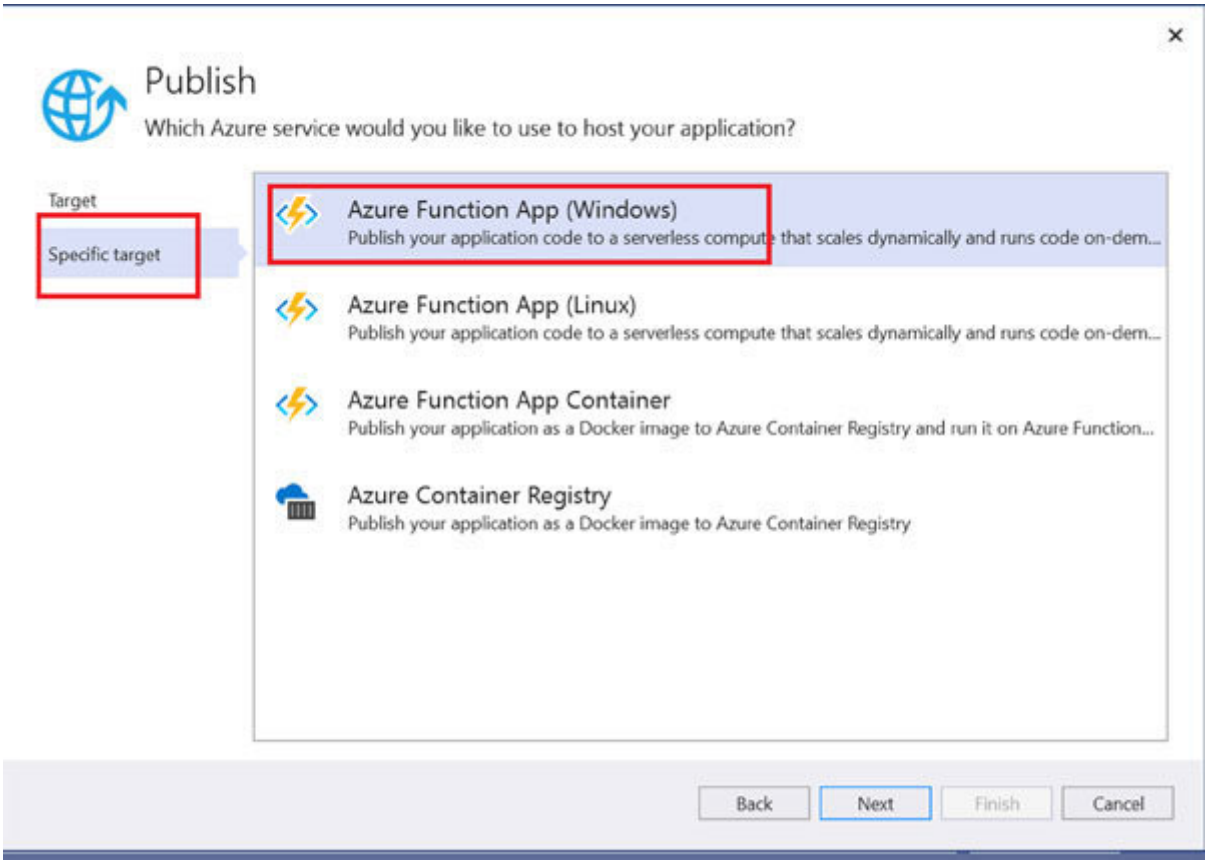


Figure 3.47: *Select specific target to publish*

Select the existing Azure Functions available in your Azure subscription or create a new Azure function app. You need to provide values to parameters when prompted while creating a new Azure function app. Please refer to *figure*

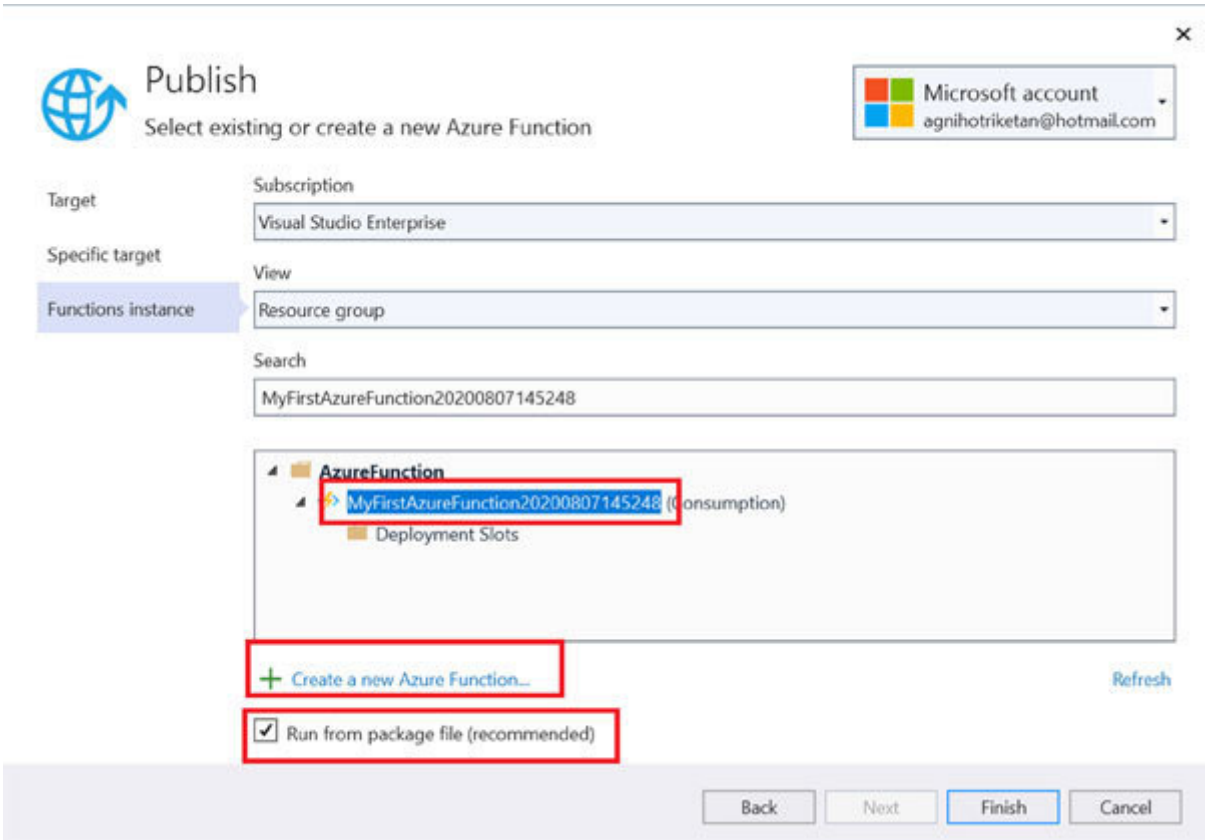


Figure 3.48: *Select existing Azure function or create one*

Hold on; your publishing activity is not yet complete. Now, click on the **Publish** button to start the publishing process from Visual Studio. Please refer to *figure*

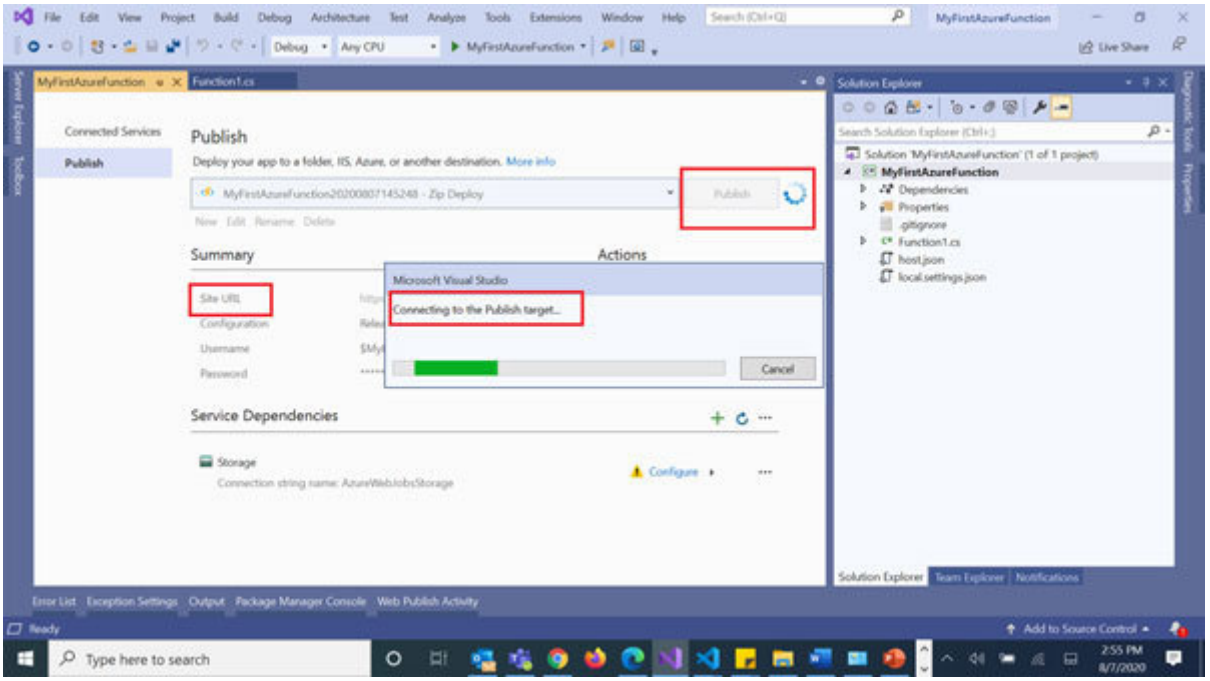


Figure 3.49: *Publish your Azure Function from Visual Studio*

When we deploy, Azure Functions from Visual Studio are precompiled **.dll** files pushed to Azure. Hence, we cannot edit them in the portal after deployment. You can change the configurations like the Azure Storage connection string.

Azure CLI

Azure CLI is a flexible and powerful command-line interface tool that helps you create and manage Azure resources through commands. Azure CLI is mostly used to automate Azure Resource Management and speed up the work.

Azure CLI is available for all Azure services and available to install on Windows, Linux, and macOS. You can also run the Azure CLI tool in cloud shell and Docker. You can run Azure CLI commands against a single Azure subscription or use multiple subscriptions per command.

Use the following URL to install Azure CLI on your system:

https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest

You can easily start learning and using Azure CLI by running in the Azure cloud shell. To use Azure cloud shell, you need to use terminal by clicking the terminal icon as it is highlighted in *figure*

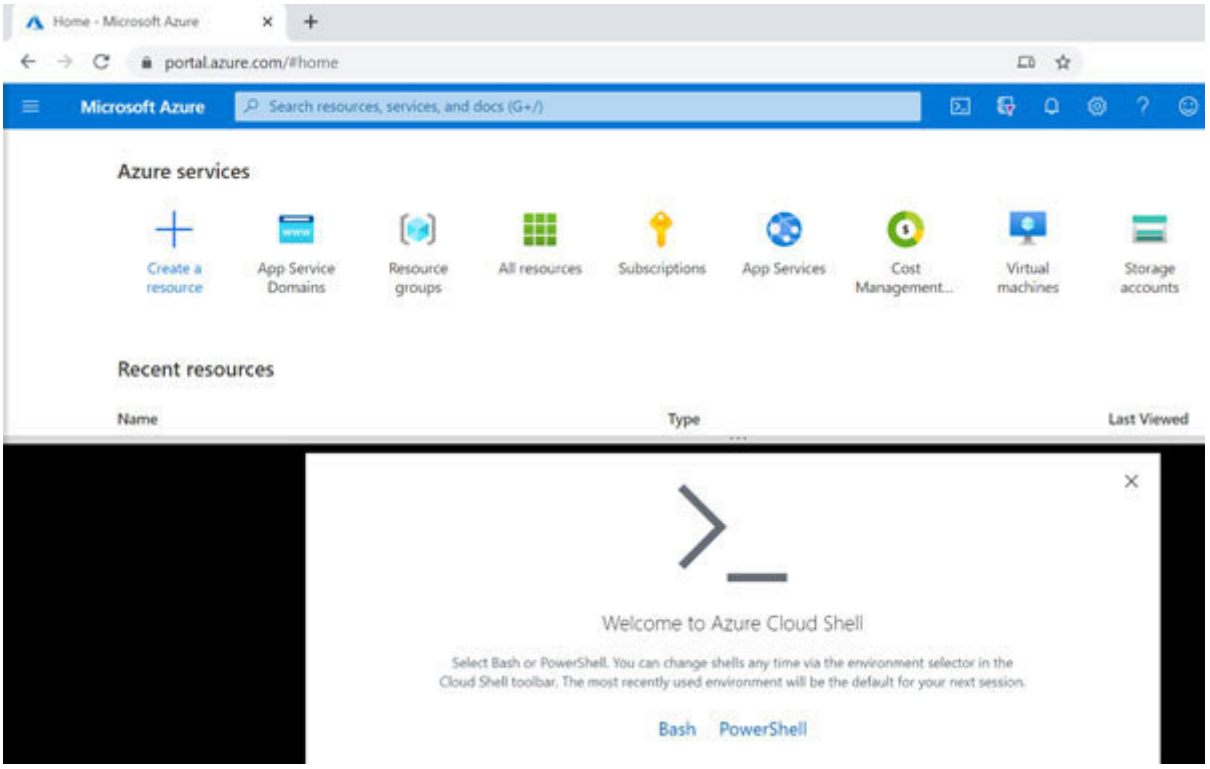


Figure 3.50: *Azure Cloud Shell*

You can get started with Azure Cloud Shell using the **az help** command. You can choose a shell of your choice from the list as highlighted in *figure*

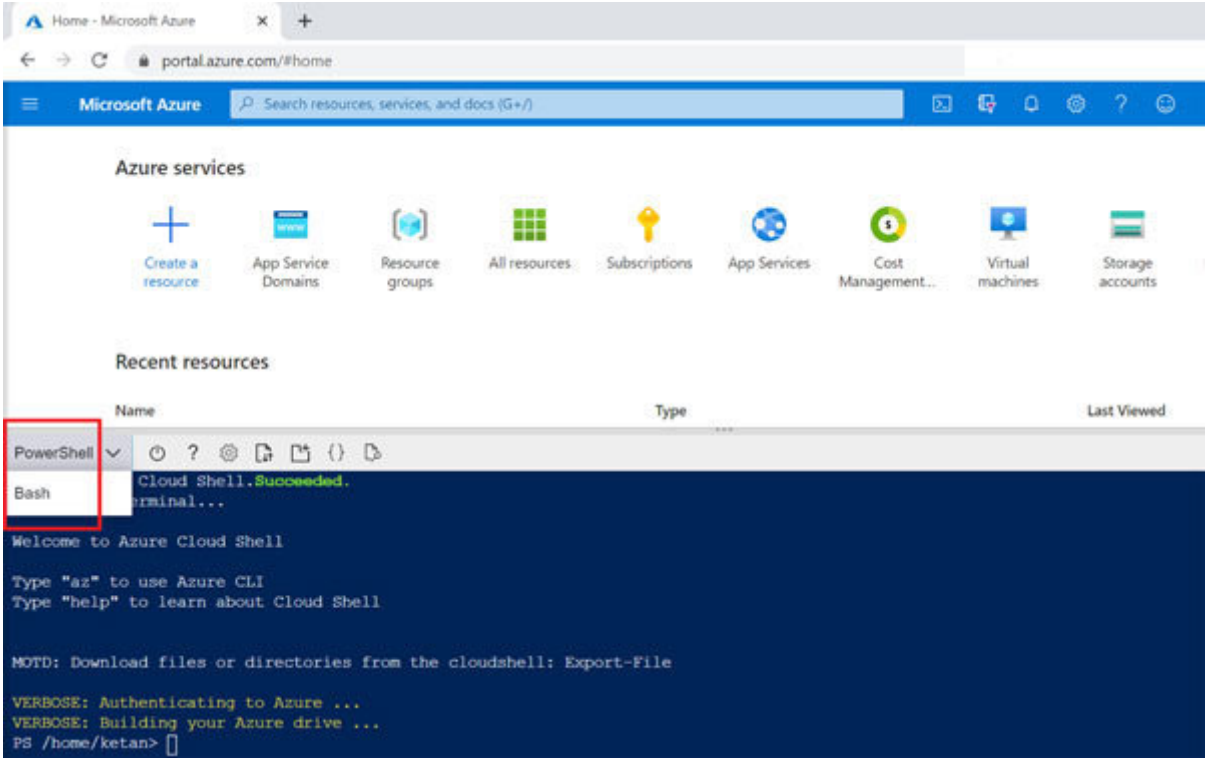


Figure 3.51: *Choose Shell*

Creating a function using Azure CLI

As you read about Azure CLI and Azure Cloud Shell in the last section, let's use Azure Cloud Shell to create a sample Azure Functions using Azure CLI commands in Cloud shell. Let's understand set CLI commands to create the Azure function.

The first step is to create a sample function app project. Please refer sample command below to create a dotnet function app with the name 'LocalFunctionProj.'

func init LocalFunctionProj –dotnet

Move to created project folder by changing the directory using the following command:

cd LocalFunctionProj

Now, create an HTTP trigger function using the following command:

func new --name HttpExample --template "HTTP trigger"

Use the command to start the function.

Now, let's publish this function app on Azure, which we created using CLI commands.

Create resource group

To get started, you must create a resource group to group all resources logically.

Refer sample command below with parameters like name and location to create a resource group. Please refer to *figure*

az group create --name CLIAzureFunction --location westus

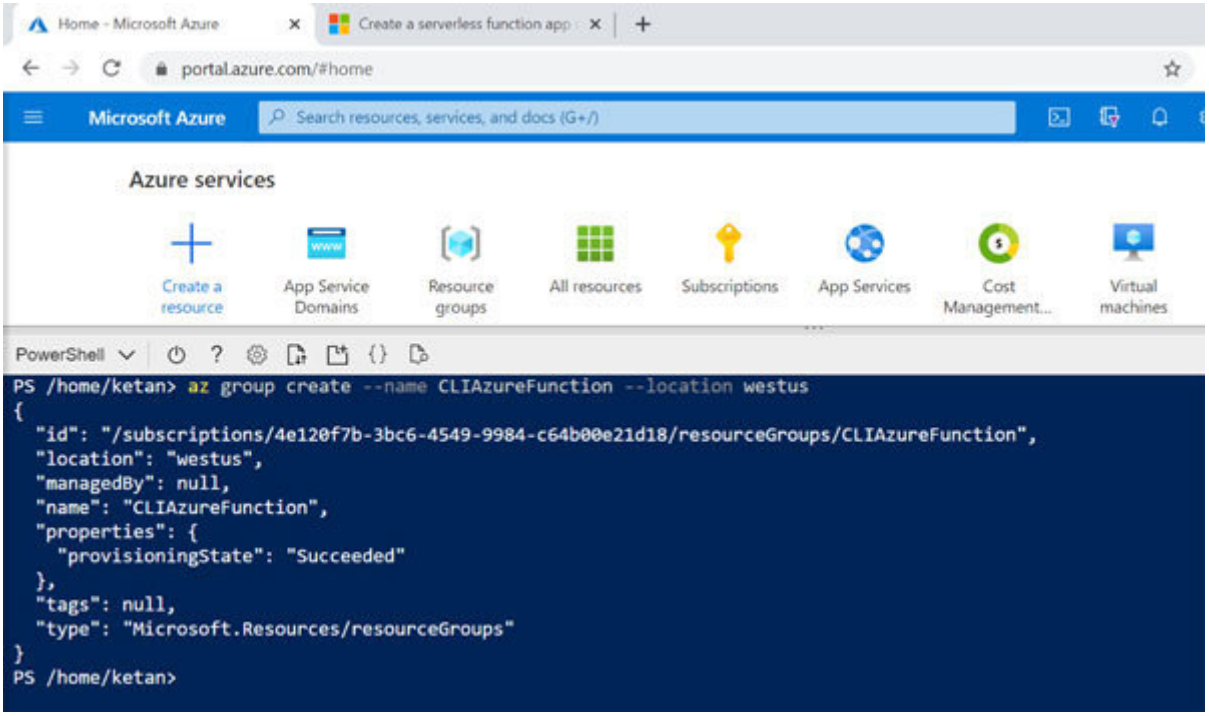


Figure 3.52: *Create a resource group*

Create an Azure storage account

Azure Cloud Shell needs Azure file share to persist files. If you do not have any storage mounted, then you must create a storage account. Please refer to *figure*

To create a storage account, you need to pass value to parameters like name, location, resource group, and SKU. Please refer to the following sample CLI command:

az storage account create --name azurefunction11082020 --location westus --resource-group CLIAzureFunction --sku Standard_LRS

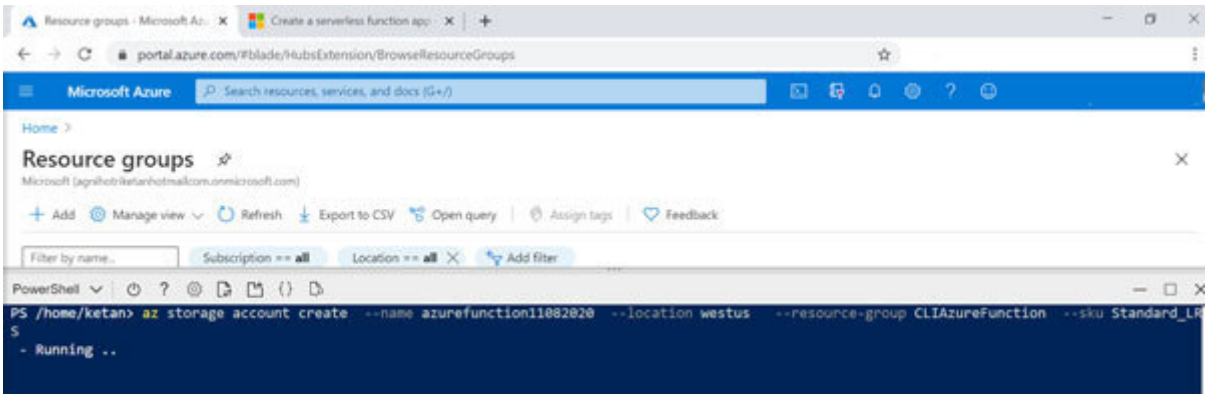


Figure 3.53: *Create a storage account to use a file share*

Create a serverless function app

You must pass parameters like name, storage-account name, consumption-plan location, resource-group, and function app version.

Please refer to the sample CLI command to create the function app. Please refer to *figure*

az functionapp create --name CLIAzureFunction11082020 --storage-account azurefunction11082020 --consumption-plan-location westus --resource-group CLIAzureFunction --functions-version 2

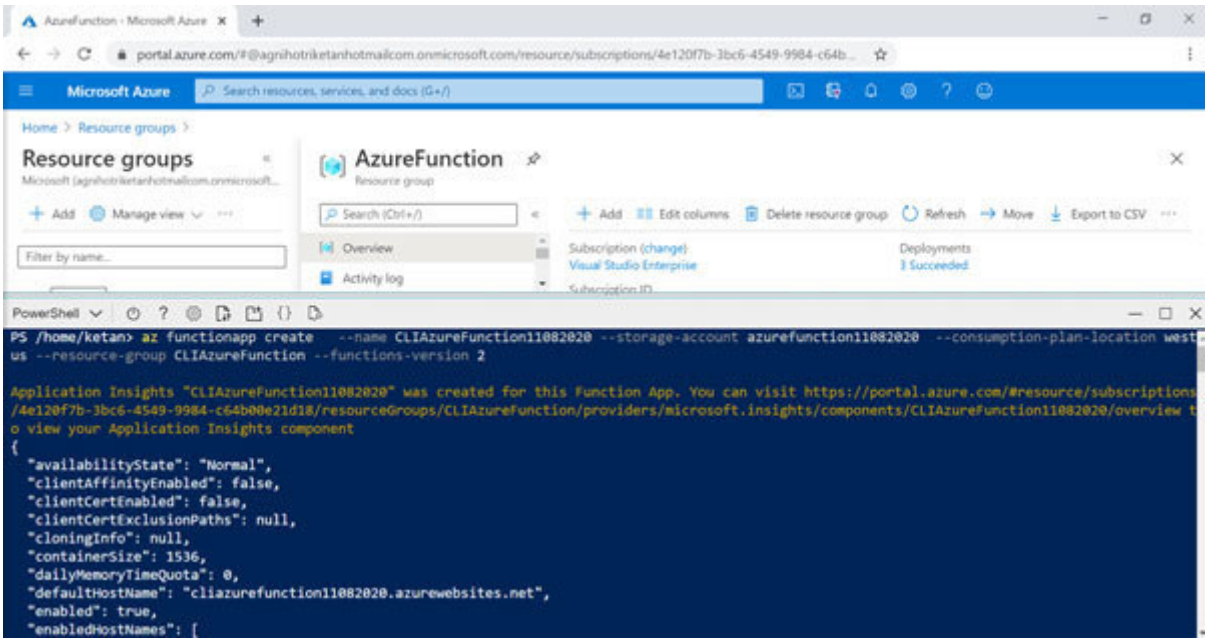


Figure 3.54: *Create a function app*

Deploy the function app to Azure

The last step is to deploy the local functions project to the function app in Azure using the following publish command:

func azure functionapp publish

You can use the Application Insights live metrics stream in a browser by executing the following CLI command:

func azure functionapp logstream --browser

Application Insights help the developer and DevOps team analyze business activities, monitor application state and application performance. It works well with a variety of platforms.

Creating a function using an ARM template

An ARM template is a JSON file that consists of resources to deploy and the properties for those resources. The template enables the deployment without the need to write the commands to create resources. The template has sections like parameters, variables, user-defined functions, resources, and outputs.

To create a function from an ARM template, you first must have Azure Functions created locally either in Visual Studio, Visual Studio Code, or CLI. The steps to create the same are discussed in previous sections of this chapter. You can get sample ARM templates by visiting

The resource template above will create four Azure resources as follows:

The Azure Functions require an Azure storage account

A consumption hosting plan for the function app

The function app itself

An Application Insights resource

To deploy the template and create resources, follow these steps:

Save the above template into the local drive as a file.

Open the Azure PowerShell. If you don't have it installed already, you may follow the instructions at https://docs.microsoft.com/en-us/powershell/azure/install-az-ps to get it installed.

You may also choose to follow the steps using the Azure portal PowerShell. The same can be accessed by clicking on the icon in the Azure portal top menu.

This will open the PowerShell on the portal at the bottom of the page, as shown in *figure* or open the Windows PowerShell and follow the next steps.

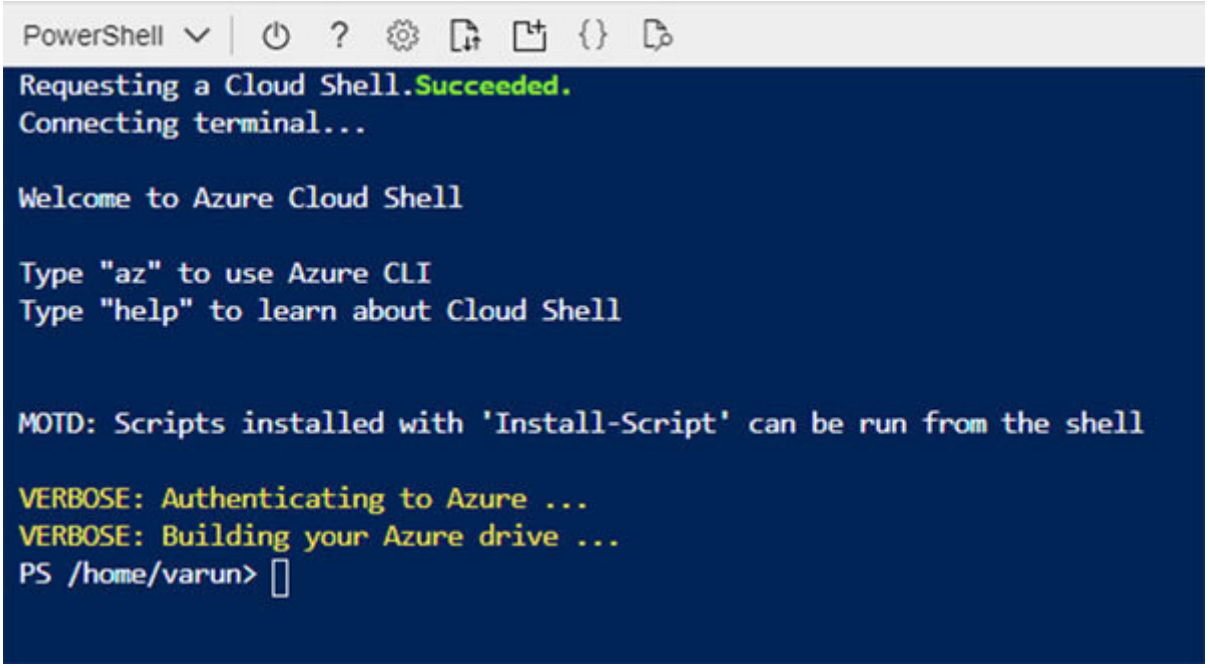


Figure 3.55: *PowerShell in the Azure portal*

Type the following command to sign in to Azure using your credentials, as shown in *figure*

Connect-AzAccount

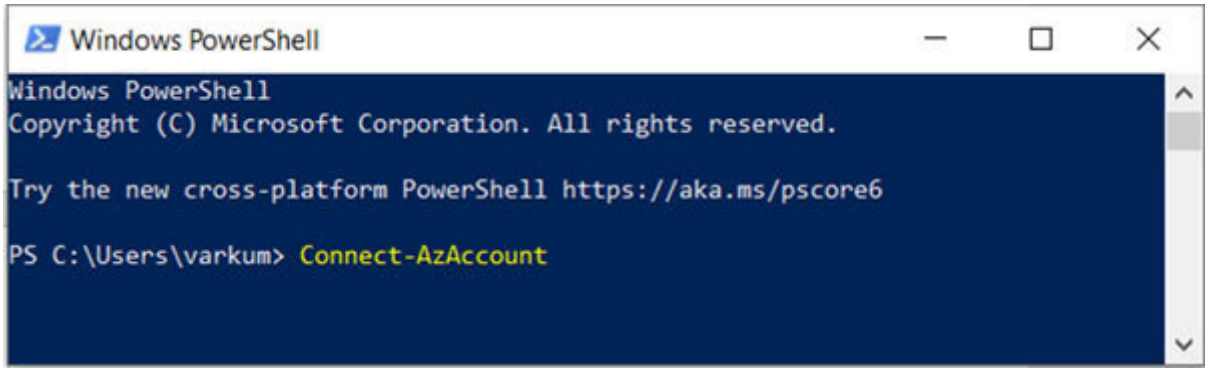


Figure 3.56: *PowerShell command to sign in to Azure*

This will open the Azure sign-in page, as shown in *figure* Enter your Azure credentials and click on on the next page.

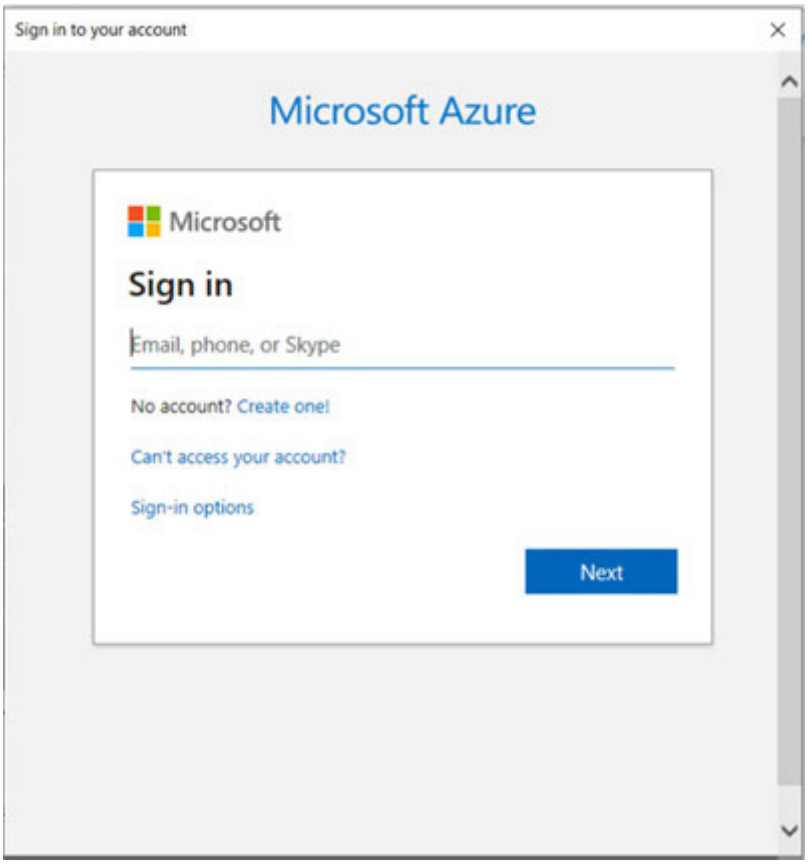


Figure 3.57: *The Azure sign-in page*

After successful sign in, enter the following commands on the PowerShell window. The example of the same is shown in *figure*

$resourceGroupName = “” <# Example: sample #>
$location = “" <# Example: eastus #>
$templateFile = "" <# Example: D:\template.json #>
New-AzResourceGroup -Name $resourceGroupName -Location "$location"

New-AzResourceGroupDeployment -ResourceGroupName $resourceGroupName -TemplateFile $templateFile

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\varkum> Connect-AzAccount

Account             SubscriptionName                    TenantId                             Environment
-------             ----------------                    --------                             -----------
wiservarun@live.com Visual Studio Enterprise Subscription 68d4dd9b-3a0e-43ae-b385-78f415dc1e37 AzureCloud


PS C:\Users\varkum> $resourceGroupName = "sample"
PS C:\Users\varkum> $location = "eastus"
PS C:\Users\varkum> $templateFile = "D:\template.json"
PS C:\Users\varkum> New-AzResourceGroup -Name $resourceGroupName -Location "$location"


ResourceGroupName : sample
Location          : eastus
ProvisioningState : Succeeded
Tags              :
ResourceId        : /subscriptions/da400927-72d7-4ace-9c06-c5796bc847b0/resourceGroups/sample



PS C:\Users\varkum> New-AzResourceGroupDeployment -ResourceGroupName $resourceGroupName -TemplateFile $templateFile


DeploymentName          : template
ResourceGroupName       : sample
ProvisioningState       : Succeeded
Timestamp               : 8/16/2020 2:14:34 PM
Mode                    : Incremental
TemplateLink            :
Parameters              :
                          Name                  Type                       Value
                          ====================  =========================  ==========
                          appName               String                     testfunctionappca1
                          storageAccountType    String                     Standard_LRS
                          location              String                     eastus
                          runtime               String                     dotnet
                          appInsightsName       String                     testfunctionappcaai1
```

Figure 3.58: *The PowerShell command to create the Azure resources for the function app*

Once the deployment is completed, you may verify the same by going to the **Settings → Deployments** section of the resource group in the Azure portal, as shown in *figure*

Figure 3.59: *Azure resource group deployment section*

You can also view the summary of the deployment by clicking on the template link. The deployment overview page created by the preceding steps is shown in *figure*

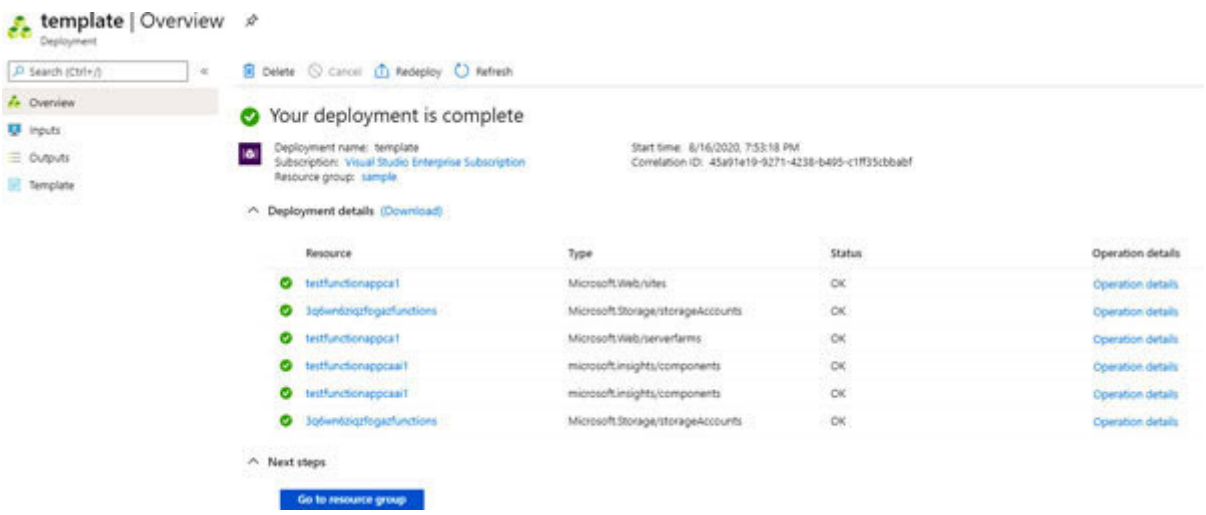


Figure 3.60: *Deployment overview page in the Azure portal*

Deleting the resource group

The function app created with the selected options creates multiple resources in the resource group, each with some cost. You may be billed for these resources based on your account and service pricing. Therefore, if you don't need these resources anymore, it is important to delete these resources. However, if you plan to use them further, you may choose not to delete those keeping the cost in mind.

You can delete all the resources in a resource group by deleting the resource group itself. You can select the resources individually or select all to delete the specific resources in the resource group, keeping the resource group. The following are the steps to delete the resources:

On the portal home page, click on **Resource group** and select the resource group you want to delete. You can also open the resource group from your function app by clicking on the **Resource group** name in the Overview section of the function app, as shown in *figure*

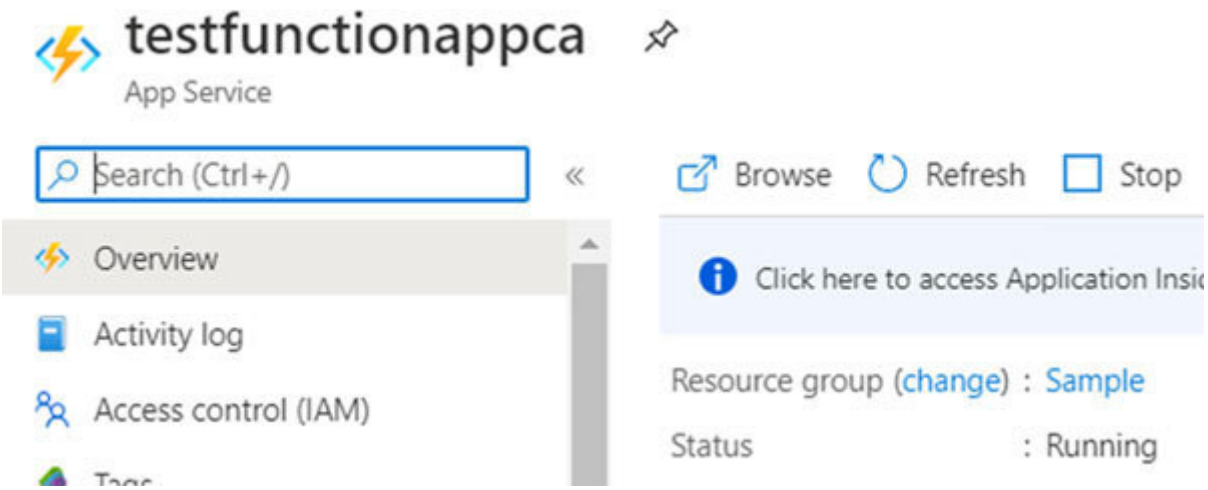


Figure 3.61: *The function app overview section*

On the Resource group page, review and verify the resource list for deletion. The resource group list is shown in *figure*

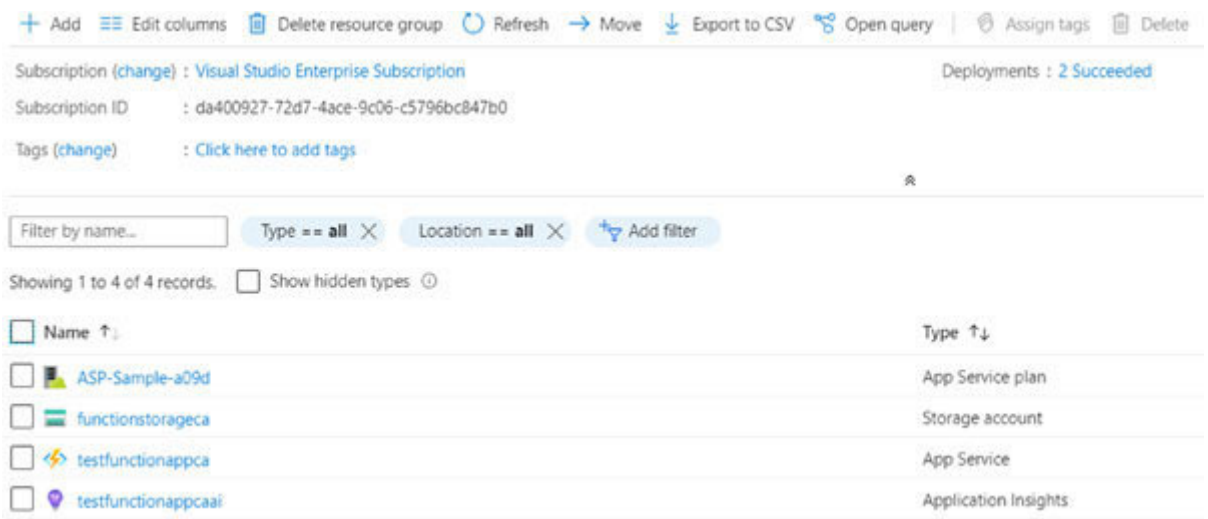


Figure 3.62: *Resource list on the resource group page*

Click on **Delete resource group** in the top menu or select specific resource names and hit

A new pane will open on the right to ask for confirmation. Review the list again, type the resource group name in the text box provided, and hit as shown in *figure*

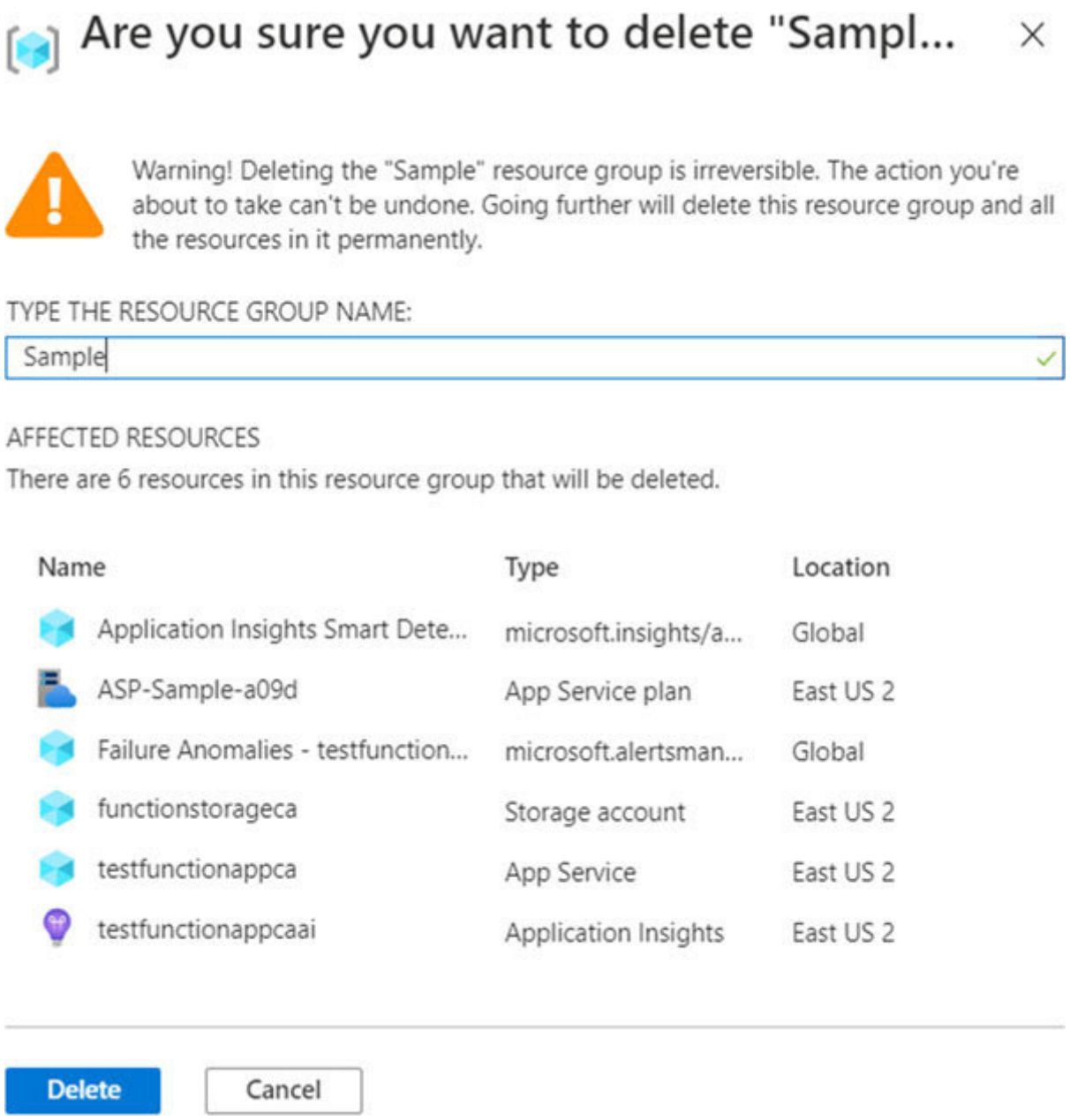


Figure 3.63: *Delete resource group confirmation pane*

It may take a couple of minutes to delete the resources. A notification will appear post deletion and can be viewed by

clicking on the notification icon at the top of the page. Please refer to the following ways of deleting the resource group:

You may also delete the Azure resource by passing the resource group name to the following CLI command:

az group delete --name CLIAzureFunction

To remove the Azure resource group and its resources from the current subscription using PowerShell, type the following command:

Remove-AzureRmResourceGroup

To delete a resource without deleting the resource group, use the following command:

Remove-AzureRmResource cmdlet.

Conclusion

Azure function app can be created with multiple options, including the Azure portal, Visual Studio, Visual Studio Code, Azure CLI, and ARM templates. Once the function app is created, you can create Azure Functions inside the function app using the pre-defined templates available to trigger a function on the HTTP request, Timer, through Service Bus Queue or topic, Azure Storage Blob, or Queue, and so on. You can also test your function through the + option on the Azure portal or debug locally if created using Visual Studio or Visual Studio Code.

You can easily deploy multiple resources using an ARM template and few commands. Remember to delete the resources when no longer required to save cost. In the next chapter, we will discuss various triggers and bindings available for the Azure functions.

Points to remember

Azure functions can be created through multiple options - Azure portal, Visual Studio, Visual Studio Code, Azure CLI, and ARM templates.

You can develop Azure Functions using multiple ways. But you should not change the function developed locally with a function developed from a portal in the same function app. Function created from the local machine should be managed locally only and not in the Azure portal.

By installing Visual Studio core tools and extensions, developers can make their life easy.

If you publish code to an existing function app, then the current content of that app will be overridden.

Application Insights Live Metrics Stream helps to analyze and monitor the deployed function app.

An ARM template can be used to define the infrastructure and configuration with declarative syntax. It can be used to duplicate the exact infrastructure and configurations to create another function app.

Azure Function can have one or more trigger specific individual functions, and all of them share the same local and hosting configurations.

CHAPTER 4

Azure Function Triggers and Bindings

In the previous chapter, we learned the different ways to create Azure Function like Azure Portal, Visual Studio, VS Code, **command-line interface** (**CLI**), and **Azure Resource Manager** (**ARM**) templates. In this chapter, we will see different triggers and binding options available to invoke Azure Functions and to connect Azure Functions to other resources without hard-coding access in the code. We will also learn how the binding data is passed and used in the functions.

Structure

In this chapter, we will cover the following topics:

Overview of triggers and bindings

Understanding triggers with a sample use case

Using Azure functions to convert a monolithic architecture into microservices

Azure Functions Integrations in the portal

Azure Blob storage triggers and bindings

HTTP triggers and bindings

Sending an email notification using SendGrid

Timer trigger and bindings

Objective

After studying this chapter, you should understand the concept of triggers and bindings for Azure Functions. You should also understand the concept of microservices and how to build those using Azure Functions. Finally, at the end of the chapter, you will learn the most common trigger and binding implementations.

Overview of triggers and bindings

A trigger is what invokes an Azure Function on a particular event. An Azure Function can have only one trigger. *Figure 4.1* shows an event triggering an Azure function:

Figure 4.1: *Event triggering an Azure function*

Triggers usually have an associated payload that is processed by the function. The various types of triggers supported by Azure Functions are as follows:

HTTP This trigger executes the function on an HTTP request. We had created this HTTP-triggered function in the previous chapter.

Blob This trigger executes the function when a Blob is updated, or a new blob is created in the monitored storage account.

Queue This trigger executes the function when a new message has arrived in the monitored Azure storage queue.

Event Hub This trigger executes the function when an event is received by the Azure Event Hub or IoT Hub.

Timer This trigger executes the function when on a particular time or defined interval of time.

Service Bus This trigger executes the function when a new message arrives in an Azure Service Bus queue or topic.

Cosmos DB This trigger executes the function when an insert or update happens in the Azure Cosmos DB collection.

Event Grid This trigger executes the function when an Event Grid topic receives a new event.

Kafka This trigger executes the function when a new event has arrived in an Event Grid topic.

RabbitMQ This trigger executes the function when a new message has arrived in a RabbitMQ queue.

Dapr This trigger executes the function when a new message is added to the platform-agnostic Dapr topic.

Note: The direction for the trigger is always "in."

Bindings are a way to connect other resources to the Azure Functions. Bindings can be defined as an input, output, or both.

Bindings provide the data to the function as the function parameter.

The various bindings supported by Azure Functions are shown in *figure*

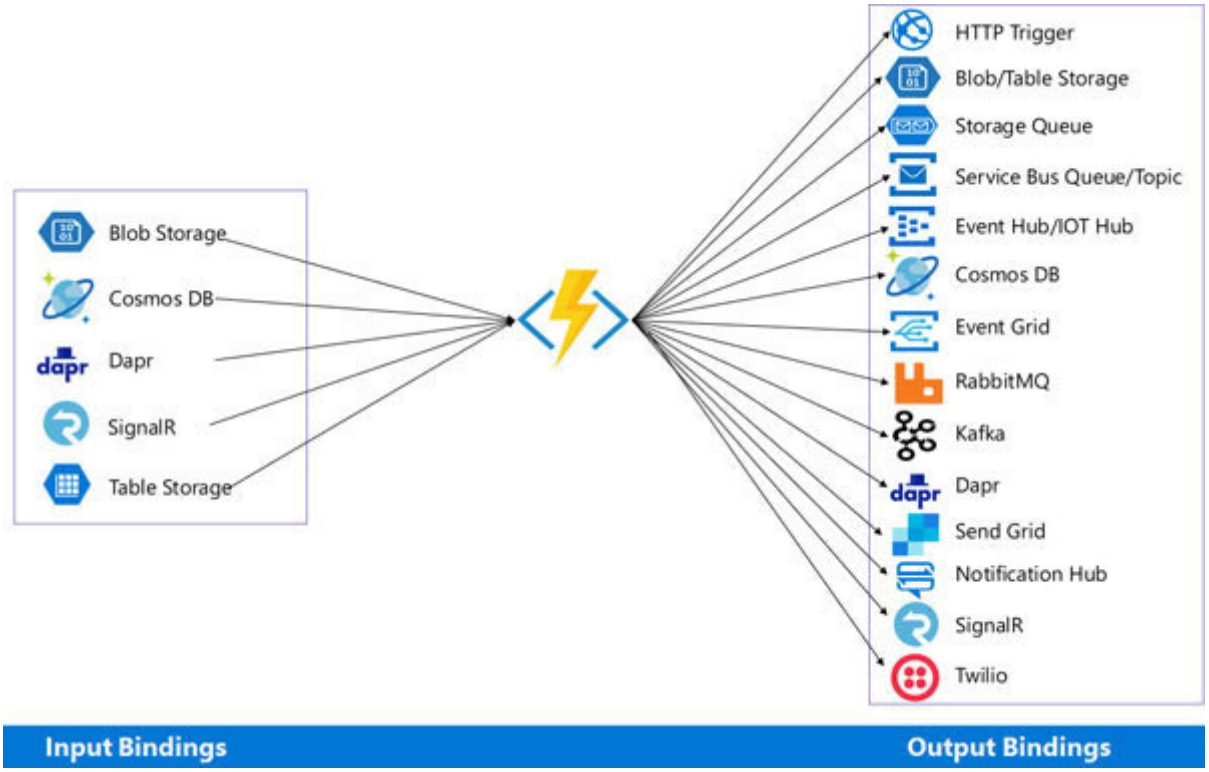


Figure 4.2: *Azure Functions supported bindings*

Note: Triggers and bindings for the Azure Functions are configured through the functions.json files so that you can avoid hard-coding values directly in the function code.

Understanding triggers with a sample use case

Mostly, web applications have a UI and a backend, which is generally an API. The API sends a communication to multiple other components such as database, backend jobs, other APIs, and so on. In the cloud scenario, this can be different databases, storage, queues, and so on.

Now let's understand the concept of triggers and bindings with a sample use case. We have a Web UI, and our API is an HTTP trigger Azure function that responds to the HTTP requests from the UI. Let's assume this in our use case. It's accepting an HTTP Post request with the request body and an image. The function is sending the body of the request to a Service Bus queue to be stored in DB and uploading the image to Azure Storage for further processing.

A Service Bus queue trigger function will get triggered when the message is added to the queue with the message as payload. The function will process this message store it into a Cosmos DB.

This will execute the third function, a Cosmos DB Trigger function, which processes data and sends the notification to the user through Twilio or SendGrid. There is also a Blob Trigger function to process the images stored in a Blob. *Figure 4.3* shows the high-level flow of the use case.

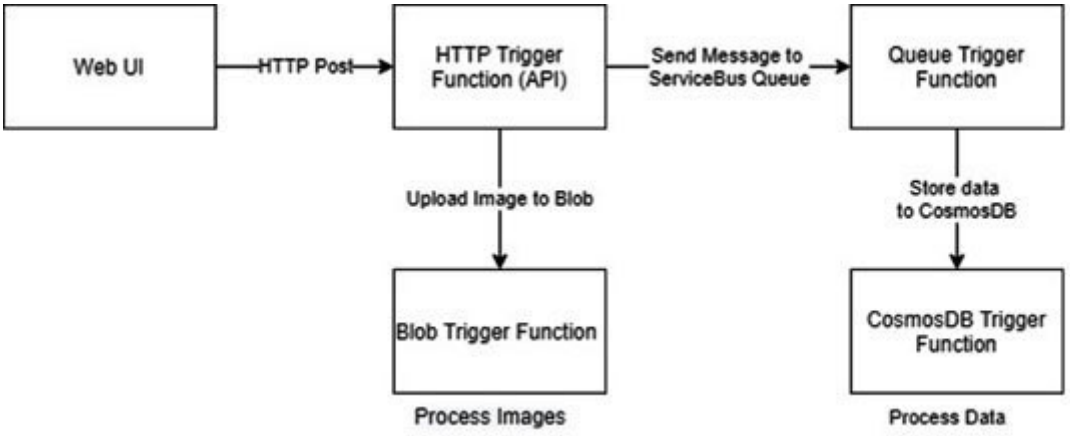


Figure 4.3: *High-level flow of the sample use case*

This is a hypothetical example. Your application may be having one or more of these functionalities that can be implemented with a lot of ease with the help of Azure function triggers. With the help of a rich set of triggers and bindings provided by Azure Functions, you can create highly scalable applications with less code and much lesser time. These powerful bindings allow you to integrate your application with a variety of other Azure services in just a few steps. You can even break down your big application components into smaller functions for faster deployment, granular scalability, and quick troubleshooting. One such example is discussed in the next section.

Using Azure Functions to convert a monolithic architecture into microservices

Microservices is the architecture style of designing a software application to collect smaller units of build and deployment. In a microservices architecture, the system is designed as modular as possible. Moreover, microservices are distributed, loosely coupled, and can be written in different programming languages.

Let's discuss a real-world example for a website to sell SSL Certificates. The website is very simple, with a button integrated with a third-party payment provider.

So, a web server responds to web requests with simple HTML, CSS, and JavaScript. It will have a web API method to handle the callbacks from the payment provider. The web server sends messages to a queue for any new request. There is another component to listen to this queue, generate an encrypted certificate, and send it over the email to the customer. Then, there are more components to let the user report an error, which calls a web API, another API to check if the certificate is valid. Finally, a nightly job to generate daily reports and save them somewhere. *Figure 4.4* shows the architecture of the application.

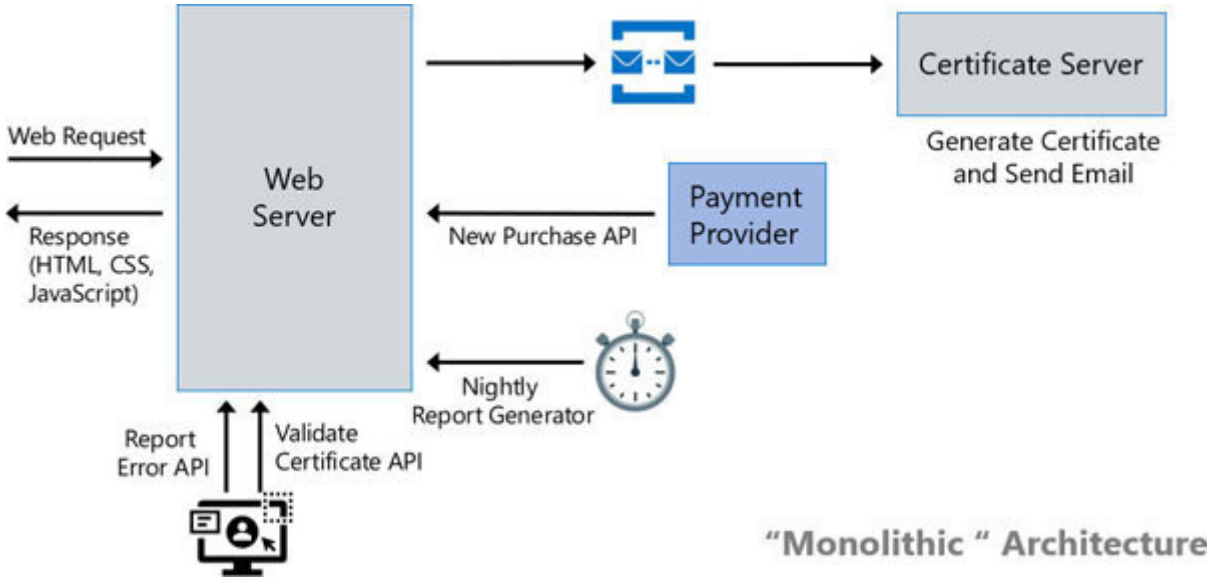


Figure 4.4: *Architecture of the use case*

If you see, we started with a simple website growing into a full-blown application with multiple responsibilities. This is a story of many applications that starts small with one server. But before you know it, one server is doing everything and ends up with a monolithic architecture that's very hard to scale. Moreover, monolithic applications become a hindrance to agile development and cause deployment issues. Any change required in the codebase required the entire solution not to be deployed. This makes DevOps quite complex and slow, and the time to production increases rapidly.

Now, let's see how Azure Functions could help us in this scenario. Functions let us break our monolithic application into smaller, scalable components. We start with a new purchase web-hook handled by an HTTP trigger Azure function that posts a message onto a queue. This message is read and processed by another Azure function that generates the license and puts it into

blob storage, which triggers another Azure function to email it to the customer. The error reporting API is now an HTTP-triggered Azure Function that writes a row into Azure Table Storage. Another HTTP trigger function for the Validate License API performs a database lookup of that license. For the nightly report, a timer triggered function. Figure 4.5 shows the breakdown of the application using Azure Functions.

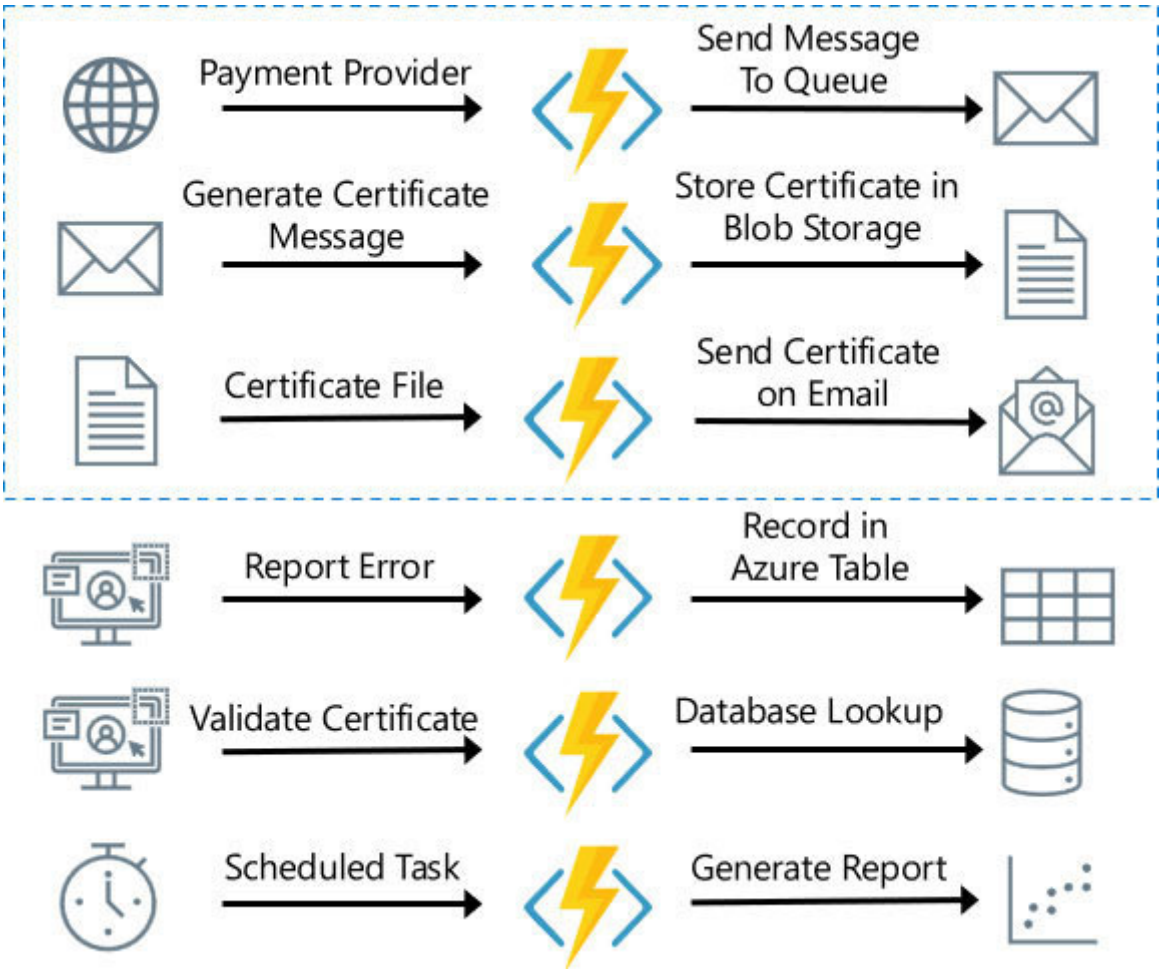


Figure 4.5: *Breakdown of an application using Azure functions*

Our web server can now be replaced with static content hosted in blob storage or a smaller web server.

With this, we have managed to decompose our complex monolith application into a set of loosely coupled functions. You could even think of them as microservices taken to the next level with the freedom of deploying and scaling independently. This makes the DevOps much less time-consuming.

Let's discuss some of the benefits of the microservices architecture we have created for our scenario:

The small and independent functions are easy to create, understand, and maintain

Each function can be scaled independently based on the application load

Each function can be deployed and tested individually based on the changes

Multiple teams can work together at the same time on different functions

You can practice Polyglot programming, writing code in multiple languages to take advantage of functionality not available in a particular language

However, as with everything, there are few disadvantages as well with the preceding design:

With multiple functions, we need to maintain more codebases

If the common code is not planned properly, such as sharing through nuggets, it causes either duplicate code or stepping back to monolithic architecture

Deployment pipelines need to be set up for each function individually to take advantage of faster deployments, and if they are not created properly, it becomes difficult to maintain

More intense test plans need to be prepared to cover multiple functions based on the languages they are written in

Azure Functions Integrations in the portal

Please follow these steps to create bindings for the function in the Azure portal:

Click on **Integration** in your HTTP-triggered function (created in the previous chapter). You will find the option in the left menu of your function, as shown in *figure*

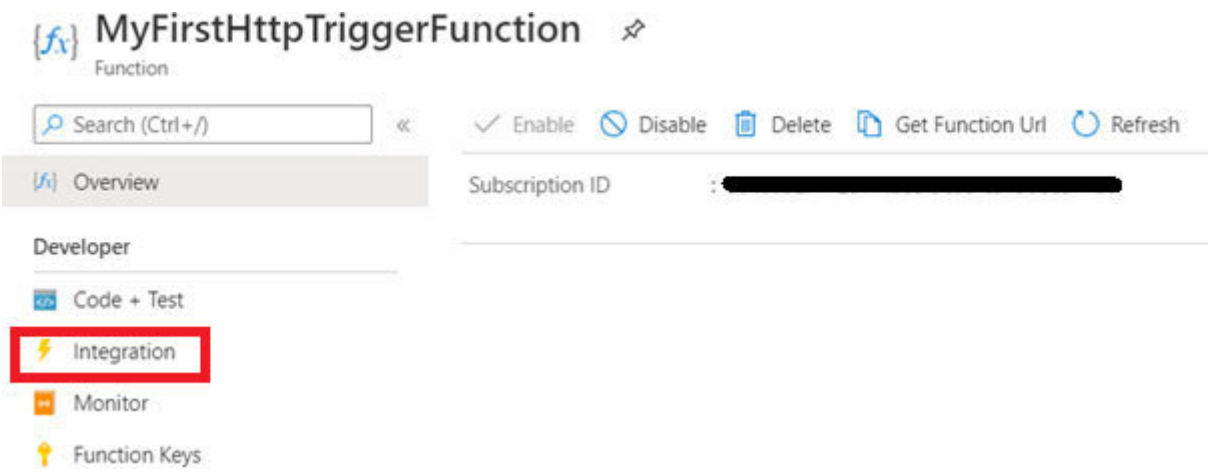


Figure 4.6: *Azure Functions left menu*

This will open the **Integration** page with default triggers and bindings, as shown in *figure*

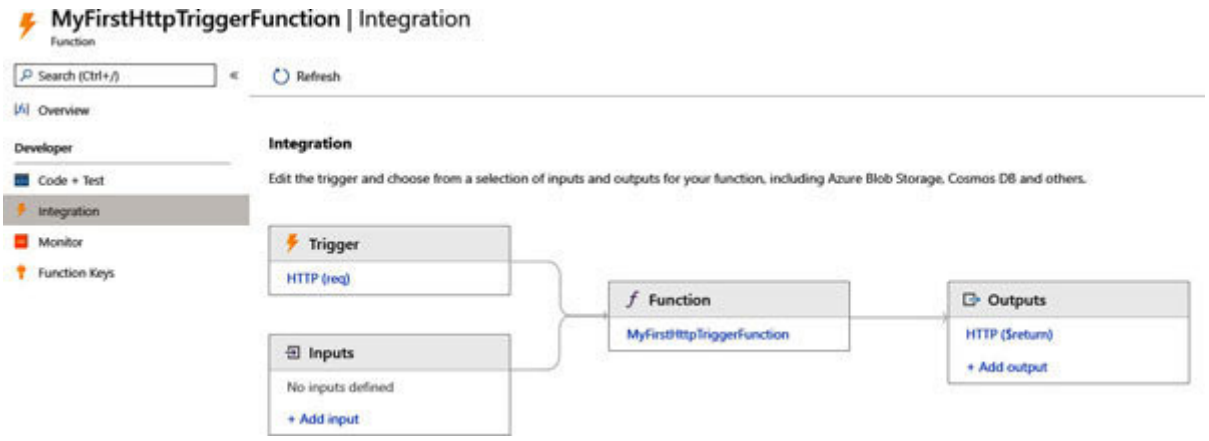


Figure 4.7: *Azure Functions Integration page*

Click on **Add Input** or/and **Add output** to add triggers and bindings from the portal.

This will open a pane on the right, as shown in *figure 4.8* and *figure*

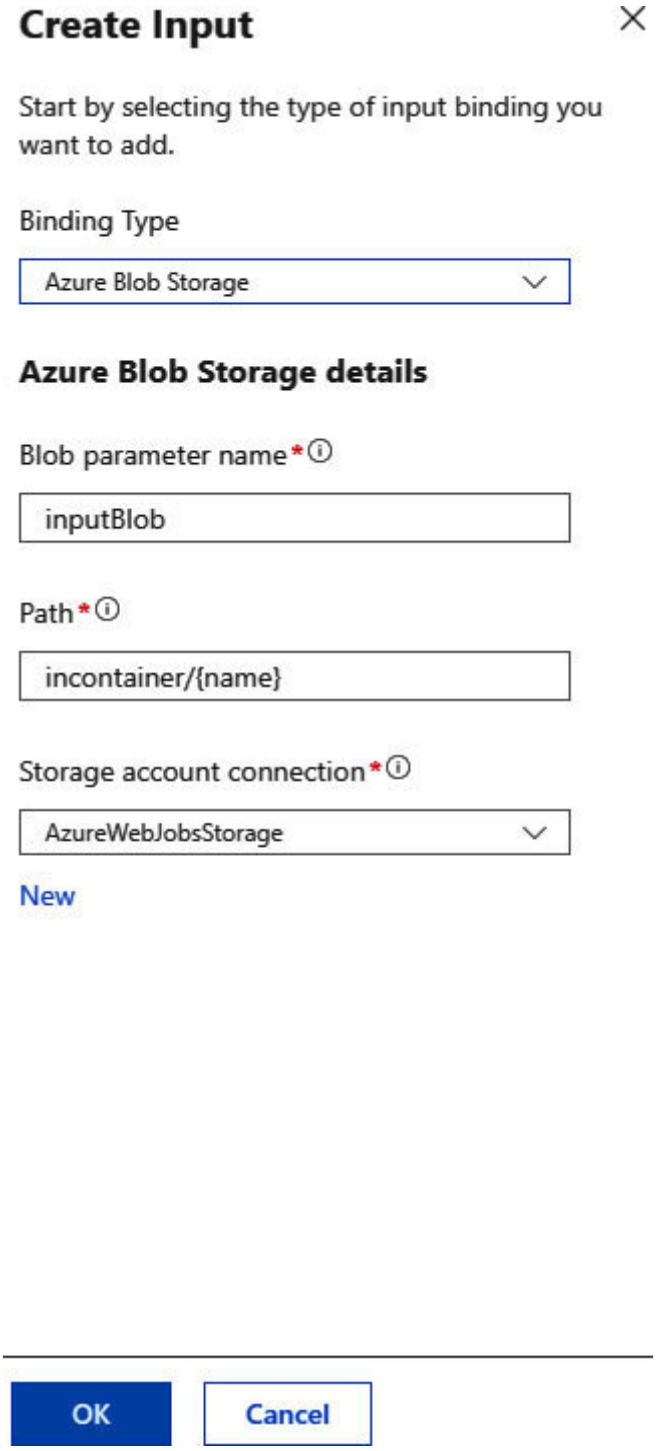


Figure 4.8: *Azure Function Input Binding*

Figure 4.9: *Azure Function Output Binding*

Select **Binding** provide other required details, and click

HTTP triggers and bindings

An HTTP-triggered function gets invoked through an HTTP request. It can be used to build serverless APIs or mobile backends, and so on. The process of creating an HTTP-triggered function was discussed in the previous chapter.

The HTTP Trigger function uses the **HttpTrigger** attribute to configure the function. The attribute takes the authorization level and the allowed HTTP methods as the constructor parameters.

Configuring the custom HTTP endpoint for your function

The default endpoint for an HTTP trigger function is in the following form:

http://.azurewebsites.net/api/?

However, you can customize your function route using the optional route property in the **functions.json** file. For example, the following **function.json** file defines a route property for an HTTP trigger configure the function route to **http://.azurewebsites.net/api/cloudavenue/technical/101** instead of the default route above:

```
{

[

    {
```

```
      {

"out"

      }

   ]

}
```

This configuration allows two parameters in the URL department and id. You can then change your function code to read and make use of these parameters.

Here's an example:

```
IActionResult req, string department, id, ILogger log)

{

var message = ID:

return OkObjectResult(message);
```

}

Azure Blob storage triggers and bindings

Integrating with Blob storage allows you to build functions that react to changes in blob data as well as read and write values.

A Blob storage triggered function gets invoked when a blob is created or updated in the Azure Blob storage. The content of the blob is passed as an input to the function.

Note: For a Blob storage triggered function, you should have general-purpose or V2 storage accounts with hierarchal namespaces.

The following is the example of a default Blob trigger function created through the Visual Studio that writes a log when a blob is added or updated in the **samples-workitems** container:

```
Connection = Stream myBlob, string name, ILogger log)

        {

Blob trigger function Processed blob\n

        }
```

In the function, the {name} part of the first parameter of the BlobTrigger attribute provides the file name of the triggering blob that can be used in the code.

The Blob Trigger function uses the BlobTrigger attribute to configure the function. The attribute takes the container path, connection, and optional blob name pattern as the constructor parameters.

You can also add the **StorageAccountAttribute** that takes name of the app setting of the storage account as the constructor parameter, as shown in following function code:

```
{

    ....

}
```

Azure Functions Blob input and output bindings

With the help of the Blob input bindings, you can access the blob data in the function with other triggers. This content will be passed as the parameter to the function by adding the following BlobAttribute:

Stream myBlob

The Blob output binding allows the creation, modification, and deletion of the blob using the same BlobAttribute but the **FileAccess** parameter as as shown in the following:

Stream myBlob

Sending an email notification using SendGrid

Sending notifications to system users is the crucial component of any software system nowadays. SendGrid is a widely used third-party service to send emails to users for various purposes like newsletters, invite users, reset passwords, business, promotional notifications, etc. SendGrid is a popular cloud-based tools-compliant tool that ensures email sending is simplified, decoupled, secure, and highly scalable. SendGrid is a crucial tool while working in restricted environments.

Configuring SendGrid using the Azure portal

To configure SendGrid, the first step is to get SendGrid's unique API Key. You can leverage your existing Azure account to get a free SendGrid account.

Visit **https://portal.azure.com/#home** and search for SendGrid in the Azure marketplace. To create a SendGrid account, you need to provide account details and select the pricing tier as per your need. Please refer to *figure*

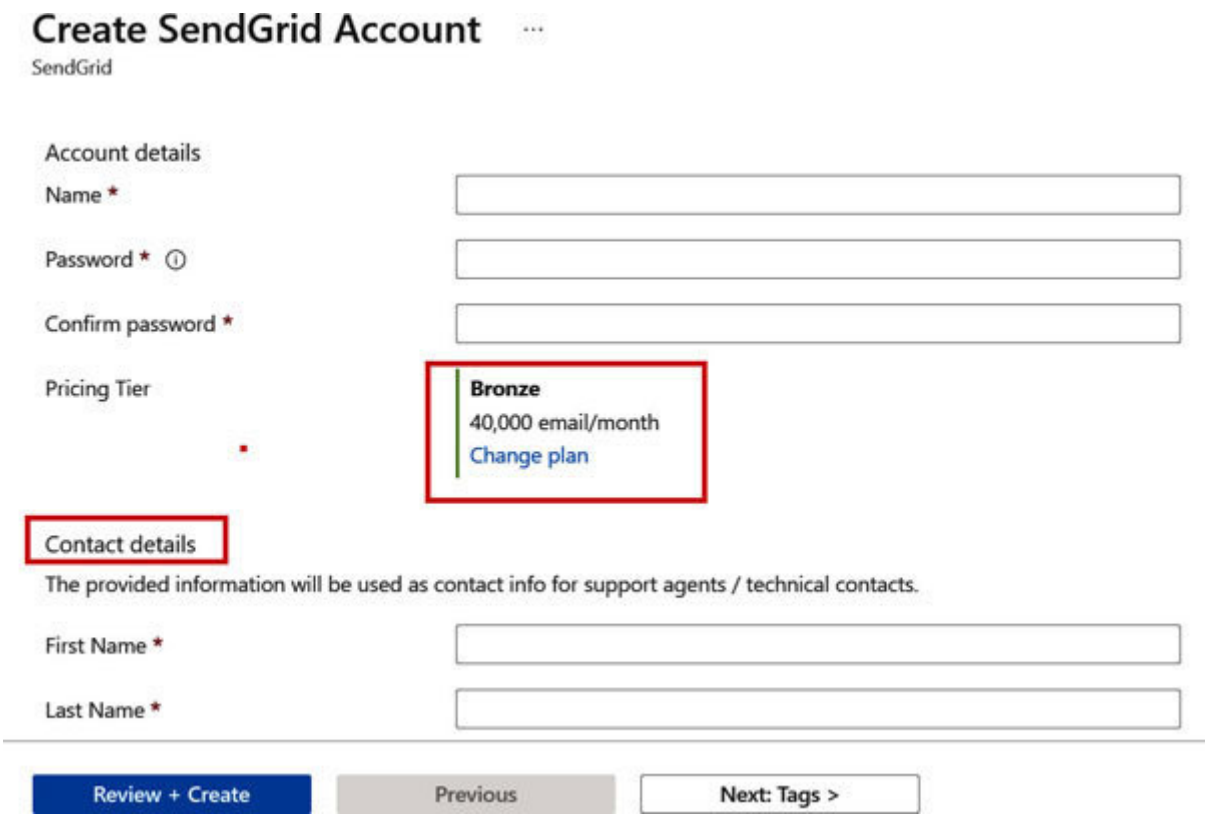


Figure 4.10: *Create SendGrid account using Azure*

Now, you need to visit your SendGrid account once provisioned by clicking It will redirect you to Please refer to *figure*

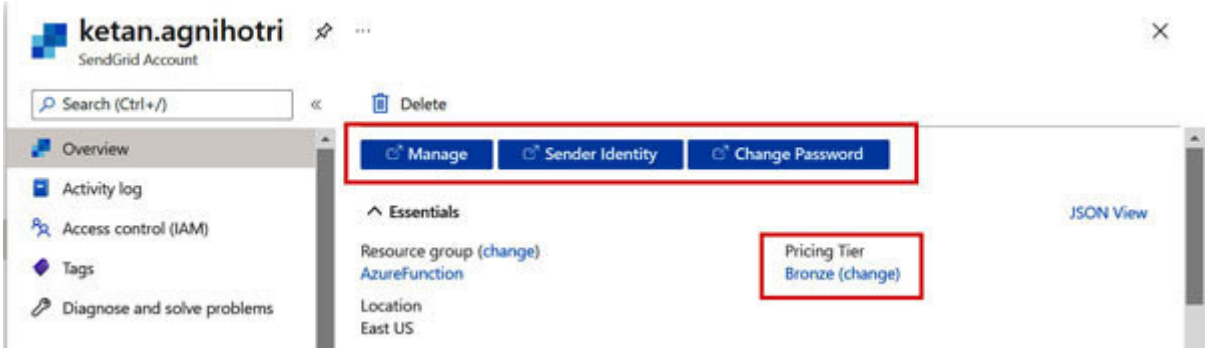


Figure 4.11: *Manage SendGrid account*

From the SendGrid portal, you need to navigate the setting section and click on **Create API** Please refer to *figure*

Figure 4.12: *Create SendGrid API key*

You need to provide key name and API key permission from the available list. Click on **Create &** Please refer to *figure*

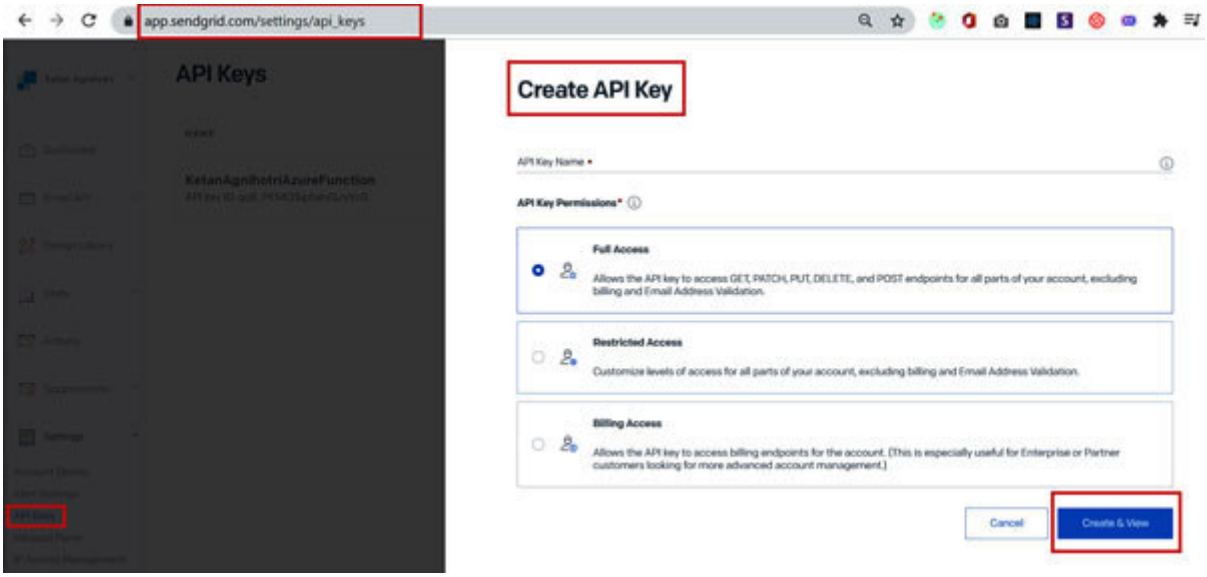


Figure 4.13: *Create and view SendGrid API key*

Please copy this API key somewhere for your reference, as this will not be visible again. Please refer to *figure*

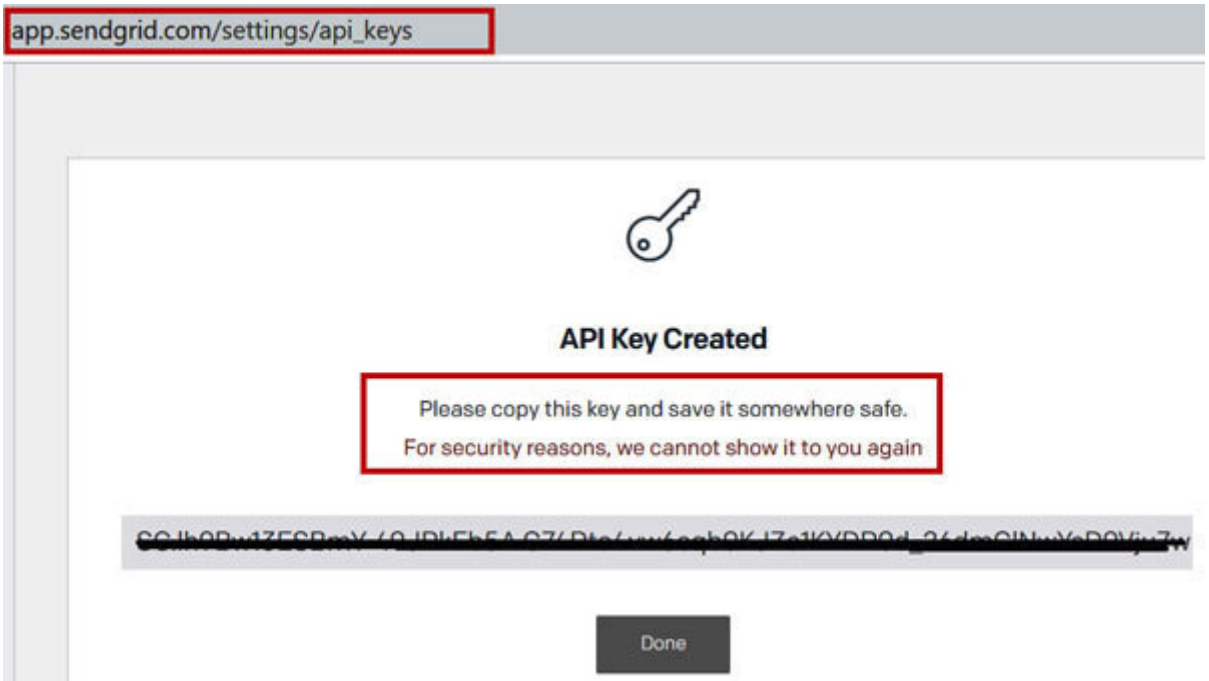


Figure 4.14: *Copy SendGrid API key for your reference*

Configure Function App

As you read about all available options to create the Azure function app service in Chapter 3: Creating your first Azure you can choose an option to create the function app. You will have a storage account to store the function code.

Create an function app and add a SendGrid API key value (copied API key) pair to App Settings, and save the settings. Please refer to figure

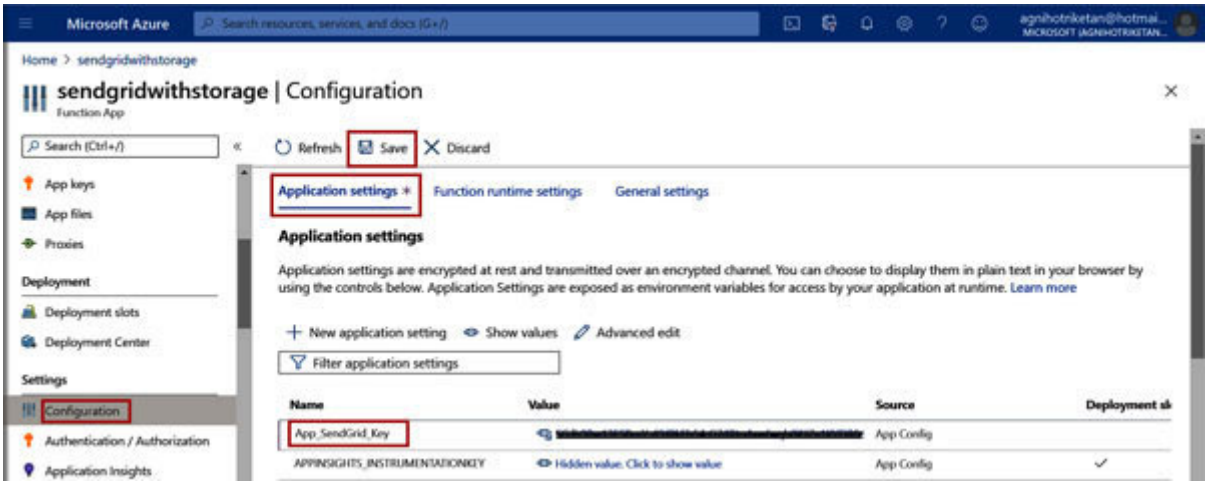


Figure 4.15: *Add SendGrid API key in application settings*

There are multiple ways to use Azure Functions for sending an email notification. There is already a pre-configured SendGrid function template available in the Azure portal as well. You can send emails based on any event like message arrival in a queue,

blob update using triggers like HttpTrigger, blobTrigger, ServiceBusTrigger, and in/out bindings.

Let's understand email sending using a grid with storage queue binding and queue trigger. Now, you need to write two functions, and in the language of your choice with a unique name and authorization level.

Implementing 'AddMessage' function

Perform the following steps:

Navigate to the **Functions** menu and click on **Add** new function.

Select HTTP trigger template, and provide function name (for example, and authorization level. Please refer to *figure*

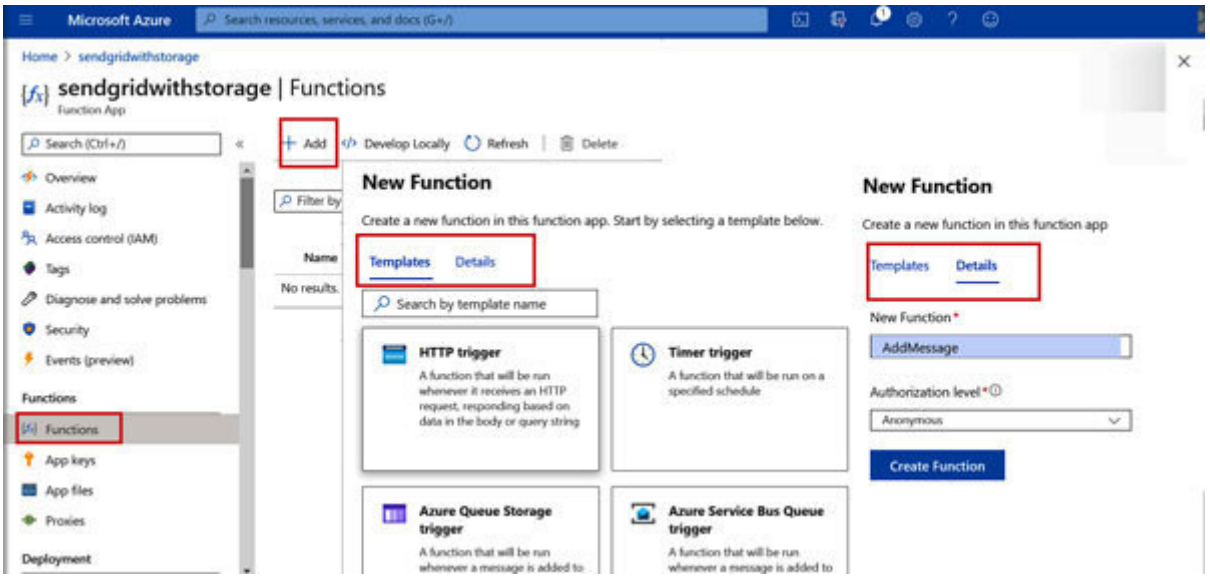


Figure 4.16: *Create AddMessage to queue storage using HTTP Trigger*

Now, the next step is to add a storage queue binding to the HTTP trigger function. Navigate to **Integration** and click on **Add**

At the time of adding queue output, select binding type as Azure storage queue. Then, you must keep a note of the message

parameter name in which you will add the message with the function code and provide a queue name. Please refer to figure

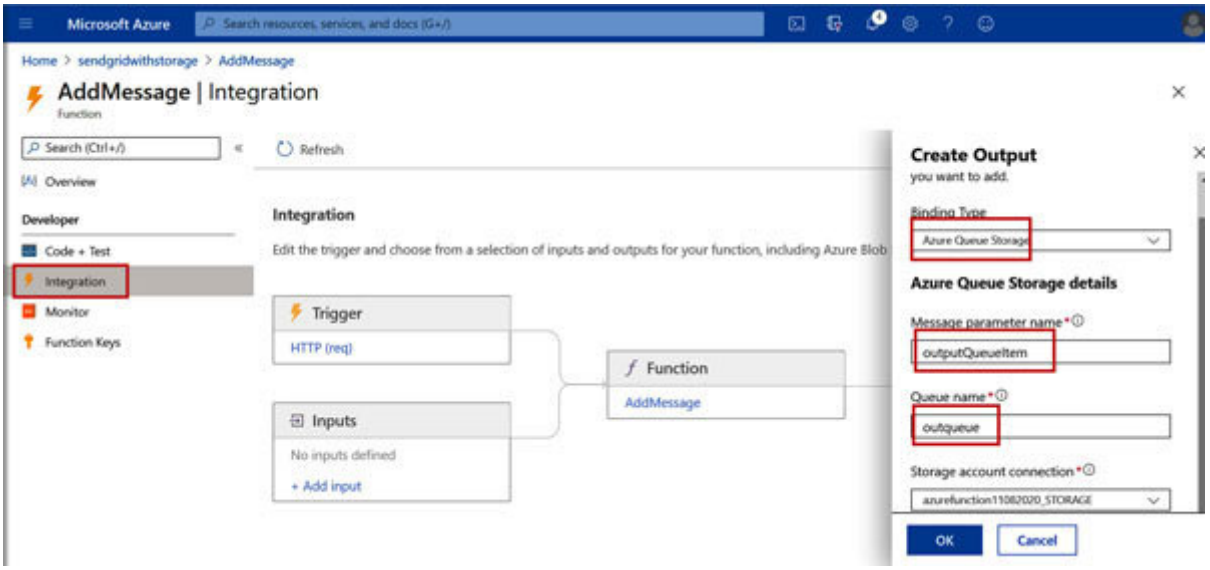


Figure 4.17: *Add queue binding details*

After adding queue as output, your HTTP trigger will add a message to a given connection string to queue storage. Please refer to figure

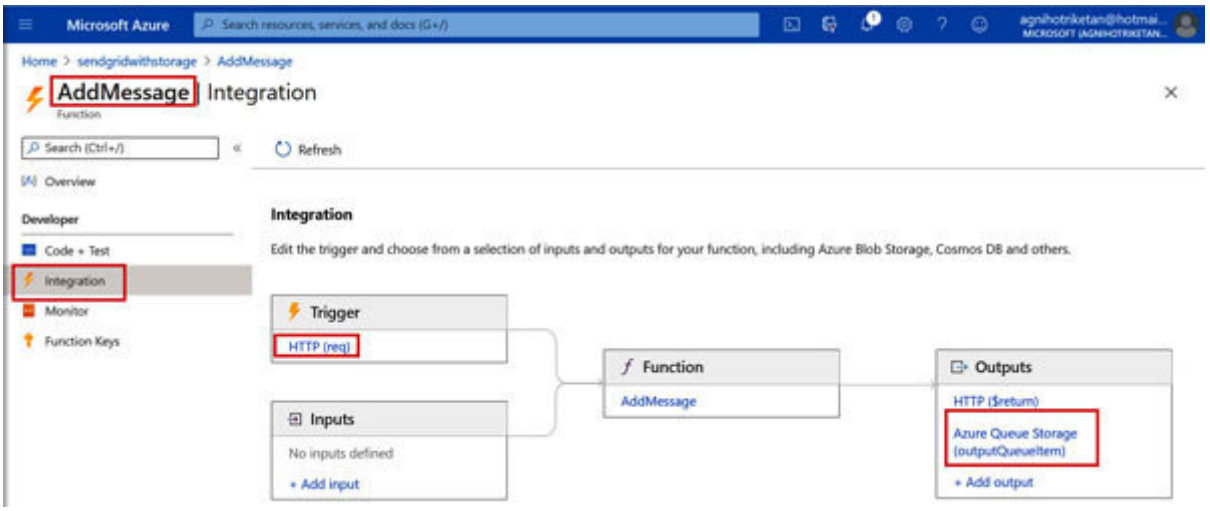


Figure 4.18: *Set output of the function*

To add a message to a queue, you need to write a line of code, Please refer to *figure* You can check the bindings by selecting

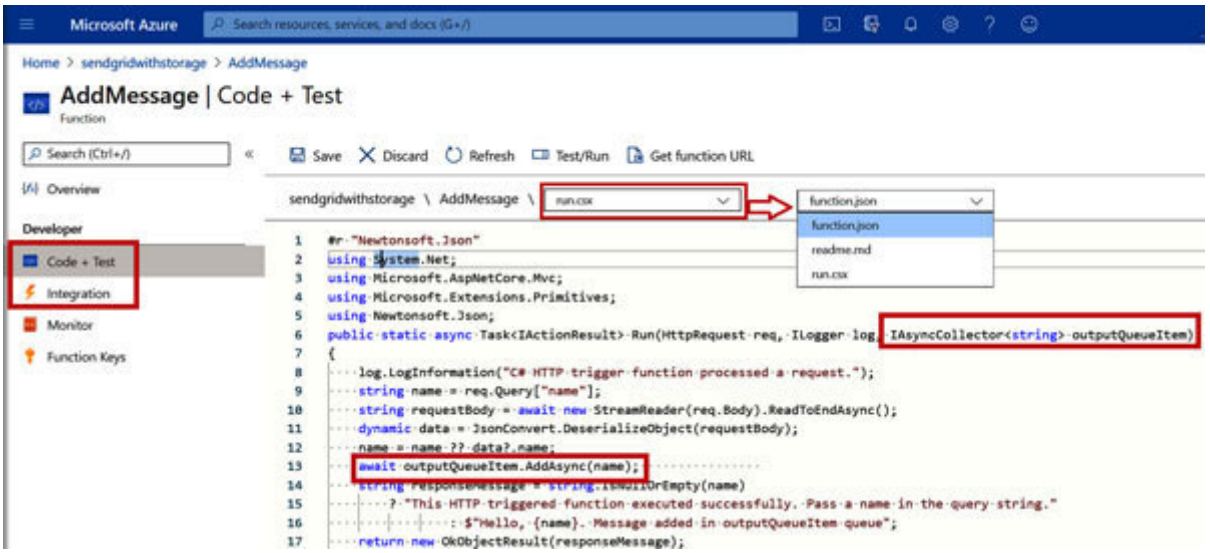


Figure 4.19: *Add email string as a message in the storage queue*

To test the configuration you made, you must copy the HTTP trigger function URL and run it in a browser. Please refer to *figure*

Figure 4.20: *Execute AddMessage function using URL and pass email id in a query string*

You can check whether the message got added into a queue using Azure portal or Azure storage explorer (that is, queue name provided while adding binding to queue storage). You must install

the free Azure Storage Explorer tool from This tool helps you to manage your Azure storage resources from any platform easily. Please refer to *figure*

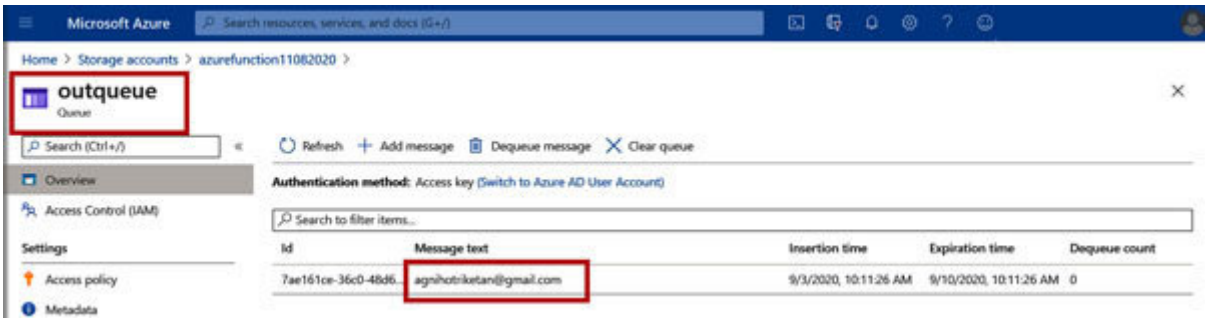


Figure 4.21: *Message added to the queue will be picked by the SendEmail function*

Implementing 'SendEmail' function

After adding an email address as a message to the queue, you need to write another function in the Azure function app. That function will be triggered on receiving a message in a queue. This function will pick this message (email address) and send an email with the subject, message, and so on:

Navigate to function from the left menu and new function of type Azure queue storage trigger. Please refer to *figure*

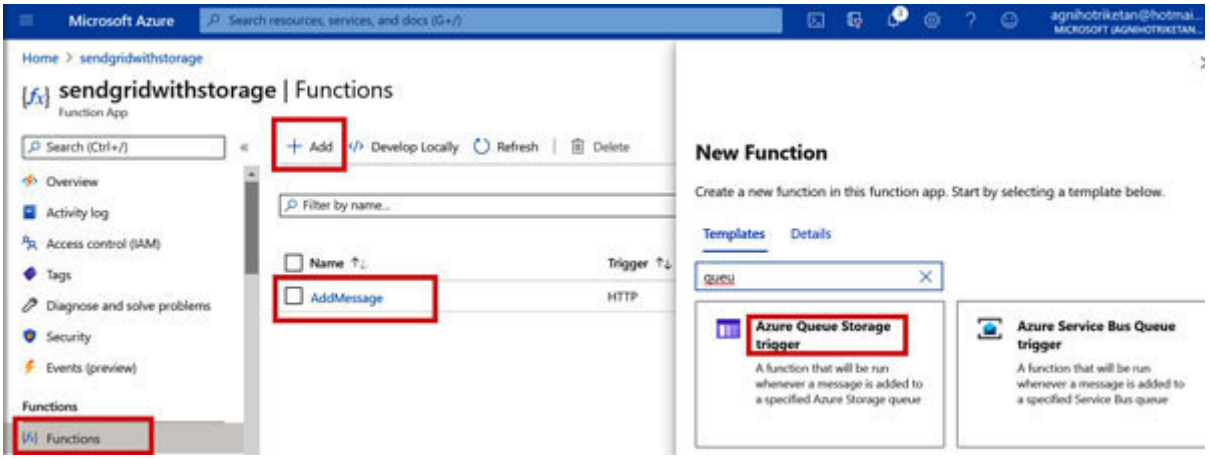


Figure 4.22: *Choose queue storage trigger template*

To create the **SendEmail** function, provide Azure function name, queue name, and storage account connection. Please refer to *figure*

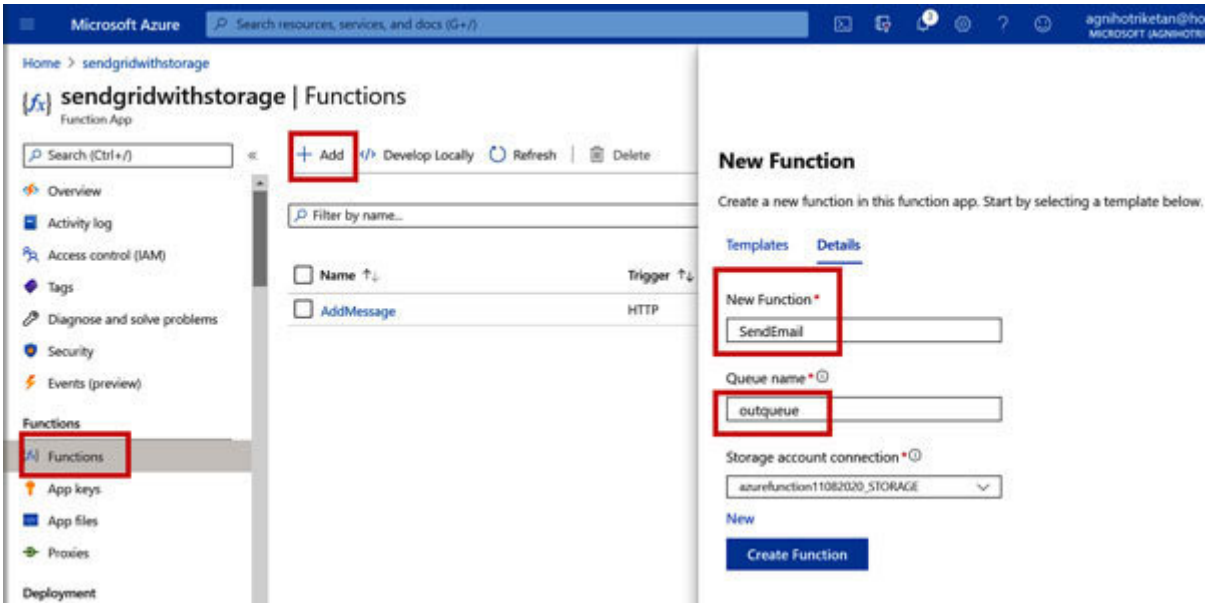


Figure 4.23: *Create SendEmail function*

After creating the function, you need to add SendGrid type output binding. You must provide binding type, SendGrid API key, message parameter name, and email information like to address, from address, email subject, and email message. Please refer to *figure*

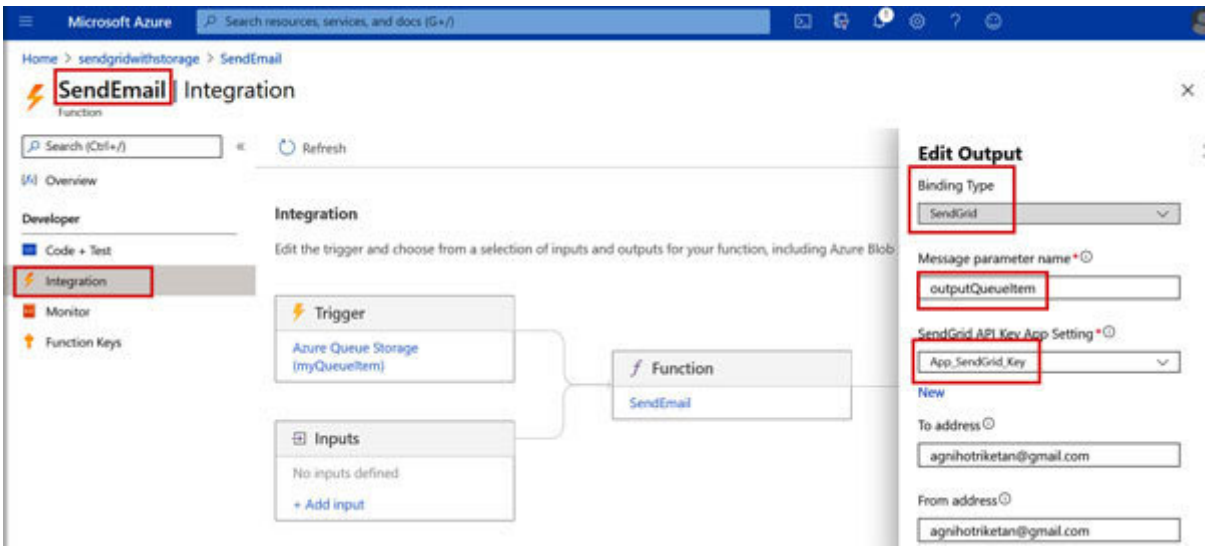


Figure 4.24: *Add SendGrid binding and pass SendGrid API key from application settings*

If you have created a function with available templates, modify the function code, add a reference to SendGrid library, and function You must define an email message object with members like to, from, email subject, and email body by adding source code in the **run.csx** file. Please refer to *figure*

Save Discard Refresh Test/Run

sendgridwithstorage SendEmail run.csx

```
#r "SendGrid"
using System;
using SendGrid.Helpers.Mail;

public static void Run(string myQueueItem, out SendGridMessage outputQueueItem, ILogger log)
{
    outputQueueItem = new SendGridMessage();
    outputQueueItem.SetSubject("Your message received in Queue storage..!!");
    outputQueueItem.SetFrom("noreply@k10.in");
    outputQueueItem.AddTo(myQueueItem,"Ketan Agnihotri ");
    outputQueueItem.AddContent("text/html", "Your message received in Queue storage..!! ");
    log.LogInformation($"myQueueItem: {myQueueItem}");
    log.LogInformation($"outputQueueItem: {outputQueueItem}");
}
```

Figure 4.25: *A set email message body*

To understand function triggers, input, and output visually, you may navigate the integration menu from the left side, as highlighted in *figure*

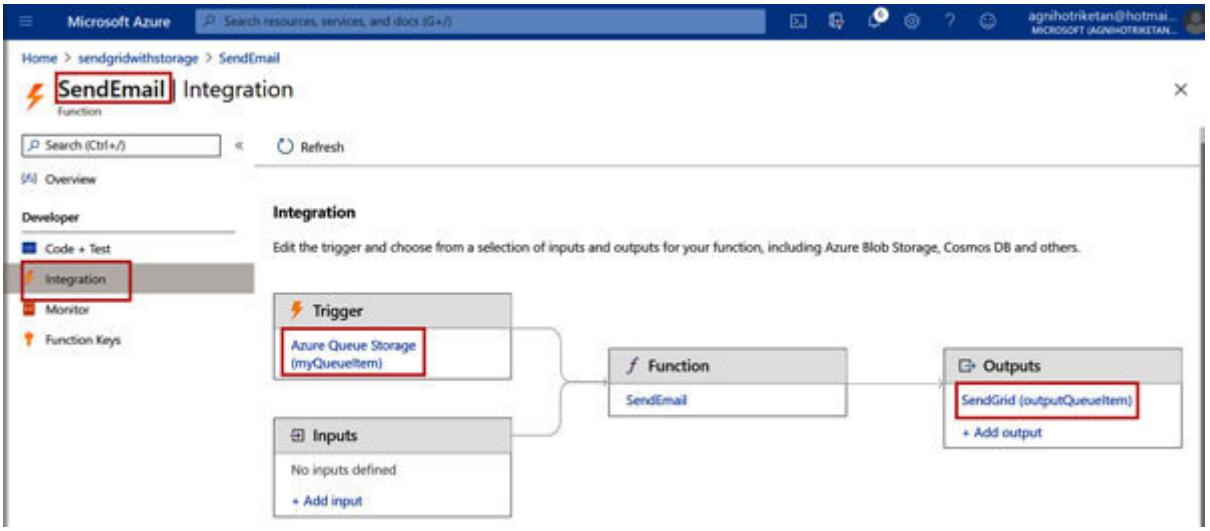


Figure 4.26: *Visual representation in the integration section*

You can also use visual studio code to check the code of the function created on the portal. You need to use the Azure extension to sign in and check the resources to browse function bindings.

You can click on the function name and understand the binding details in the JSON file. Please refer to *figure*

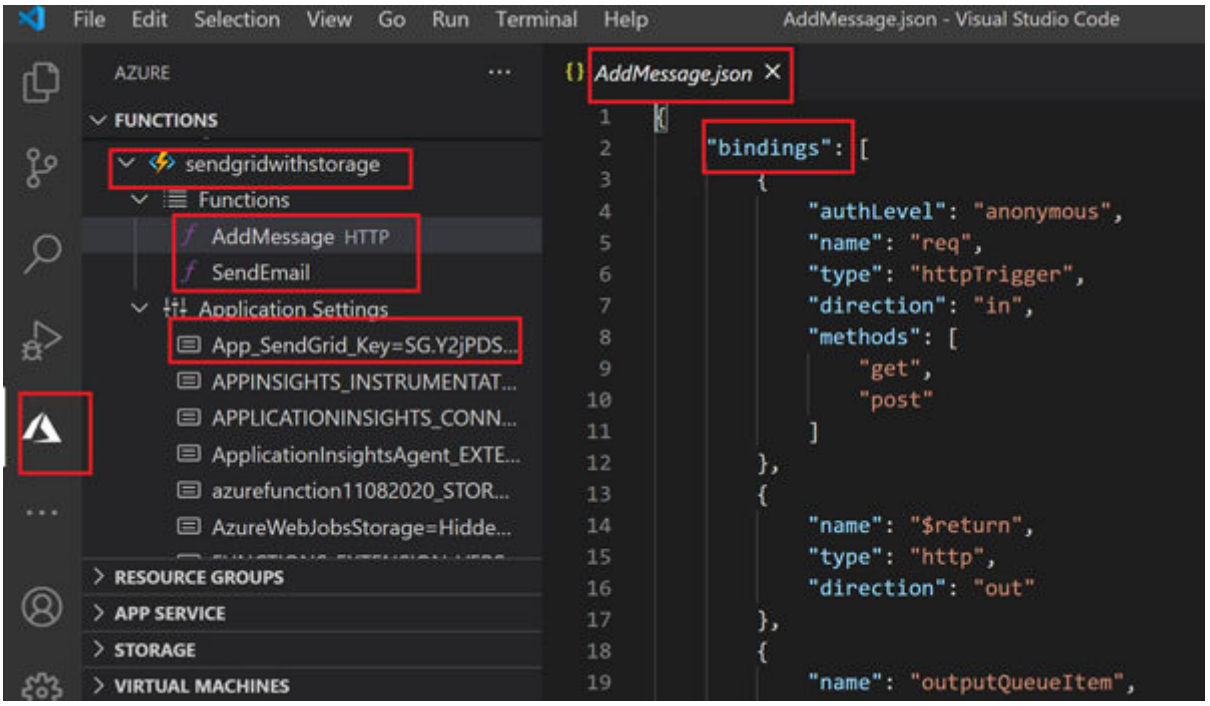


Figure 4.27: *Check created bindings for AddMessage Function using visual studio code*

The following is how the **function.json** file looks like for the function:

```
{
[
        {
[
"post"
            ]
        },
```

{

"out"

},

{

"azurefunction11082020_STORAGE"

}

]

}

To check binding details for queue trigger function click on **SendEmail** and analyze bindings from JSON file. Please refer to *figure*

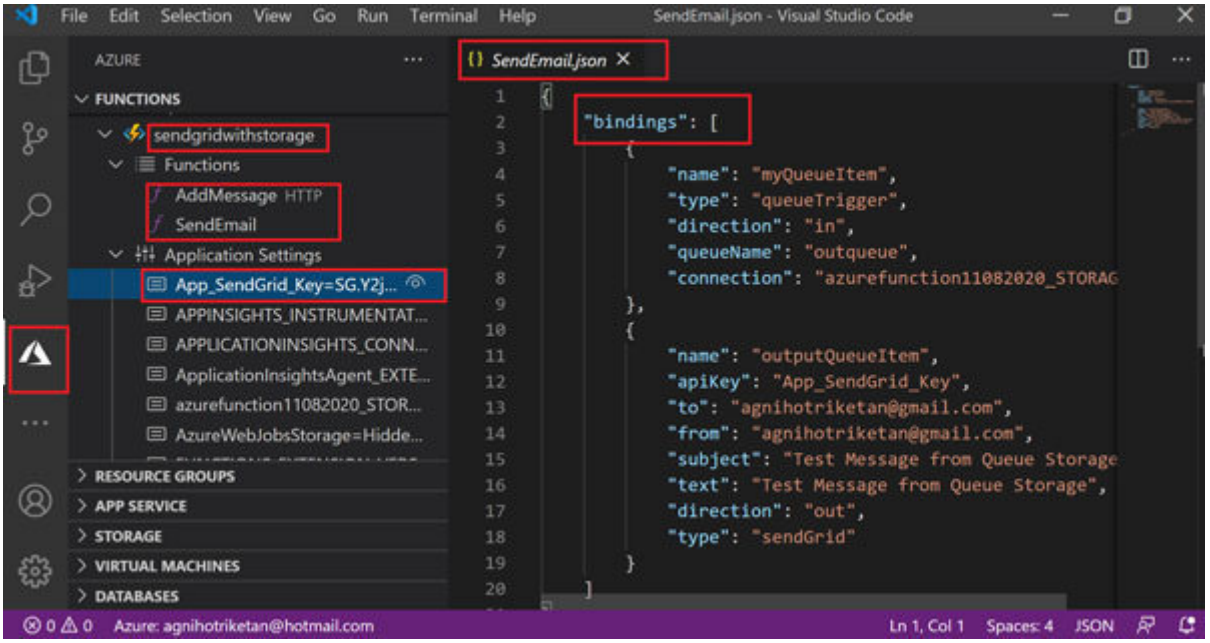


Figure 4.28: *Check created bindings for the SendEmail function using Visual Studio Code*

The following is how the **function.json** file looks like for the **SendEmail** function:

```
{

[

        {

"azurefunction11082020_STORAGE"
```

```
            },
            {
"Test Message from Queue
"Test Message from Queue
"sendGrid"
            }
        ]
}
```

After implementing the function, you need to browse the function URL now and pass your email address as a parameter. Then, you must receive an email in your inbox if everything is set up properly. Please refer to *figure*

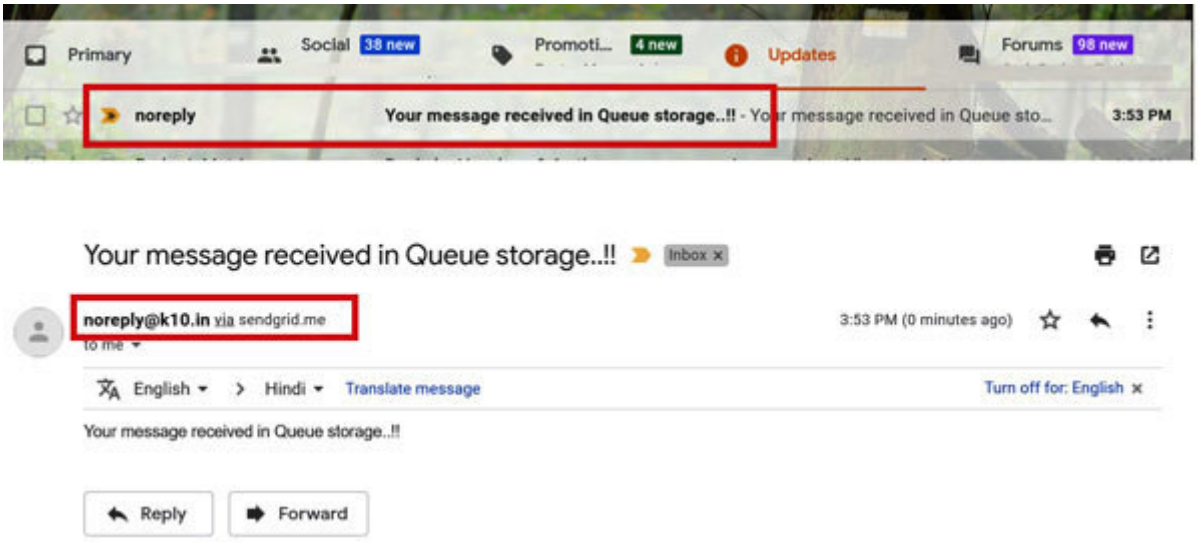


Figure 4.29: *Check received email in the mail inbox*

You can check the execution logs and exceptions and understand whether functions were triggered or not, and executed one after another. You can understand functional flow by looking at function logs. Please refer to *figure*

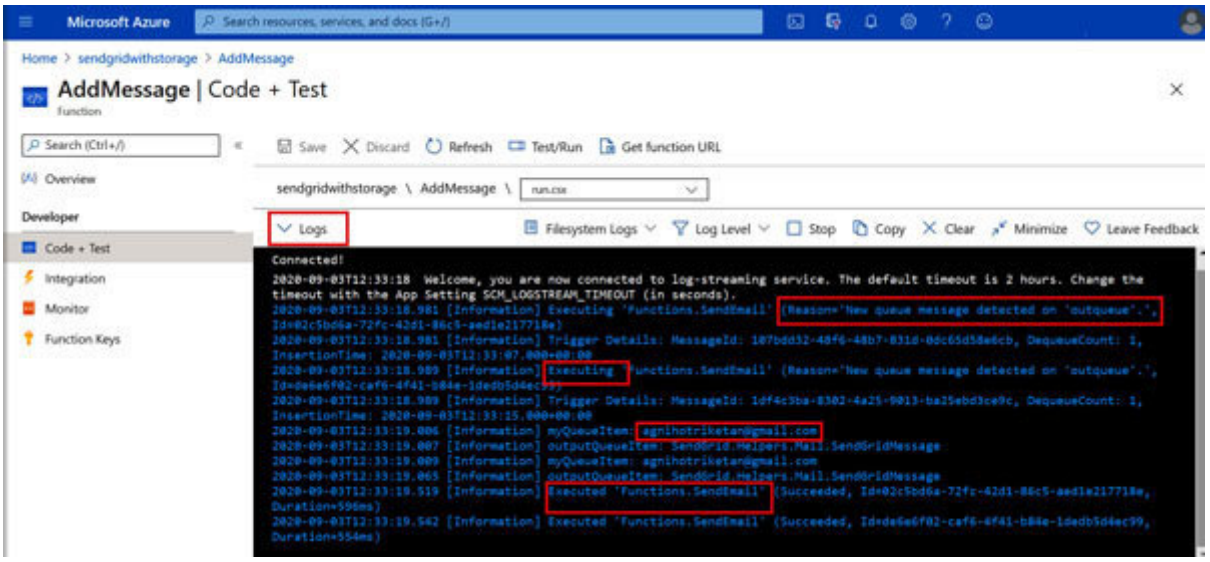


Figure 4.30: *Analyse execution logs to understand triggers and bindings*

You can make changes to these written functions like make as the POST method, accepting the JSON object with more information as payload, and so on. You can play with available triggers and bindings.

Timer trigger and bindings

Timer triggers are used to create a schedule to invoke a function. We can create a schedule and define time. Whenever a timed event occurs, it will trigger a function. For defining time, the CORN expressions are used. The CORN expression is a combination of various time-related properties written in a specific format, such as {second}{minute}{hour}{day}{month}{day of week}. This expression can also contain few special characters as follows:

A comma This is used to separate time properties and lists values together

A forward-slash This can be used to define the repetitive interval of time

An asterisk This is used to declare expression can have any value

A hyphen This is used to define a range of values to and from

In a CORN expression, numeric digits can refer to time in seconds/minutes/hours and days or months. To specify a day of the week, 0 to 6 digits can be used, where 0 starts with Sunday. Also, one can use English names of abbreviations up to three letters to specify a day. The names of the day are case-sensitive. By default, CRON expressions are associated with the UTC zone, but they can be configured in-app settings of the function app.

You can check the sample CRON expression by referring to *table*

Table 4.1: *CRON expression example*

So, using the CRON expression format based on schedule requirements, we can configure a timer trigger to invoke a function at a defined time. Let's see this with a practical example:

To create a timer function, select the timer template and provide the function name and schedule. Please refer to *figure*

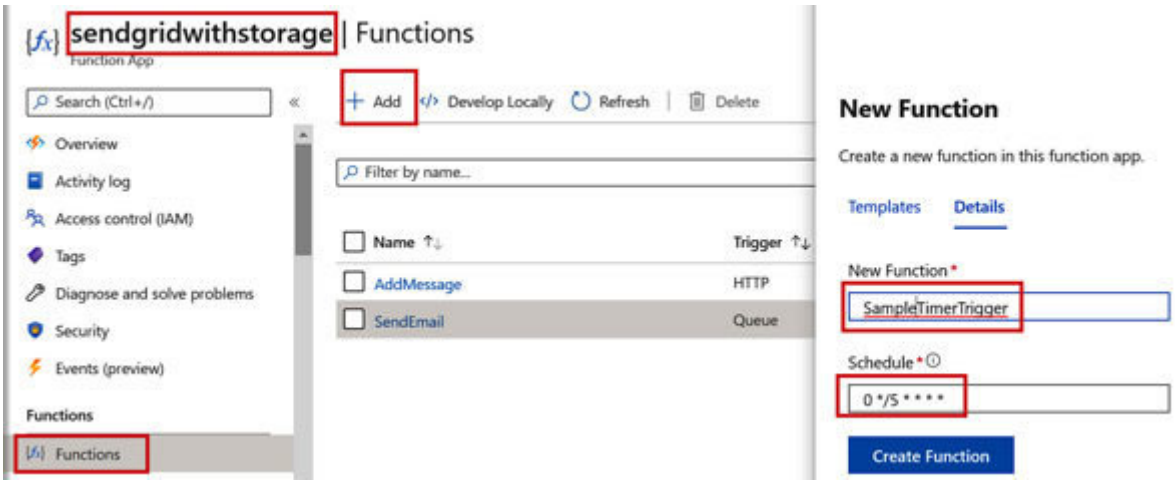


Figure 4.31: *Create a new function using time trigger template*

You can modify the schedule and change the CORN expression by navigating the left integration menu and clicking the trigger to modify the execution schedule. Then, click on Please refer to *figure*

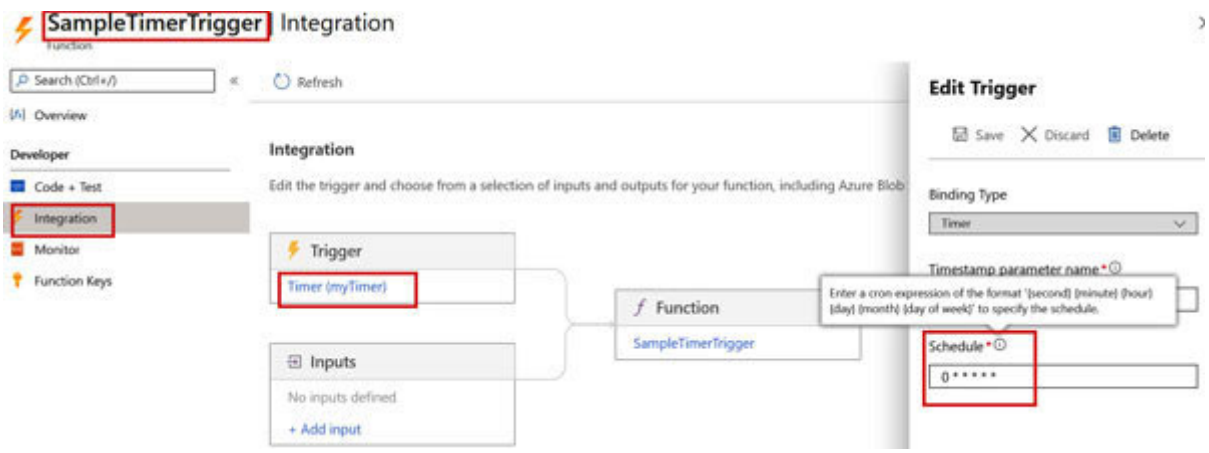


Figure 4.32: *Schedule of trigger this function can be managed using the CORN expression*

To check bindings for your timer-based function, navigate to You can also modify bindings from here. Please refer to *figure*

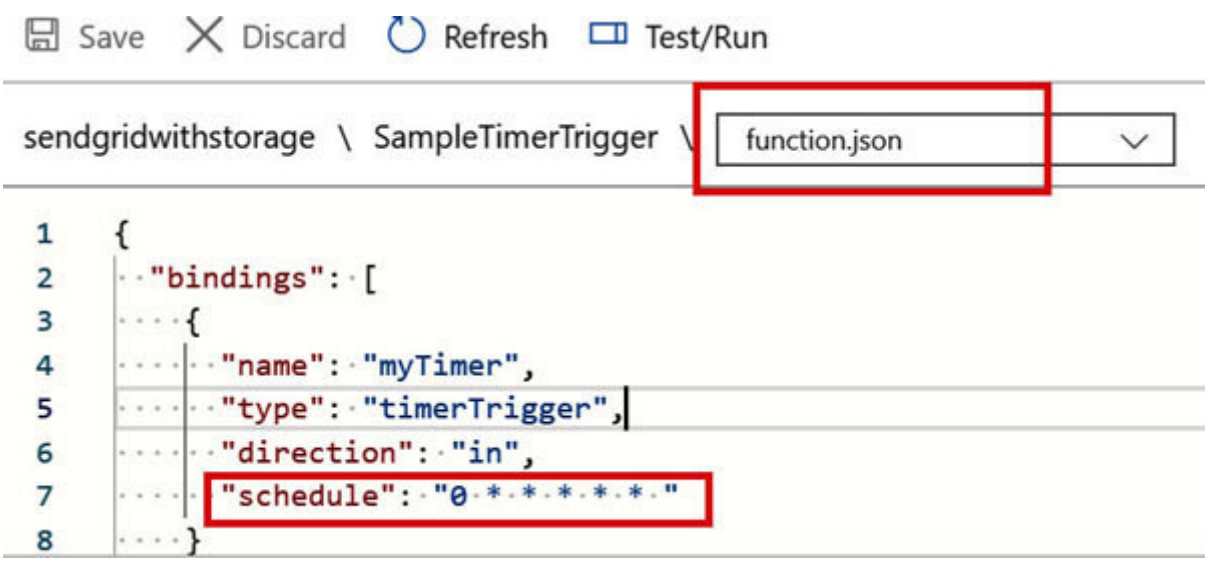


Figure 4.33: *Time trigger bindings*

Let's say you want to trigger this sample timer function every minute. Then, you can verify whether this function got triggered every minute by checking the function execution logs. Please refer to *figure*

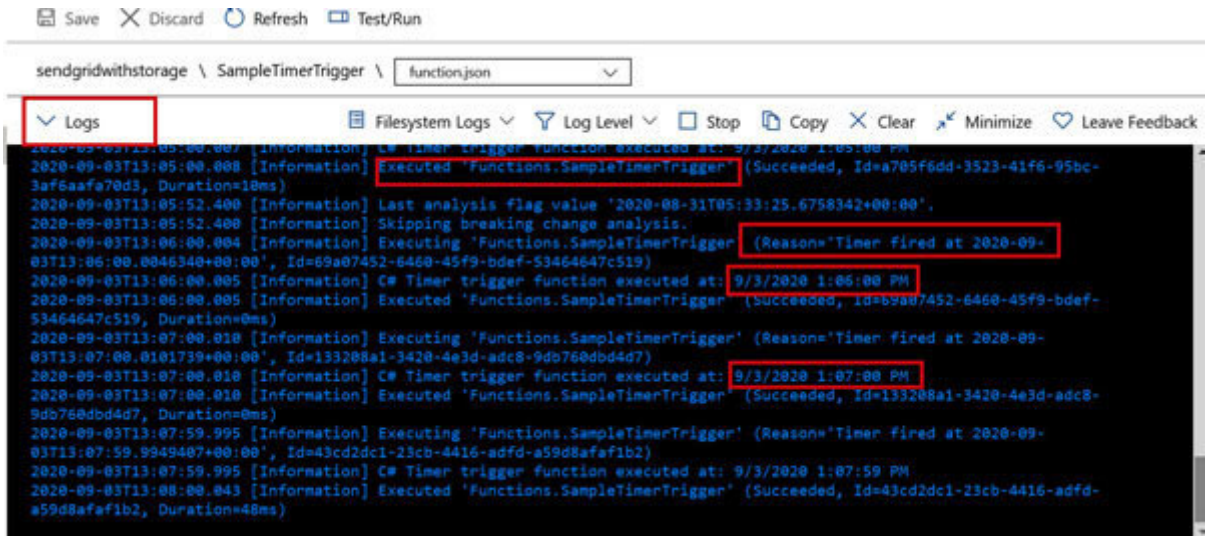


Figure 4.34: *Time based function triggered every minute*

Conclusion

Azure functions provide a rich set of triggers and bindings to invoke the functions and integrate the functions to other Azure resources. Each Azure Function can have only one trigger and multiple bindings with directions: in, out, or both. Azure Functions allows you to define bindings both in code as well as in the portal by going to the Integration page of the function. The configurations for the bindings are stored in the **function.json** file that can be easily modified based on requirements.

In the next chapter, we will discuss the function orchestrations using the Azure Durable functions.

Points to remember

Azure functions provide seamless integration with other Azure services with the help of triggers and bindings

Each Azure Function can have only one trigger but multiple bindings

The direction of triggers is always in but can be in or out for the bindings

The data from the trigger is sent to the Azure Function as a function parameter

You can create microservices architecture with the help of Azure functions that can be deployed and scaled independently

You can define bindings in code or on the Integration page of the function on the Azure portal

Triggers and bindings for Azure Function are configured through **functions.json**

A Blob storage triggered function gets invoked when a blob is created or updated in the Azure Blob Storage

To create a Blob storage triggered function, you should have general-purpose or V2 storage accounts with hierarchal namespaces

SendGrid output binding allows you to send email from your Azure Function easily

Timer triggers are used to invoke function at the scheduled time

CHAPTER 5

Durable Functions and Orchestration

In the previous chapter, we discussed various integration options available with the triggers and binding in Azure functions. In this chapter, we will learn to create long-running, stateful workflow functions using the Azure Durable Functions extension. We will understand the various concepts of Durable Functions and their working behind the scenes. We will also understand various patterns that can be implemented using Durable Functions to solve different problems. At the end of the chapter, we will create our first Durable Functions.

Structure

In this chapter, we will cover the following topics:

Overview of Durable Functions

Understanding the working of Durable Orchestrations

Underlying Azure Storage provider for Durable Functions

Durable Functions Patterns

Working with Durable Functions

Use case: Export SQL Table Records file to blob using Durable Functions

Handling Exceptions in Durable Functions

Objective

After studying this chapter, you should understand the concept of Durable Functions, the working of Durable Orchestrations, and the concepts of sub-orchestration. You will also understand the backstage of Durable Functions and underlying storage. Later in the chapter, you will learn the Durable Functions patterns and the type of problems they solve. At the end of the chapter, you will learn to handle exceptions in Durable Functions, create and deploy Durable Functions on Azure.

Overview of Durable Functions

It is essential to design applications stateless that need to be hosted in the cloud to leverage the autoscaling and dynamic provisioning. However, sometimes it's important to store the state for the application for specific business needs. Durable Functions provides a way to handle state, durability, and reliability in a serverless architecture.

Durable Functions is an extension to Azure Functions that lets you define stateful workflows with the help of orchestrator functions.

Note: The extension, behind the scenes, manages the checkpoints and states to ensure the function doesn't have to start execution from the beginning for any crash or each replay.

The Durable Functions app consists of different Azure functions among the orchestrator function, activity function, Entity function, and Client function.

Orchestrator functions

An orchestrator function allows writing stateful workflows in code by defining the actions and the order of execution for those actions. Orchestrator functions can be long-running with a lifespan of an instance from seconds to months or even never-ending.

Orchestrator functions are triggered by an orchestration trigger binding that reacts to messages enqueued into a task hub. A task hub is a logical container in function's Azure resources, which is used for orchestrations. The orchestration function can define different actions such as activity functions, sub-orchestrations, HTTP triggers, etc.

Note: To test an orchestrator function in the Azure Portal, you need to define a client function that starts an orchestrator.

Client functions

The client function is the function that is responsible for initiating the orchestrator function. Client functions deliver the message to the task hub, to which the Orchestration trigger binding reacts. Any function with durable client output binding can be a client function. For example, HTTP-triggered function, Azure Storage Queue triggered function, Cosmos DB triggered function, and so on.

Activity functions

Activity functions are the functions that perform the actual work and are called by the durable function. There are no restrictions on activity functions for the type of work they can perform, and they can perform any work based on the business logic defined by you. The activity function can also return the results or data to the orchestrator function. Orchestrator and activity functions can only interact with each other through the same task hub.

Note: You can trigger an activity function only from an orchestrator function.

Figure 5.1 shows how the client function, orchestrator function, and activity functions are called and their responsibilities:

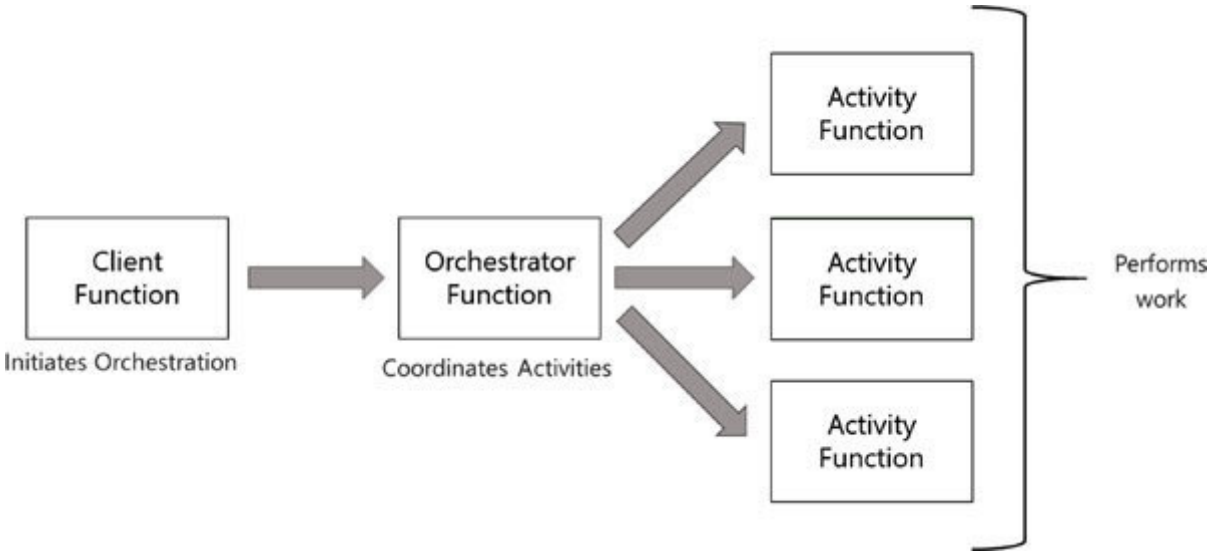


Figure 5.1: *Calling of orchestrator and activity functions*

Entity functions

Entity functions are the special type of functions with an entity trigger used to define operations for fetching and updating the state or durable entities. Entities are the smallest unit of service with a unique identity and maintain an internal state. They have the following properties:

Each entity can perform a set of operations, and that might change its state

Entities can communicate with other entities and orchestrations through messages in reliable queues

Each entity has its unique ID consisting of the entity name and entity key to differentiate entities with the same name

Entity functions can be invoked from both the normal as well as Durable functions.

Understanding the working of Durable Orchestrations

Orchestrator functions are used to orchestrate the execution of other Durable functions. The progress of the execution is automatically checkpointed when the function The state is not lost on process recycles or the restart of the VM.

Each instance of an orchestrator function has a unique ID which is autogenerated GUID. You can also provide your instance ID string, but it should be unique across all instances and should follow the following rules:

Must be less than 256 characters

Must not start with @

Must not contain /, \, #,? and control character

These instance IDs are used for the monitoring and troubleshooting that we will learn in the next chapter.

Orchestrator functions maintain the execution state using the event sourcing cloud design pattern. The Durable Task Framework stores all the actions performed by the orchestration in an append-only store and not only the latest state. This helps in

eventual consistency for transactional data and provides complete history for audit.

The await or yield operation in an orchestrator function gives the control of the orchestrator thread back to the Durable Task Framework. The framework dispatcher stores the new actions scheduled by the orchestrator function in an Azure Storage table, and the orchestrator function is unloaded.

When an activity function returns or an orchestration function is given some work to perform, the orchestrator re-executes the start and rebuilds the local state called During replay, if the code calls another activity function, the Durable Task Framework checks the History table for the execution history. If the function has already been executed, then the orchestrator function doesn't call it again and continues to the next line of code.

Figure 5.2 shows the working of the orchestrator using execution history:

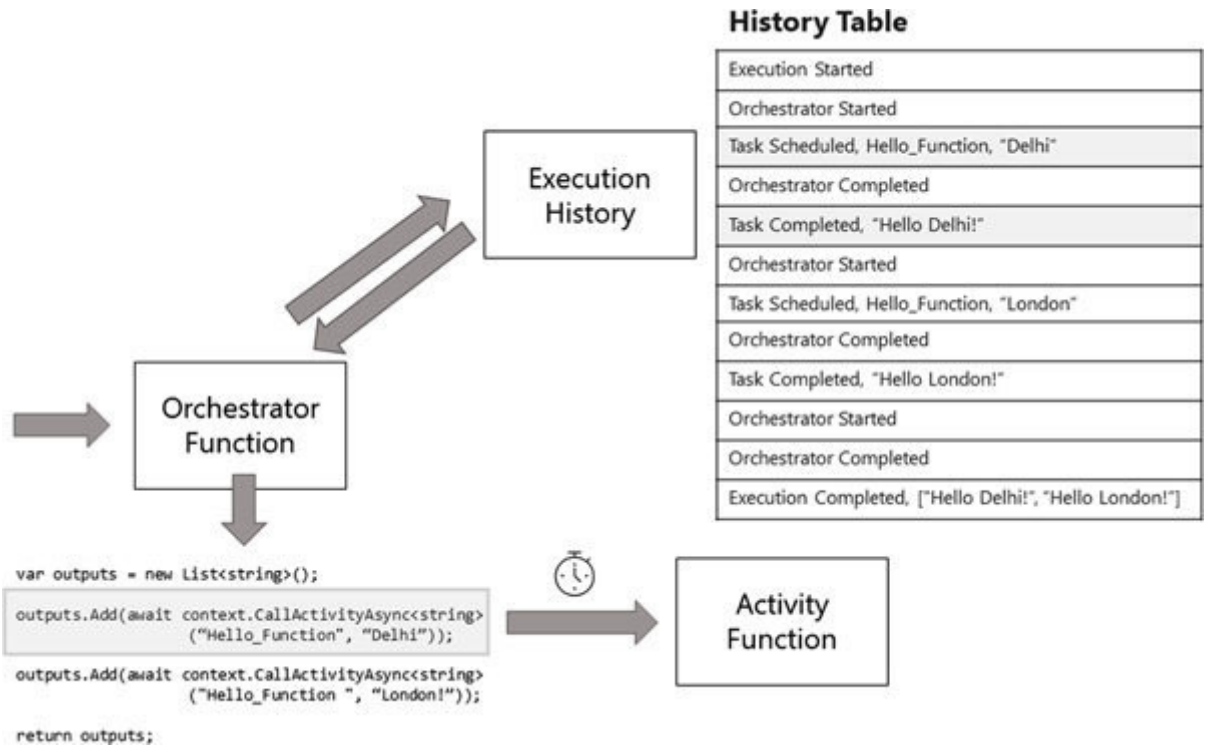


Figure 5.2: *Working of orchestrator using execution history*

To understand how orchestrators use execution history to replay and rebuild their local, let's discuss the execution of a simple orchestrator function as shown in the preceding figure.

The Orchestrator function calls a very simple activity function, which returns **of** where the name of the city is passed as the function parameter.

The following are the steps that will be performed during the orchestration:

A client function calls the orchestrator function.

The orchestrator starts executing until it reaches and awaits code.

On await, the orchestrator checks the execution history of the particular activity function call.

If there is no record of the activity function being called or completed, which is true in the first execution, the orchestrator schedules the task, add a record to the history table, and shuts down.

The scheduled activity function gets executed and completes the task.

A record is added to the history table for task completion along with the output.

Since the Orchestrator code has more steps to be performed, it starts executing again from the beginning and builds up its local state.

Again, when the orchestrator reaches an await, it checks the execution history. This time there is a record present for the activity function being completed.

The output, thus stored, is added to a list of strings in the orchestrator function.

The orchestrator continues and similarly calls the next activity function, and steps would repeat for each await.

Finally, the orchestrator runs to completion and returns the output list.

Handling race condition

Orchestrator functions are single-threaded functions and don't require handling race conditions in general. However, a race condition might occur while communicating through an external system.

Orchestrator functions can define *critical sections* with the help of **LockAsync** to handle such conditions. Only one instance of an orchestrator function can enter a critical section at a time. Other instances will not be allowed to enter the critical section until the previous instance comes out.

The following code in *figure 5.3* shows an orchestrator function that defines a critical section:

```
public static async Task Synchronize(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var lock = new EntityId("LockEntity", "lockid");

    using (await context.LockAsync(lock))
    {
        // critical section code
    }
}
```

Acquire lock and Controls entry into critical section

Removes lock at the end of block

Figure 5.3: *Critical section in an Orchestrator function*

Sub-orchestration

Sometimes a larger Orchestrator function needs to break down into multiple smaller functions or need one more level of parallelism for the orchestration function. This can be achieved with the help of sub-orchestration.

A sub-orchestrator function can be called from an orchestrator function using the following two methods, shown in *figure*

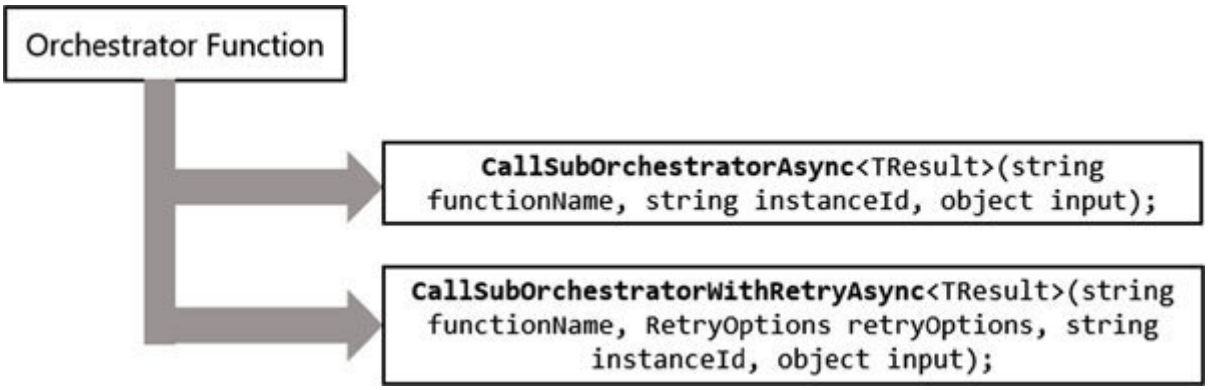


Figure 5.4: *Sub-orchestrator functions*

Underlying Azure Storage provider for Durable Functions

Durable Function has unique underlying scaling capabilities, which improves process execution. Those capabilities are closely dependent on Azure Storage. Let's understand them one by one.

History table

Azure Storage table that contains history events of orchestration running within Task Hub is a History table. The name format of the history table is

Azure Table storage is a cloud service that stores data in a NoSQL structure. Each row has an entity with and which can be used to query data from table storage.

When the orchestrator function runs, the rows from the history table are loaded into the memory, and the events are replayed in the orchestrator function to the previous checkpoint state. Figure 5.5 shows the history table during the first execution and replay.

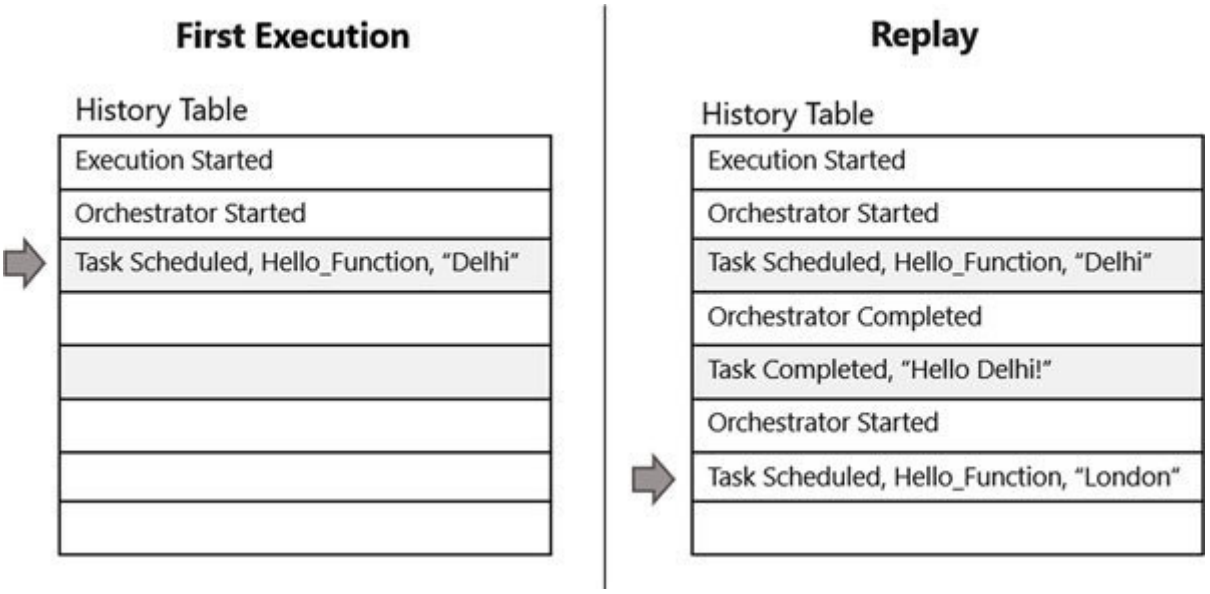


Figure 5.5: *History table during the first execution and replay*

This architecture refers to one of the cloud design patterns known as the *sourcing* You can understand more about this by visiting

Instance table

The Instance table contains the status of all orchestration and entity instances within a task hub. When an instance is created, a new row is added to the instance table. Instance table is used by **GetStatusAsync** (.NET) and **getStatus** (JavaScript) HTTP APIs.

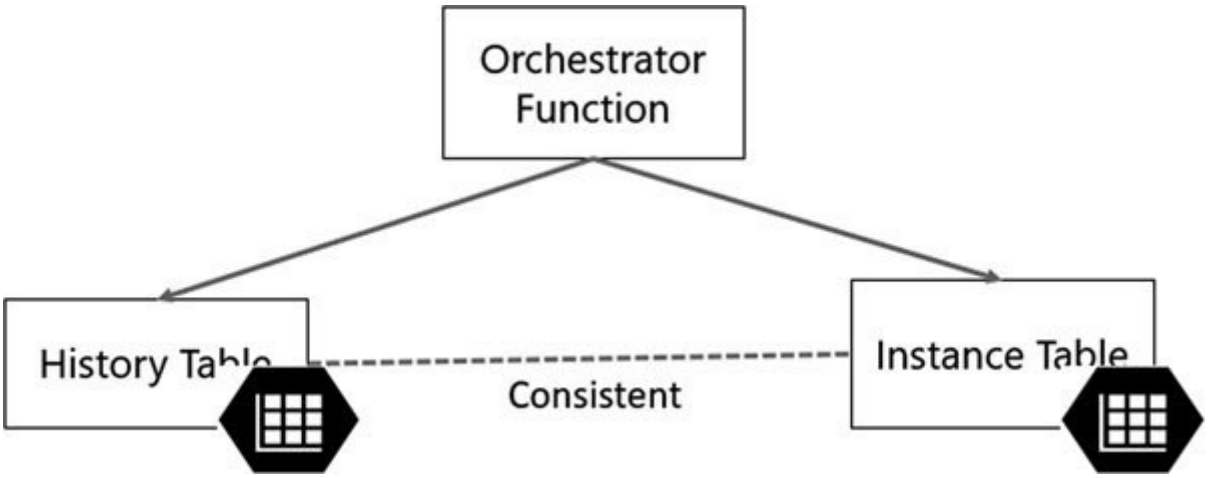


Figure 5.6: *Instance table is kept consistent with history table*

As shown in *figure* the Instance table is kept consistent with the history table. This architecture refers to one of the cloud design patterns known as the **and Query Responsibility Segregation** pattern.' You can understand more about this by visiting

Like normal Azure Functions, durable functions orchestrator and activity function need some trigger to execute. For this, internal queues from the function app's task hub are used to guarantee

message delivery. Durable functions have two types of queues as follows:

Work-item queue

Earlier in this book, we have seen queue triggers for Azure Functions; work-item queues are the same as these basic queues. In each task hub of Durable Functions, there is a single work-item queue. Work-item queue is mainly used to trigger activity functions by dequeuing one message at a time. Inputs required for running these activity functions and some metadata that states information like which activity functions should be triggered is stored as a message in this type of queue. Figure 5.19 show the triggering of activity functions through a work-item queue.

Figure 5.7: *Triggering of activity functions through a work-item queue*

Control queue

Control queues are used to trigger orchestrator and entity functions. There can be multiple control queues in each task hub, and this number depends on partitions configured in the task hub. This is because the orchestrator and entity function instances are stateful and need to be load-balanced across the control queues. Control queues can have various types of messages throughout the execution of orchestrator function, right from any control messages, responses from activity function execution, and so on. One orchestrator instance can accept and process multiple messages at a time as a batch. *Figure 5.20* show the triggering of Orchestrator functions through multiple Control queues.

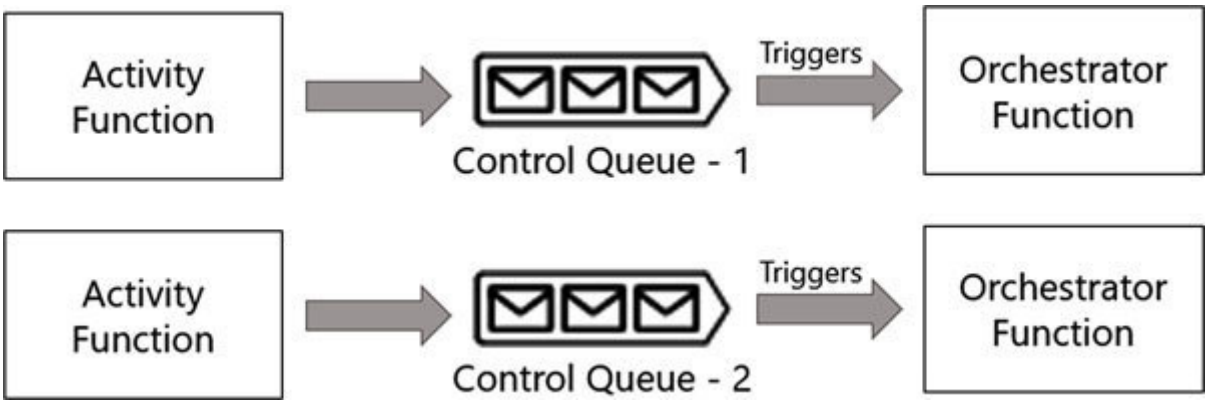


Figure 5.8: *Triggering of orchestrator functions through multiple control queues*

Durable Functions patterns

You can use Durable Functions patterns to write stateful functions based on the requirements. The following are the most popular application patterns that can be used to implement Durable Functions:

Function chaining pattern

Function chaining allows multiple functions to execute in sequence. In chaining, the output of the function executing first can become the input to the next function, as shown in figure

Figure 5.9: *Function Chaining Pattern*

The following code in figure 5.10 is an example of a durable function implementing the Chaining pattern:

```
[FunctionName("ChainingFunctionExample")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("Function1", null);
        var y = await context.CallActivityAsync<object>("Function2", x);
        var z = await context.CallActivityAsync<object>("Function3", y);
        return await context.CallActivityAsync<object>("Function4", z);
    }
    catch (Exception)
    {
        // Error handling.
    }
}
```

```
Function Chaining- Output of each function
Becomes the input to the next function
```

Figure 5.10: *Chaining pattern code sample*

In the preceding example, the orchestrator function has called four activity functions and is used to invoke each activity function by passing parameters. Each function returns the output of type objects assigned to the variables x, y, and z. These are then provided as input to the next function, **Function3,** and **Function4,** respectively. The Durable Functions framework checkpoints the progress of the function instance on the await call. The function instance resumes from the preceding await call in case of a crash or replay.

Fanning out and fanning in pattern

Fanning out and fanning in allows the functions to execute multiple functions in parallel. The output of the functions is then aggregated to get a result. An example could be a reconciliation of multiple accounts in parallel and consolidating the results to get the current balance, as shown in *figure*

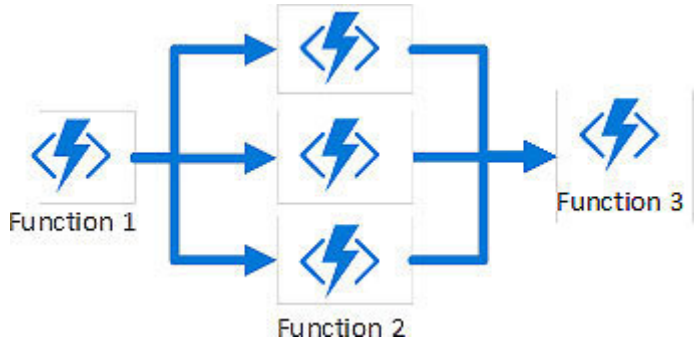


Figure 5.11: *Fanning-out and fanning-in pattern*

The following code in *figure 5.12* is an example of a durable function implementing fanning out and fanning in a pattern:

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

    string[] Values = await context.CallActivityAsync<string[]>("Function1", null);
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<string> task = context.CallActivityAsync<string>("Function2", Values[i]);
        parallelTasks.Add(task);
    }
    await Task.WhenAll(parallelTasks);

    int concatString = parallelTasks. Concat(t => t.Result);
    await context.CallActivityAsync("Function3", concatString);
}
```

Function executed in parallel, with the help of the dynamic list of tasks.

Aggregate the return values of all functions and send as input to Function3.

Figure 5.12: *Fanning out and fanning pattern code sample*

In the preceding example, the orchestrator function has called three instances of the activity function. **Function2** in parallel, with the help of the dynamic list of tasks, after the execution of **Task.WhenAll** is called to await the response of each function that returns a string value. The output of all the instances of **Function2** are concatenated (aggregated) and passed as an input to The Durable Functions Framework checkpoints the progress of the function instance on the

Async long-running APIs pattern

Async long-running APIs pattern provides the mechanism to the external clients to get the state of long-running operations. The long-running API redirects the client to another endpoint to get the status of the execution so that client is not left hanging. This Durable function pattern supports implementing the asynchronous request-reply cloud design pattern, as shown in *figure*

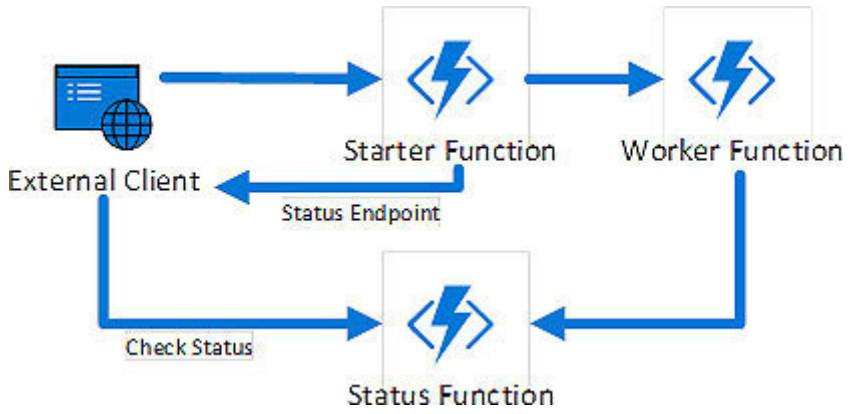


Figure 5.13: *Async long-running APIs pattern*

The following code in *figure 5.14* is an example of a starter HTTP function implementing the async long-running APIs pattern:

```
[FunctionName("AsyncAPIExample")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route = null)] RequestDetails request,
        [ServiceBus("requestqueue", Connection = "ServiceBusConnectionAppSetting")]
        IAsyncCollector<Message> queueMessage,
        ILogger log)
    {
        string statusURL =
                  $http://{Environment.GetEnvironmentVariable(STATUS_ENDPOINT")}/RequestStatus/{reqid}";

        var message = JsonConvert.SerializeObject(request);
        Message msg = new Message(Encoding.UTF8.GetBytes(message));
        msg.UserProperties["RequestGUID"] = message.RequestID;
        msg.UserProperties["RequestSubmittedAt"] = DateTime.Now;
        msg.UserProperties["RequestStatusURL"] = statusURL;

        await queueMessage.AddAsync(msg);
        return (ActionResult) new AcceptedResult(rqs, $"Request Accepted for Processing, StatusURL:
                {statusURL}");
    }
```

Function sends message to the Service bus queue and returns with the "Status Function" URL to the client to poll for status.

Figure 5.14: *Async long-running APIs pattern code sample*

In the preceding example, the starter function is an HTTP trigger function that takes **RequestDetails** as a post message and enqueues the same to the Service Bus queue and returns with the Endpoint to the client to poll for status.

Monitor pattern

Monitor pattern is the reverse of the async long-running API pattern. Instead of returning the status endpoint to an external client to monitor a long-running job, the long-running monitor function keeps checking an external endpoint to monitor. For example, monitoring the temperature of a remote machine and take some action based on temperature going lower or higher than the certain threshold, as shown in *figure*

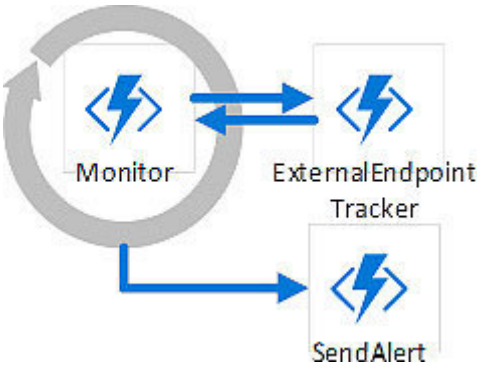


Figure 5.15: *Monitor Pattern*

The following code in *figure 5.16* is an example of a durable function implementing the monitor pattern:

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

    while (context.CurrentUtcDateTime < expiryTime)
    {
        var temp = await context.CallActivityAsync<string>("GetTemperature",
                                                           machineId);

        if (temp > 35)
        {
            await context.CallActivityAsync("SendAlert", machineId);
        }

        var nextCheck = context.CurrentUtcDateTime.AddSeconds(pollingInterval);
        await context.CreateTimer(nextCheck, CancellationToken.None);
    }
}
```

Function keeps polling "GetTemperature" on a regular interval to get the external machine's temperature.

Sends Alert if the temperature goes higher then the threshold.

Orchestration sleeps until polling interval.

Figure 5.16: *Monitor pattern code sample*

In the preceding example, the orchestrator function keeps polling on a regular interval to get the external machine's temperature and call an activity function – to send an alert if the temperature goes higher than the threshold.

Human interaction pattern

The human interaction pattern allows input from a person in an automated process. The function sends the notifications to the person and receives the response asynchronously. You can define logic for expiry or alternative in case of the required person is unavailable. For example, an organization's expense system in which any expense raised by an employee must be approved by the manager within the time frame defined in the organization's policy or should get expired, as shown in *figure*

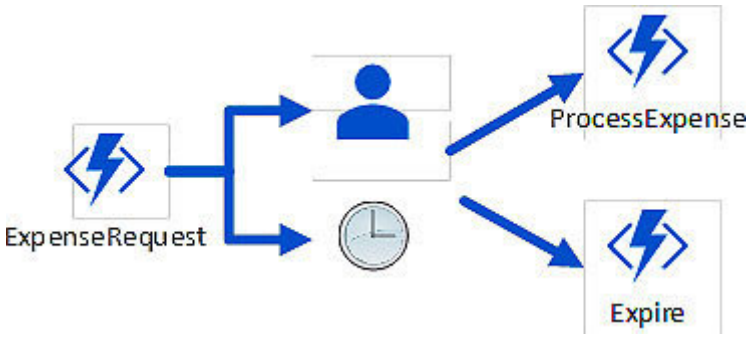


Figure 5.17: *Human interaction pattern*

The following code in *figure 5.18* is an example of a durable function implementing the human interaction pattern.

```
[FunctionName("HumanInteractionExample")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    await context.CallActivityAsync("ExpenseRequest", null);
    using (var timeouttoken = new CancellationTokenSource())
    {
        DateTime maxDateTime = context.CurrentUtcDateTime.AddHours(48);
        Task requestTimeout = context.CreateTimer(maxDateTime, timeouttoken.Token);
        Task<bool> approvalEvent = context.WaitForExternalEvent<bool>("ApprovalEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent, requestTimeout))
        {
            timeouttoken.Cancel();
            await context.CallActivityAsync("ProcessExpense", approvalEvent.Result);
        }
        else
        {
            await context.CallActivityAsync("Expire", null);
        }
    }
}
```

Function raises an Event of approval

If action taken within time the request gets processed otherwise gets expired after the lapse of the time.

Figure 5.18: *Human interaction pattern code sample*

In the preceding example, the orchestrator function raises approval and keeps track of the event for 48 hours. If the manager takes action within this time, the expense requests get processed; otherwise gets expired after the lapse of the time.

Aggregator pattern

An aggregator pattern allows you to aggregate event data from multiple sources over a period. The clients can query the data for its current state. This pattern can also be used to aggregate data from multiple sources and then, based on data, decide which function to be called. An example can be a sports API getting events data from multiple sources for a game and should return the current state on request, as shown in *figure*

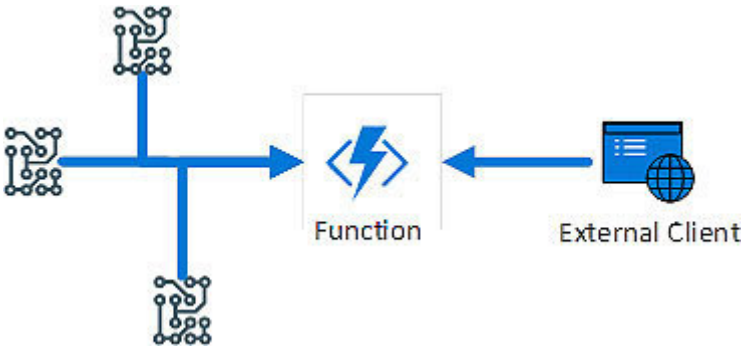


Figure 5.19: *Aggregator pattern*

The following code in *figure 5.20* is an example of a durable function implementing an aggregator pattern:

```
[FunctionName("AggergatorExample")]
public static async Task Run(
    [EventHubTrigger("game-events")] EventData eventData,
    [DurableClient] IDurableOrchestrationClient entityClient)
{
    var game = (string)eventData.Properties["game"];
    var delta = BitConverter.ToInt32(eventData.Body, eventData.Body.Offset);

    // The " GameResult/{game}" entity is created on-demand.
    var entityId = new EntityId("GameResult",game);
    await entityClient.SignalEntityAsync(entityId, "setscore",
                                                        delta);
}

[FunctionName("GameResult")]
public static void GameResult([EntityTrigger] IDurableEntityContext ctx)
{
    int currentScore = ctx.GetState<int>();
    if(ctx.OperationName.ToLowerInvariant() == "setscore")
    {
        int score = ctx.GetInput<int>();
            ctx.SetState(currentScore + score);
    }
    else
        ctx.Return(currentScore);

}
```

Eventhub binding to get game scores from externa/ sources

Eventhub binding to get game scores from externa/ sources

Sets Score

Returns Score

Figure 5.20: *Aggregator pattern code sample*

In the preceding example, the function is a durable function with Event Hubs binding to get game scores from external sources. The function calls the entity function to perform operations like **setscore** or and so on.

Working with Durable Functions

Durable functions are ideal for developing workflow-based stateless applications. The durable function helps to develop modern cloud-based applications to leverage dynamic autoscaling services. Azure Durable Functions are an extension to Azure Functions and support stateless as well as serverless computing architecture.

As discussed earlier, the Durable Functions has the following components:

Orchestrator This function is as a client that will invoke the orchestrator function.

Orchestrator The orchestrator function enables us to develop stateful workflows by calling activity functions asynchronously. This function can store output returned by activity functions in local variables and can be passed as input to subsequent activity functions.

Activity These functions are called by the orchestrator function and contain the function business logic. Please refer to *figure*

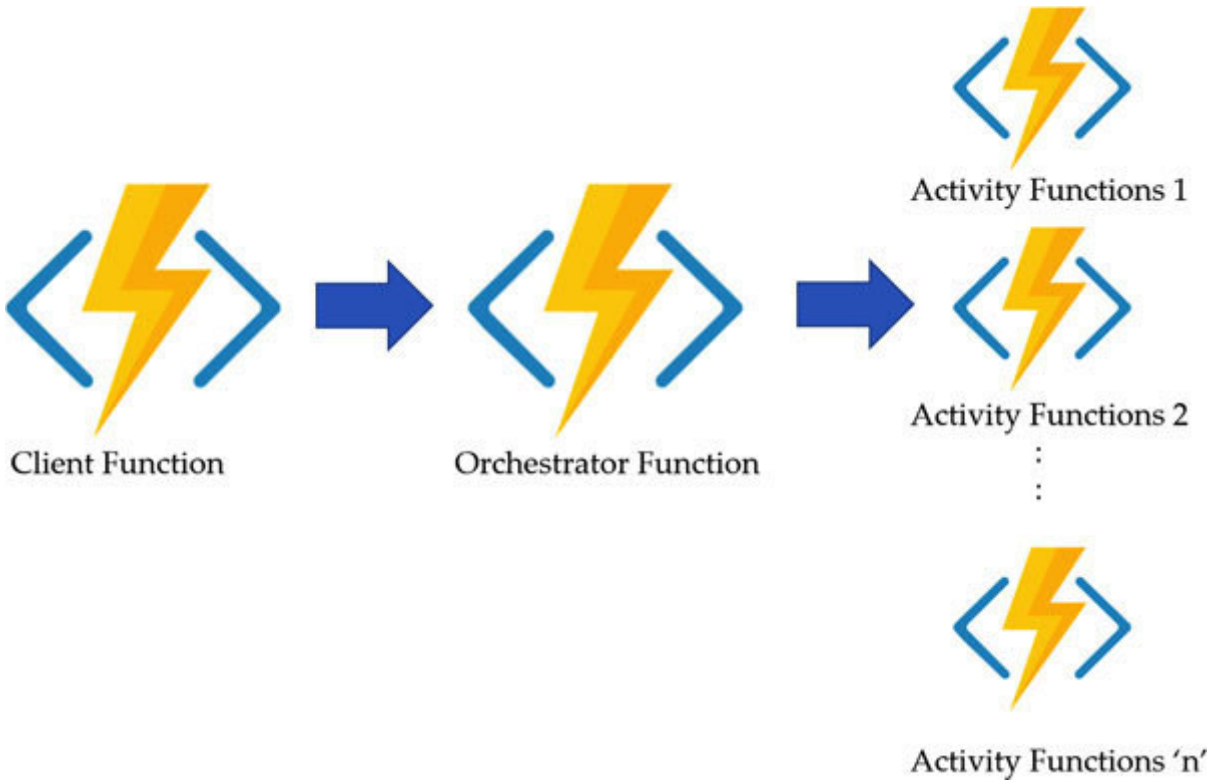


Figure 5.21: *Durable Functions*

Let us create an Azure Durable Function using the Azure portal:

As we understood the steps to create an Azure function app by selecting the required parameters in *Chapter 3: Creating your first Azure* you need to create one function app. After creating the Azure functions app, click on **Functions** from the left menu. Please refer to *figure*

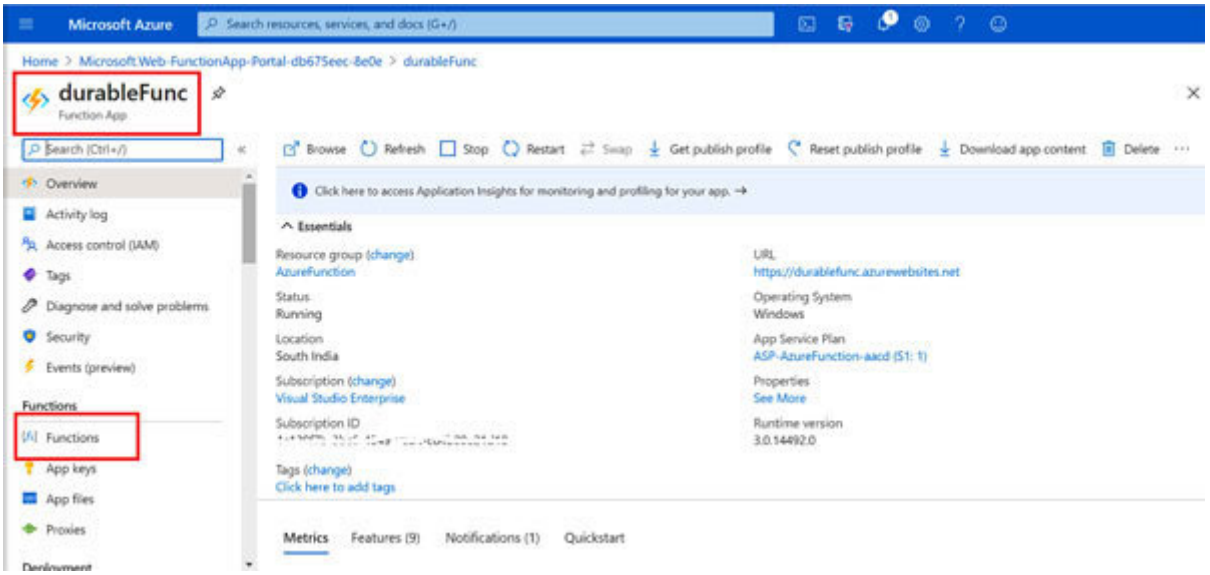


Figure 5.22: *Add new durable functions*

Now, you need to add three new functions by selecting existing templates. Create an HTTP starter function, an orchestrator function, and an activity function without changing existing code generated by templates. Please refer to *figure*

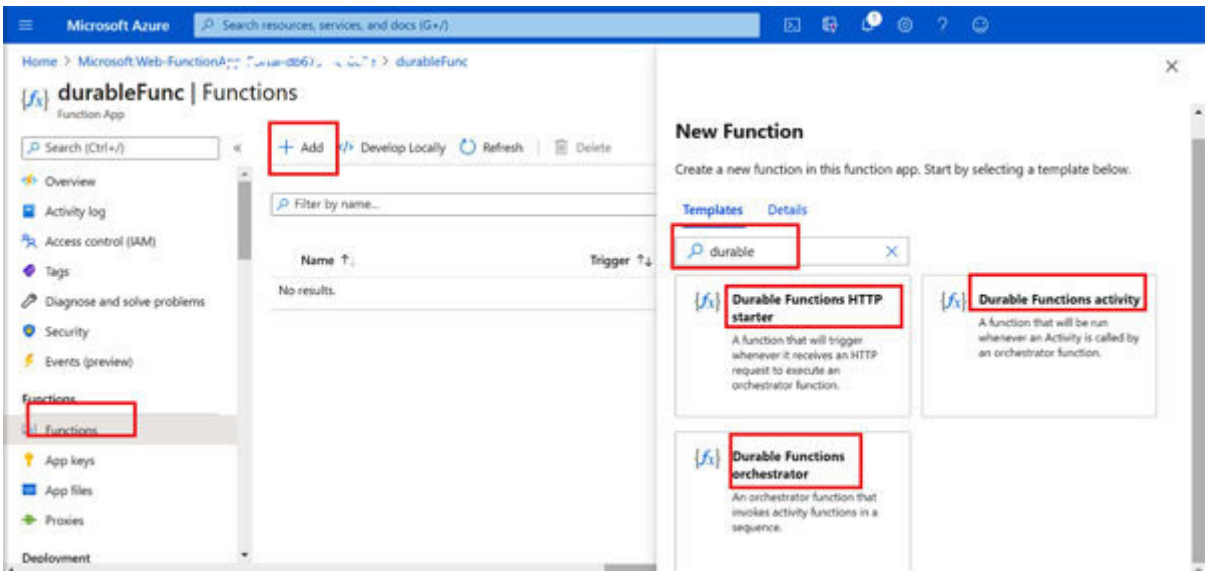


Figure 5.23: *Durable functions templates*

You can now verify the functions generated and their trigger types by navigating to the functions section from the left panel. Please refer to *figure*

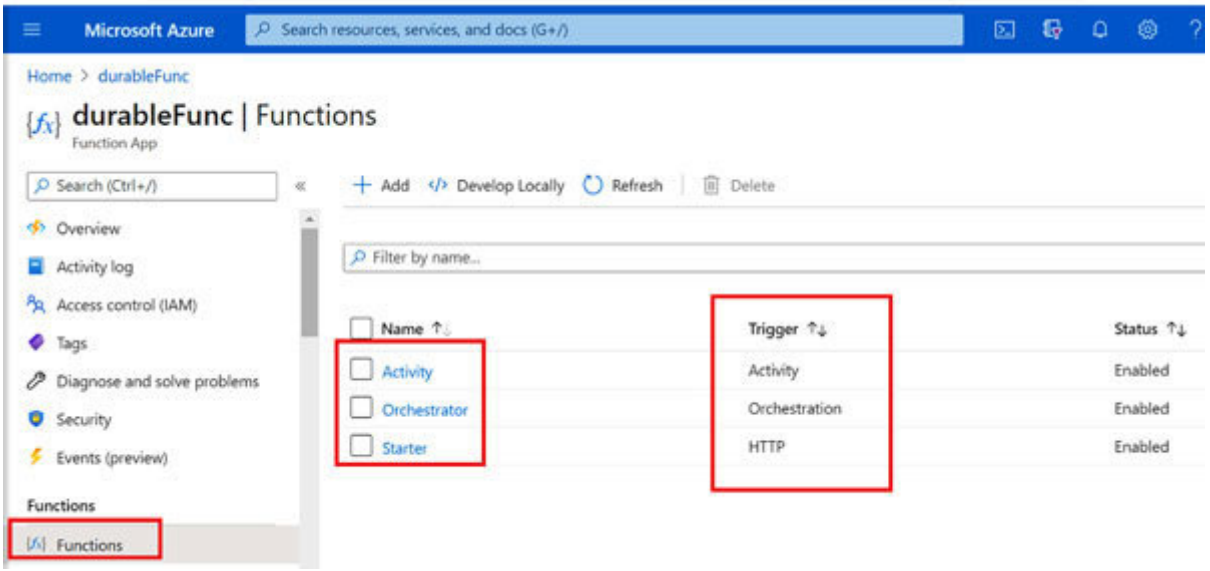


Figure 5.24: *Durable Functions created using available templates*

The starter function is an HTTP trigger, which works as a gateway and accepts the function name with inputs to be executed. It uses the **StartNewAsync** method of the **DurableOrchestrationClient** to invoke the orchestration function. Please refer to *figure*

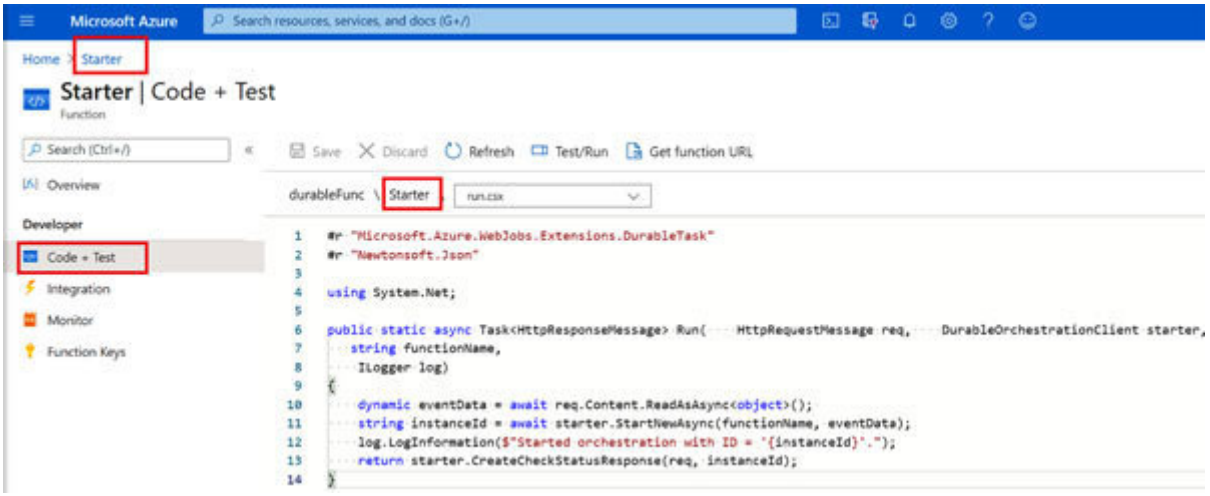


Figure 5.25: *Sample starter functions*

You can check the binding information in Please find binding JSON for starter function as follows:

```
{

[

    {



[
```

“get”

]

},

{

“out”

},

{

“in”

}

]

}

The orchestrator function will invoke activity using the **CallActivityAsync** function of The orchestrator function is responsible for managing the activity functions. You can invoke the same activity function multiple times by passing different inputs. Please refer *figure*

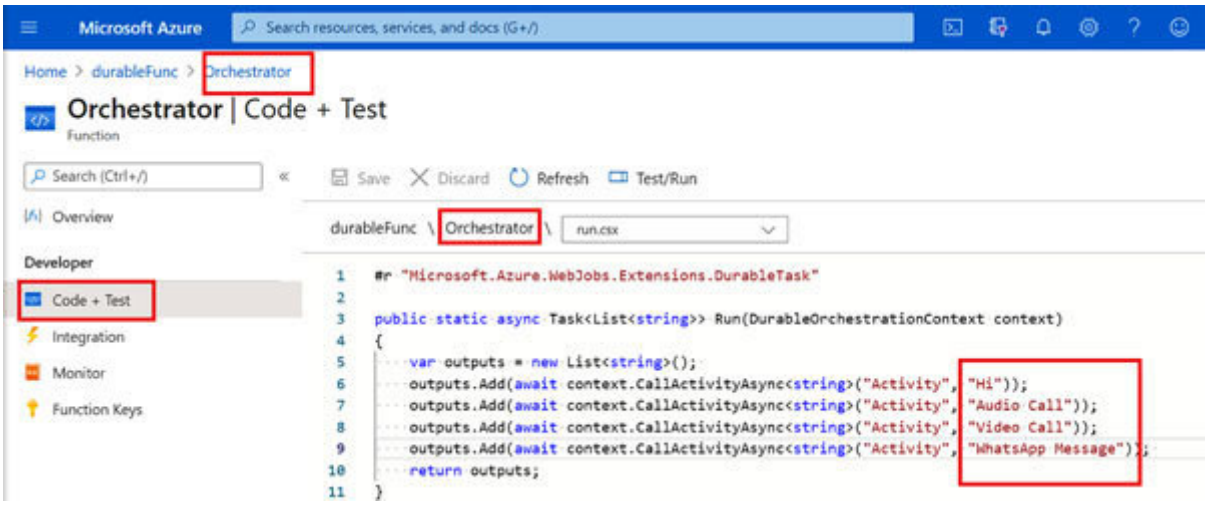


Figure 5.26: *Sample orchestrator function*

Here's the binding JSON for the orchestrator function as follows:

```
{

[

    {
```

```
"in"

      }

   ]

}
```

The activity function has actual business logic and acts as steps in the workflow which the orchestrator manages. Please refer *figure*

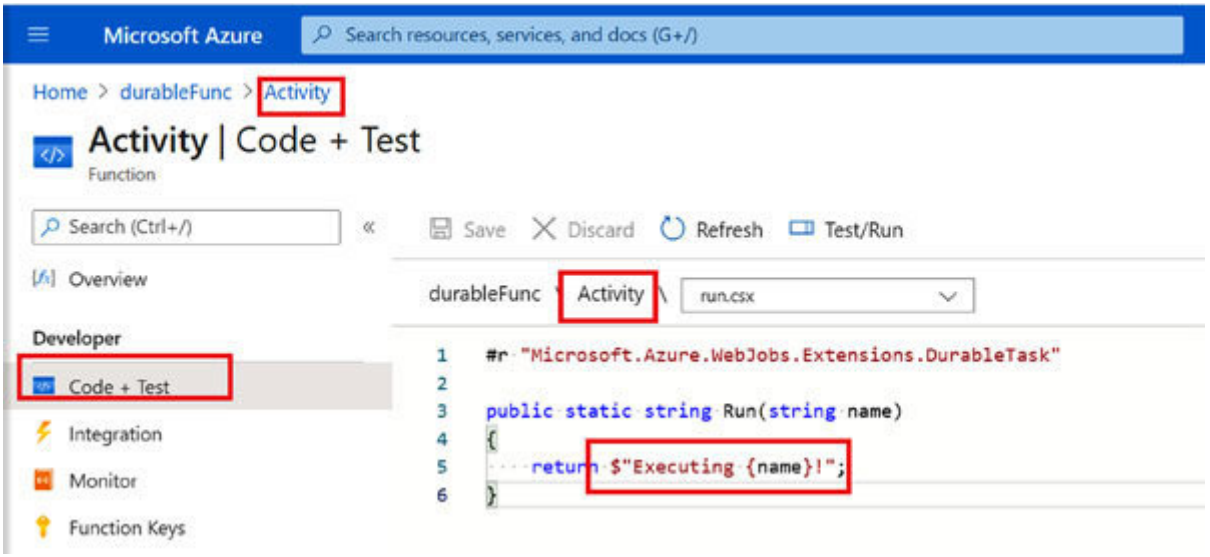


Figure 5.27: *Sample activity function*

Here's the binding JSON for the activity function as follows:

```
{

[
```

```
      {
"in"
      }
   ]
}
```

Now, you have created an orchestration client, an orchestrator function, and an activity function. The next step is to invoke and test the workflow of these functions.

To test the workflow, Navigate to the **Starter** function, and click on the **+** option. Then you need to pass the orchestrator' parameter value as orchestrator and click You can also use the postman tool to run this durable function using function URL. Please refer to *figure*

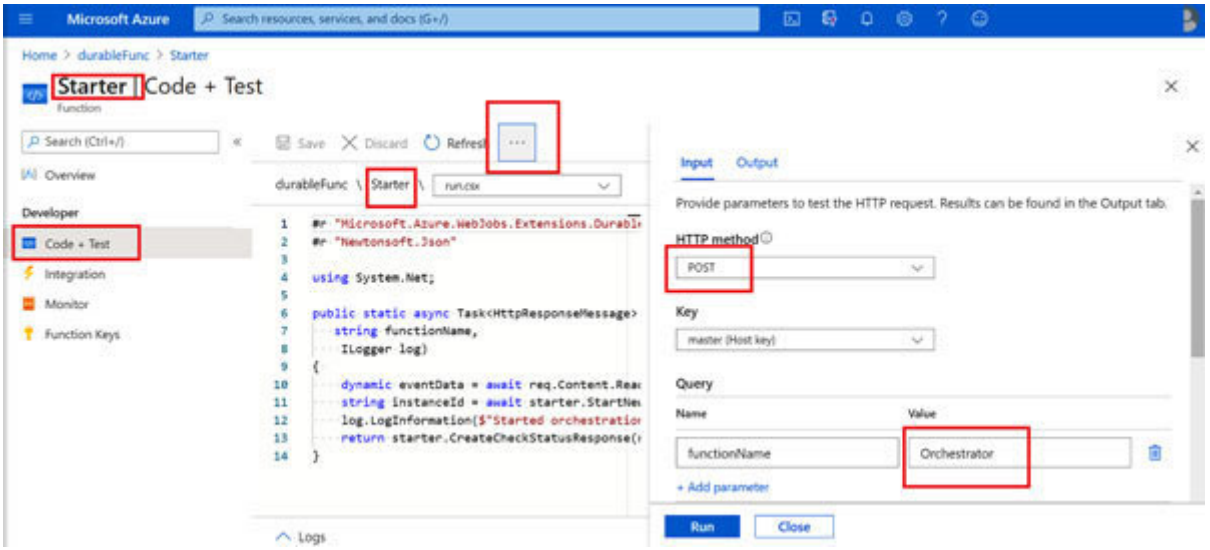


Figure 5.28: *Invoke starter functions*

After successful execution of Durable function workflow, you should see execution logs by event type in Azure storage table **DurableFunctionHubHistory** and Please refer to *figure*

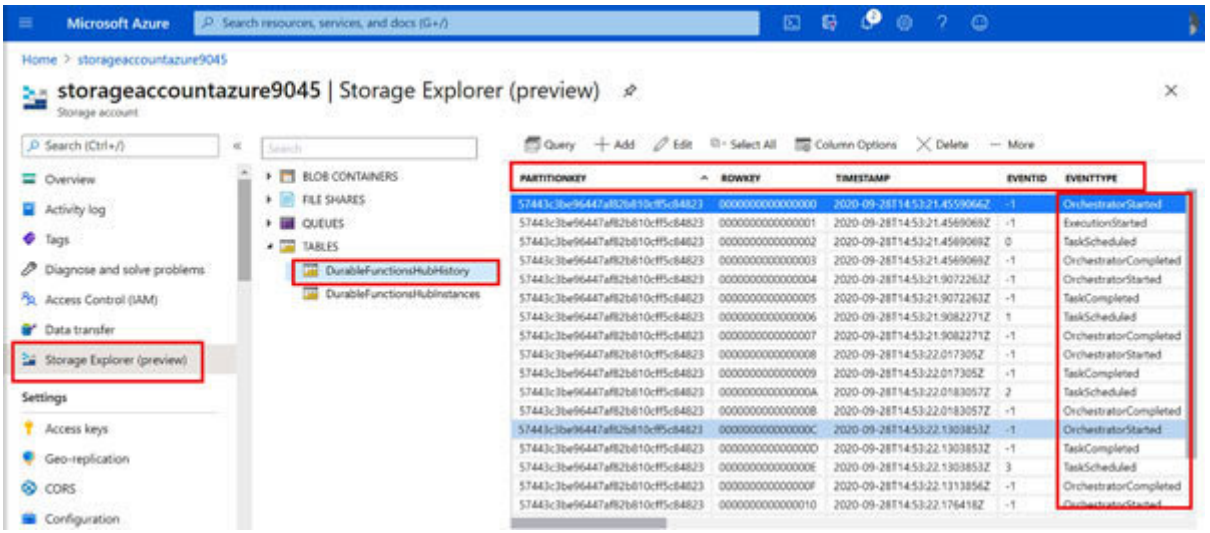


Figure 5.29: *The DurableFunctionHubHistory table*

You can see the output of function execution against instances in the **DurableFunctionsHubInstances** storage table. Please refer to *figure*

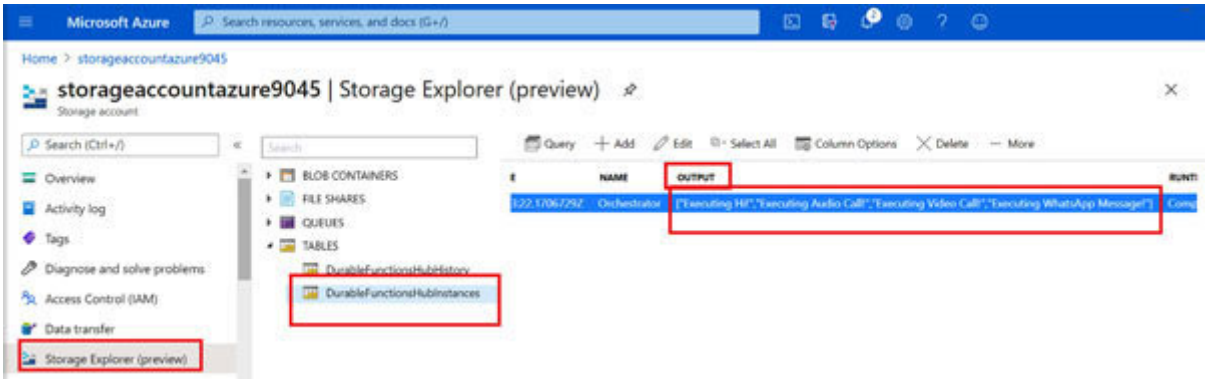


Figure 5.30: *The DurableFunctionsHubInstances table*

You can use the Azure storage explore tool to browse these tables to understand and track the execution history. If something goes wrong while executing this durable function, then navigate to monitor tab of that function to get a detailed error explanation.

The durable function supports reliable execution of the function, which means that it automatically resumes its previous state in case of a crash. This feature is called *and* This is achieved because the execution history is stored in the history table. The storage connection string to these tables can be found in the application settings section of the function app.

USE CASE: EXPORT SQL TABLE RECORDS FILE TO BLOB USING DURABLE FUNCTIONS

CENARIO:

In this section, we will solve a common business case using the orchestrator function as an activity function. You must import and export data from one data source to another data source manually in day-to-day life. This activity might be periodic, like monthly, weekly, yearly, and so on. We will implement an SQL table data export functionality using a workflow implemented using Azure Durable Functions in this use case. We can export this data in various files like JSON, XML, and TXT.

Before starting the actual implementation of the use case, let's understand the serverless solution for this by referring to *figure*

The first step will be to connect to a database or data source from our program

Then, passing this retrieved data to the next step as input

Writing that data into a file type of your need or choice

Uploading a generated file to Azure blob

Using blob trigger and SendGrid, you can send that file as an attachment or its downloadable URL in an email

Export SQL Table Records file to blob using Durable and Activity Functions

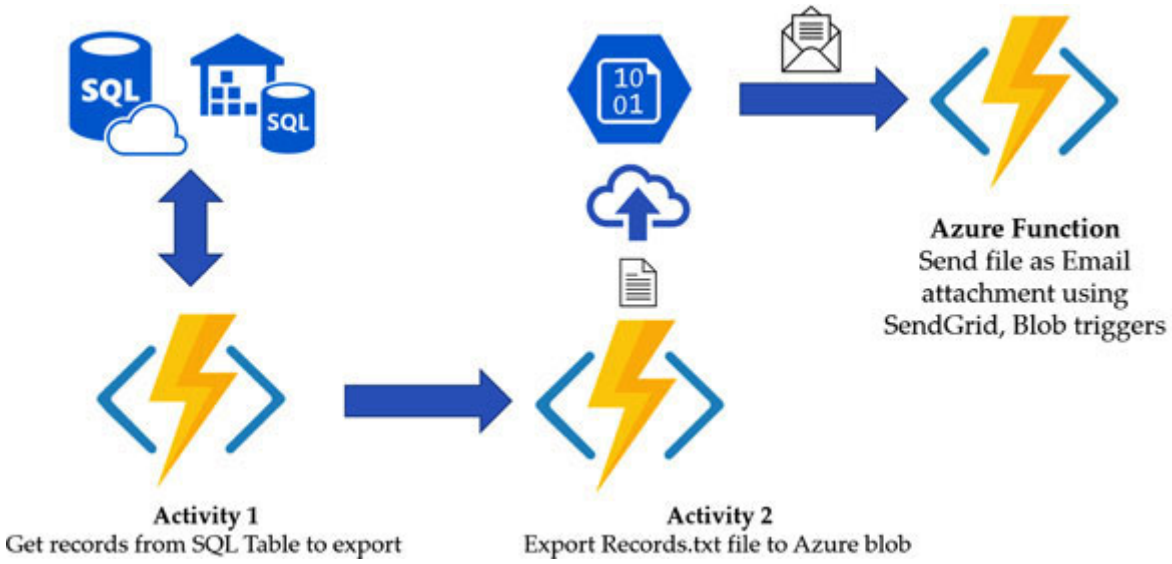


Figure 5.31: *Data export use case*

To build this use case, we will use Visual Studio. Before starting the actual implementation of the workflow, you must have an SQL database and data in the SQL table of the database you want to export in a file. If you do not have any database or data table, create a sample database and add some test data in one SQL table sample. Let's create an employee table in the Azure SQL database to test. Please refer to *figure*

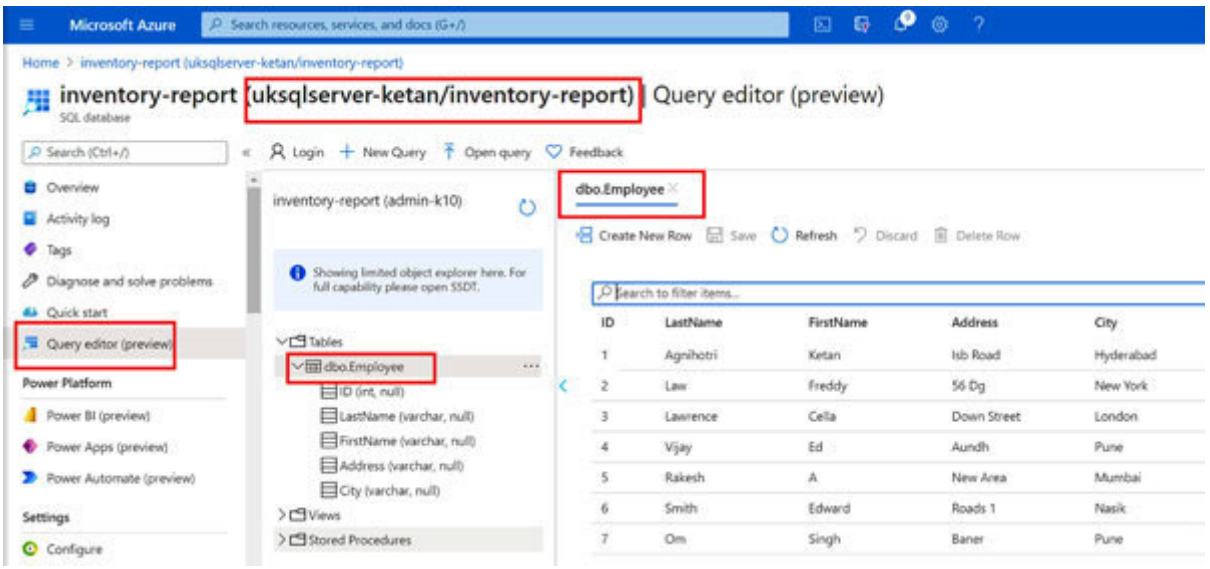


Figure 5.32: *Sample SQL data table*

We have already checked the detailed steps to create the Azure function using Visual Studio in *Chapter 3: Creating your first Azure*

To create a Durable function using Visual Studio, open Visual Studio and create a new project by selecting the Azure functions template. Then, the next step is to choose the Azure Functions runtime as well as the trigger type. You may select blank. Also, you can add a new Azure function by choosing Azure Functions v3 (.NET Core) runtime, Storage Account information later, and function access type. Please refer to *figure*

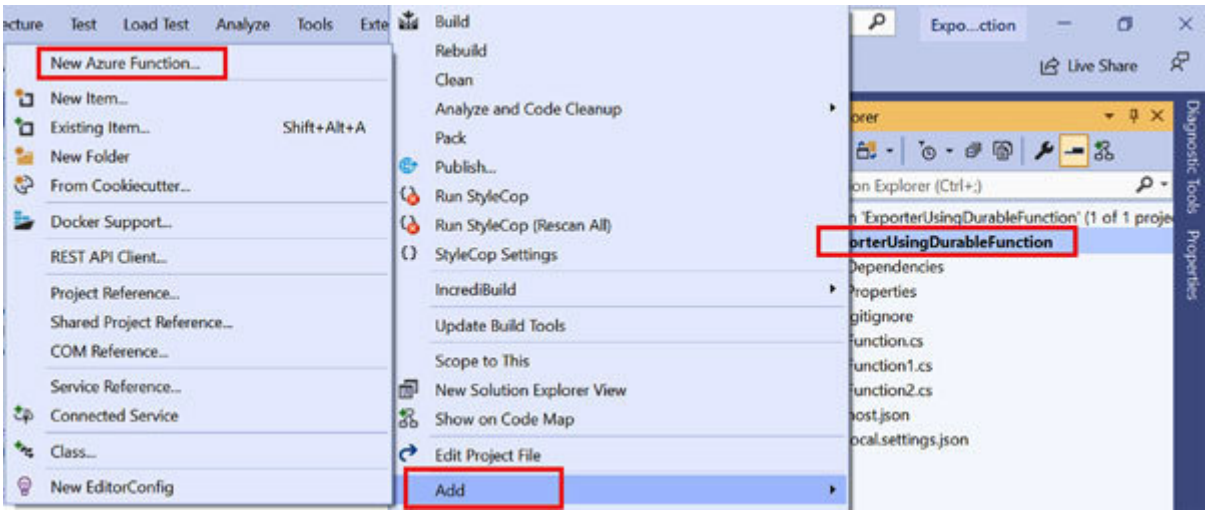


Figure 5.33: *Add a new Azure function*

While adding a new Azure function, you must select **Function** as a function type. This will create the following functions:

This is a client function that can invoke the durable orchestrator.

This is the actual durable orchestrator which will invoke and manage the activity functions.

Visual Studio creates this simple sample activity function. We can remove or modify these default functions.

Now open **function.cs** and implement the starter function. Please refer code for the starter function below, which invokes the orchestrator.

```
Task HttpRequestMessage req,

[DurableClient] IDurableOrchestrationClient starter,

ILogger log)

{

// Function input comes from the request content.

string instanceId = await

orchestration with ID =
```

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

}
```

Please refer to *figure*

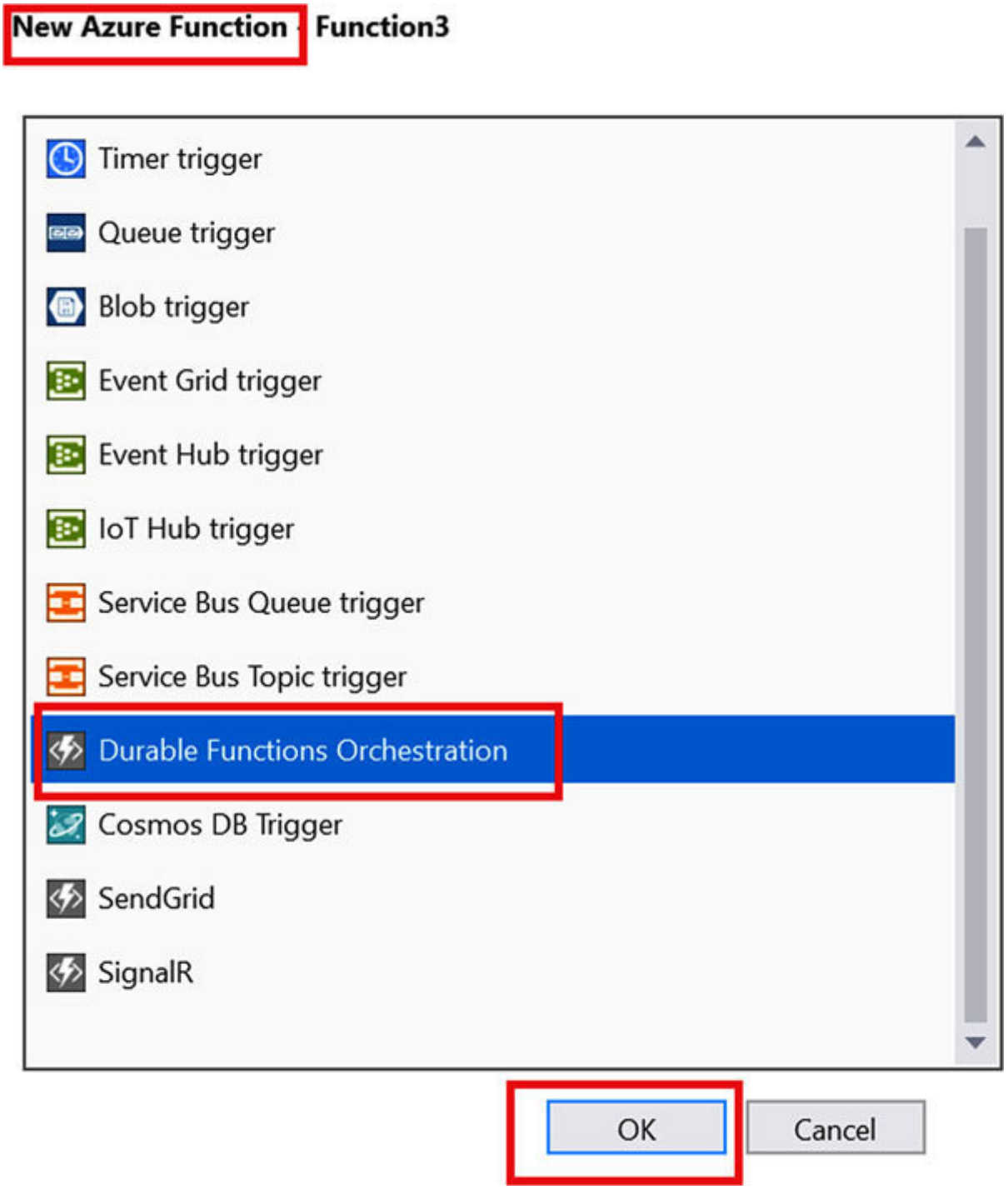


Figure 5.34: *Select Durable Function orchestration*

The next step is to implement an orchestrator function that will pass the database/data source connection string to the activity function.

Please visit the connection strings section of the left panel of the previously created database to retrieve the connection string. Please refer to *figure*

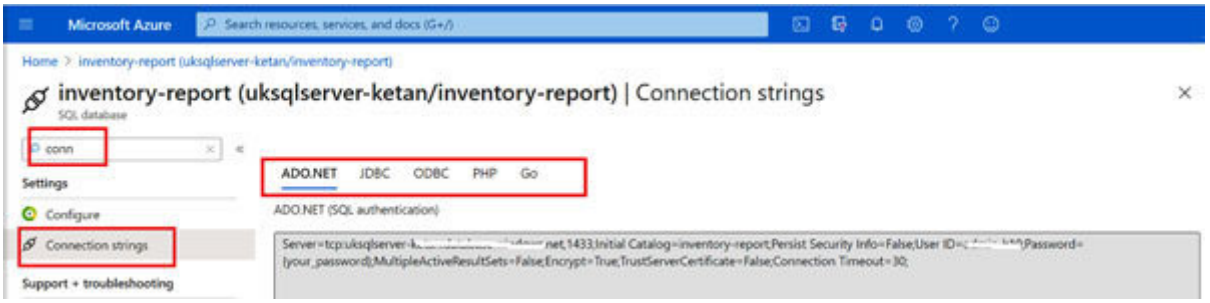


Figure 5.35: *Get the database connection string*

Once this is done, we need to modify the orchestrator function as follows:

```
RunOrchestrator(

IDurableOrchestrationContext  context)

{
```

```
var outputs = new

        {

await "Welcome to Data

        };

string dbConnectionString =

string myString = await dbConnectionString);

myString));

return outputs;

}
```

In this function, function workflow must define. We will first retrieve data from the SQL table. Then, we will write it in a **.txt** file and upload that file to blob. You need to write two activity functions included in the orchestrator workflow and will get executed one after another in sequence.

Activity function 1: Getting records from a SQL table to export

Please refer to the following code to retrieve records from a SQL data table. This function will create line separated string with records from the SQL data table. This will be passed as input to the next activity function. You may refactor the following code as per your need:

```
string connString, ILogger log)

{

        SqlCommand comm = new SqlCommand

        {

            Connection = new SqlConnection(connString)

        };

        comm.CommandText = @"select * from

        comm.Connection.Open();
```

```
                SqlDataReader sqlReader = comm.ExecuteReader();

                StringBuilder records = new StringBuilder();

while (sqlReader.Read())

                {



+ "\t" + + "\t" + + "\t" + + "\t" + +

                }

                sqlReader.Close();

                comm.Connection.Close();

return records.ToString();

}
```

Activity function 2: Exporting records.txt file to Azure blob

Please refer to the following code to write the retrieved SQL table. This function will write a byte array of a string of records into To give a unique file name, **datetime.now** is used. This **MemoryStream** will be uploaded to blob.

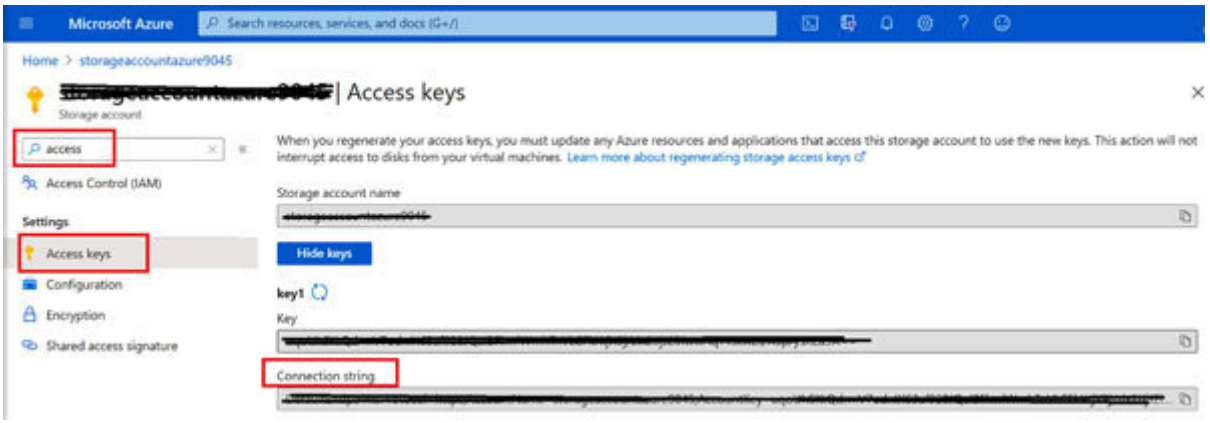


Figure 5.36: *Get storage account connection string*

Before uploading this stream, you need to connect to the Azure storage container using a storage connection string which can be retrieved from the access section of a storage account. Please refer to *figure* You may refactor the following code as per your need:

Task string myString, ILogger log)

```
{

var myByteArray = Encoding.UTF8.GetBytes(myString);

using ms = new MemoryStream(myByteArray))

                {



CloudStorageAccount storageAccount =

var myClient = storageAccount.CreateCloudBlobClient();

var container =

var done = container.CreateIfNotExistsAsync();

var currentTime =

var fileName = "empsqldata" + currentTime +

CloudBlockBlob blockBlob =
container.GetBlockBlobReference(fileName);

                  ms.Position =

await blockBlob.UploadFromStreamAsync(ms);
```

```
                    };

}
```

Azure Function to send a file as an email attachment using SendGrid and Blob triggers

In *Chapter 4: Azure Function Triggers and* we discussed a use case of sending an email using a queue trigger. Using similar steps, you can implement a blob trigger to send this exported data file as an attachment, or if the file is huge, you can share a downloadable URL in the email message body.

Running and testing Azure Durable Function locally

After implementing previous steps, you can run this function app locally by pressing the *F5* key. This will build your source code and open a terminal. You need to fix build errors to invoke the function app successfully. Please refer to *figure*

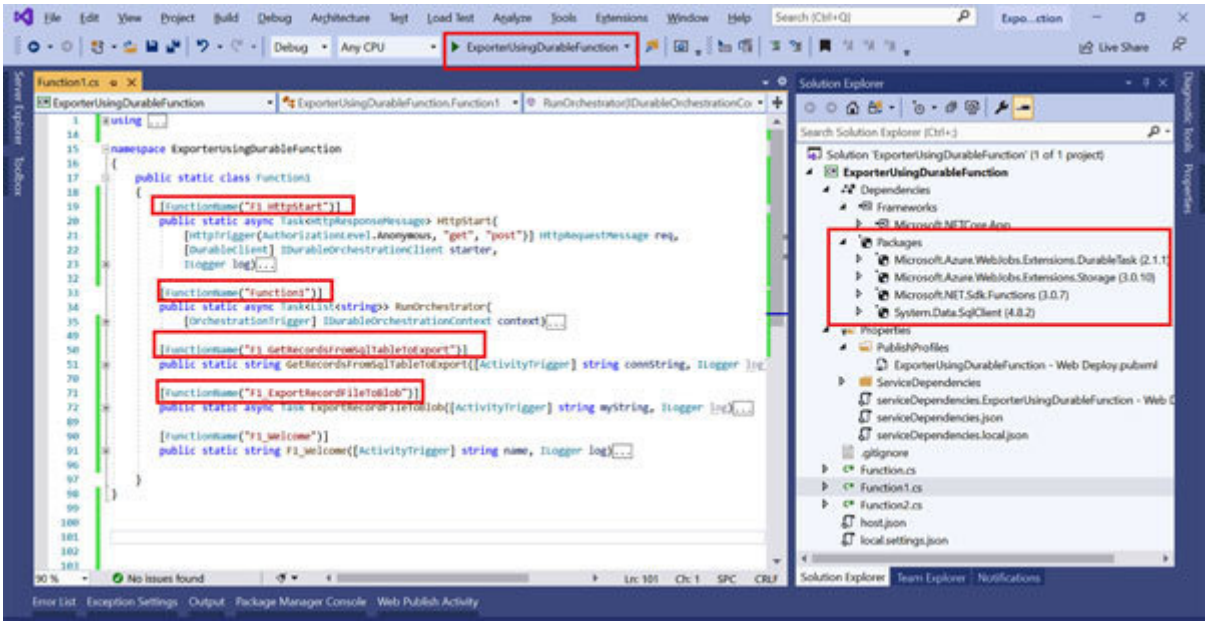


Figure 5.37: *Run Durable Functions*

After successfully building and running the durable function, you should see a terminal window, as shown in *figure* This terminal window will have a list of available functions. Now you can invoke the HTTP-trigger-based starter function from the browser, postman, or any other client-side tool. You can easily understand the function execution flow by analyzing the execution logs from the terminal.

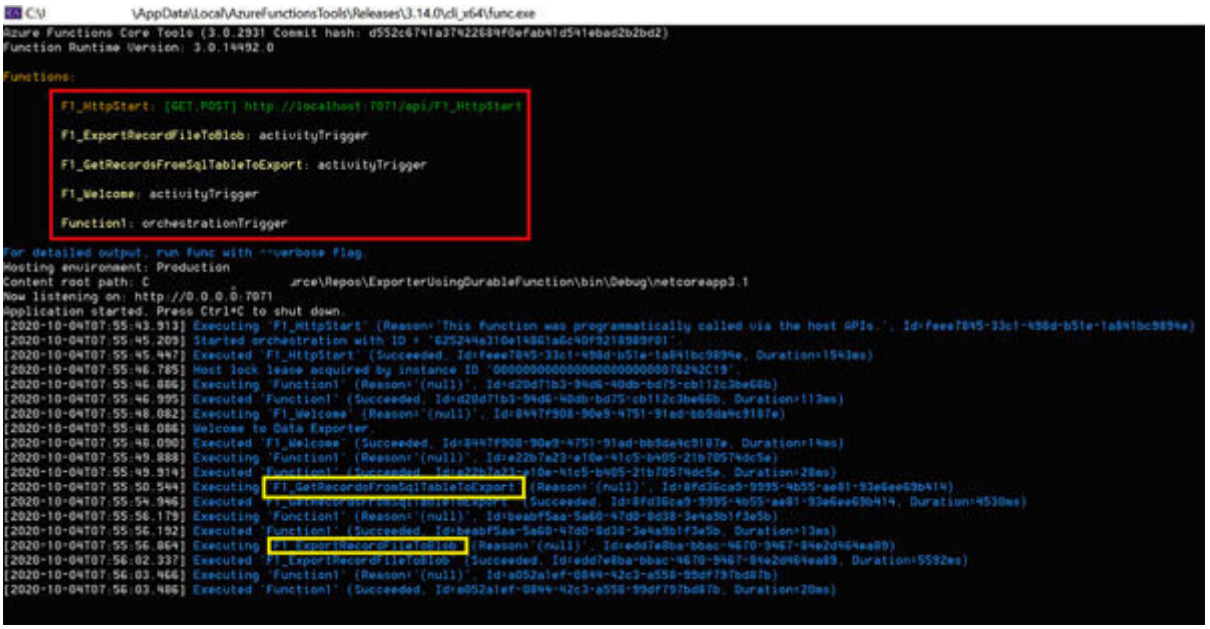


Figure 5.38: *Durable Functions terminal and execution logs*

After successfully executing the durable function workflow, you must be able to see a txt file in your connected storage blob. You may download this file and check the content of this file. Please refer to *figure*

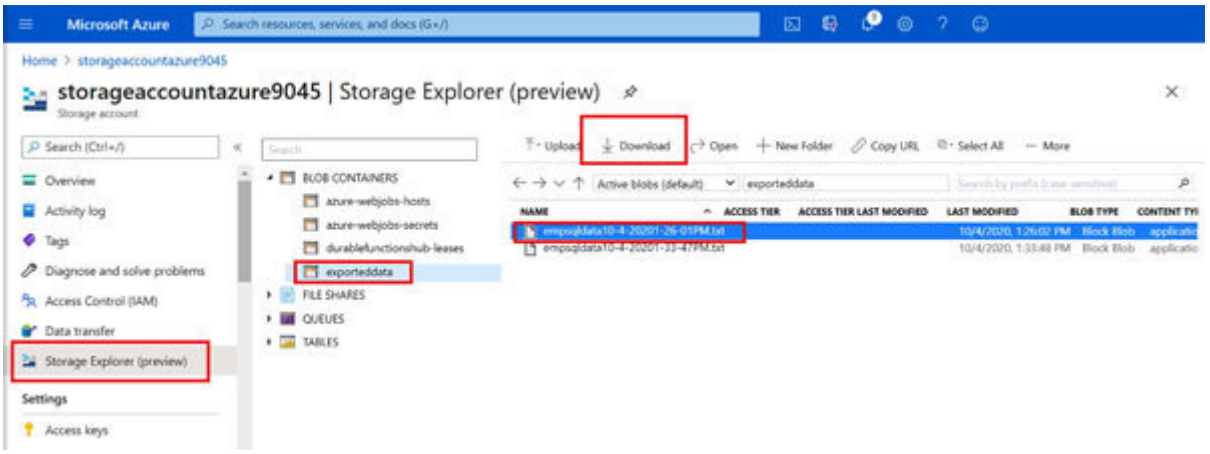


Figure 5.39: *Uploaded date files*

Publishing Azure Durable Function from Visual Studio

After verifying the implemented solution on a local machine, you can publish this durable function from the visual studio itself. To publish this durable function, right-click on **Project** and select

If you haven't yet created a function app, you must create a function app from Visual Studio. Please refer to *Chapter 3: Creating your first Azure Function* to understand the detailed steps of publishing Azure function from Visual Studio. Please refer to *figure*

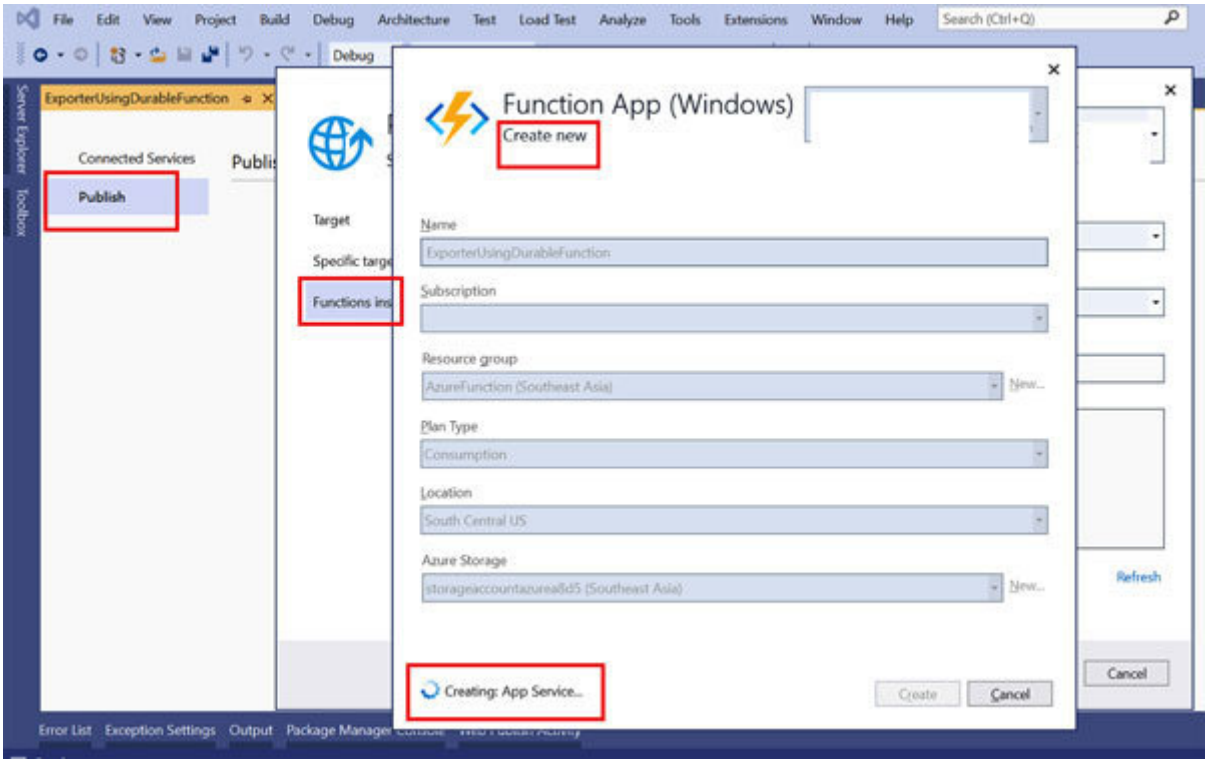


Figure 5.40: *Create Azure function from Visual Studio*

To publish, you must provide the requested data like function app, configuration, runtime storage account configuration by publish wizard. This step will set up a publishing profile in your Visual Studio. After clicking on the **Publish** button, zip upload technology will be used to publish the zip of build artifacts of the Azure function. Please refer to *figure*

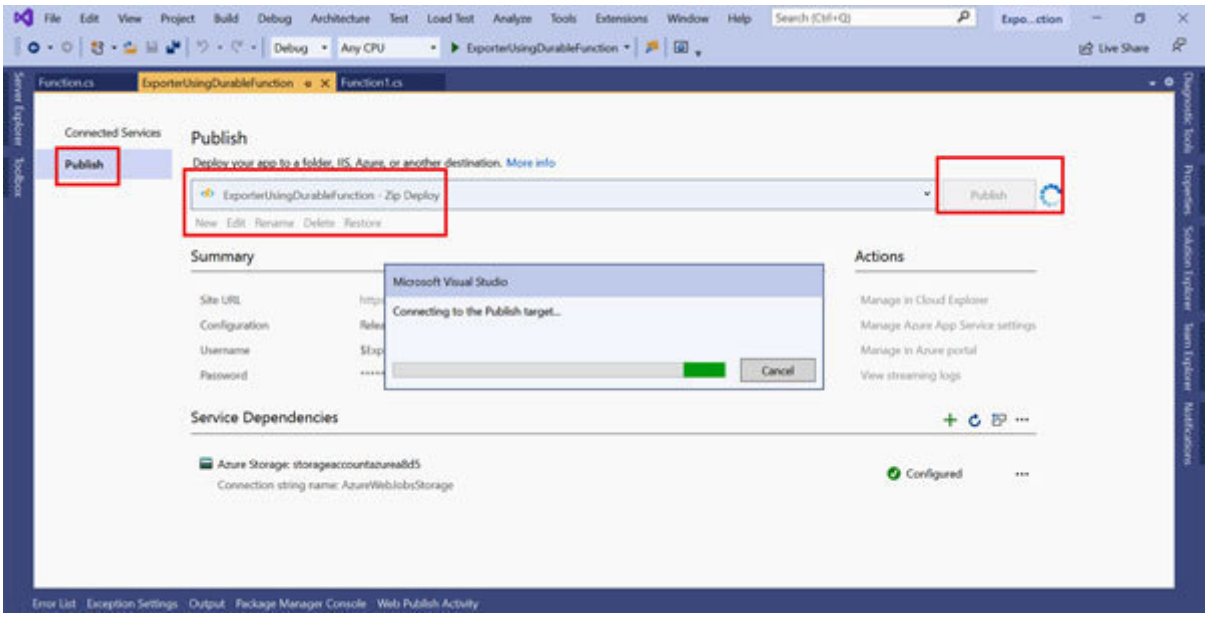


Figure 5.41: *Publish the Azure function*

After successful publishing, you can verify the functions available in the Azure portal or Azure extension in visual studio code. Please refer to *figure*

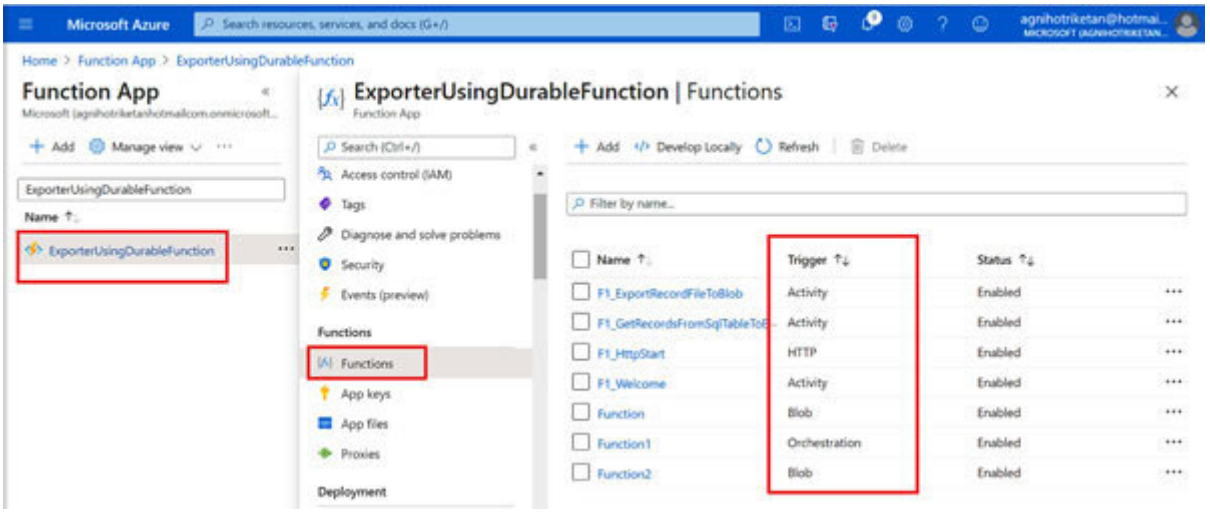


Figure 5.42: *Deployed Durable Functions*

By implementing this use case, you have learned how to develop a function app that can export the data from one source to another using Durable Functions. This type of function app will reduce the repetitive manual efforts of exporting data in regular intervals.

Handling exceptions in Durable Functions

Durable Functions allows you to handle exceptions through the built-in exceptions handling programming language you are using, such as Try-Catch-Finally for C#. However, there are few features Durable Functions provides to handle exceptions as mentioned as follows:

Propagation of error messages to the parent orchestrator When an activity function or a sub-orchestration throws an exception, it propagates up to the orchestrator function It also contains an **InnerException** to have the same message as the original exception.

Automatic retry on failure Durable Functions provides the **CallSubOrchestratorWithRetryAsync()** method to call activity function with retry on failure. You can specify an automatic retry policy as a parameter to this function. The retry policy has the following options:

Max number of The maximum number of retry attempts to be performed

First retry The time to wait before the first retry attempt

Backoff Exponential backoff allows to increase the waiting time between retires after each retry failure instead of retrying after a

fixed amount of time

Max retry The maximum amount of time to wait in between retry attempts

Retry The maximum amount of time to spend doing retries

Handle: The user-defined callback to determine if a function should be retried

Unhandled The unhandled exceptions are auto-handled by the durable functions by returning the status as failed for the activity function and logging the exception. But it's always a best practice to implement the error handling mechanism provided by the language runtime instead of relying on the Durable Functions Framework.

Conclusion

Durable Functions can support long-running processes, process monitoring, parallel execution of functions (fan-out, fan-in), and central error handling. The underlying durable task framework does the reliability management, state management, and tracking management by provisioning and managing storage queues and tables. The Azure durable function is an extension to Azure Function and an easy way to implement the workflows. In the next chapter, we will understand the security configuration in Azure Functions.

Points to remember

Durable Function is an extension to Azure Functions which implements workflows by writing a minimal amount of code

Stateful functions and serverless computing are achievable using durable functions

The orchestrator function calls and manages multiple activity functions that are responsible for implementing logic

The sub-orchestration feature allows to orchestrate several activities using the **CallSubOrchestratorAsync** or the **CallSubOrchestratorWithRetryAsync** methods and used where a retry mechanism is needed

Durable Functions use storage for communication and state management behind the scenes

Durable Function patterns can be used to solve complex scenarios in the applications

Exceptions handling in Durable functions can be achieved through the built-in exceptions handling of programming language

CHAPTER 6

Configuring Security for Azure Functions Security

We discussed the Azure Durable Functions and functions orchestration to implement long-running, workflow, and stateful functions in the previous chapter. In this chapter, we will discuss planning and implementing security for Azure Functions. Planning for secure development and operation of serverless functions is much the same as for any web-based or cloud-hosted application.

Structure

In this chapter, we will cover the following topics:

Function access keys

Azure active directory authentication

TLS/SSL certificate for function apps

Using managed identities for functions

Network security

Objective

After reading this chapter, you should understand the various available ways for securing Azure Functions, including keys, certificates, azure active directory authentication, managed identity, and so on. You will also learn how to add network security, putting functions in virtual networks, enabling CORS and access restrictions, and whitelisting IP addresses for your Azure Functions app. Finally, you will learn to implement network security using API management.

Function access keys

Azure function access keys provide a default mechanism to secure your HTTP-triggered functions. You need to provide an access key while making an HTTP request to your function endpoint. This is a good way to secure your APIs to a certain level but should not be the only security you implement to your functions. Also, it requires sharing the keys to all the clients in case of new key creation or renewal. Azure functions provide four types of access keys as follows:

Function These are function-level keys that allow access to a specific function only for which the key has been defined.

Host These are also function level keys, but unlike function keys, these allow access to all the functions within the function app.

Master Master keys are the admin-level key that provides all-access provided by the host keys and administrative access to the framework rest APIs. These keys should not be shared with the clients.

System System keys are the system-managed keys created by specific extensions called by internal components. These keys are used by certain triggers such as event grid and durable functions to call durable task extension APIs.

Accessing host, master, and system keys

You will find host, master, and system keys in the **App Keys** section under **Functions** in the left menu of your function app, as shown in *figure*

Figure 6.1: *App keys*

As you can see, by default, the Azure function provides one master key with the name and one default host key.

Accessing function keys

You will find the function keys in the **Function Keys** section under **Developer** in the left menu of the specific function in the function app, as shown in *figure*

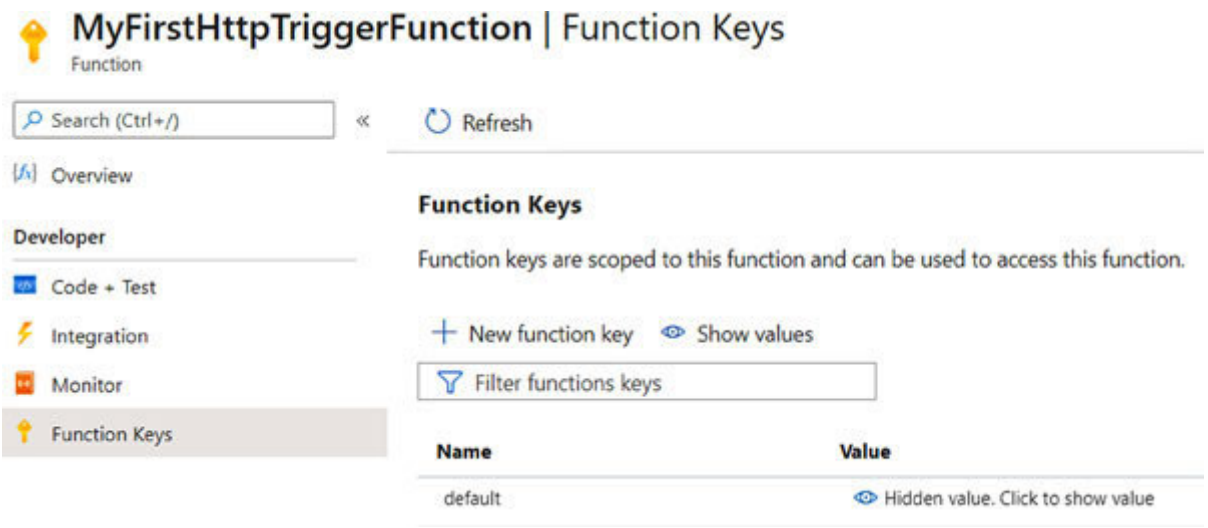


Figure 6.2: *Function keys*

Creating a new key

Follow the given steps to create host keys:

For host keys, click on the **Host** option on the page, as shown in *figure*

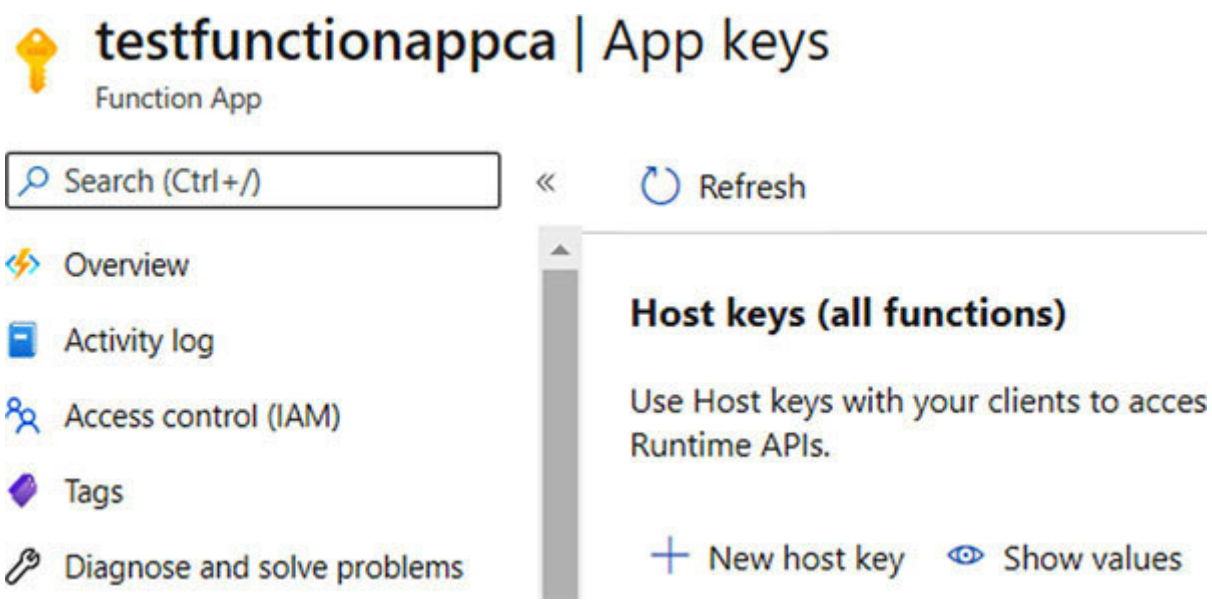


Figure 6.3: *New host key*

For function keys, click on the **function** option on the page of the specific function as shown in *figure*

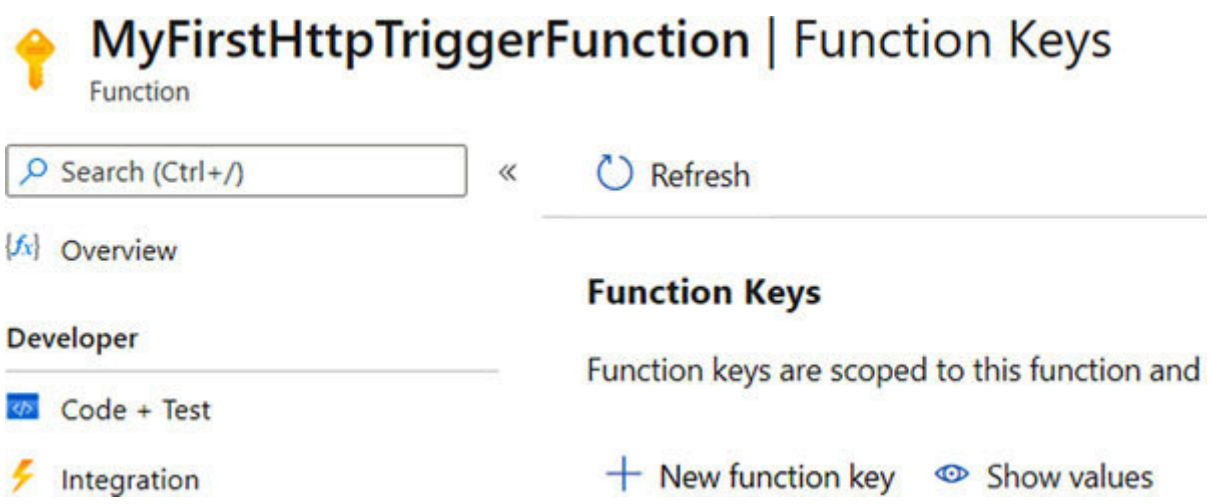


Figure 6.4: *New function key*

This will open a right pane asking for a name and value, as shown in *figure*

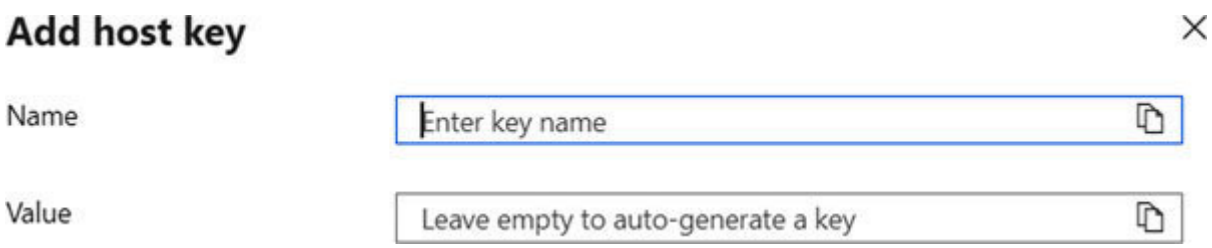


Figure 6.5: *Add host key*

Provide a name and value you want to use as the key. You can leave the value blank to get a system-generated value.

Click on

Renew an existing key

You can renew the host key, master key, and system key by following the given steps:

Click on the **Key** option next to the specific key, as shown in *figure*

Name	Value	
_master	Hidden value. Click to show value	Renew key value
default	Hidden value. Click to show value	Renew key value

Figure 6.6: *Renew key value*

It will ask for a confirmation, as shown in *figure*

Renew key value

The existing key value of default will be replaced by a randomly generated key.

Renew Cancel

Figure 6.7: *Renew key-value confirmation*

Click on

Azure active directory authentication

You can configure your Azure Functions to use the Azure active directory for the authentication. The advantage of using the Azure active directory authentications is the delegation of authentication to an external identity provider, which simplifies development and minimizes user administration. This implementation supports the federated identity design pattern. You can configure your function app for Azure AD authentication quickly following these steps:

Open your Function app in the Azure portal.

Click on the **/** option in the left menu of your function app, as shown in *figure*

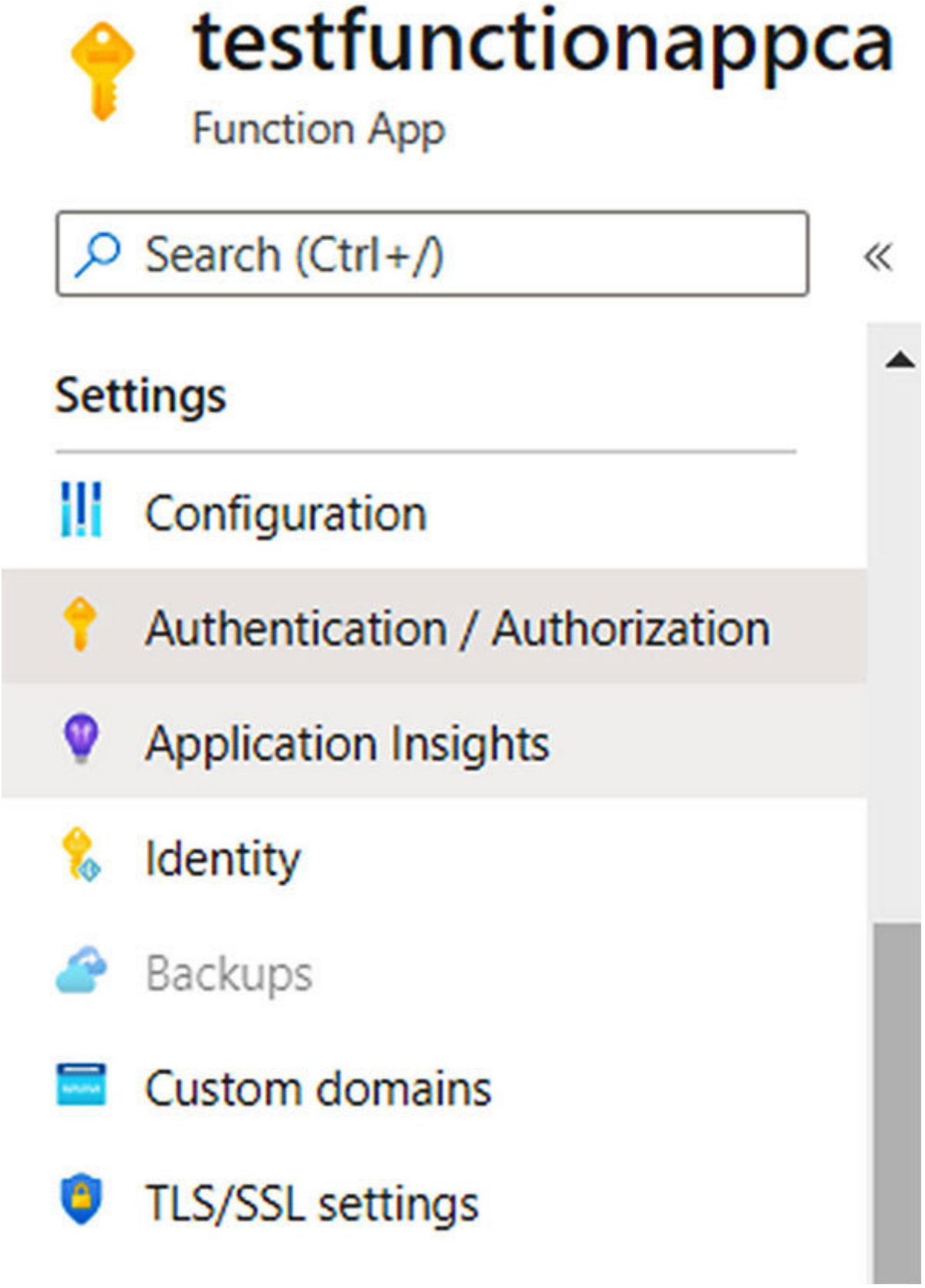


Figure 6.8: *Authentication/Authorization option in the menu*

This will open the **Authentication / Authorization** page with the default **App Service Authentication** disabled, as shown in *figure*

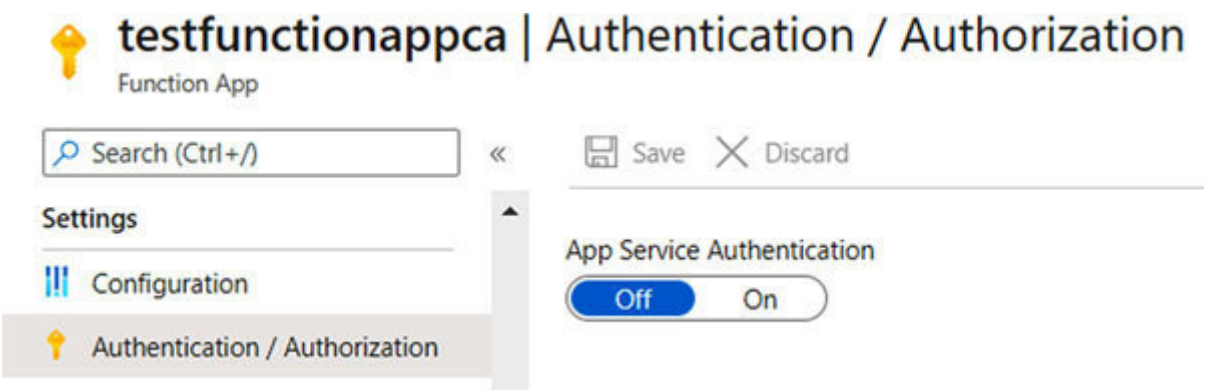


Figure 6.9: *Authentication/Authorization enabling the option*

Toggle the **On/Off** button to enable the authentication for your function app. Here you see multiple authentication provider options as shown in *figure*

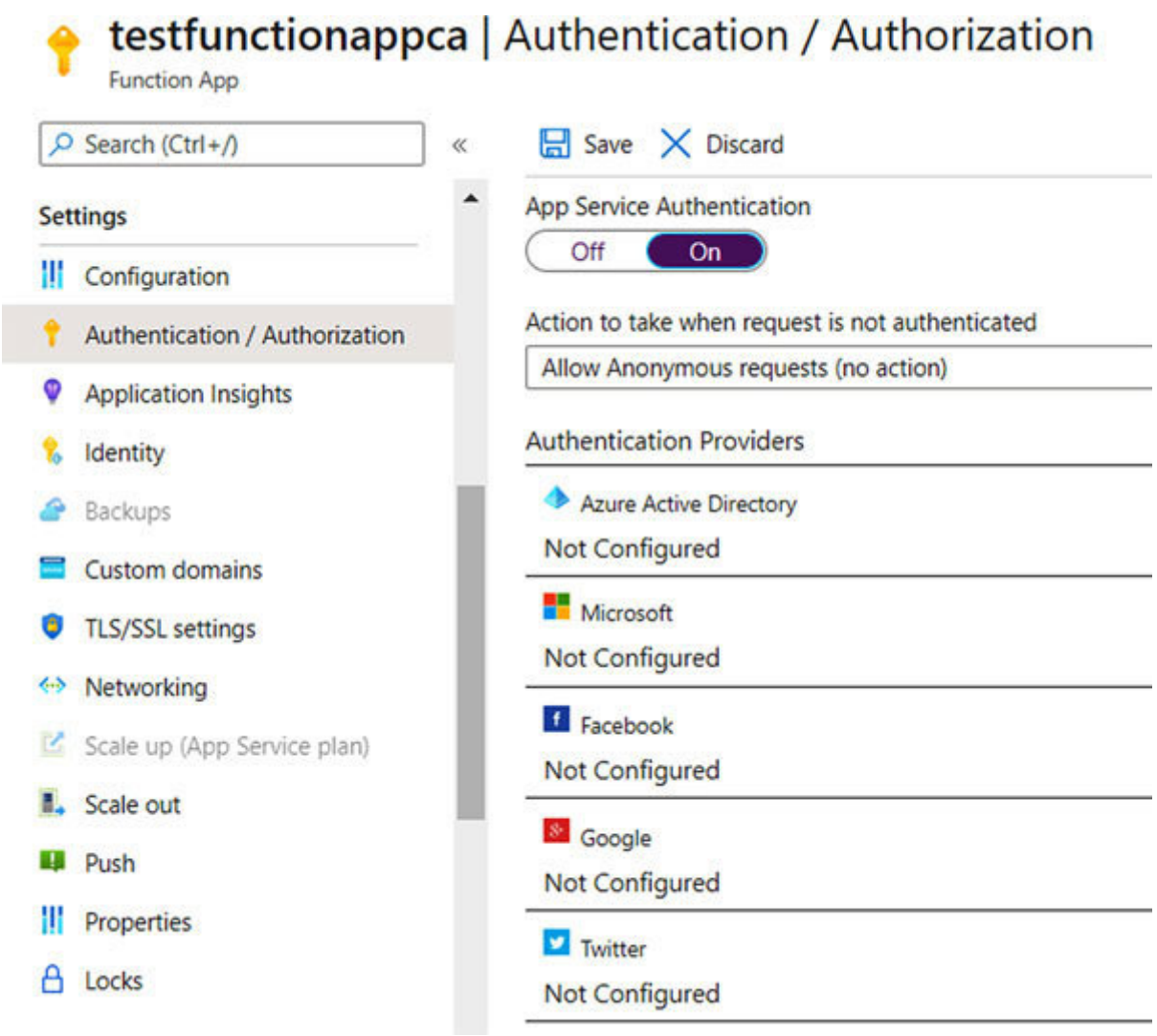


Figure 6.10: *Authentication/Authorization providers*

Click on **Active** to open the **Settings** page, as shown in *figure*

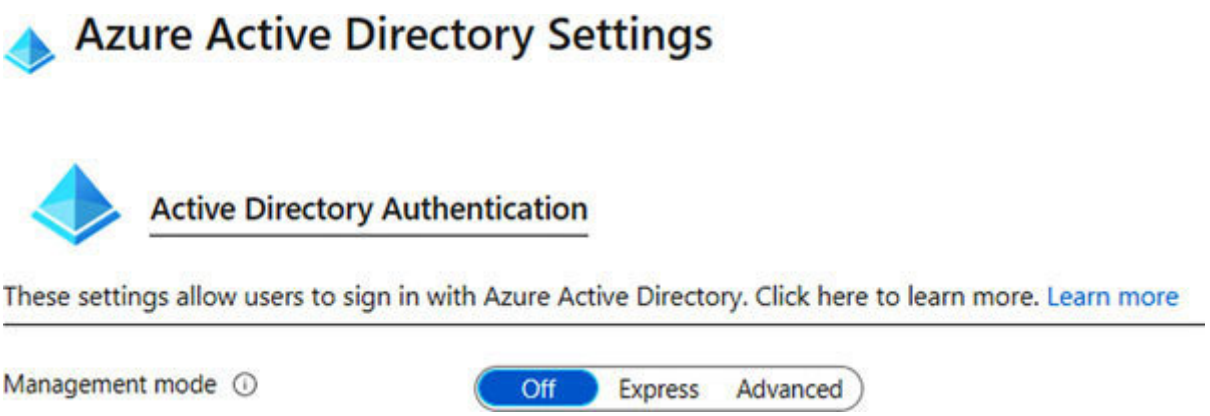


Figure 6.11: *Azure active directory management mode*

Click on to change the Management mode to express. This will provide options as shown in *figure*

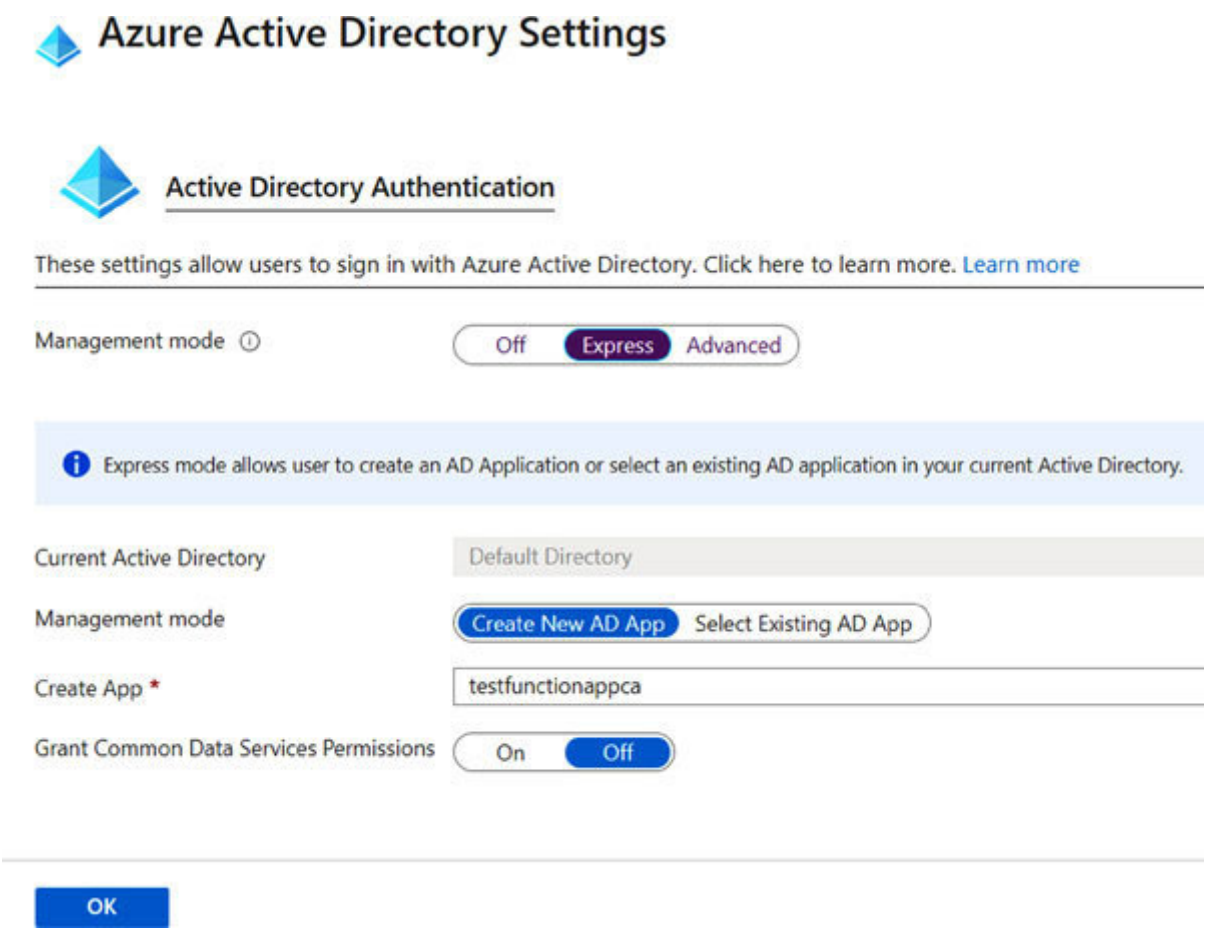


Figure 6.12: *Azure active directory settings*

You can choose to create a new AD app or select an already existing AD app.

Click on **OK** to register the app in the Azure active directory, and then click on

Configuring Azure active directory authentication advanced settings

To configure your function app for Azure AD authentication through advanced settings following the given steps:

Follow the steps for Express Settings as listed previously.

Click on to change the management mode to advanced. This will provide options as shown in *figure*

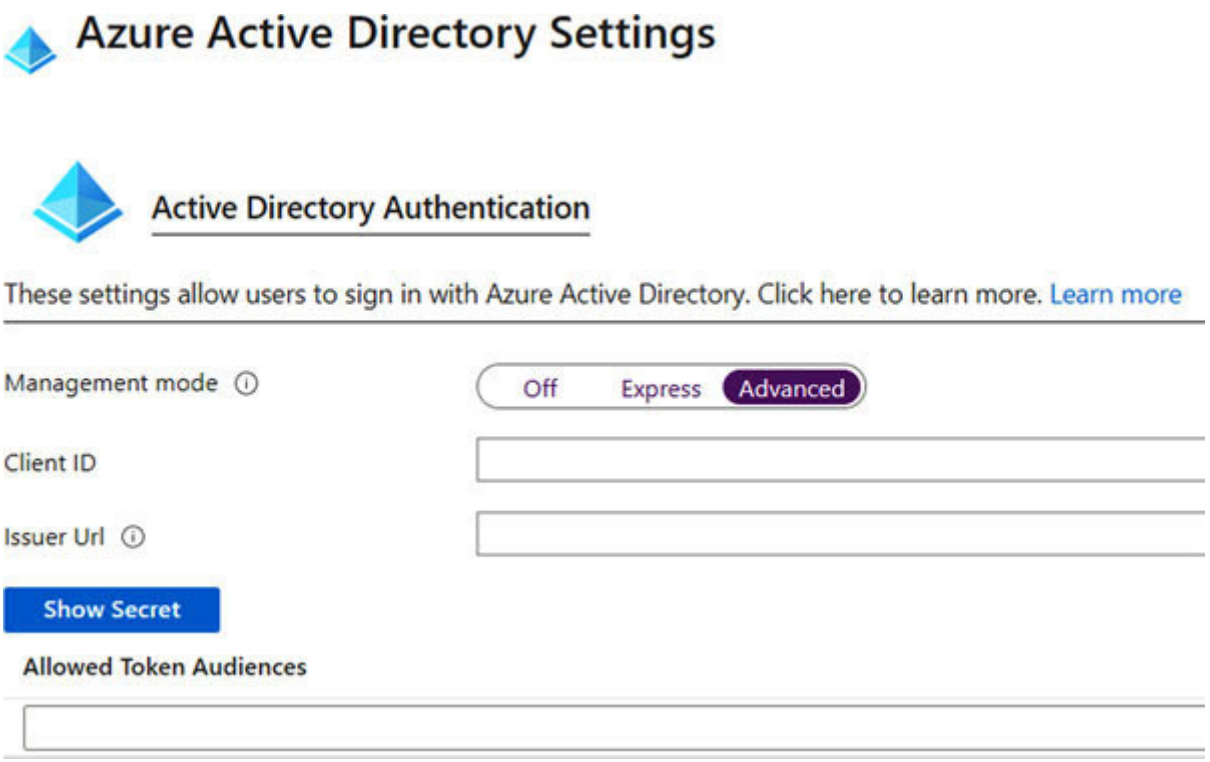


Figure 6.13: *Azure active directory advanced settings*

This will ask for the client ID and issue URL. To get these details, follow these steps:

Go to your Azure active directory in the Azure portal.

Click on **App Registration** and then click on **New** as shown in *figure*

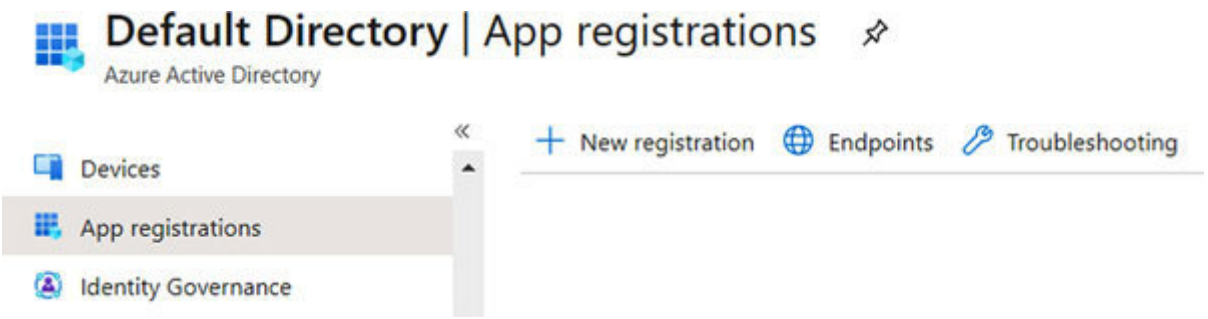


Figure 6.14: *Azure active directory app registration*

On the **an** page, provide an app registration name and choose who can access the app, as shown in *figure*

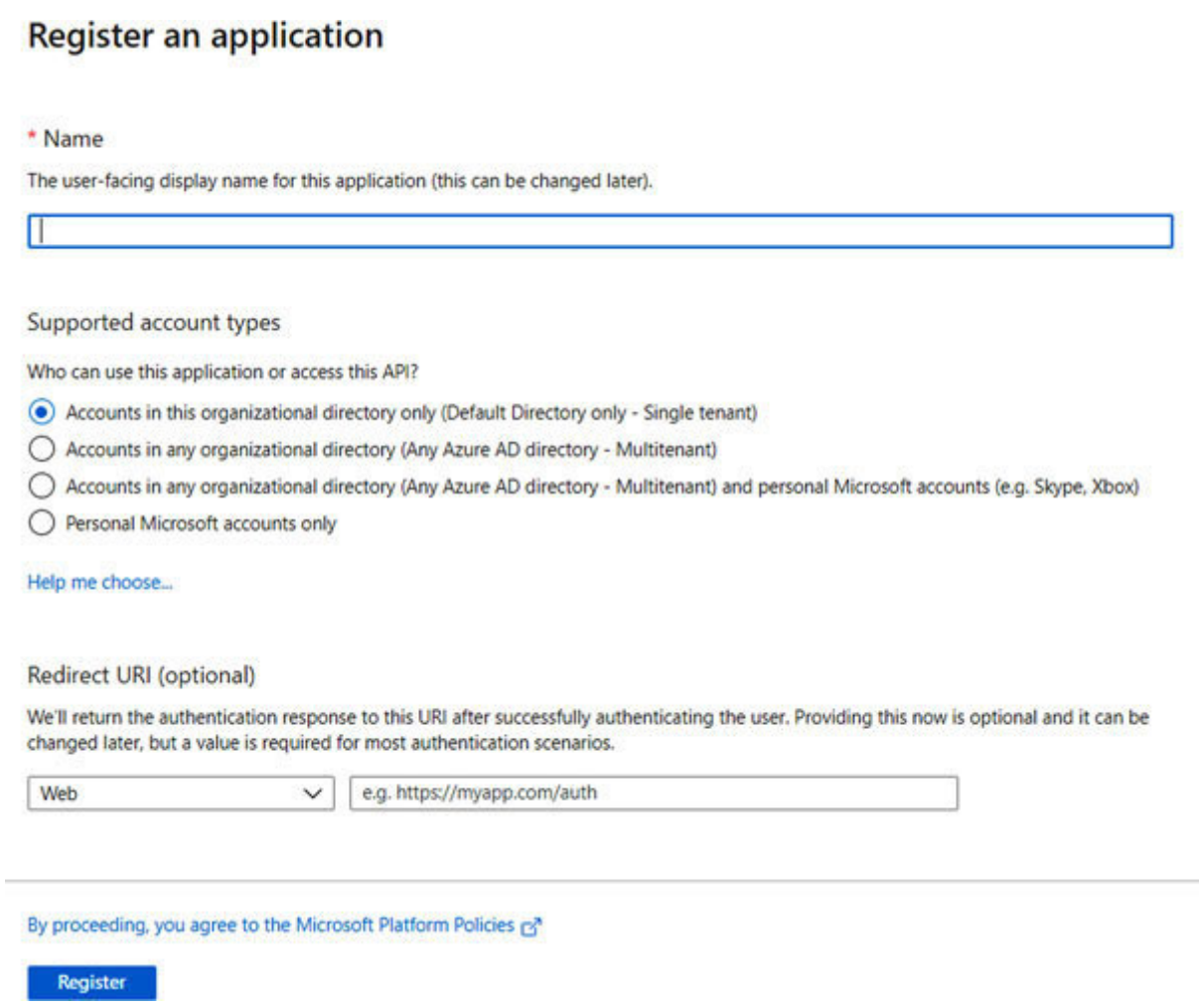


Figure 6.15: *Azure active directory – register an application*

In the redirect URI, select web and type **URL >/.auth/login/aad/** For example,

Click on

Once the app is registered, this will automatically take you to the overview section, as shown in *figure*

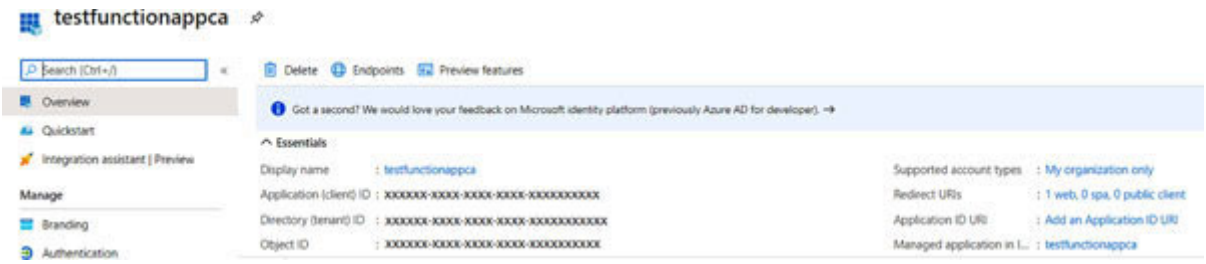


Figure 6.16: *Azure active directory app overview*

Copy and keep these details with you.

Fill in the details as copied in the preceding step.

Click on **OK** to register the app in the Azure active directory, and then click on

TLS/SSL certificate for function apps

Azure Functions allow you to configure your protocol settings to enable HTTPS only and set a minimum TLS version that helps to improve the security of your function app by transmitting information over an encrypted connection between the client and server. You can also add TLS/SSL bindings that let you specify which certificate to use while responding to the requests to a specific hostname.

Configuring protocol settings and TLS/SSL bindings

You can configure the protocol settings and TLS/SSL bindings by following the given steps:

Click on the TLS/SSL settings under Settings in the left menu of your function app, as shown in *figure*

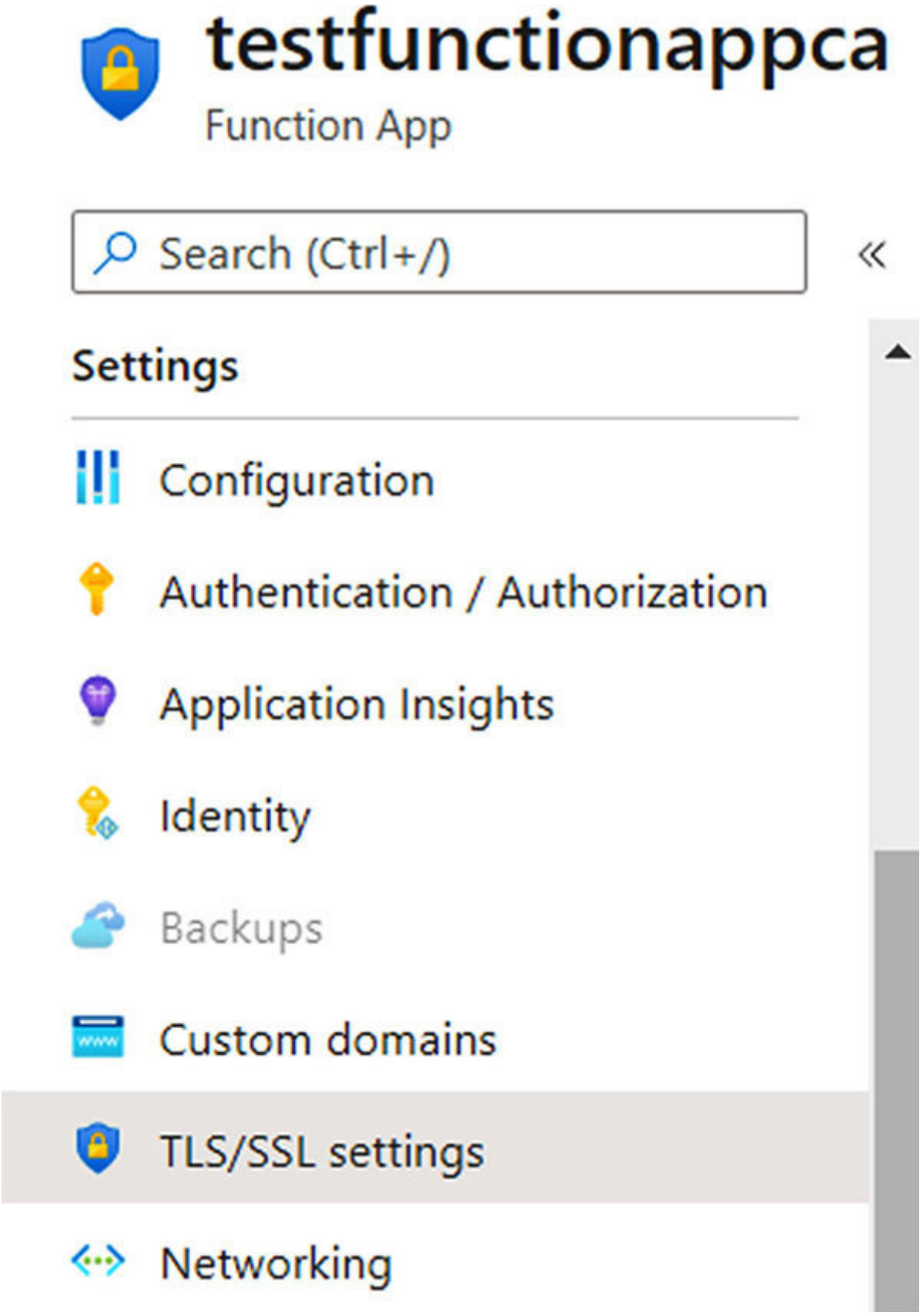


Figure 6.17: *Function app TLS/SSL setting menu options*

This will open the TLS/SSL settings page, as shown in *figure*

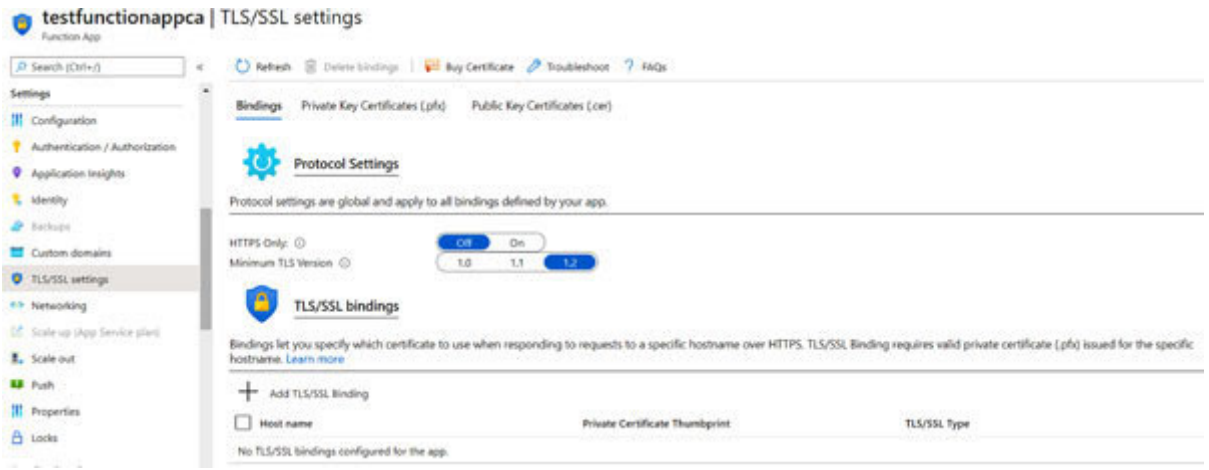


Figure 6.18: *Function app TLS/SSL setting pages*

On the TLS/SSL settings page, you can enable the **HTTPS Only** feature to redirect all HTTP traffic to HTTPS, set minimum TLS version, or add binding by clicking on the **TLS'SSL** option.

Note: TLS/SSL binding requires a valid private key certificate (.pfx) for the hostname.

Managing private key certificates

Private key certificates can be used for TLS/SSL bindings and loaded to the certificate store for your app to consume. Azure functions provide the options for private key certificates given as follows:

Buy certificates

You can buy a new certificate by clicking on the option on the TLS/SSL settings page of your function app, as shown in *figure*

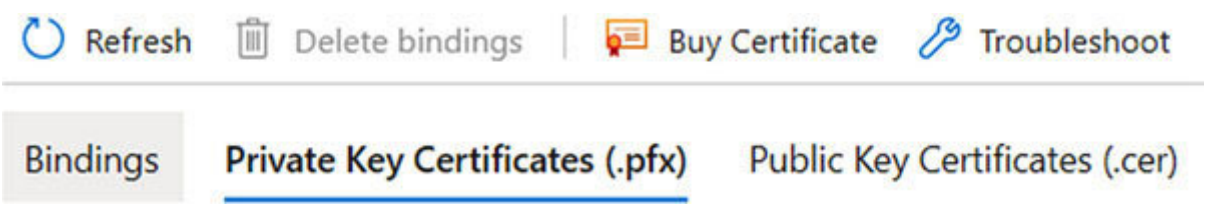


Figure 6.19: *Function app buy TLS/SSL*

It allows you to purchase the certificate from GoDaddy, manage SSL certificates quickly, and provide the following features:

Secure one web or function app and its sub-domains (wildcard SSL)

One year validity and auto-renewal

A **domain-validated** certificate with an easy and intuitive validation process

Maintains the certificate in Azure key vault

SHA-2 and 2048-bit encryption - the strongest on the market

Compatible with all major browsers

The certificate can be imported to any function app or web app. It can be exported and used in any services or directly from the KeyVault secret it is stored.

Import app service certificate

You can import the newly purchased certificate from the option or an already existing certificate through the **App Service** option, as shown in *figure*

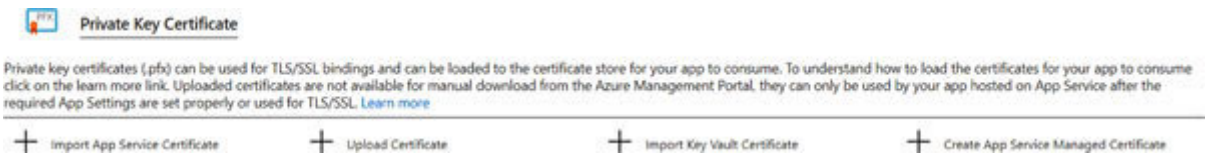


Figure 6.20: *Function app import app service certificate option*

When the import is done, you can see the certificate in the private key certificates list.

Upload certificate

You can choose to obtain a certificate from an external certificate provider and upload the same to your function app.

Perform the following steps to upload the certificate manually:

Click on the option.

This will open a right pane, as shown in *figure*

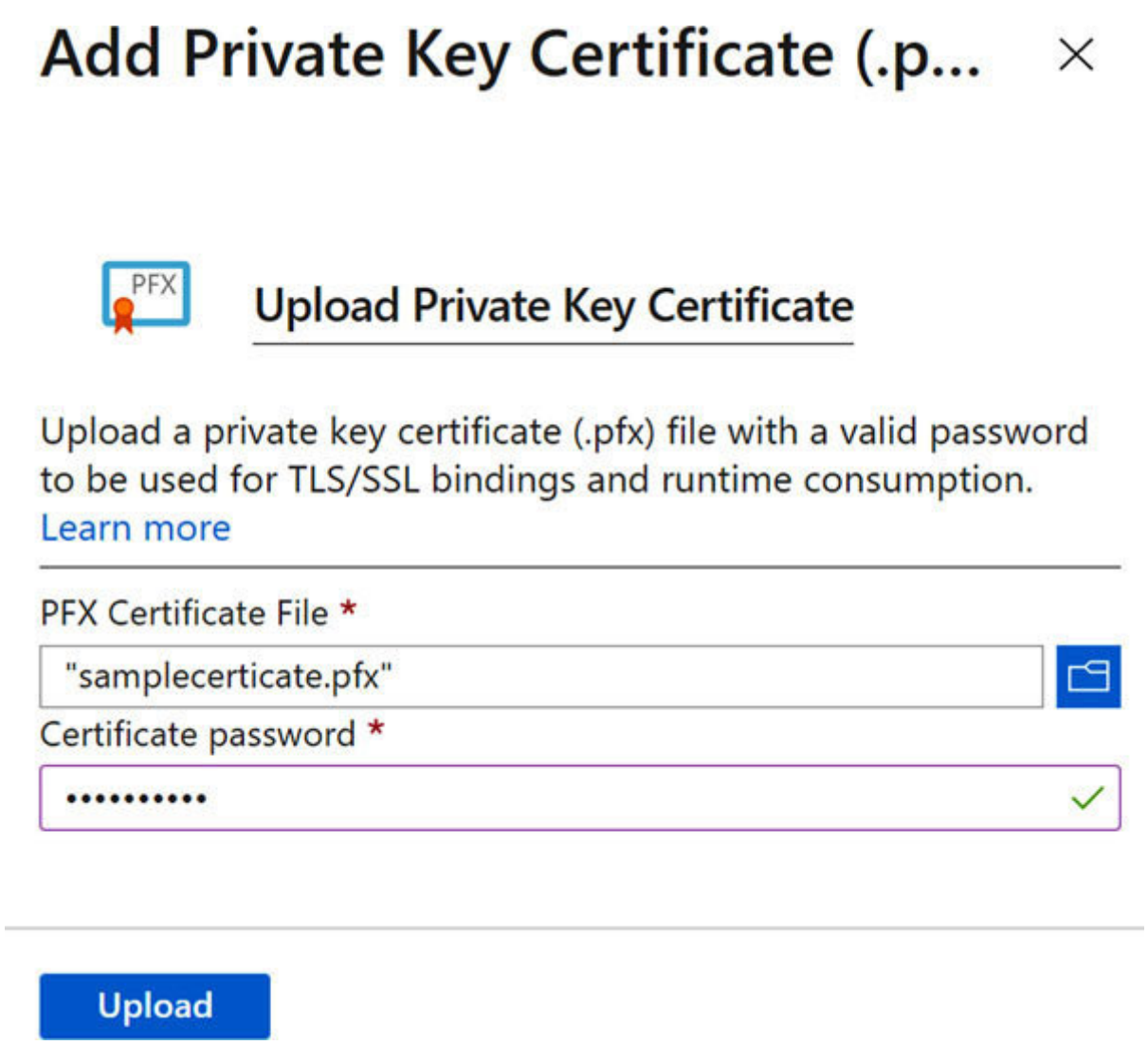


Figure 6.21: *Add private certificate pane*

Select your PFX file and input the password that you created while exporting the PFX file.

Click on **Upload.**

Create app service managed certificate

App service managed certificates provide you an option to create a certificate that can be used for SSL binding for the selected

sub-domain. These certificates are free of cost and fully managed by app service. However, there are few limitations with the app service managed certificates listed as follows:

They support only sub-domain hostnames and not the naked domains

They do not support wildcard certificates

They do not support A type records in the DNS

Note: If your function app is on the app service plan, make sure that it is not in the F1 or D1 tier. Custom SSL is not supported in these tiers.

Uploading public key certificates

You can upload a public certificate to your function app by following the given steps:

Go to **Key Certificates** on your page.

Click on the **Public Key** option.

This will open a right pane, as shown in *figure*

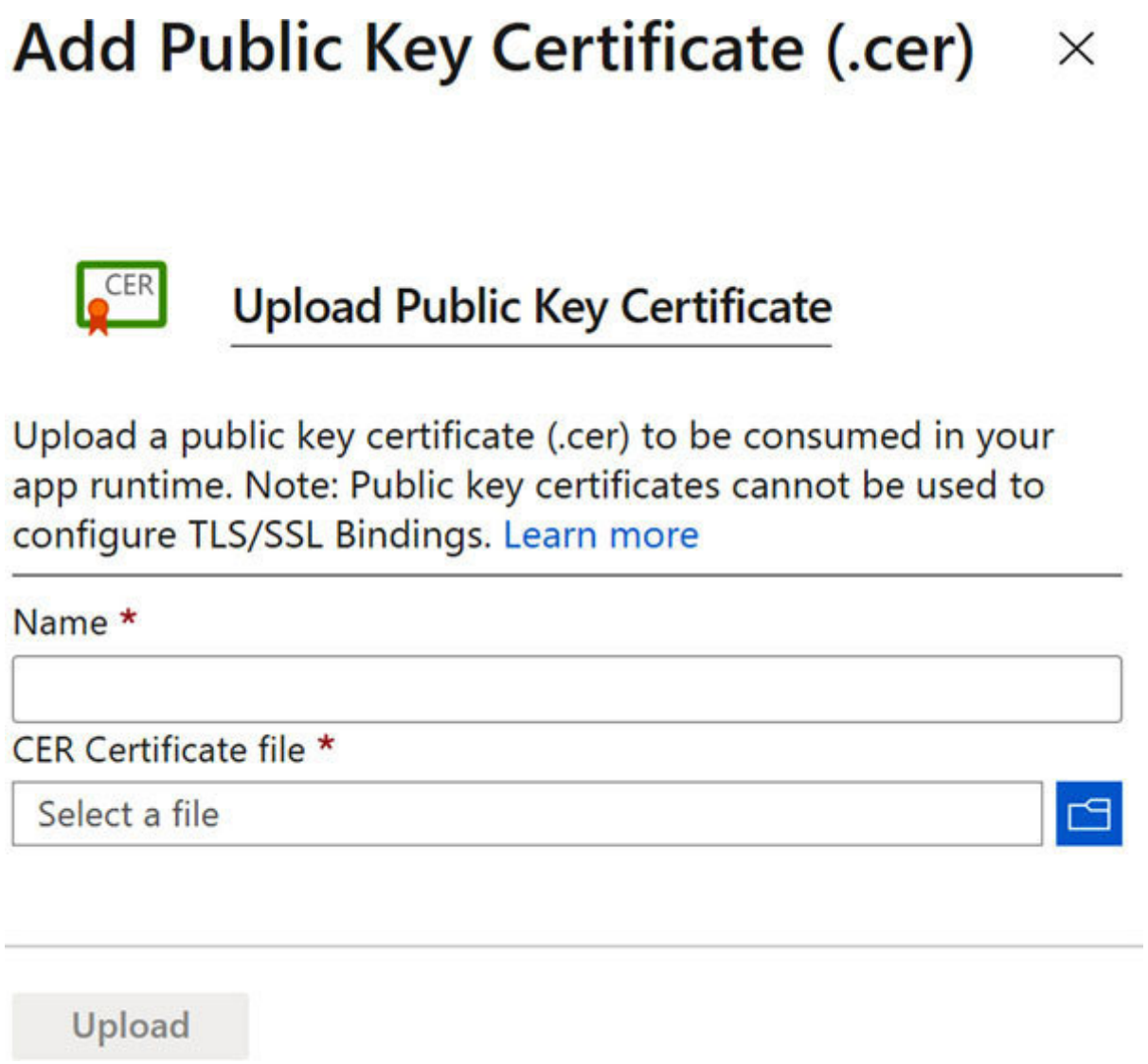


Figure 6.22: *Add public certificate pane*

Input the name and select your CER file.

Click on

Configuring auto-renewal settings

You can configure the auto-renewal settings of your certificate to avoid the consequences of the expiry. To set auto-renewal, follow these steps:

Search App Service Certificates in the portal search.

Select the certificate you want to set auto-renewal for from the list on the App Service Certificates page.

Click on **Auto Renew Settings** under **Settings** in the left menu.

Toggle **Renew App Service** to

Click on

This will auto-renew the certificates 60 days before the expiry. By default, app service certificates have a one-year validity period.

Using managed identities for functions

Azure managed identities allow the function to access the Azure active directory protected resources without storing and managing secrets. Azure provides two types of managed identities:

A system-assigned identity, which is associated with the function app and gets deleted with the function app. A function app can have only one system-assigned identity.

A user-assigned identity is a separate Azure resource assigned to the function app. A function app can have multiple user-assigned identities.

Enabling the system-assigned identity

To enable the system-assigned identity for your function app, follow these steps:

Open your function app in the Azure portal.

Click on the option in the left menu of your function app, as shown in *figure*

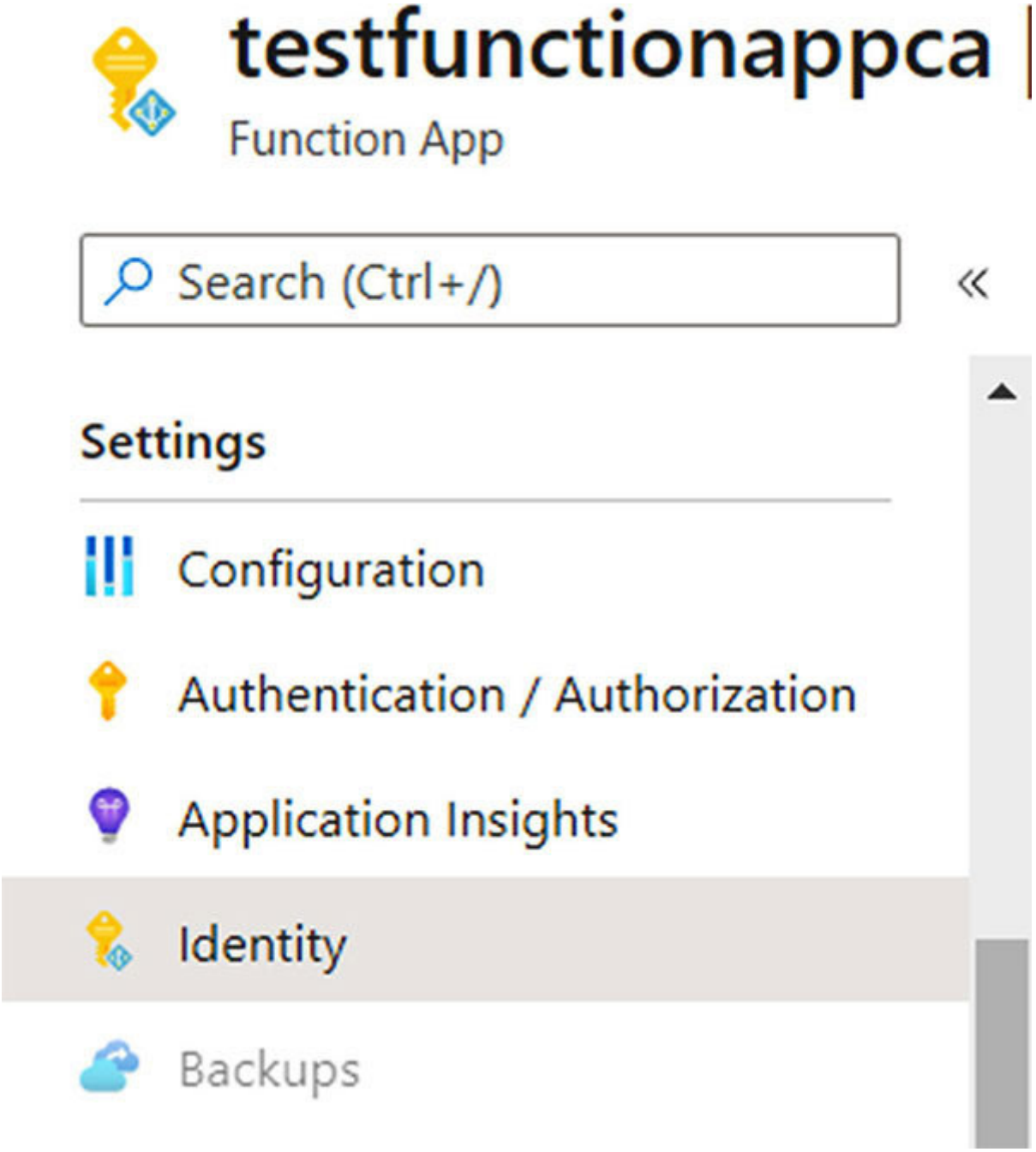


Figure 6.23: *Identity option in the menu*

This will open the identity page with the default status as disabled, as shown in *figure*

Figure 6.24: *Function app identity page*

Toggle the status button to enable the system-assigned identity for the function app.

You can also use the following PowerShell command to enable the system-assigned identity by providing the function app name and resource group:

Set-AzWebApp -AssignIdentity $true -Name $functionappname -ResourceGroupName $resourceGroupName

To enable system-assigned identity through Azure CLI, use the following command:

az webapp identity assign --name --resource-group

You can also enable the system-assigned identity during the resource created through the ARM template by adding the following property in the resource definition:

```
"identity": {
"type": "SystemAssigned"
}
```

Creating the user-assigned identity

To enable the user-assigned identity for your function app, follow these steps:

Open your **Function app** in the Azure portal.

Click the option in the left menu of your function app, as shown in *figure*

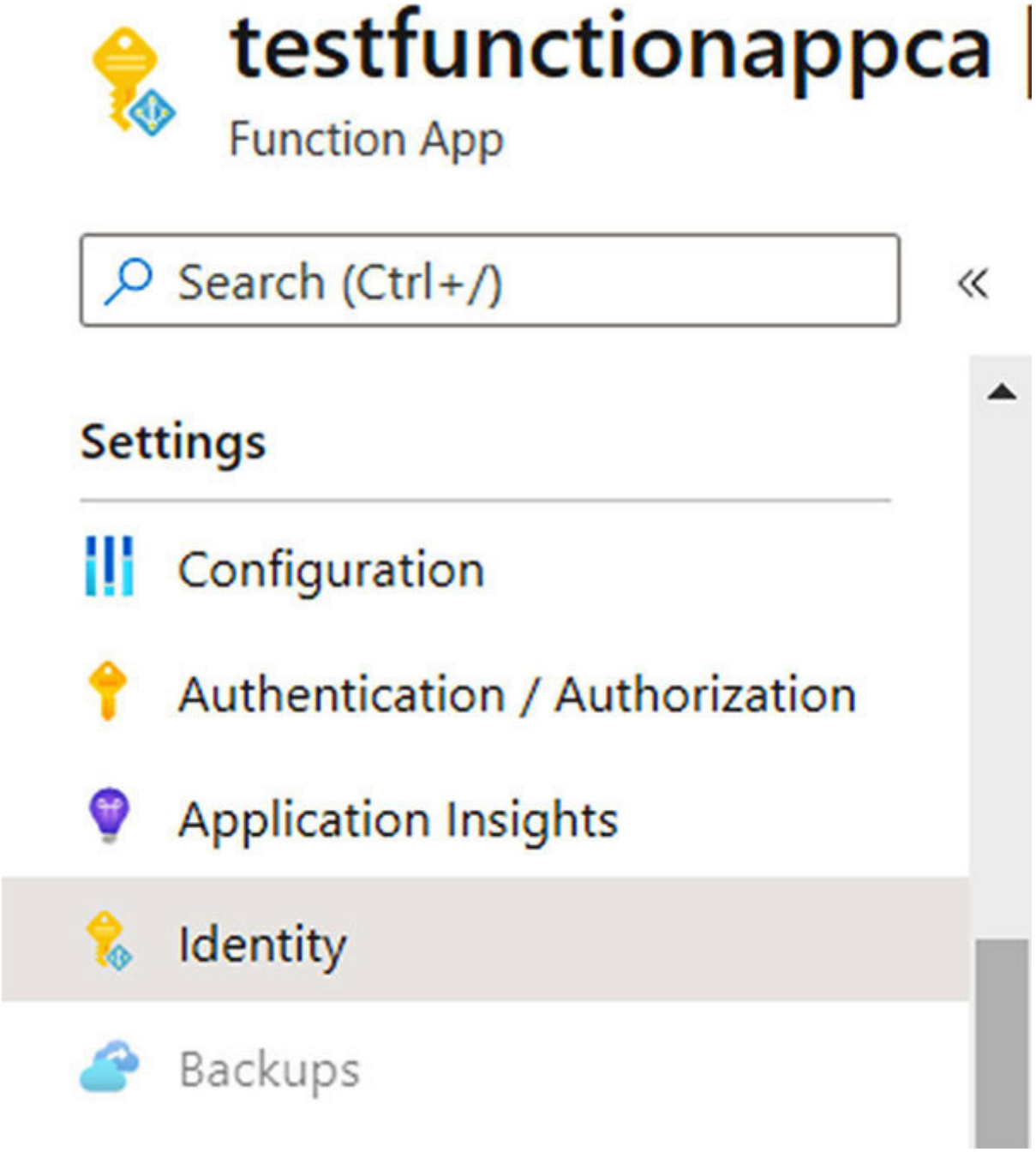


Figure 6.25: *Identity option in the menu*

This will open the **Identity** page with tabs for and

Click on the tab and then click on the option, as shown in *figure*

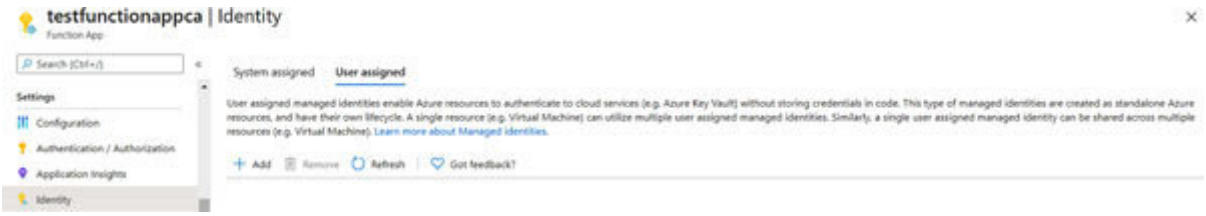


Figure 6.26: *User assigned identity*

Select the **User assigned** identity.

Click on

You can use the following PowerShell command to create a new user-assigned identity by providing the resource group:

$userAssignedIdentity = New-AzUserAssignedIdentity -Name $userAssignedIdentityName -ResourceGroupName $resourceGroupName

To create a user-assigned identity through Azure CLI, use the given command:

az identity create -g GROUP> -n ASSIGNED IDENTITY NAME>

You can also create the function app with the user-assigned identity during the resource creation through ARM template by adding the following property in the resource definition:

"identity": {
"type": "UserAssigned",
"userAssignedIdentities": {

"": {}
}
}

The here is the resource ID of the desired identity.

Network security

To keep your Azure Function production-ready, you need to secure them so that unauthorized access can be reduced. In today's modern era, securing your application is one of the foremost critical assignments as information breaches may happen. So, you need to secure your written Azure Functions. Azure functions give you a lot of options with an easy-to-use setup. Network security focus on the network protocols, ports, allowed or denied connected services.

Use Case 1: Integrating Azure Functions with an Azure virtual network

In this use case, your Azure Function should be able to connect to the internet as well as a resource in the Azure virtual network. Before starting the actual implementation of the use case, let's understand the serverless solution for this by referring to *figure*

Create a function app in the premium plan.

Create a virtual machine in a virtual network without a public IP address.

Connect the function app to the virtual network.

Create a function proxy to access resources of the virtual network from inside the virtual network.

Figure 6.27: *Integrate Azure Function with an Azure virtual network*

Creating a function app in the premium plan

The first step is to create a function app resource from the menu or Azure home page. While creating a function app, a select premium plan. You can choose an existing storage account or create a new storage account. You will need this storage account in the next step, 'select diagnostics storage account.' Please refer *figure*

Home > Function App >

Create Function App

Hosting

Storage (New)	
Storage account	storageaccountvneta841f
Plan (New)	
Plan type	Premium
Name	ASP-VnetAF-8430
Operating System	Windows
Region	Central US
SKU	Elastic Premium
Size	Small
ACU	210 total ACU
Memory	3.5 GB memory

Create < Previous Next >

Figure 6.28: *Select premium function app plan while creating*

We have already checked the detailed steps to create the Azure function using Visual Studio in *Chapter 3: Creating your first Azure*

Create a virtual machine in a virtual network without a public IP address

After creating a function app with a premium plan, you need to add your existing virtual machine to the virtual network or create a new Azure resource of type virtual machine from the menu.

While creating a virtual machine, you must select your subscription and provide virtual machine instance information like name, username, password, resource group, region, size, an image in the basic tab.

Now, navigate to the networking tab and create a new virtual network. While creating a virtual network, you must provide information like virtual network name and address range, subnet name, and address range. The subnet size (10.10.1.0/24) defines how many interfaces can be added to the subnet. Your IP address/24 subnet will provide you 254 host addresses.

Now, select the public address value as so that the virtual machine will have a private IP address only and can be accessed by the resource present in the same virtual network.

Please refer to *figure*

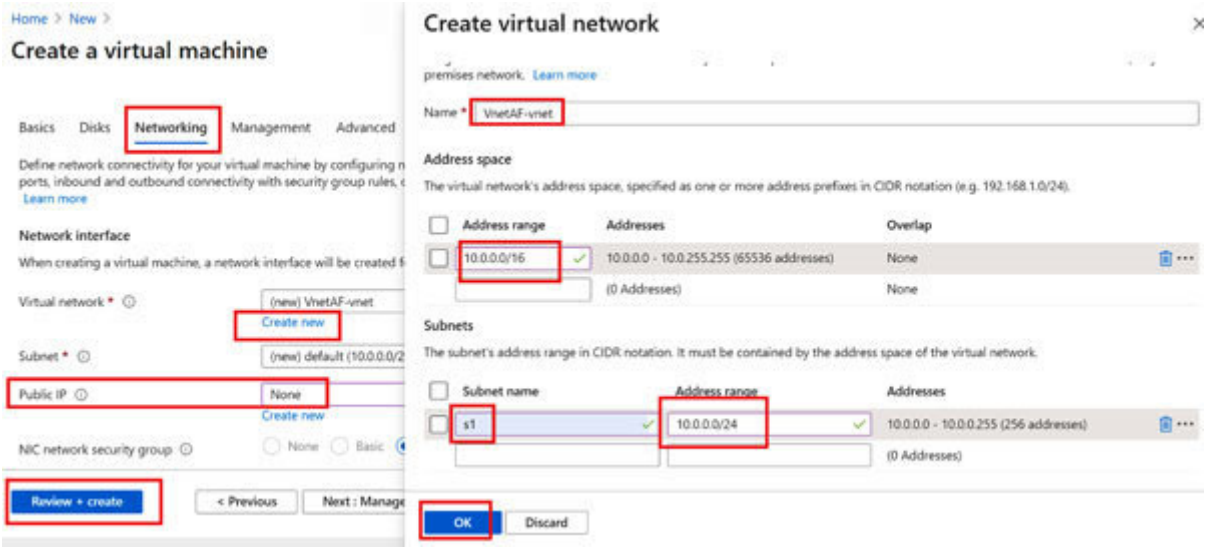


Figure 6.29: *Create virtual network and subnet while creating a virtual machine*

After creating a virtual machine, you can confirm the networking information like public and private IP address in the properties section of a virtual machine. Please refer to *figure*

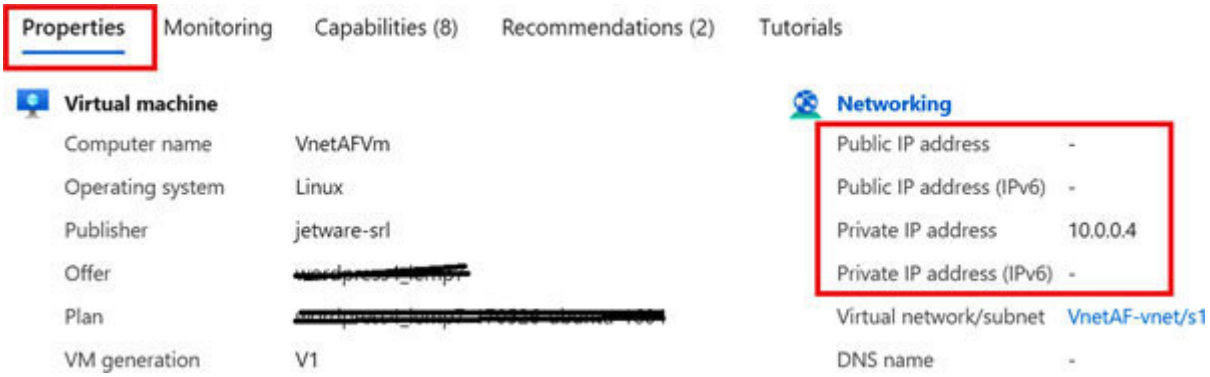


Figure 6.30: *Virtual Machine accessible on a private network only, not public*

Connecting the function app to the virtual network

To connect your previously created function app to a virtual network, you need to search networking in the left menu of the function app. You have multiple options in the function app to manage your networking requirements. As you have created a function app with a premium plan, you can add a VNet integration.

Please refer to *figure*

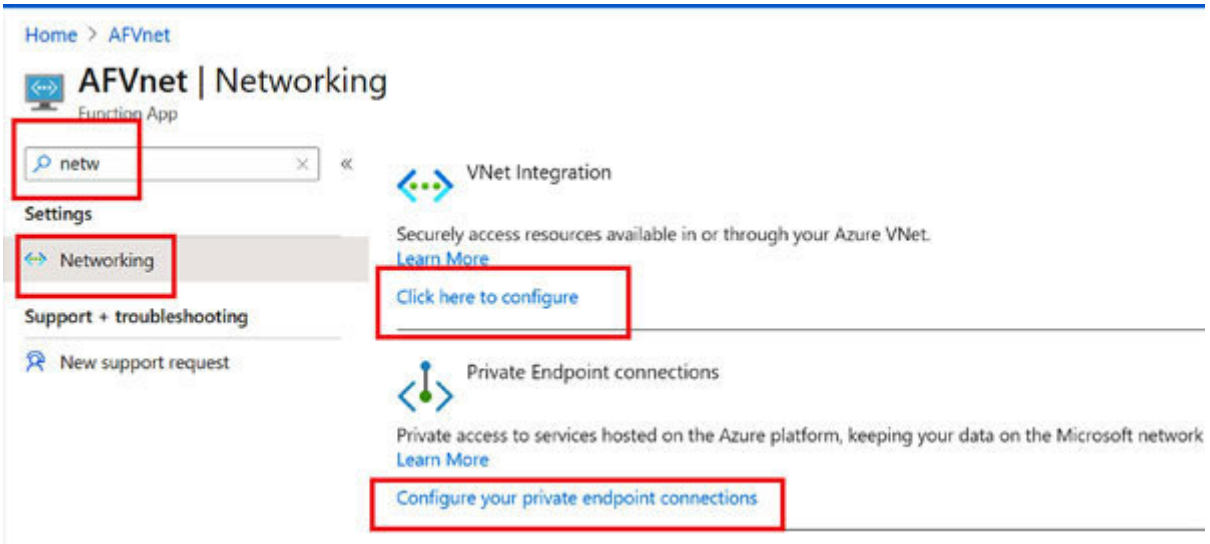


Figure 6.31: *Add VNet integration*

Click on **VNet** In the VNet configuration, click on **Add** In **Network Feature** use the settings shown in *figure* You must choose the same address block used by your virtual machine. You should only have one address block defined.

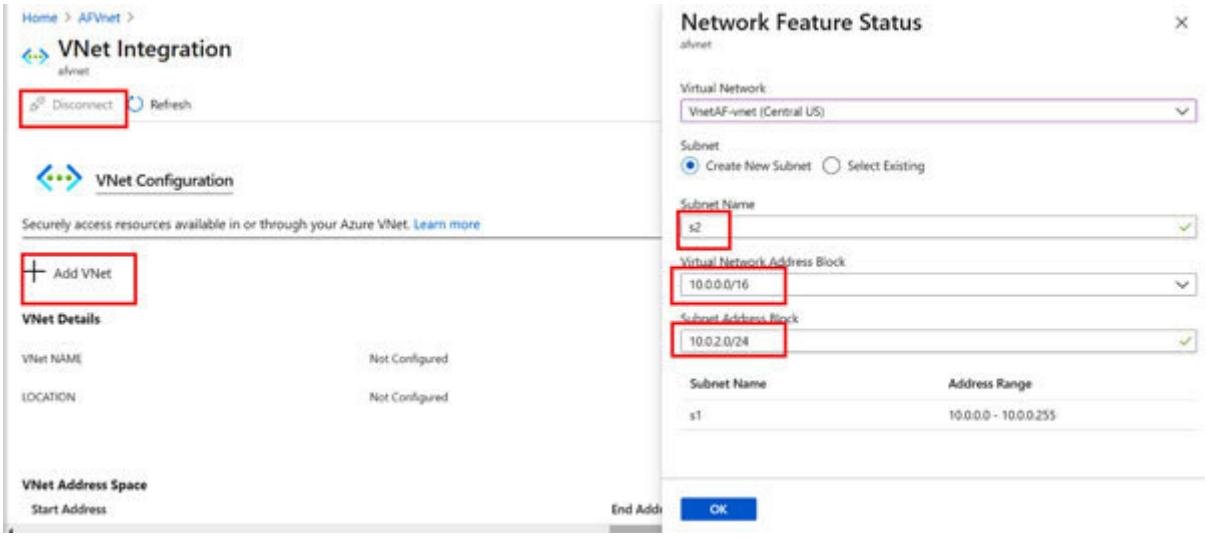


Figure 6.32: *Create virtual network and subnet while creating a virtual machine*

You can also verify the VNet integration, as shown in *figure* Click on **Disconnect** to remove the VNet integration.

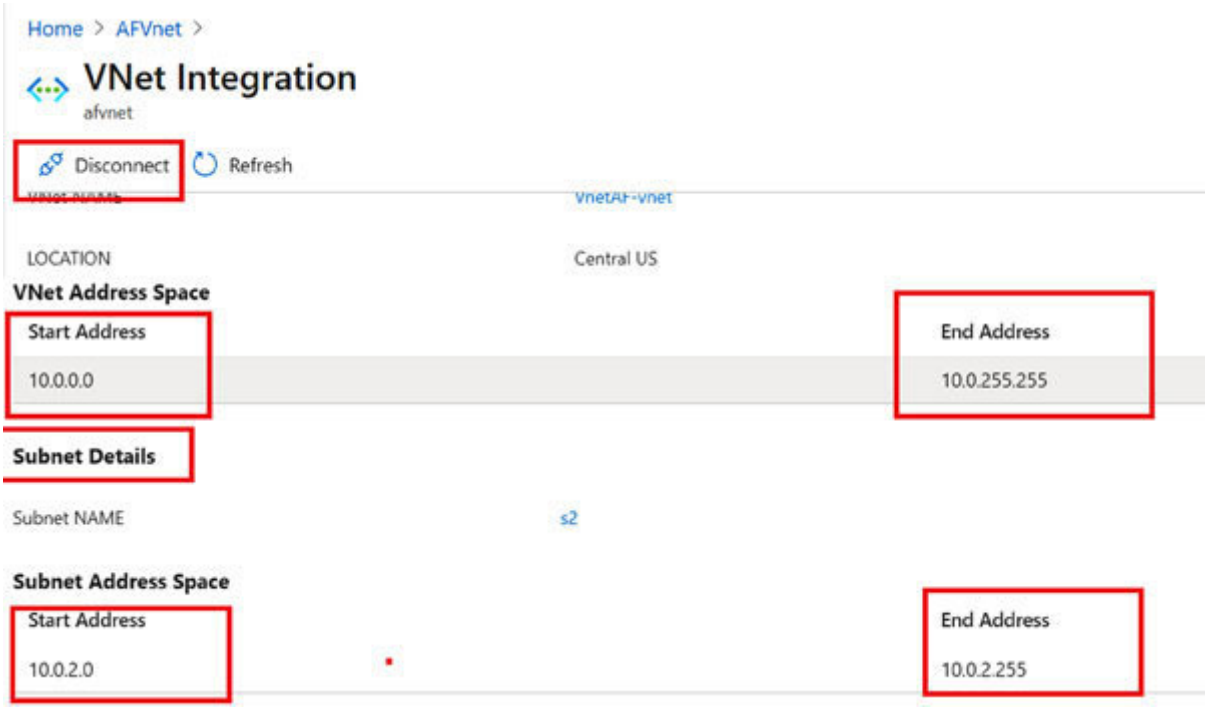


Figure 6.33: *VNet integration details*

Now your function app should be able to access the virtual network and the resource inside it.

Creating a function proxy to access resources of the virtual network from inside the virtual network

With the VNet integration empowered, you'll create a proxy in your function app to forward requests to the VM running within the virtual network.

In your function app, select **Proxies** from the left menu to add proxy. Proxy has settings like name, route template, and backend URL. Backend URLs can be used to access the resources of a virtual machine. Please refer proxy setting in *figure*

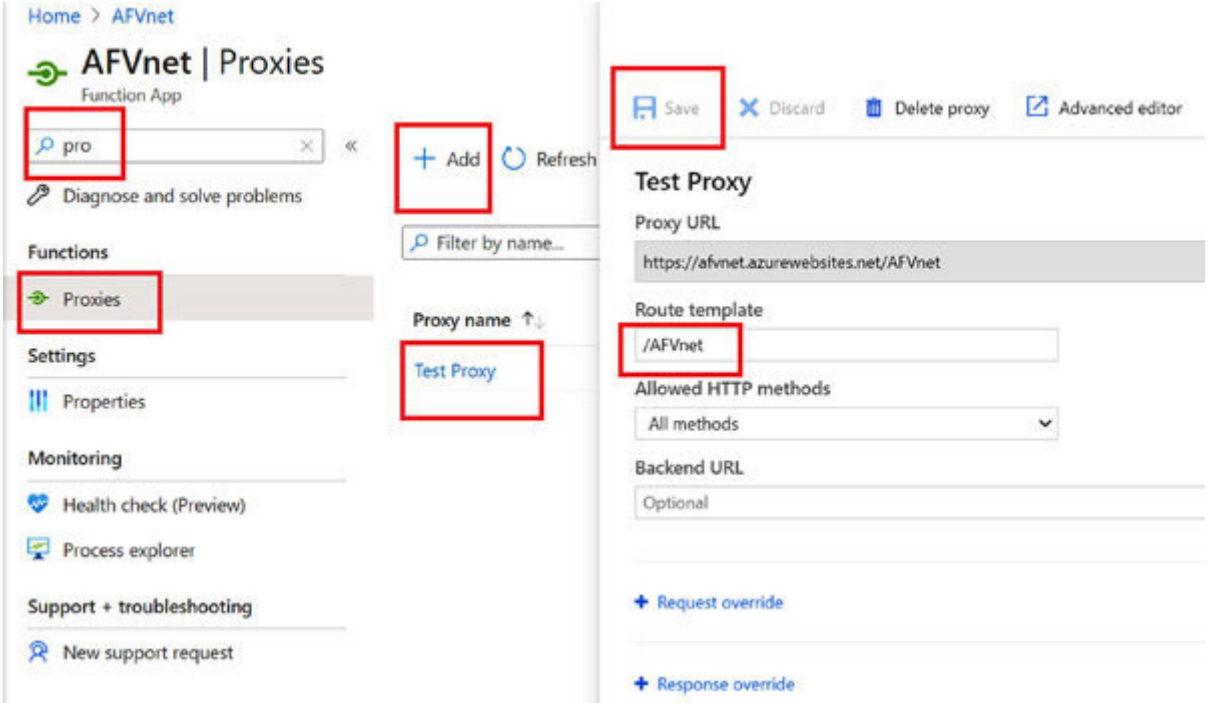


Figure 6.34: *Set up proxy*

In this use case, you understood the concept of accessing the resources by adding virtual network integration between function app and virtual network. This same use case can be applied to multiple scenarios like API usage among the other Azure resources.

Enabling CORS for Azure Function

Cross-Origin Resource Sharing allows JavaScript code running in a browser on an external host address to connect with your backend. Let's say you want to allow your **https://localhost:8080** locally running application to access the backend. In that case, you need to specify the external host record that should be permitted to establish cross-origin calls (For example, To permit all external hosts, add a record with the value "*" and remove all other roots from the list.

To set CORS for your Azure Function, search and navigate to the CORS section of Azure Function in the Azure portal. Now, you can add and save hostnames in the allowed origins sections. Please refer to *figure*

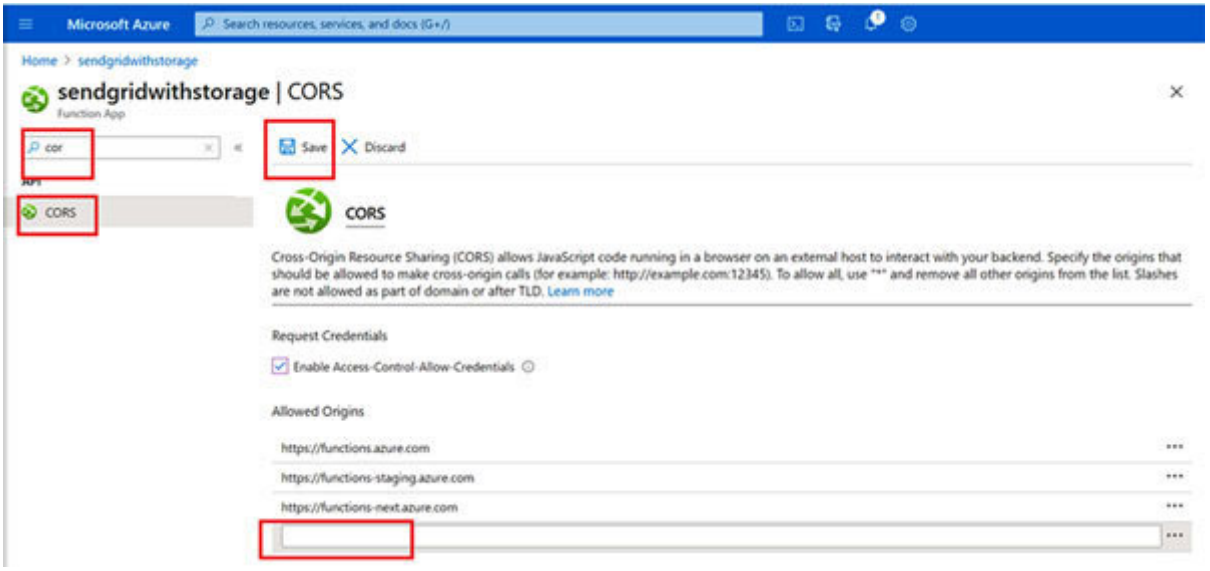


Figure 6.35: *Configure CORS in Azure Functions*

With these settings, you should successfully enable CORS for your Azure Functions and make it accessible for the host provided.

Enabling access restrictions

In the networking section of the Azure function app, you can configure additional security and restrictions using IP whitelisting. IP whitelisting is an access control security feature that permits few recognized entities to get specific applications and services. To add access restriction, click on the **Networking** section of the function app. Please refer to *figure*

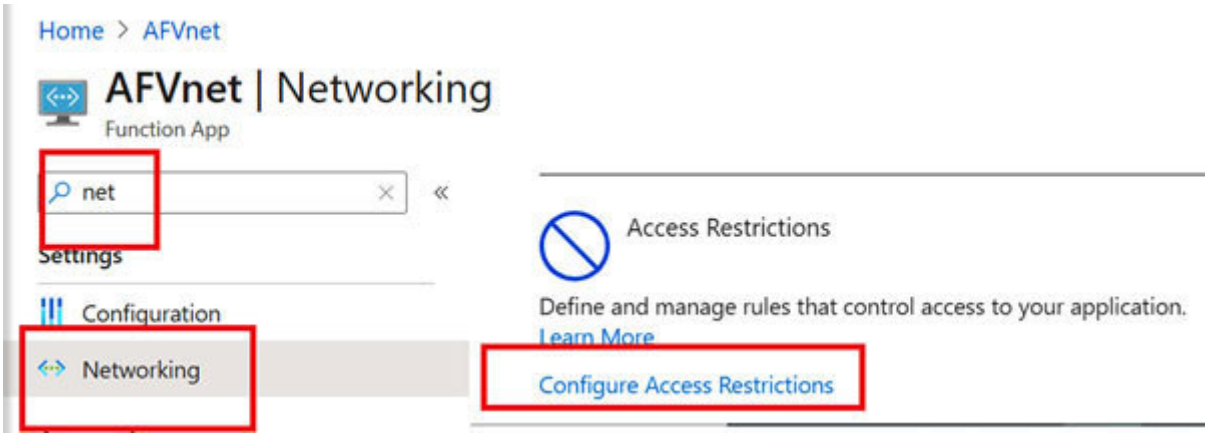


Figure 6.36: *Configure access restriction in Azure Functions.*

To add access restriction, click on **Add** and provide values like rule name, priority value, action, and IP address block. As soon as a rule is created, it will be added to the existing list. Please refer to *figure*

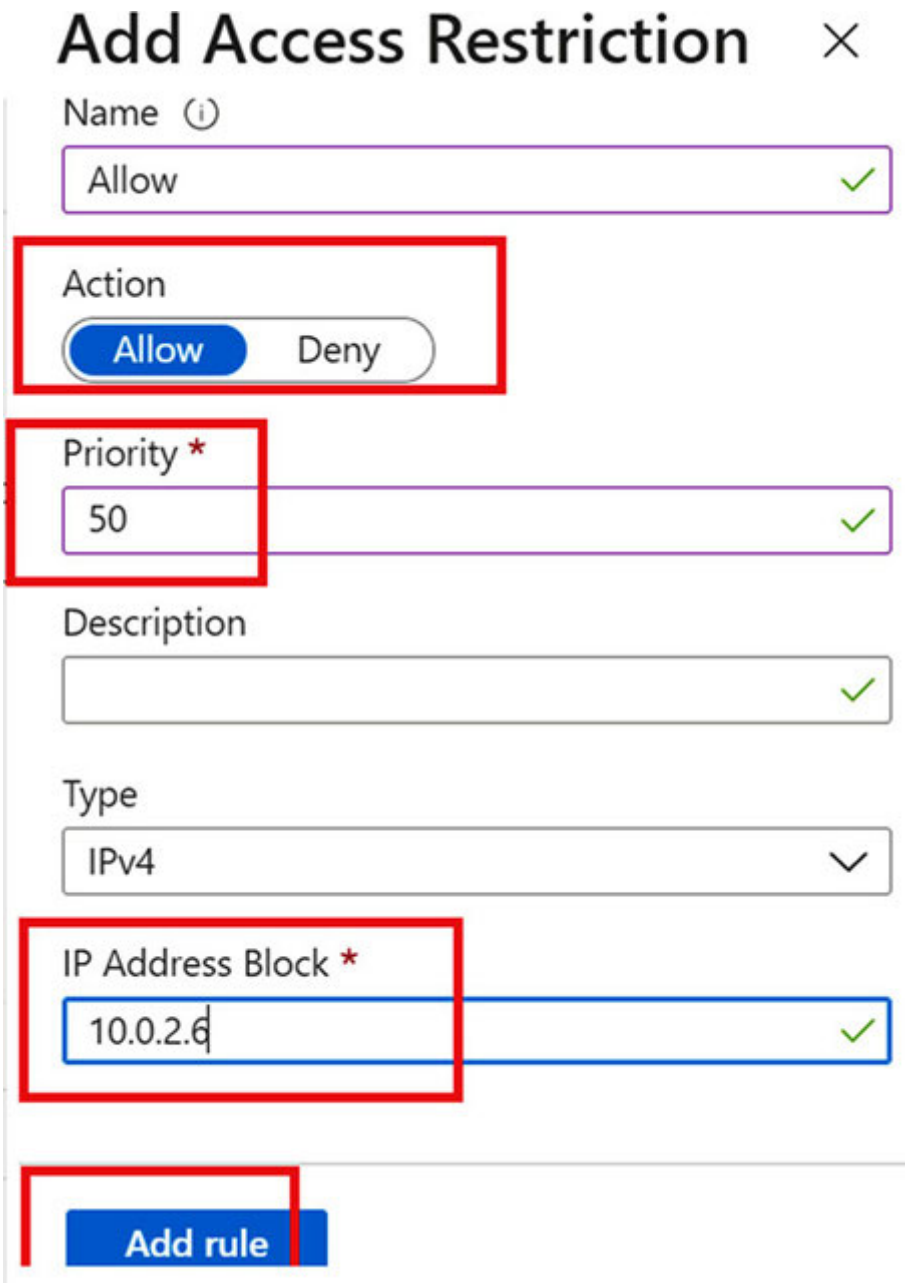


Figure 6.37: *Add access restriction rule*

If you want to restrict function app access to a limited IP address or organizational applications, you add access rules with allowing or deny actions. These rules act like instructions that are executed by given priority value. Please refer to *figure*

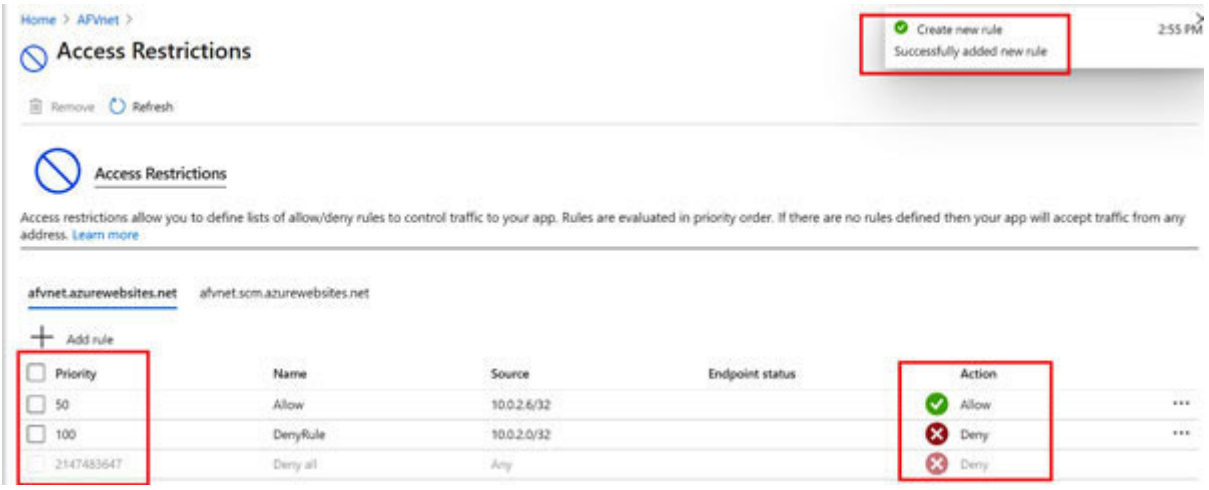


Figure 6.38: *Access restriction list with action and priority*

If you try to access the function by URL from a blocked IP address, you will get a 403 forbidden error. These rules will ensure security by allowing and blocking the IP addresses.

Private endpoint connections

As the industry proceeds to receive serverless (and **Platform as a** arrangements, they frequently require a way to coordinate with existing assets and services. These existing assets can be databases, files, or REST APIs. While combining serverless and PaaS offerings with conventional ways was complex, you can add a private endpoint in the networking section of the Azure function, as shown in *figure*

Click on **Add** and provide required information like name, subscription, virtual network, and subnet.

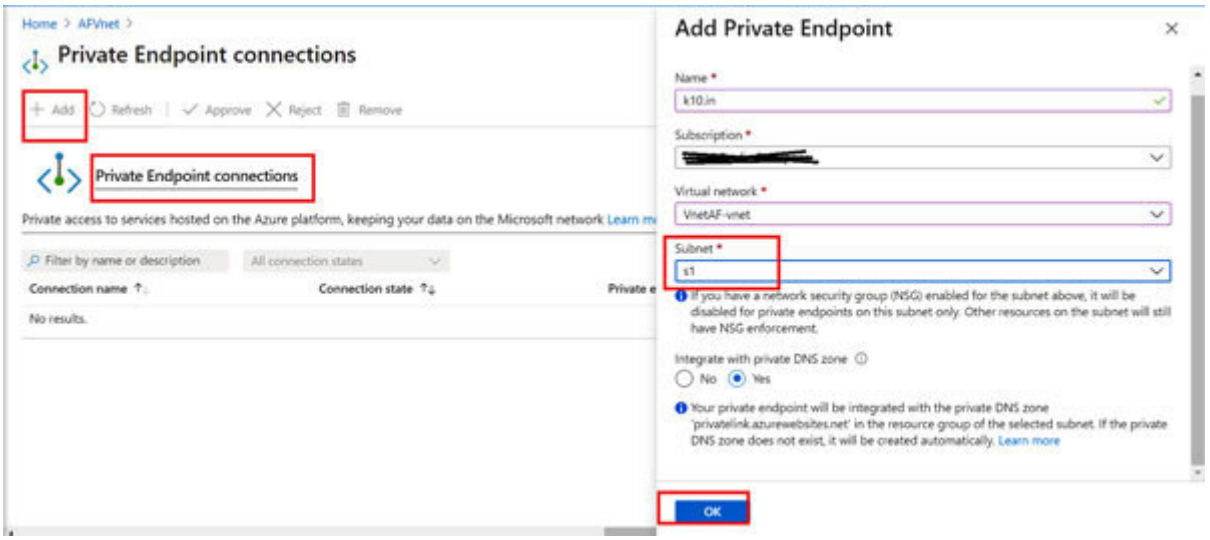


Figure 6.39: *Set up private endpoint connection*

Private endpoints guarantee that the assigned assets are accessible by the virtual network only. The Azure function app will

communicate with assigned resources using a private IP address. This gives an extra level of network-based security and control. Azure function app using a premium plan and an enabled VNet integration is required to connect private endpoint.

Network security using API Management

You can manage all your Azure functions using API management. To limit the number of API calls by client applications to your Azure function (let's say you want to restrict 10 API requests per second), we would need to create a lot of logic to handle this. With Azure API Management, you do not have to compose any custom code. All your request and response will be a route through API management. Please refer to *figure*

Figure 6.40: *API management*

Let see the process to add and configure API management for the Azure function app. Click on API management from the left blade. You can select the existing API management or create a new API management service. Please refer to *figure*

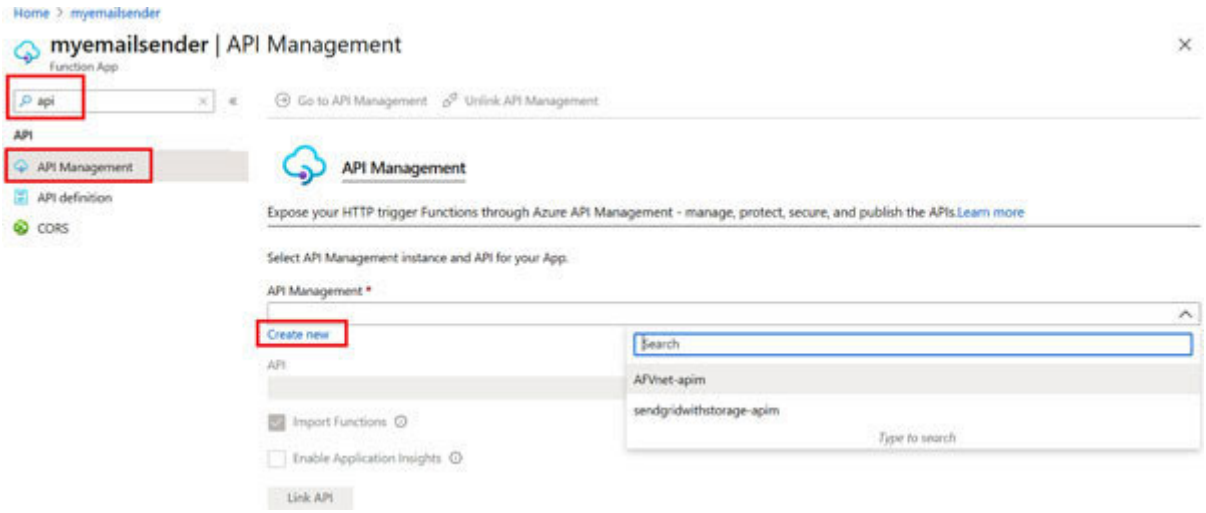


Figure 6.41: *Add API management service*

To create a new API management service, you need to provide information like name, resource group, and pricing tier (Basic/Standard/Premium/Developer(No SLA)). Please refer to *figure*

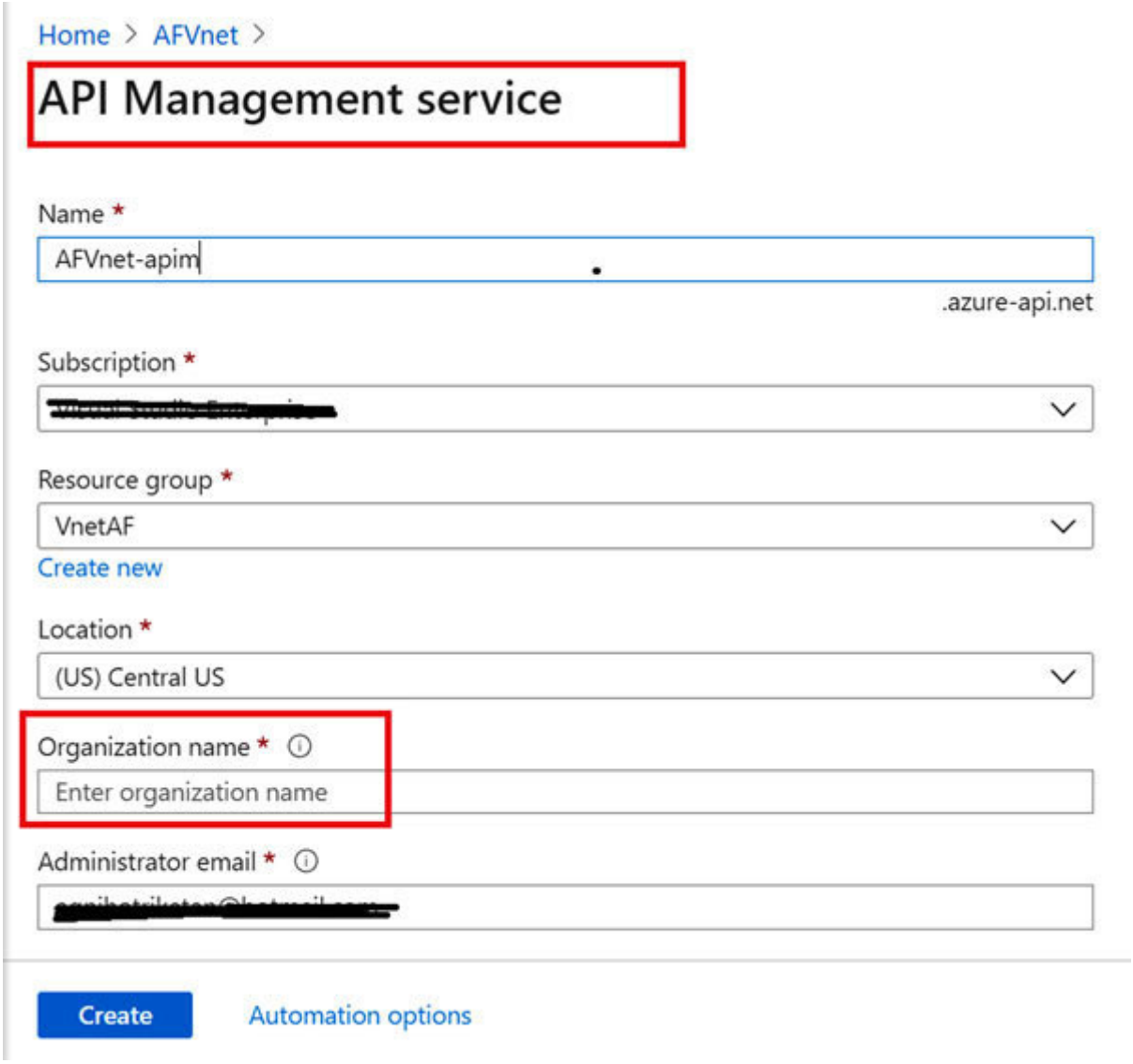


Figure 6.42: *Create API management service*

After creating an API management service instance, you need to add Azure functions as API. Choose **Function** Please refer to *figure*

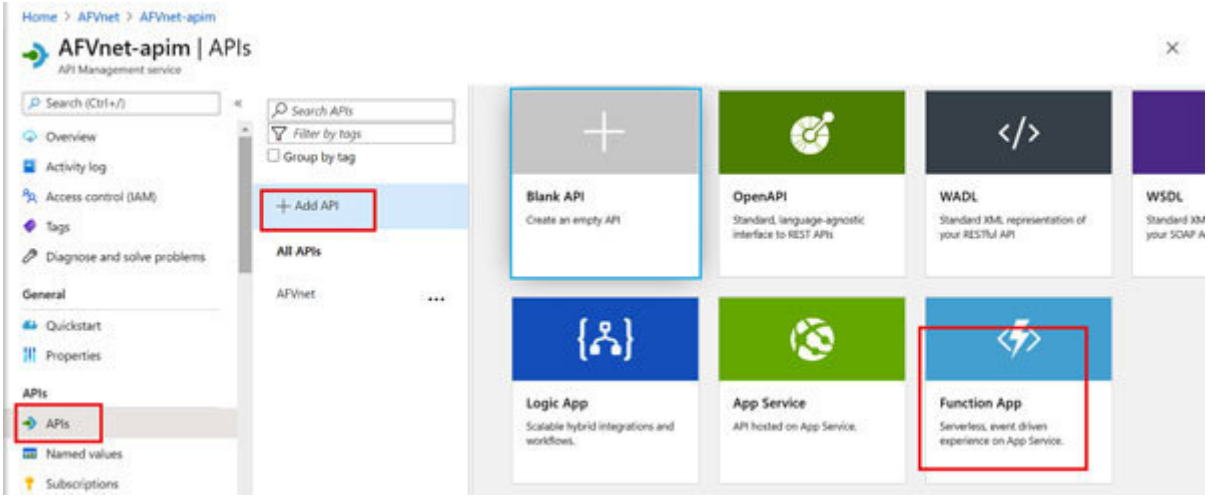


Figure 6.43: *Select type of APIs*

Now, you need to browse the function app you have and select the function app you want to configure in API management services. Please refer to *figure*

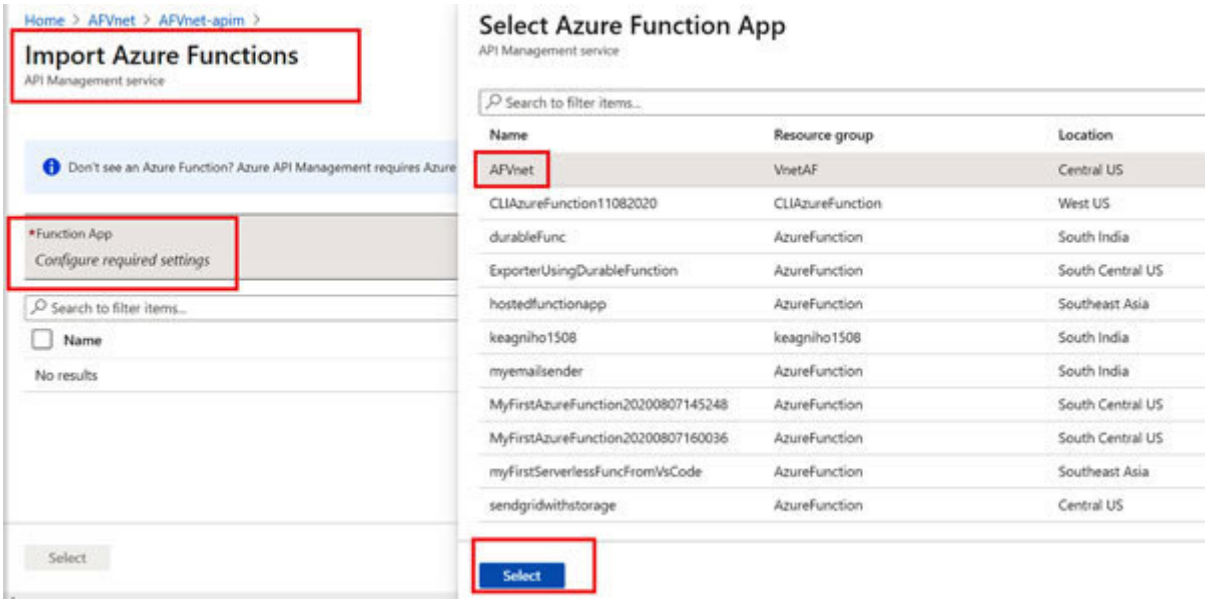


Figure 6.44: *Select your function app from the list*

After selecting the Azure function app, you must confirm the given information and click on Please refer to *figure*

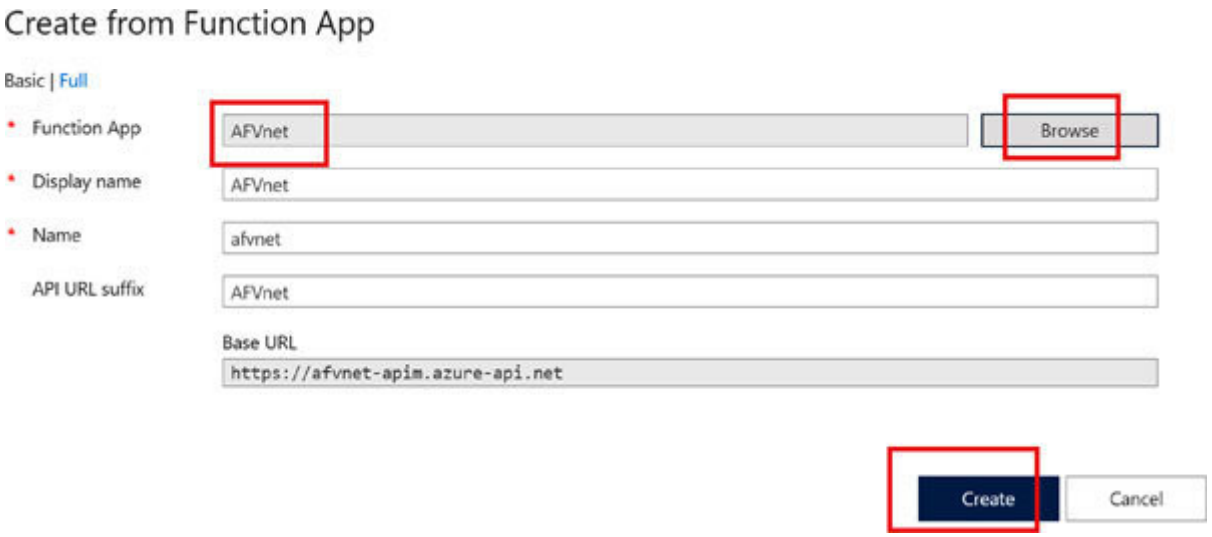


Figure 6.45: *Create API from function app*

After performing previous steps, you should see something similar to *figure 6.46* and you are done with API management and Azure function app integration. Now you can configure inbound, outbound policies like IP restrictions, limit call rate, and so on. Please refer to *figure*

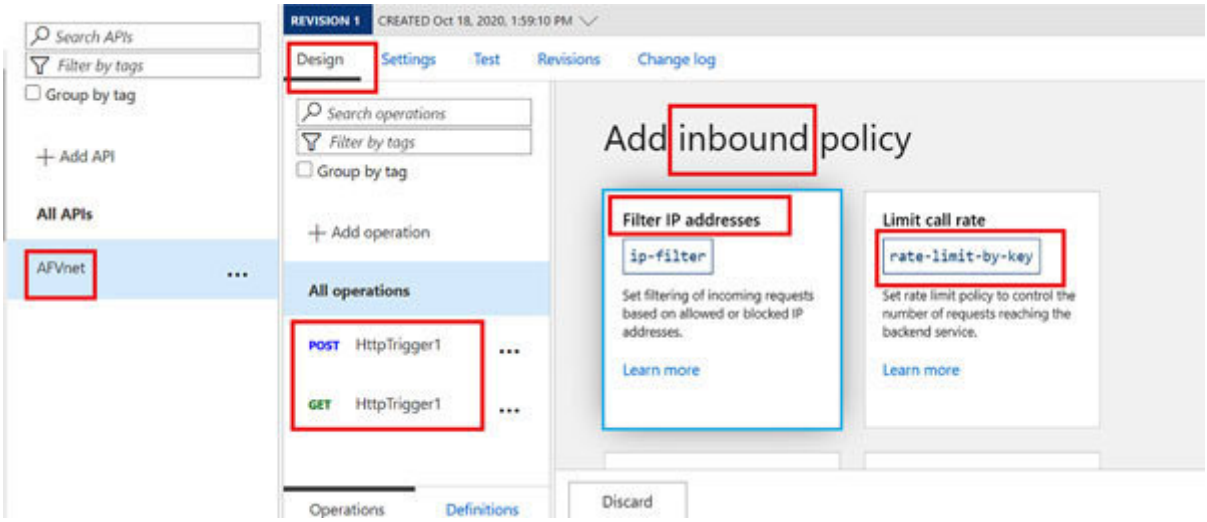


Figure 6.46: *Access restriction rules with action and priority*

Let's take an example of IP whitelisting. You need to add allowed IP addresses and blocked IP addresses as shown in *figure*

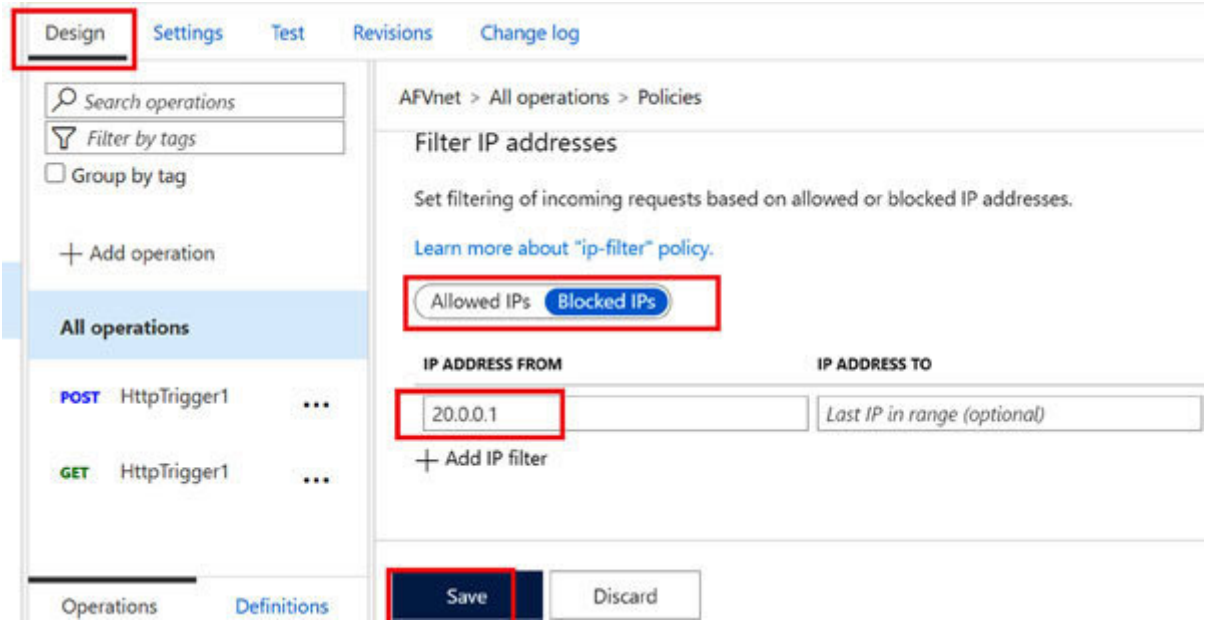


Figure 6.47: *Filter IP addresses*

After configuring the inbound policy, you can navigate to the test tab and test your API. If your IP is not in the allowed IP list, you will get a 403 IP forbidden error. Please refer to *figure*

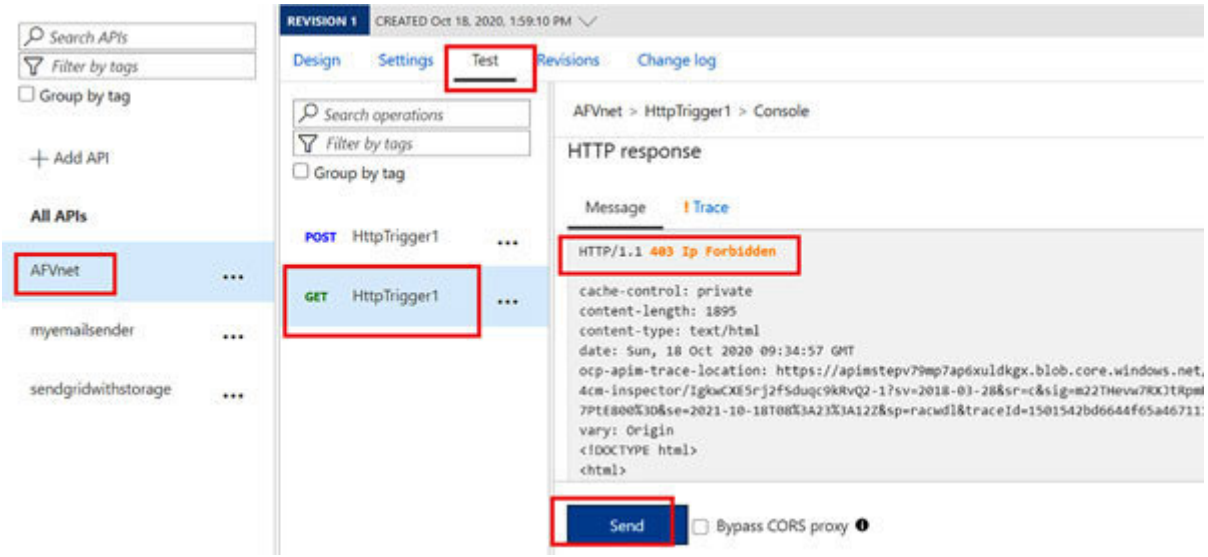


Figure 6.48: *403 IP forbidden error*

After removing the filter IP address policy, you should be able to access your function app using the URL again and should receive 200 HTTP status code as shown in *figure*

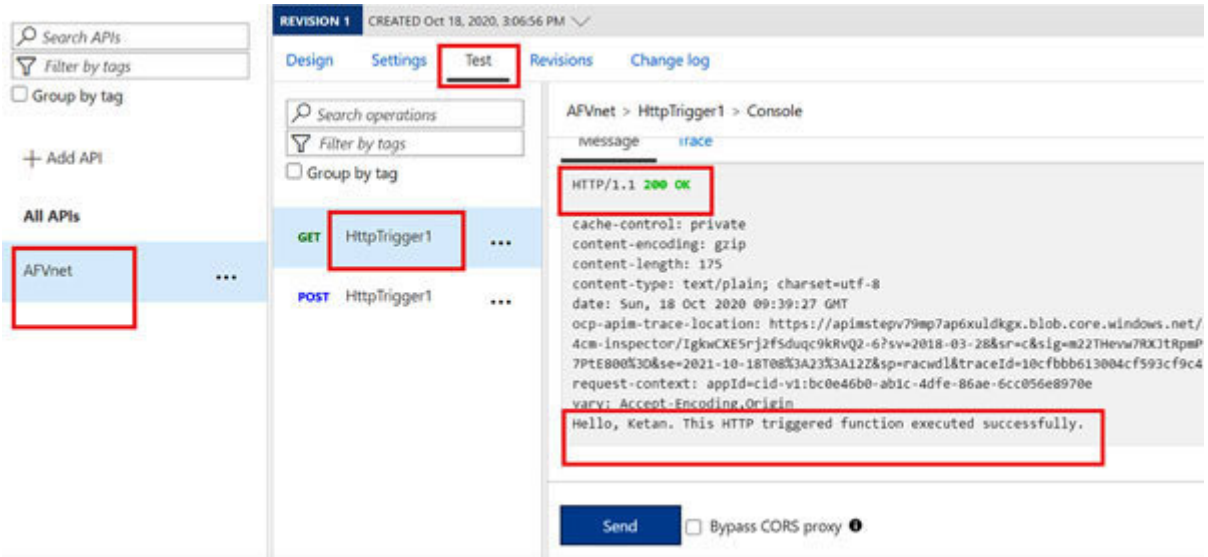


Figure 6.49: *Response after removing inbound policy*

You can have multiple inbound and outbound policies in the design section, and you can order them easily by drag and drop. You can also edit these policies using the code editor. Please refer to figure

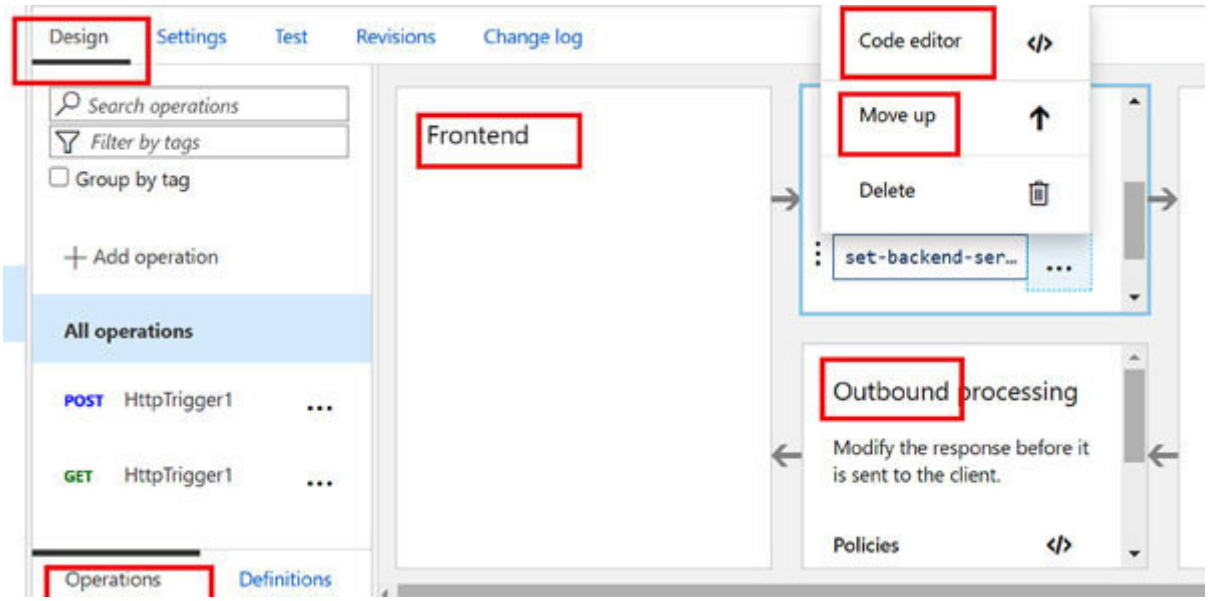


Figure 6.50: *Manage policies*

Let's configure the throttling of requests using inbound policies. You want to restrict your client application to make 1 API call to your Azure function in 5 seconds. To implement this scenario, you need to add a limit call rate inbound policy, as shown in figure

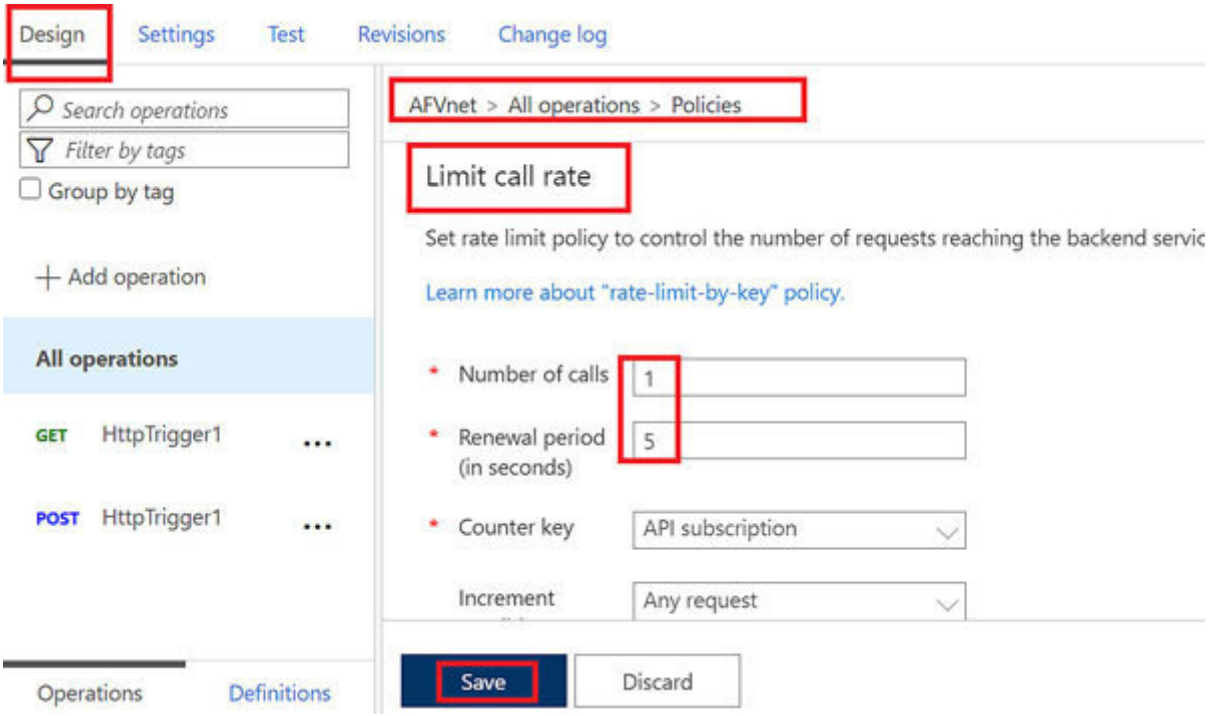


Figure 6.51: *Limit call rate*

If your client's application exceeds the number of allowed requests per 5 seconds, he should get 429 HTTP status code as **Many**

Once you are done with the policy set up, you can navigate the **Test** tab and test your function app URL.

Note: You can get more information about Azure API management by visiting

Azure function management can be modern and smooth using API management for existing backend services, making a huge difference in the function app as APIs distribution becomes easy for organizations.

Conclusion

In this chapter, we learned the importance of security in using Azure Functions and various aspects that may impact your resources' security. We have also understood multiple ways like authorization levels, IP restrictions, Azure Active Directory authentication, managed identities, and API management, which you can use to secure your Azure Functions with practical examples and avoid any security issues and vulnerabilities. Depending on the real-time scenarios, you can choose any techniques to improve the security of your function app. In the next chapter, you will read about continuous integration and continuous deployment of a function app.

Points to remember

Regular code reviews to check security vulnerabilities and errors in code are important tasks

You must configure network security rules

You must define data and function access levels as per needs using RBAC policies, using which roles can be assigned users, groups, and service principals, managed identities and key vaults, **Shared access signature** tokens

You can also impose validations on inputs provided to Azure functions

Azure functions let you restrict secure access using keys in multiple ways:

Using app service plan having Azure front door implemented

Azure **API Management** to authenticate requests

Enabling app service authentication and authorization

IP whitelisting

You must apply the recommendations suggested by the Azure security center

CHAPTER 7

Continuous Deployment for Azure Functions

In the previous chapter, we discussed how to plan and implement the security of the Azure Functions from all areas. In this chapter, we will discuss continuous deployment for Azure Functions. Planning for secure development and operation of serverless functions is much the same as for any web-based or cloud-hosted application.

This chapter covers the deployment aspect of functions using common deployment tools like Azure DevOps, GitHub actions, and so on. It also defines the various options to deploy functions on production, such as deployment slots.

Structure

In this chapter, we will cover the following topics:

Deployment methods and technologies

Continuous integration and continuous deployment using Azure DevOps

Building and deploying Azure Function using GitHub actions

Azure Functions deployment slots

Objective

After studying this chapter, you should be able to understand the continuous build process for Azure Functions. You will also learn about automating the build and triggering the release process of Azure Function using available tools.

Deployment methods and technologies

There are multiple ways for deploying your app to Azure, and the method is adopted as per needs and development cycle. When your app runs in the production environment, you are more likely to continuously publish your source control changes or use an automated publishing pipeline, which includes additional validation and testing.

Please refer to *table 7.1* to understand available deployment methods for your function app:

app:
app: app: app: app: app: app: app: app: app: app:
app: app: app: app: app:
app: app: app: app: app:

Table 7.1: *Deployment methods*

Cross-platform development and hosting support are available for Azure Functions, and hence Azure Functions can be deployed on Windows and Linux machines. Consumption, premium, and dedicated are the hosting plans available for Azure Functions.

There are multiple supported deployment technologies like external package URL, zip deploys, docker container, web deploy, source control, local Git, cloud sync, FTP, and portal editing. Not all of these technologies are supported by each type of Azure Functions.

When you make any change to Azure Function triggers, functions infrastructure must be aware of these changes. This is called **trigger**

Note: You can get more information about deployment technologies by visiting

This syncing happens automatically for few deployment technologies. However, you need to manually do syncing activity by restarting the function app or HTTP POST request for external package URL, local Git, cloud sync, or FTP.

Continuous integration and continuous deployment using Azure DevOps

Source control is the way you can keep code repositories like Git. for Working in teams, where team members collaborate and contribute to a single project, it is easy to track and maintain code with source control.

Source control integration for deployments:

Using the deployment center of Azure Functions and source control integration with repositories like git, deployments can be triggered automatically whenever any code is pushed to the functions repository. This is called **continuous integration and deployment**

This practice can be followed for projects where more frequent changes are done in code by multiple contributors. When a continuous deployment is enabled for a function app, all functions in the function app are deployed simultaneously. Also, to maintain a single source of truth for the function code as a code repository, editing functions from the Azure portal is not possible, as it switches to read-only mode.

Source code repositories options for Azure functions:

Git source control repository

Azure Repos

Bitbucket

Dropbox

OneDrive

To set up continuous deployment, a basic folder structure for the function app should be maintained. There needs to be a root folder with the host. json file, which contains the configuration details. There can be multiple subfolders to maintain each function code separately. From version 2.x, and above all functions in a single function app should be built on top of a single language stack. The next thing is the bin folder, containing all packages and libraries required by the function app. Other than this, there can be other required files specific to the language used for functions.

The folder structure example is as follows:

MyFunctionApp

| - host.json
| - Function1
| | - function.json
| | - ...
| - Function2
| | - function.json
| | - ...
| - SharedCode
| - bin

For setting up continuous deployment for Azure functions, steps are almost similar for different types of repositories. In this

chapter, we will see these steps with Azure Repos and GitHub repository.

Setting up Azure DevOps account

Azure DevOps is a modern DevOps tool for planning, developing, testing, and deploying modern apps with an efficient application release cycle. Azure DevOps is a tool to keep track of software delivery. Azure DevOps services are platform-independent.

To set up CI-CD using Azure DevOps, you must have an Azure DevOps account. Please visit **https://dev.azure.com/** to create an Azure DevOps account.

To implement build automation and deployment automation, Azure pipeline is a cloud-hosted build and release service that works with any language, platform, and cloud. By connecting to any source control like GitHub, this service can build and release changes continuously to any cloud. These build and release definitions are YAML-based. Azure Pipelines has components like build, release, library, task groups, and deployment groups. Azure Pipelines has advanced workflows with native container support and features to monitor CI/CD stages. Please refer to *figure*

Figure 7.1: *CI-CD workflow*

Create Azure DevOps project in an organization

You can create multiple projects under one Azure DevOps organization. Azure DevOps organization provides centralized control over the collection of projects which helps to connect groups of related projects. You need an organization created before creating a new project and **new project**' permission for your account.

After creating an organization, you can create a project to set up a structure of projects for effective management of business needs, which helps isolate projects from each other.

At the time of project creation, you must select parameters like project visibility, version control, work item process. Please refer to *figure*

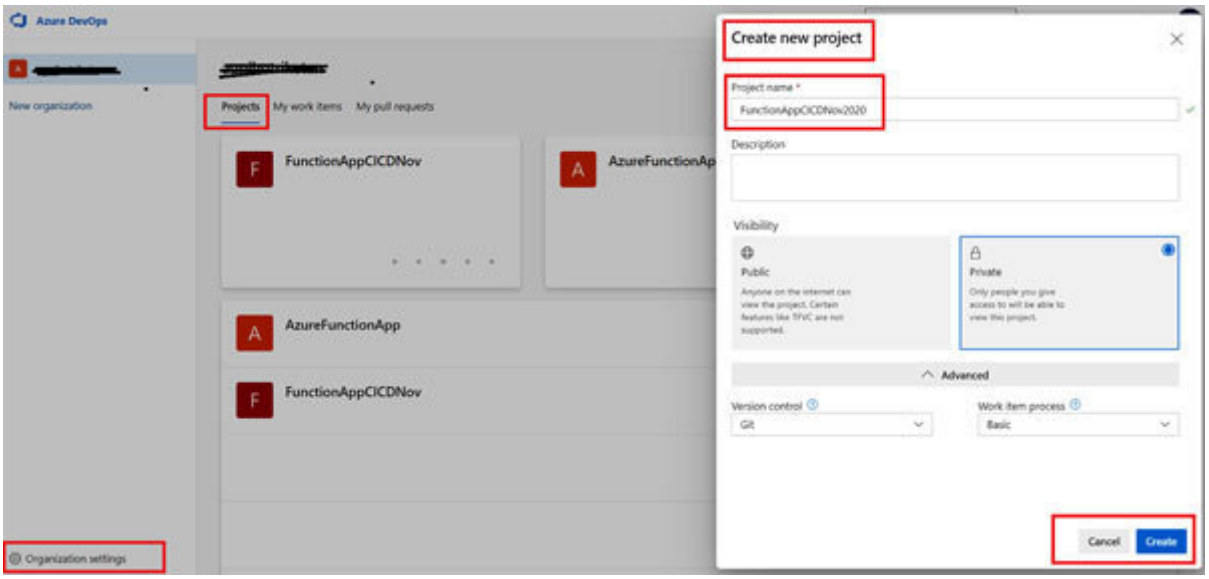


Figure 7.2: *Create Azure DevOps project (private or public)*

Pushing Azure Functions source code in Azure Repos

Azure Reops service works as distributed source controls, which include unlimited cloud-hosted private Git repository for your project. Azure Repos supports all Git clients and all IDEs, all editors. You must clone the empty repo on your machine and push the source code of your new or previously developed Azure Function. You must add a **.gitignore** file in your repo. A **.gitignore** file is a plain text file that contains paths, file extensions, and patterns to be ignored while committing source code in the repo to keep the repo lightweight. Please refer to *figure*

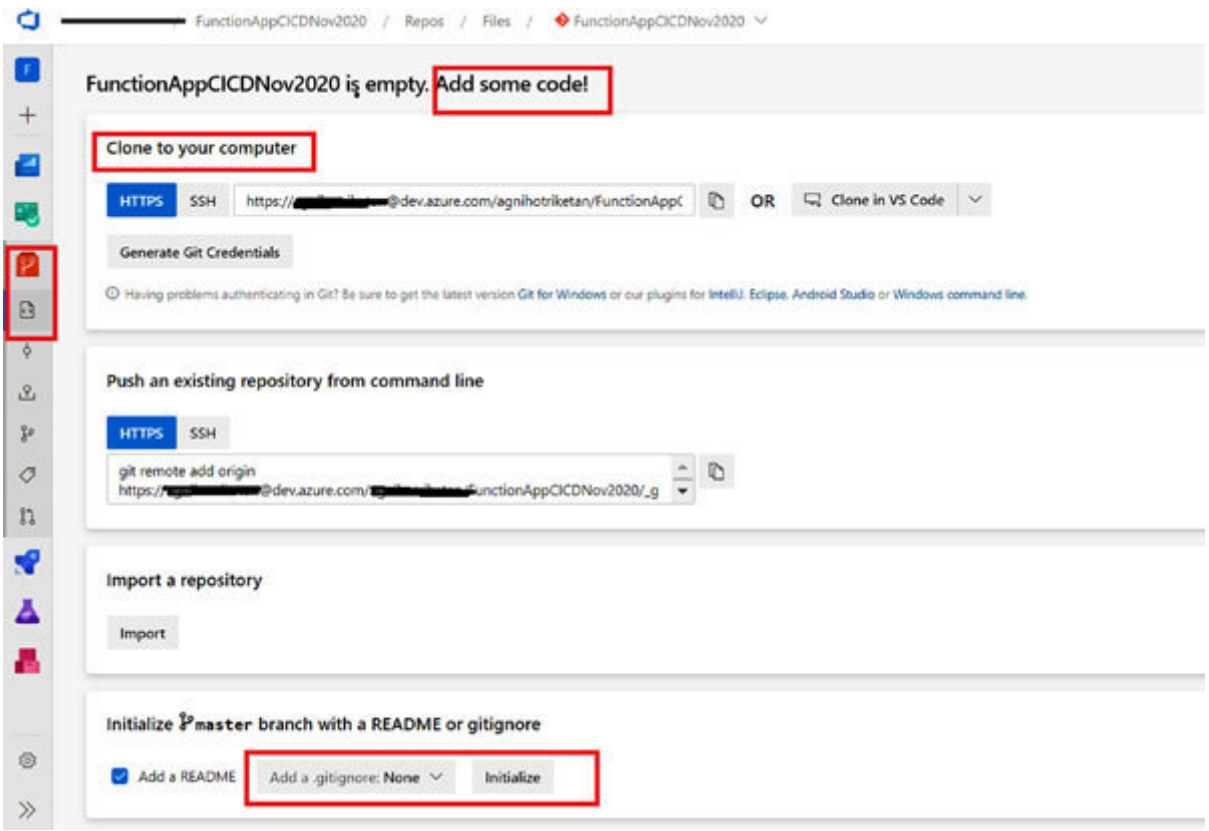


Figure 7.3: *Push Azure Function code to Azure DevOps Repos*

Azure Repos supports effective code reviews, raises pull requests, and implements branching strategies to merge the code after successful build and test case execution to maintain high code quality. Source code access management is fast and easy and can be managed by Azure AD.

Build automation (CI) - creating a build pipeline

Let's create a new build pipeline. Navigate to the pipeline option from the left menu and click on to start the process of creating a new build definition.

Now the next step is to choose the source code location. To build function app source code, you must make sure all the functions and function app configuration files properly. Once the code set up is completed, select the **classic** option, as shown in *figure*

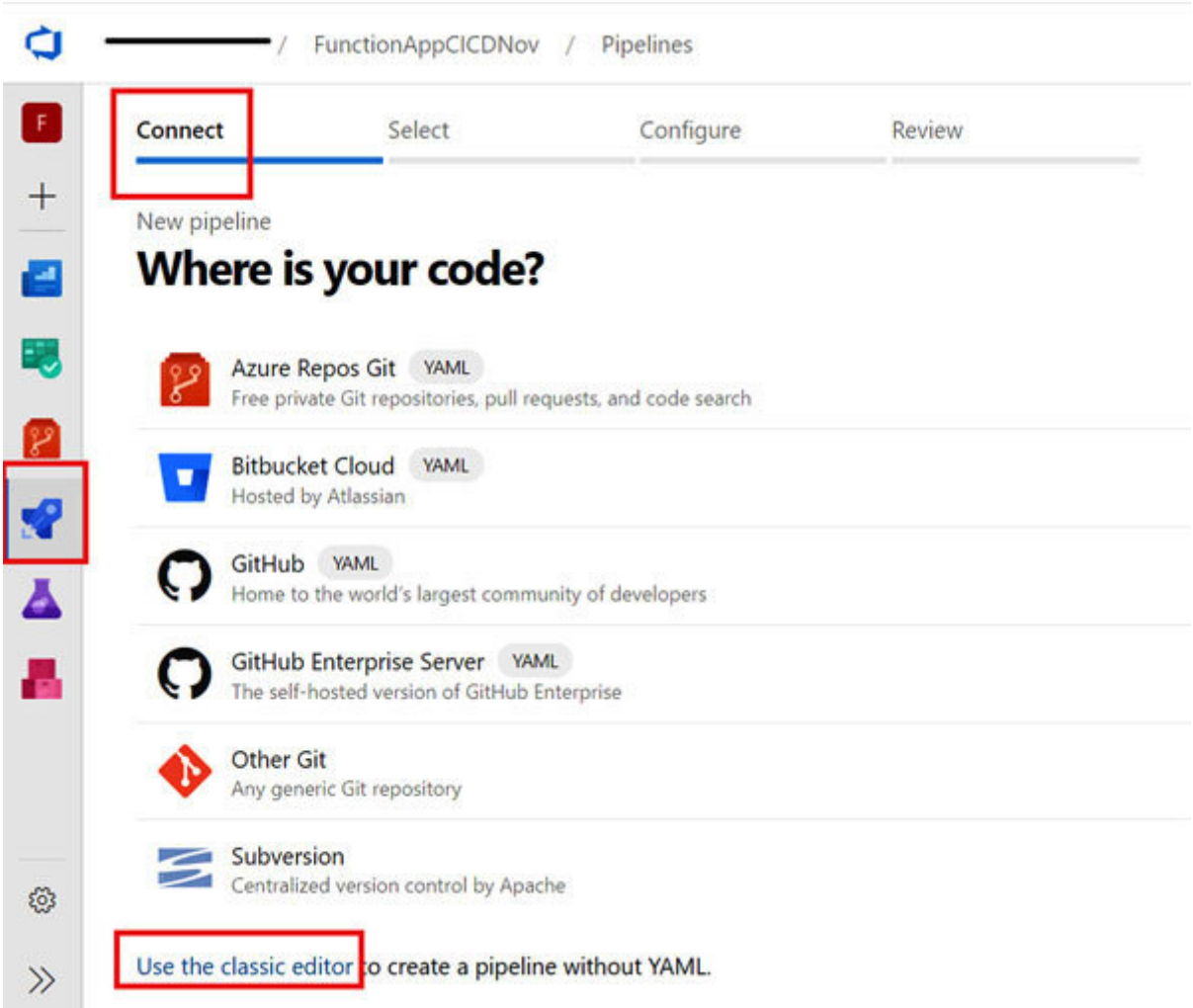


Figure 7.4: *Connect to source code*

Now you will navigate to the repository selection screen, as shown in *figure* You must select the source repository, branch off your Azure function app, and click on Make sure you have selected the correct repository and branch.

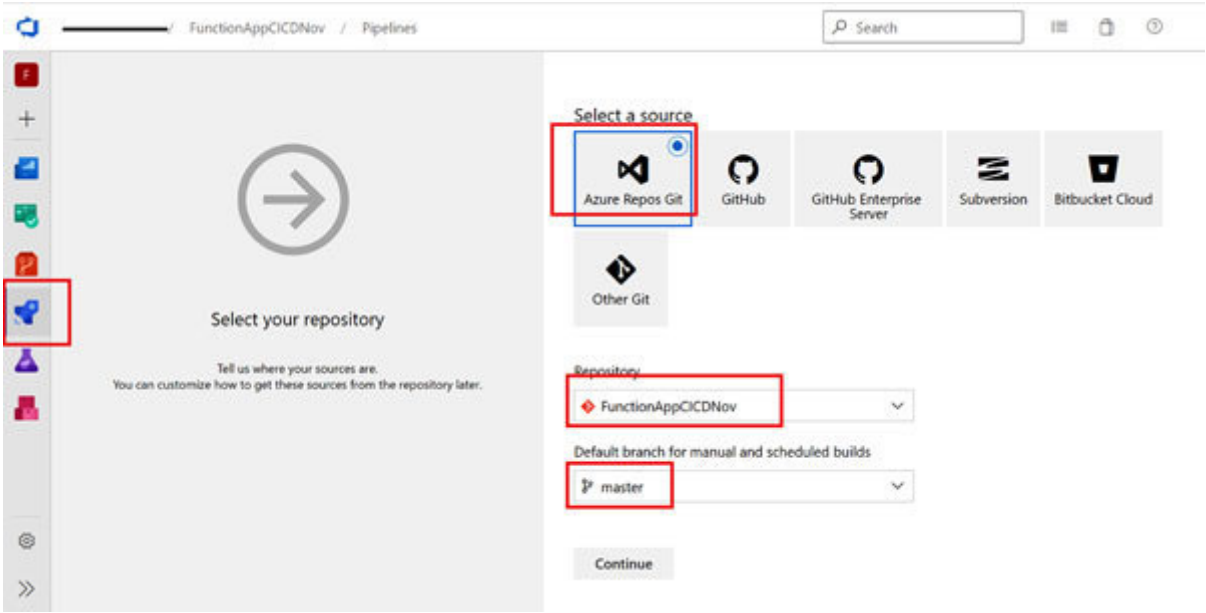


Figure 7.5: *Select your repository*

The next step is to select a build template from the available template options provided by Azure DevOps. To browse available templates, add a keyword in the search box. Azure DevOps provides a variety of templates to build your Azure Functions. You can also add an empty job and add tasks one by one per your custom workflow requirement. Select a template and click on Please refer to *figure*

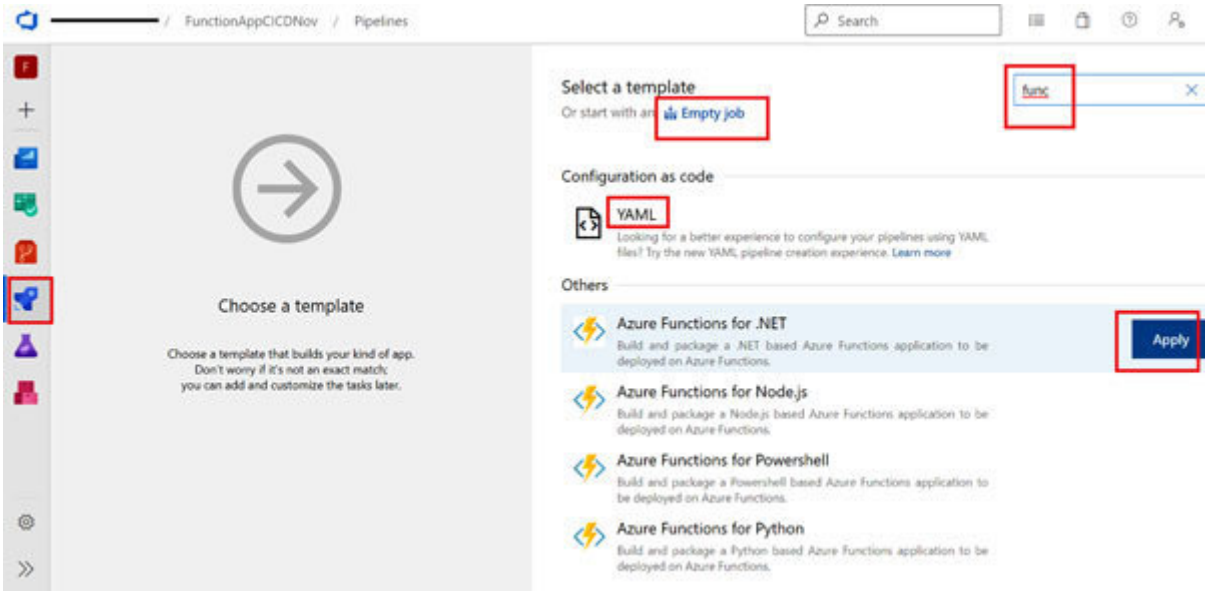


Figure 7.6: *Choose build template*

After applying the build template, each step has few attributes. You must provide values. You can add more tasks like run test cases, publish build results, and code quality checks (for example, eslint, SonarQube, and so on) from the marketplace.

After making the necessary changes, save the build definition and queue a new build. Build definition is a blueprint of tasks required to be run in sequence to build a software application. When running the pipeline, you must verify the preselected values of the agent pool, agent specification, and branch. Build an agent is a virtual machine that has required software/tools pre-installed on it. Please refer to *figure*

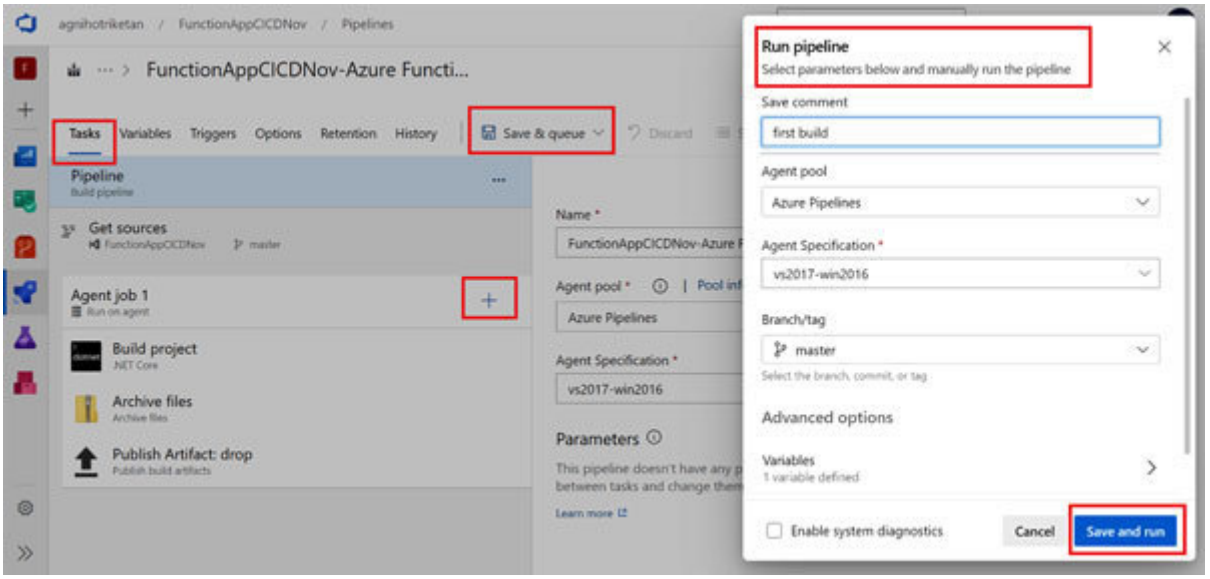


Figure 7.7: *Provide inputs and run the pipeline*

Once you queue a build, you can monitor the execution cycle of the build pipeline running on the build agent. Terminal logs are very helpful to troubleshoot the build errors and can help analyze the build process. Please refer to *figure*

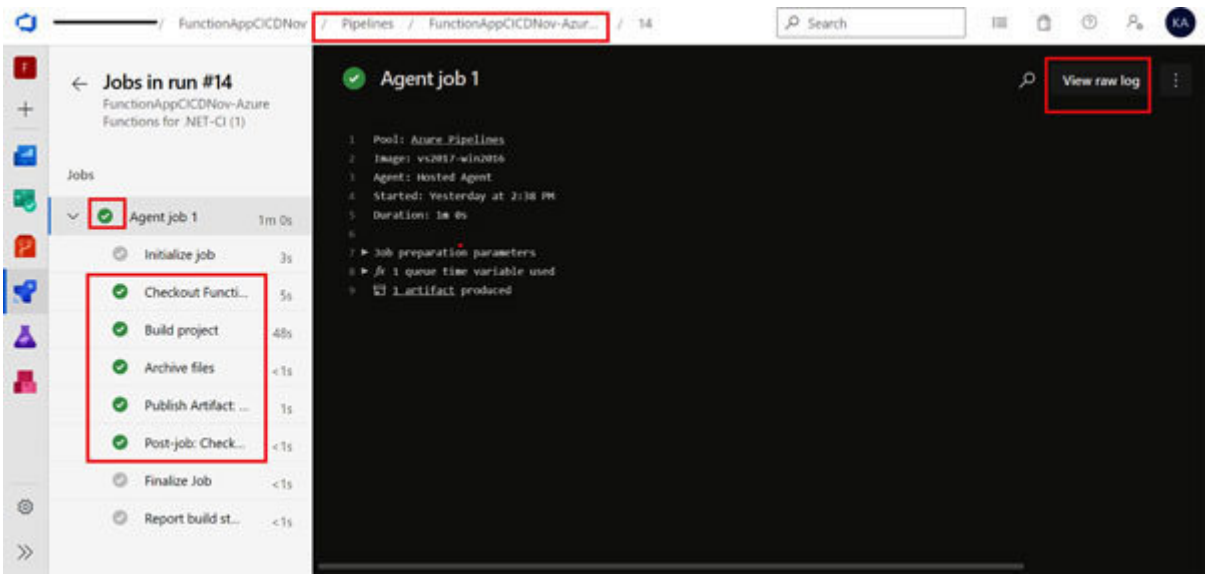


Figure 7.8: *Build logs*

Build pipeline has a **Triggers** tab, which helps you to trigger a build based on some conditions. These conditions could be based on time or a predefined schedule. You can enable the continuous integration option; once you commit to the repository, a new build will be triggered. Please refer to *figure*

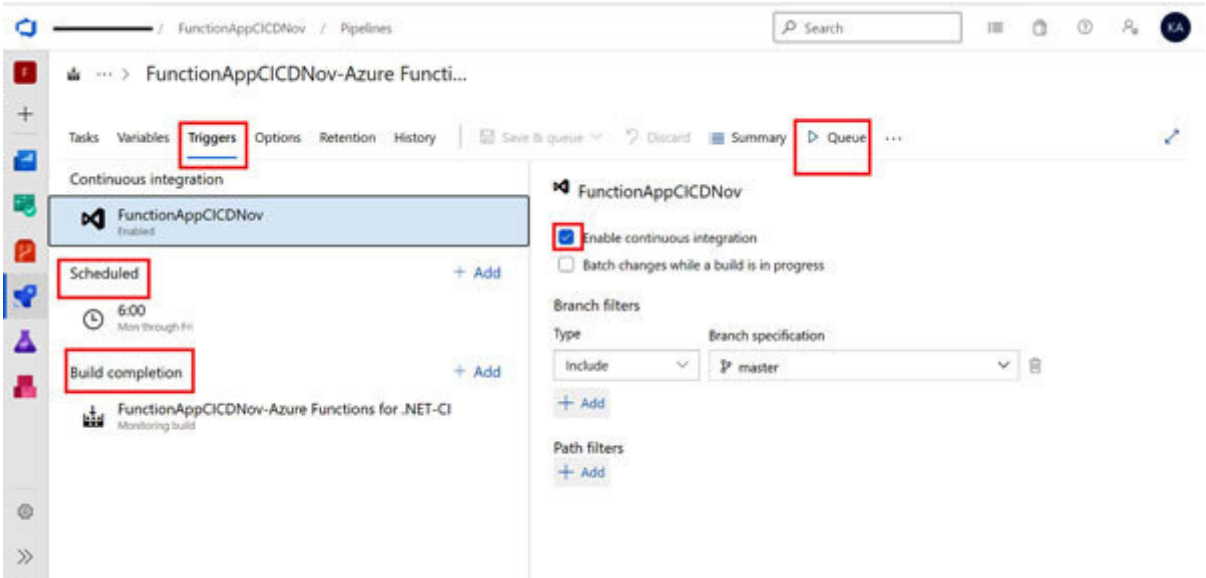


Figure 7.9: *Build triggers*

Once the build execution succeeds, you can download and check the build artifacts on your machine by clicking the **Published** button available on the build page. These build artifact/build packages will be considered as input to your release/deployment automation.

Deployment automation (CD) – Creating release pipeline

Now you must have understood the continuous integration and build automation. As a part of deployment automation, Azure DevOps has a release pipeline option.

To create a new release pipeline, navigate to the release sub-menu of pipelines. Click on **New** or import the existing release pipeline. Please refer to *figure*

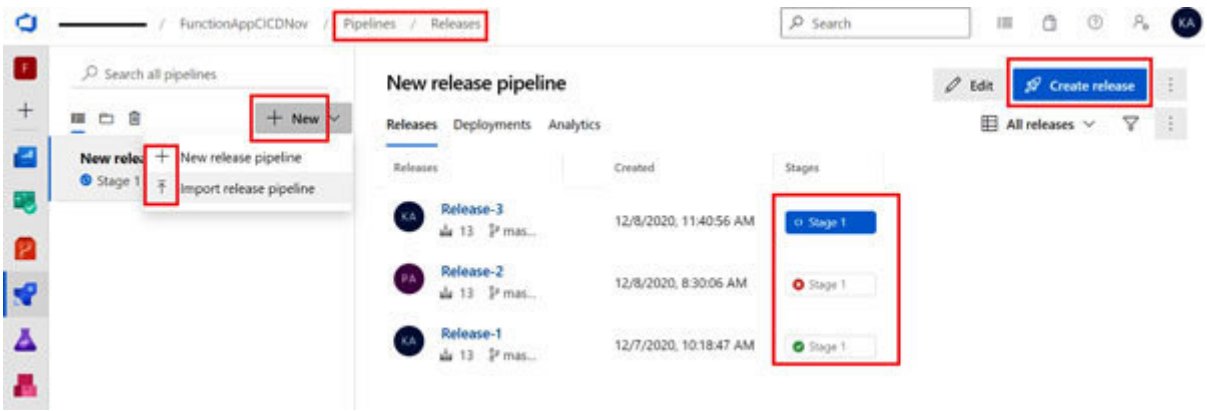


Figure 7.10: *Create release pipeline*

Just like build templates, Azure DevOps also provides release templates. Search for **the function app to** and click on Immediately, the template will ask to set up environments like dev, test, prod, and so on. You may close it or set it up as per your requirement.

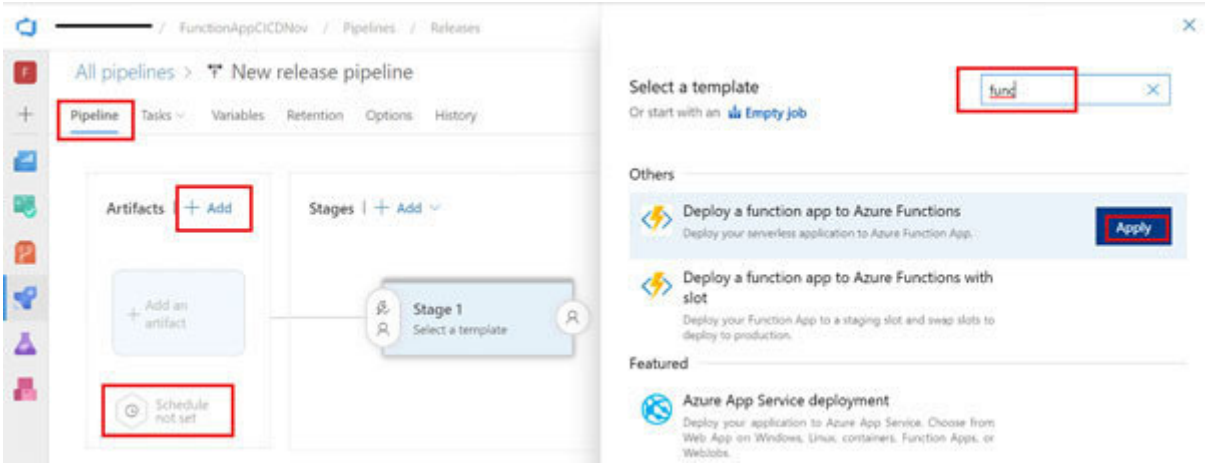


Figure 7.11: *Choose release template*

The next step is to provide build artifacts as input to the release pipeline as part of deployment automation. Click on **Add artifact** and select your previously set up build pipeline and latest successful build artifact. Please refer to *figure*

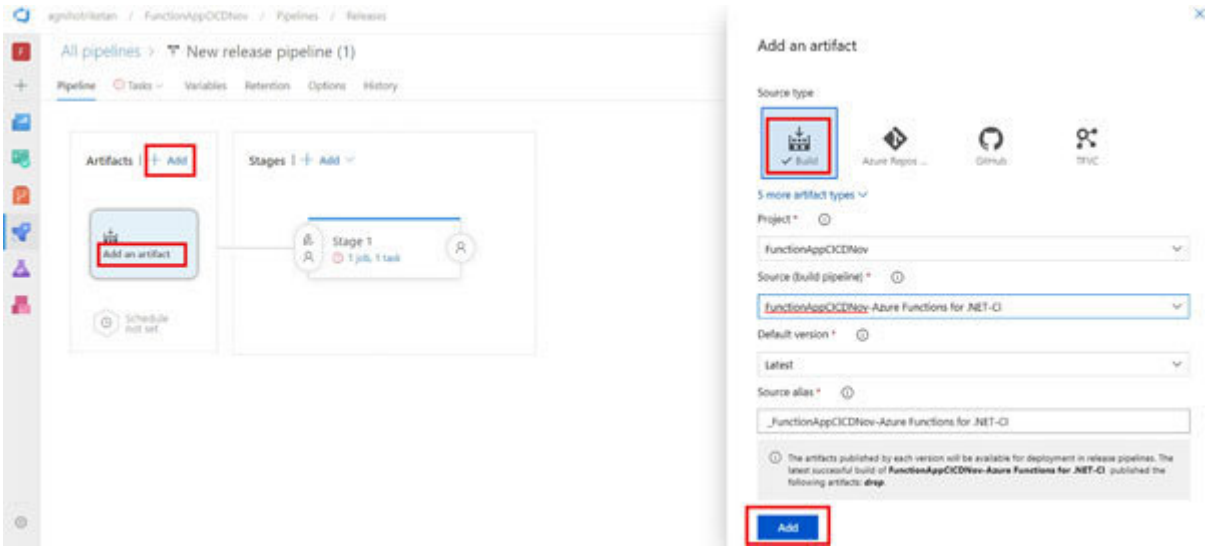


Figure 7.12: *Select build artifact*

After completing the artifact setup, you must provide the destination where these build artifacts will be published. Click on

a highlighted task of the stage, and provide the Azure subscription, Azure Function app, Function app type, and so on.

To get the list of Azure Function apps present under your Azure subscription or to connect to your Azure subscription, you must click on **Authorize** to provide the required permissions. Save all the configurations made to release the pipeline. Please refer to *figure*

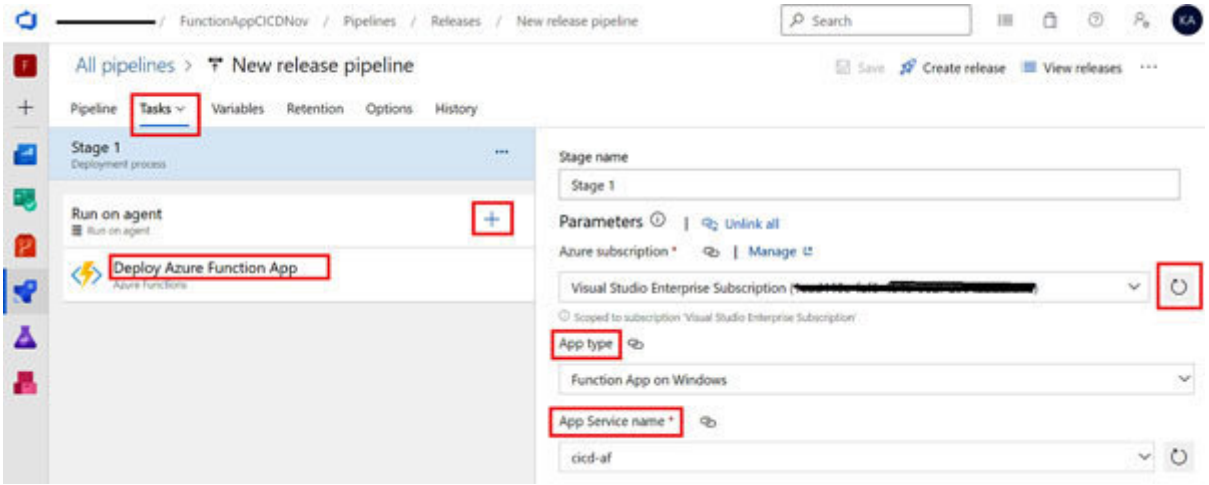


Figure 7.13: *Connect to Azure*

Now, let's create a release using this newly created release definition. Click the create release button and select build version, environment/stages from a pop-up; after this, queue a new release. After creating a release, you can track release progress, as shown in *figure*

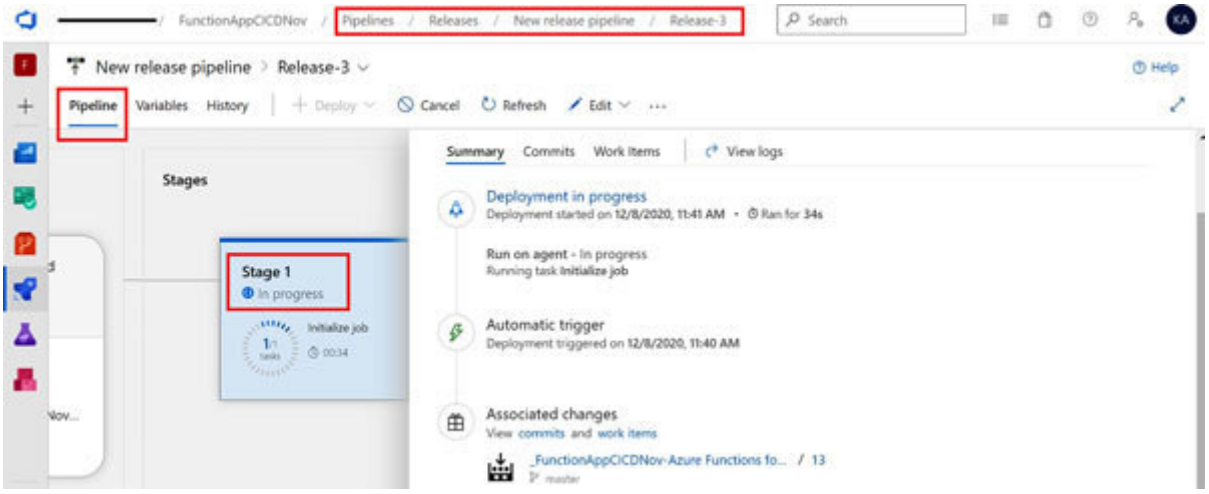


Figure 7.14: *Create a release*

You can also set up a build and release pipeline/automation using the deployment center option and connecting to Azure DevOps from the Azure portal. You can also see all the builds and their progress in the Azure portal, as shown in *figure*

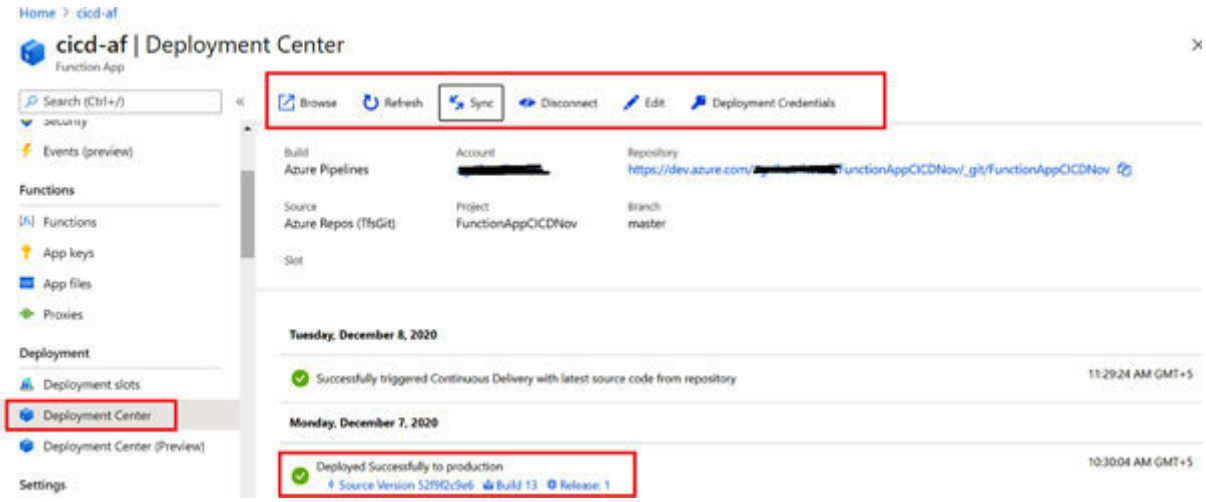


Figure 7.15: *Deployment Center*

You can enable pre-deployment approvals as well. You can also select the re-deploy option to promote a stable build and roll back

the unstable build. After a successful release, you must verify your changes on the deployed version of the Azure Function app.

Building and deploying Azure Function using GitHub actions

GitHub actions are a suite of event-driven workflows with a sequence of activities like build, test, and deploy included in it and added to your source code repository. GitHub actions make automation of activities very simple and fast using YAML-based files. These YAML files are stored in the path. Workflow has events, jobs, steps, actions, runners. GitHub has recently introduced this section in every repository. You can add your workflow, add/edit existing workflow from the marketplace, view workflow results, and analyze the build/deploy logs.

In this section, you will learn to build and deploy your previously developed Azure Functions.

For Azure Functions, the workflow YAML file has three sections like authentication, build, and deploy.

As a part of the authentication step, you must generate deployment credentials. To generate deployment credentials, navigate to your function app, and download publish profile by clicking the **Publish** option from the Azure portal, as shown in *figure*

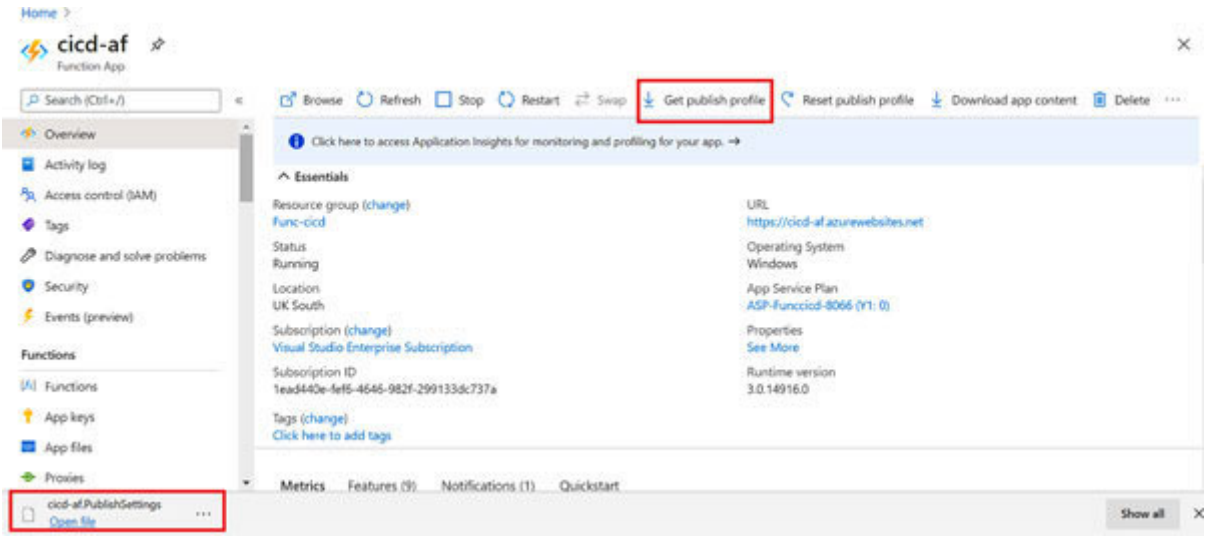


Figure 7.16: *Generate deployment credentials*

The next step is to navigate to the settings section of the GitHub repository and click on Now, copy the content of the downloaded XML file from the Azure portal and add it against a key This key will be used in the workflow YAML file to access the generated credentials. Please refer to *figure*

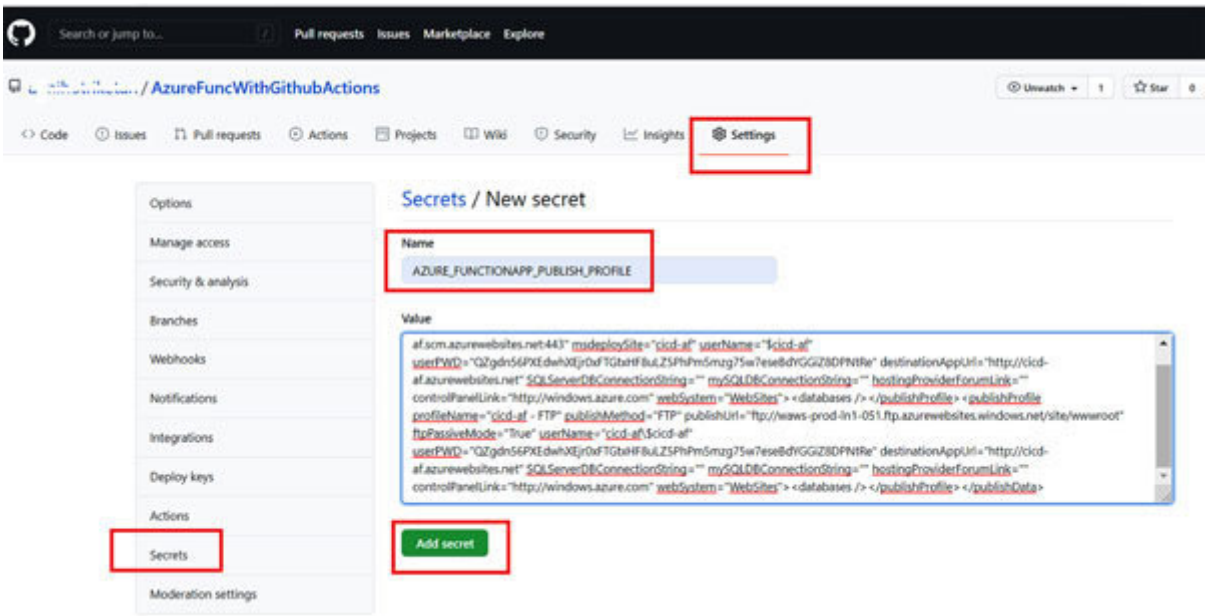


Figure 7.17: *Add deployment credentials*

Now, you can test your manually deployed Azure Function, as shown in *figure*

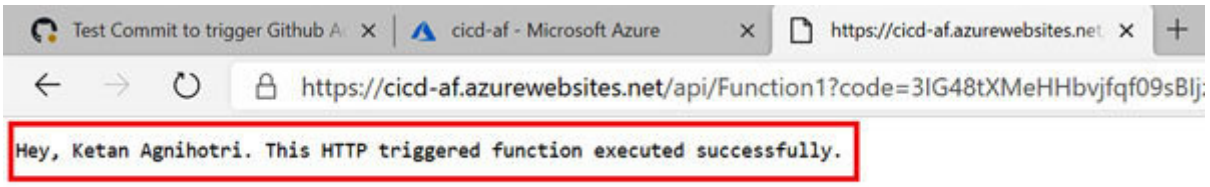


Figure 7.18: *Run sample Azure function*

Let's set up a workflow in your GitHub repository. Navigate to the **Actions** tab and click on **set up a workflow** You can set up a new workflow yourself or can use the suggested workflows. Please refer to *figure*

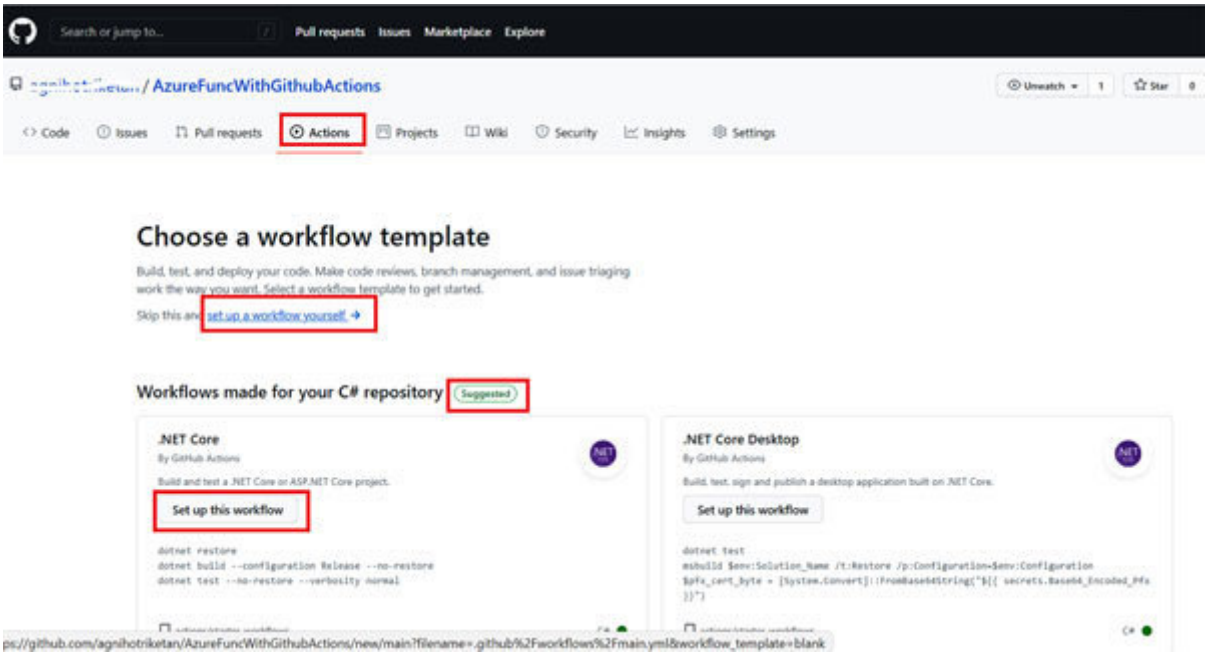


Figure 7.19: *Set up a workflow*

While creating a workflow, you need to add a **.yml** file in the path. You can get pre-build and freely available **.yml** files for your Azure Function by visiting Please refer to *figure*

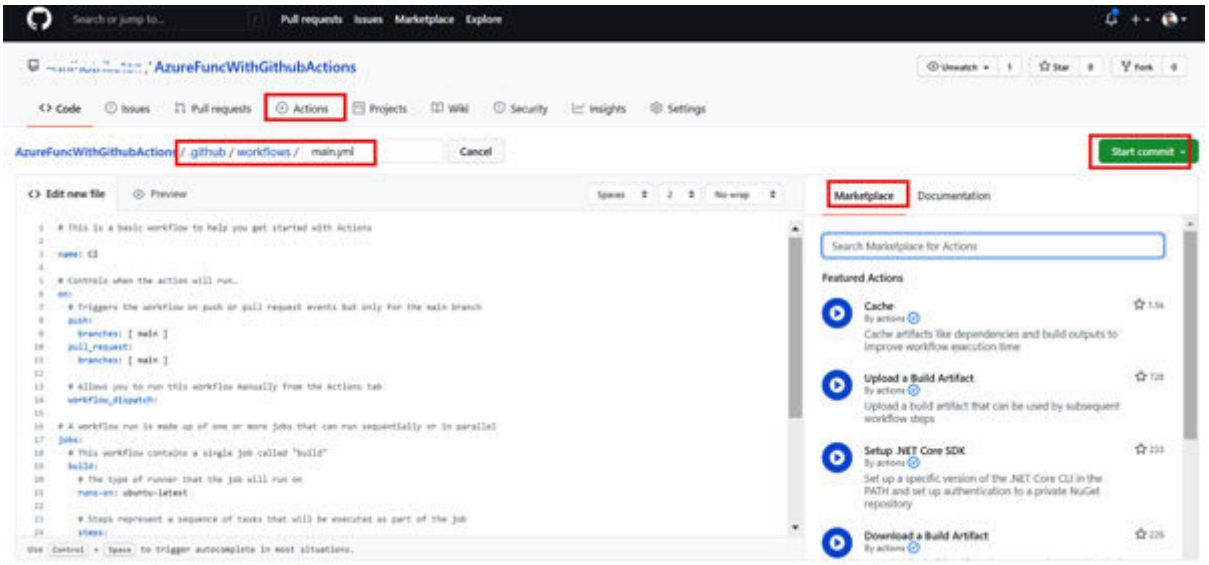


Figure 7.20: *Azure DevOps Repos*

You can copy and paste the template available **.yml** files. You must edit this **.yml** file and add details like function app name, package path, slot-name, publish-profile, and so on of your Function app. Please refer to *figure*

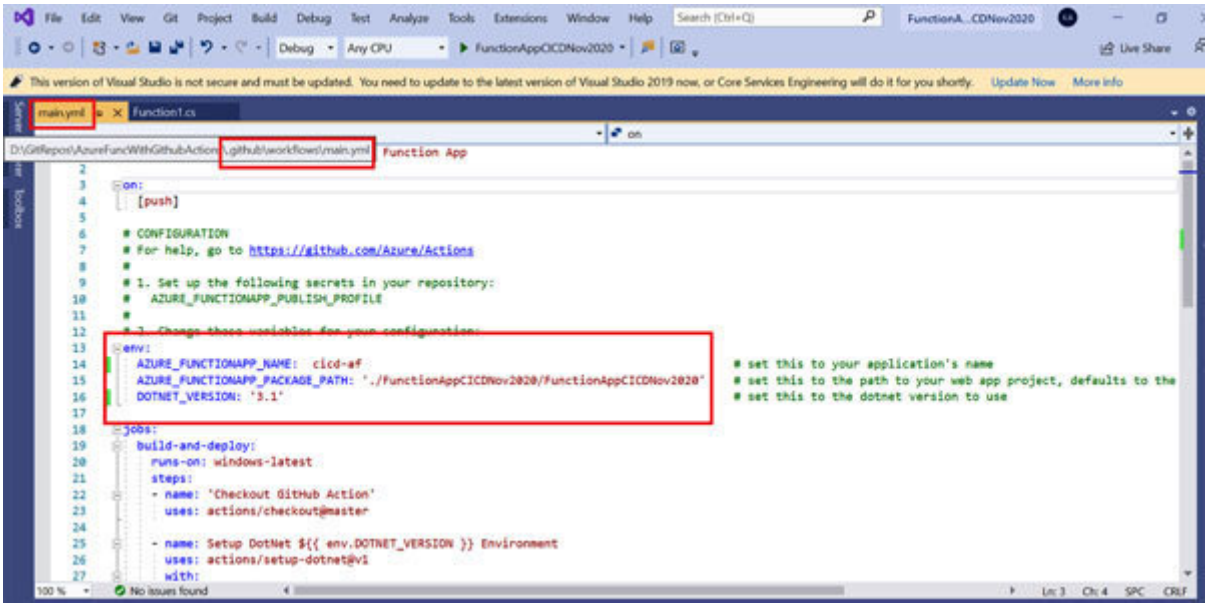


Figure 7.21: *Set up your Function app details*

After adding the workflow, let's test whether workflow settings are working as expected or not. To test it, you can make few changes, commit, and push your changes to your remote GitHub repository. Please refer to *figure*

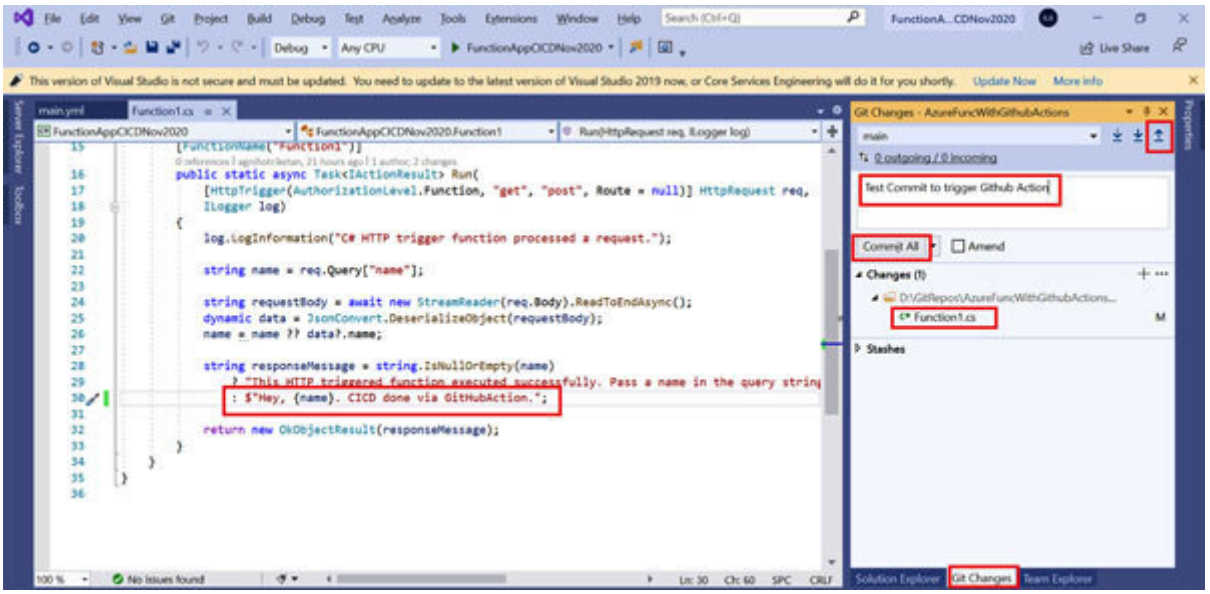


Figure 7.22: *Make changes, commit, and push your changes*

As soon as you push your changes to your GitHub repository, your workflow should be triggered, and then the build/deploy task should be queued. If it is not triggered, then you must check the workflow settings. Please refer to *figure*

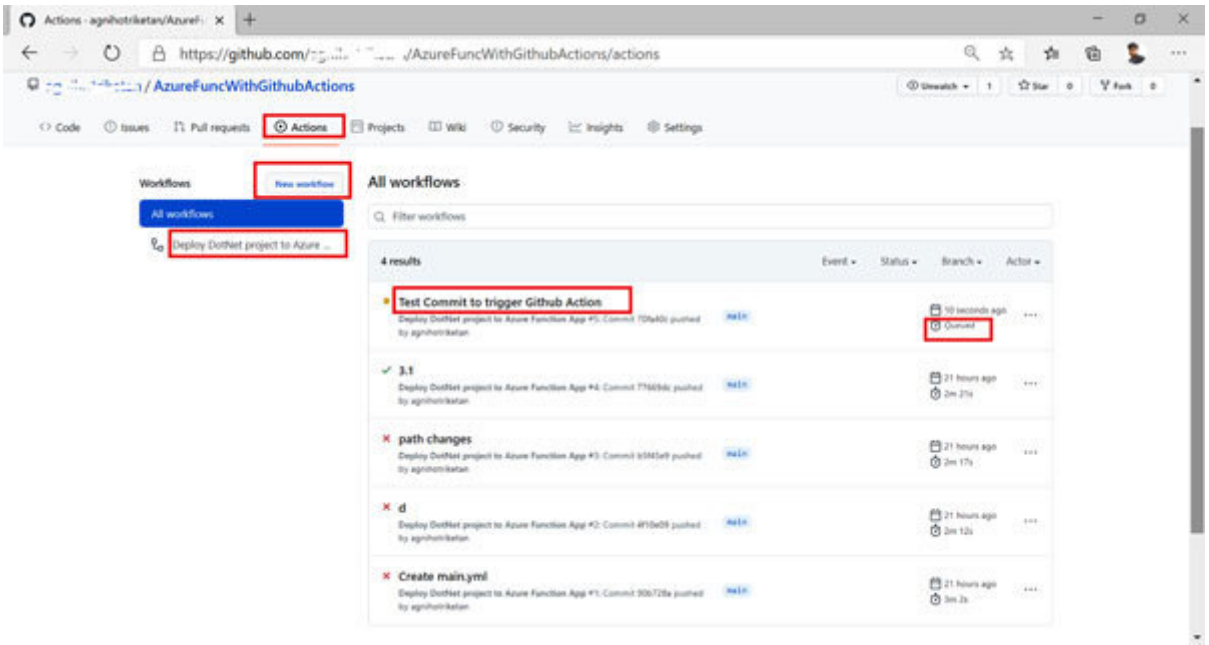


Figure 7.23: *GitHub actions got triggered*

Defined job steps will be executed in a sequence that is defined in the YAML files. You can track the progress/status of all jobs and tasks on GitHub. Build and deployment automation are part of a single workflow file. You may design multiple workflows as per your need. Please refer to *figure*

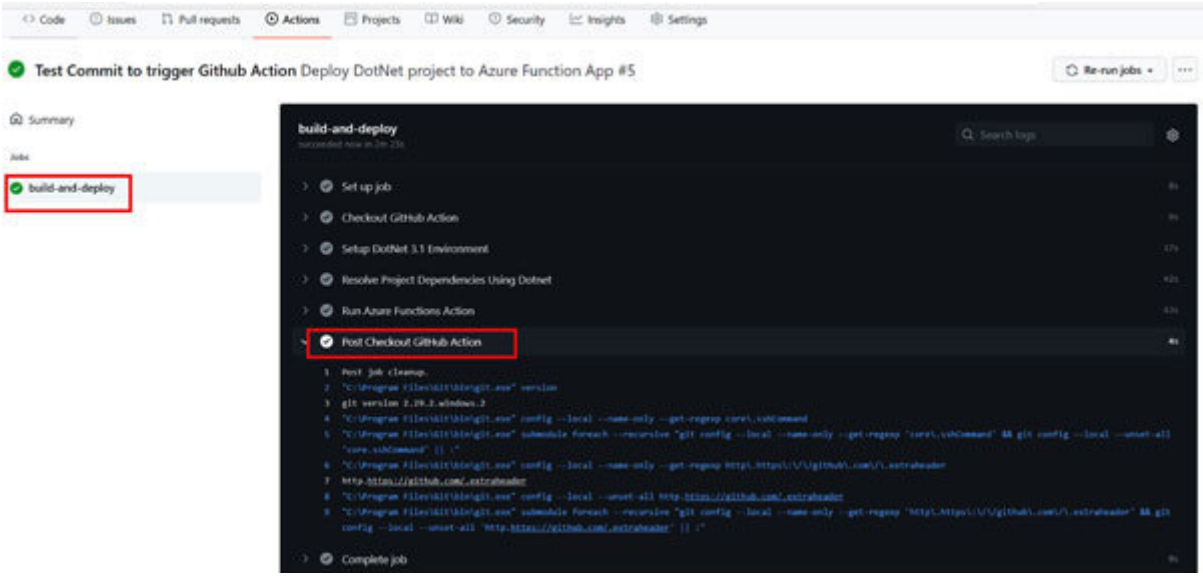


Figure 7.24: *Execution logs*

If your workflow execution succeeds, then let's execute and test your deployed Azure functions. Please refer to *figure* You should be able to see your changes in function output.

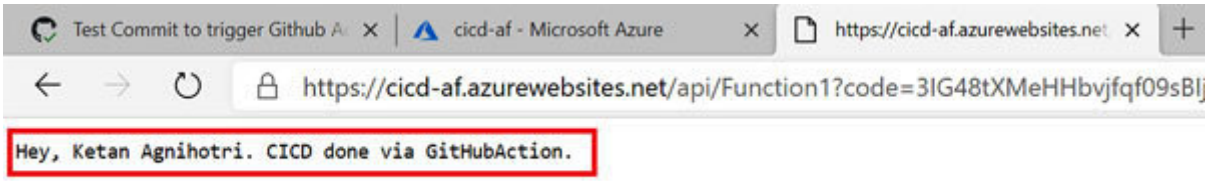


Figure *Execute and test the changes*

Note: You can get more information about GitHub actions by visiting

GitHub actions support two ways of generating deployment credentials, publish the profile and Azure service principal for RBAC. We have implemented build and deploy using publish

profiles. You may visit https://docs.microsoft.com/en-us/azure/azure-functions/functions-how-to-github-actions?tabs=dotnet#deploy-the-function-app for more information about Azure service principle. The GitHub actions use the kudu zip deploy method for deployment of Azure Functions.

Azure Functions deployment slots

Azure Functions deployment slots enable you to run different instances of Azure Functions. Slots are different environments like staging, test, dev, prod, and so on publicly exposed Azure Functions. You can swap these instances as per your need as and when it is needed. Slots are currently not available for the Linux consumption plan. You can have multiple slots for a Function app running under one app service plan. Traffic redirection using slots can reduce latency. You can use different slots for different environments. After swapping the slots, you can easily reverse the swap and bring the last known instance up and running if you face any issues.

When swapping slot, one slot is considered the source and the other one as the target slot. Steps like apply settings, wait for restart and availability, update the routing, and repeating these three operations for another slot will ensure zero downtime.

Creating deployment slots

Navigate to **Deployment slots** and click on **Add** You must provide the name of the slot and click on Please refer to *figure*

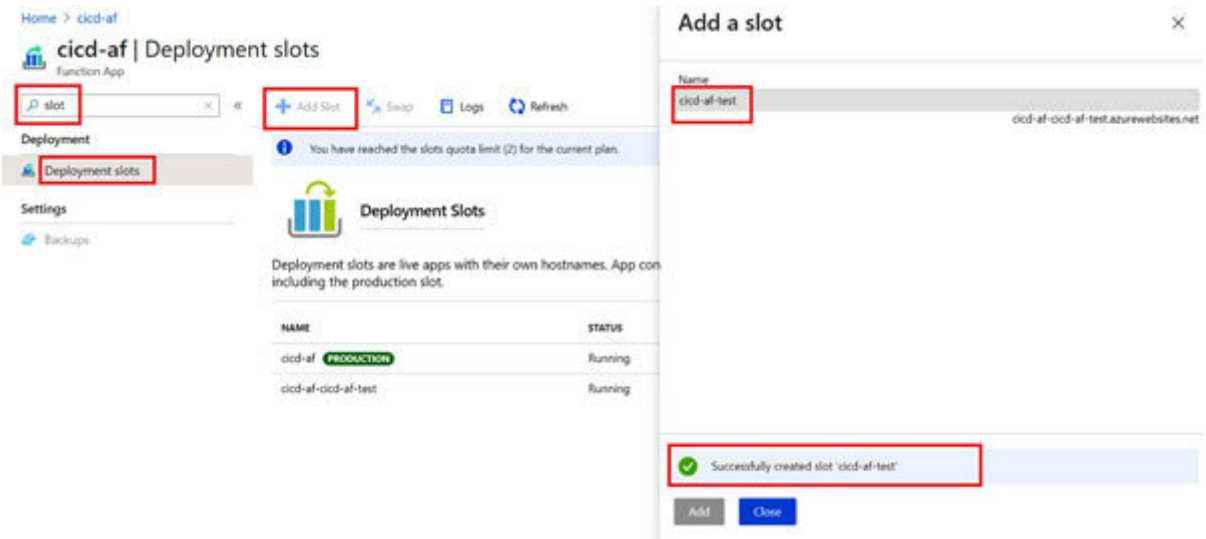


Figure 7.26: *Add new slot*

Swap slots

To swap the slots, you must select deployment slots and click on In a swap window, you must select the source and target slot. Please refer to *figure*

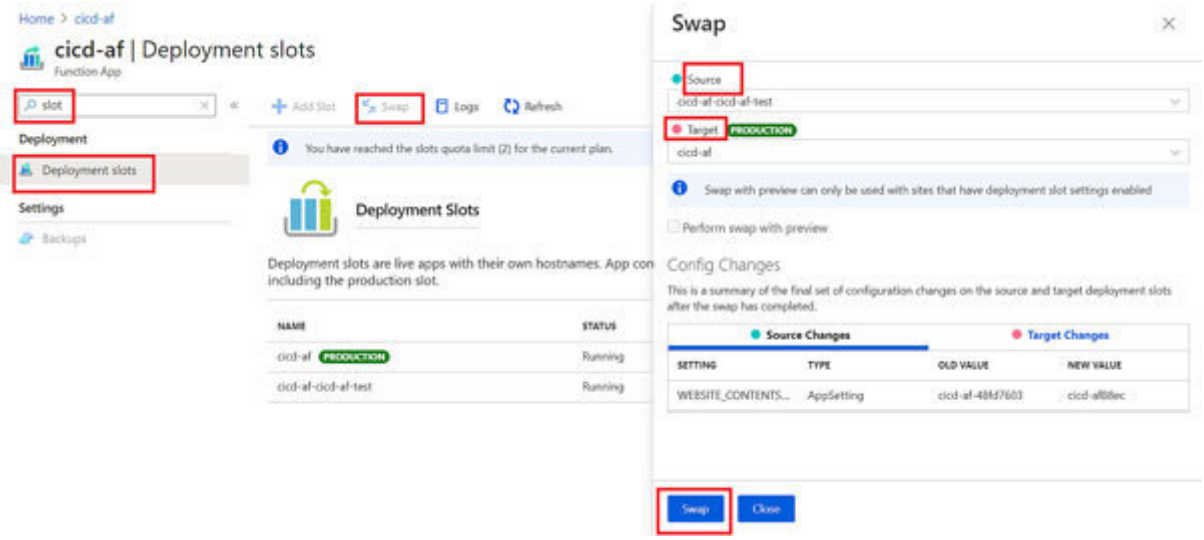


Figure 7.27: *Swap Slots*

You can review config changes at the time of swapping the slots. If the swap results in an error, then you can easily roll back the swap operation.

Conclusion

In this chapter, you learned the deployment methods and technologies supported. You also understood the steps to set up continuous build and continuous deployment. Depending on your needs, you can choose any techniques to set up, build and deploy automation of your function app. GitHub actions and Azure DevOps have inbuilt CICD pipelines templates and workflows to choose from. You can also start from a blank template and search for tasks from the marketplace to add.

Points to remember

Continuous deployment and slot features are not available for the Linux consumption plan. The consumption plan cannot be explicitly defined for Linux and will be created automatically.

The number of slots available to a function app depends on the plan. The consumption plan is allowed one deployment slot only. Additional slots are available for apps under the app service plan.

If your source code is in Azure Repos or GitHub, you must write permissions for your subscription.

If you configure continuous integration, then you must not edit your source files in the functions portal.

In zip deployment, existing files are deleted from your function app if not available in the **.zip** file, and other files are overridden.

CHAPTER 8

Troubleshooting and Monitoring Azure Functions

In the previous chapter, we learned the deployment aspect of functions using the common deployment tools. In this chapter, we will understand various ways to monitor and troubleshoot Azure functions running on production. We will learn how to connect to various monitoring and analytics options n Azure, such as Application Insights, Azure monitoring logs, and so on. We will also learn to integrate the Application Insights to Power BI to create rich visualizations for better analytics of the Azure Functions.

Structure

In this chapter, we will cover the following topics:

Integrating Azure Functions with Application Insights

Troubleshooting Azure Functions

Monitoring Azure Functions

Integrating Application Insights with Power BI

Objective

After studying this chapter, you should integrate your Azure Functions with Application Insights and learn ways to troubleshoot issues in your Azure Functions. You will also learn how to use Azure Diagnose to solve problems in real-time, connect to Azure monitoring logs, create monitoring alerts for your function, and implement performance monitoring for your functions. At the end of the chapter, you will learn how to authenticate and integrate the Application Insights to Power BI.

Integrating Azure Functions with Application Insights

Azure Functions provides in-built integration with Application Insights to collect telemetry generated, including the application logs. Application Insights provides a rich option of visualizations for the telemetry data. You can also choose to drill into the error logs, events, and metrics with the help of exceptions dashboards and Kusto query.

Enabling Application Insights

To enable **Application Insights** for Azure Functions, follow these steps:

Click on under the **Settings** section of your function app, as shown in *figure*

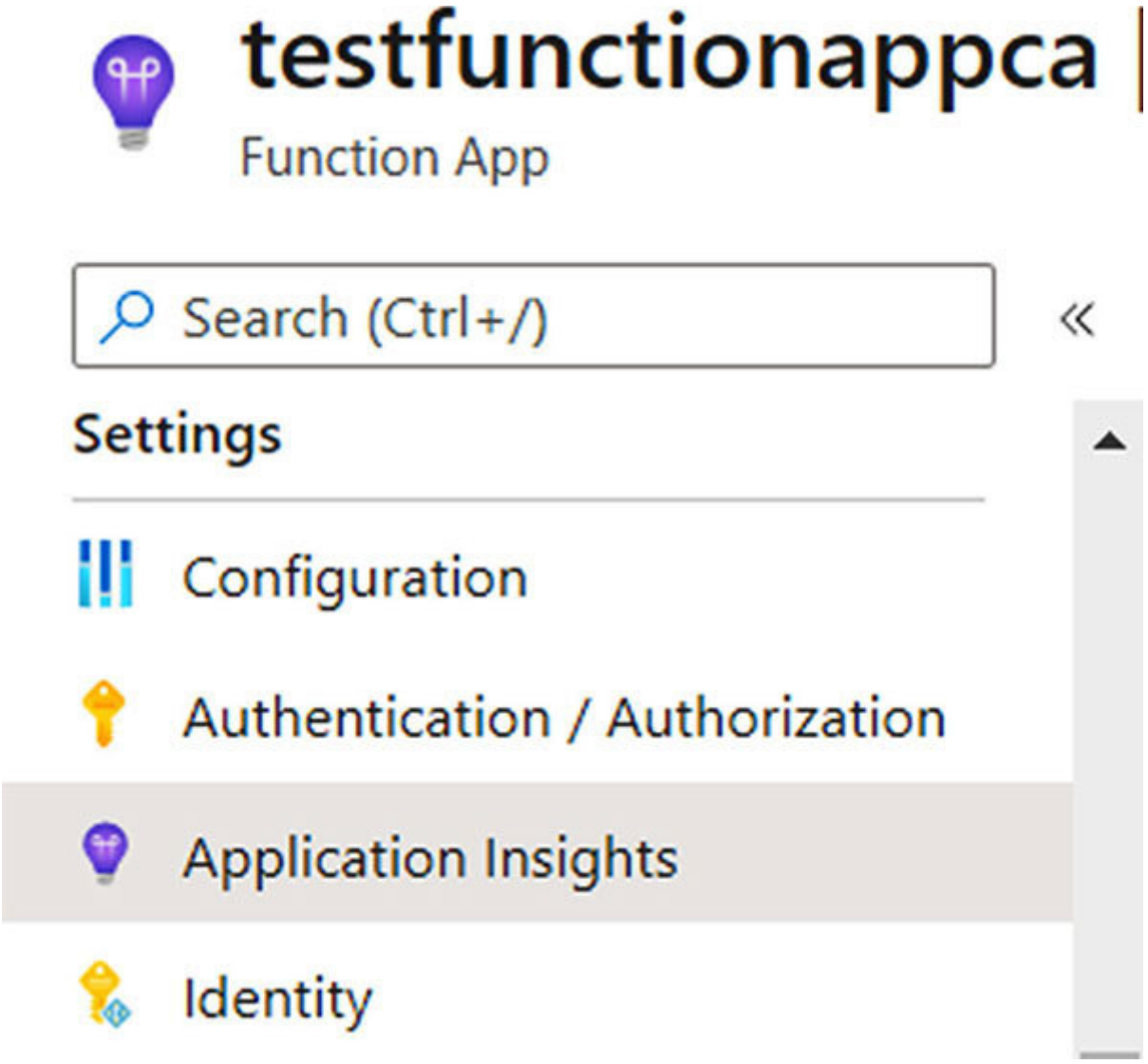


Figure 8.1: *Application Insights option in Azure Functions*

On the Application Insights setting page, click on **on Application** as shown in *figure*

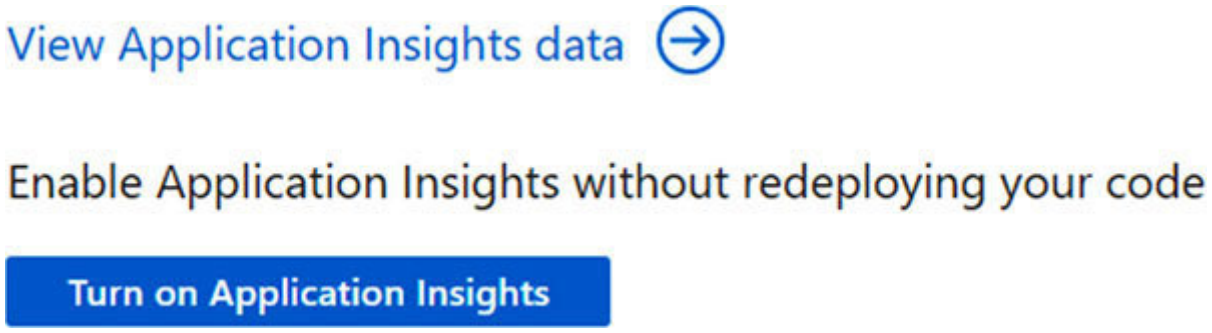


Figure 8.2: *Application Insights Settings page on Azure Function*

Choose to Create a new Application Insights resource or choose an existing one, as shown in *figure*

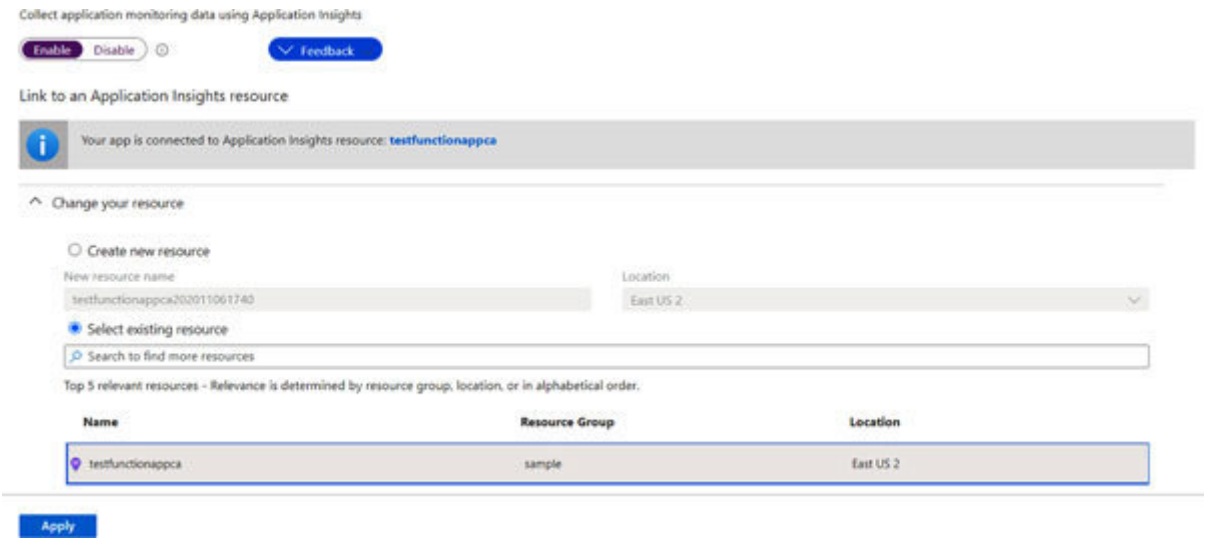


Figure 8.3: *Application Insights enabling on Azure Functions*

Click on

The Application Insight is linked to the function app through the instrumentation key in the app settings with the key name as as shown in *figure*

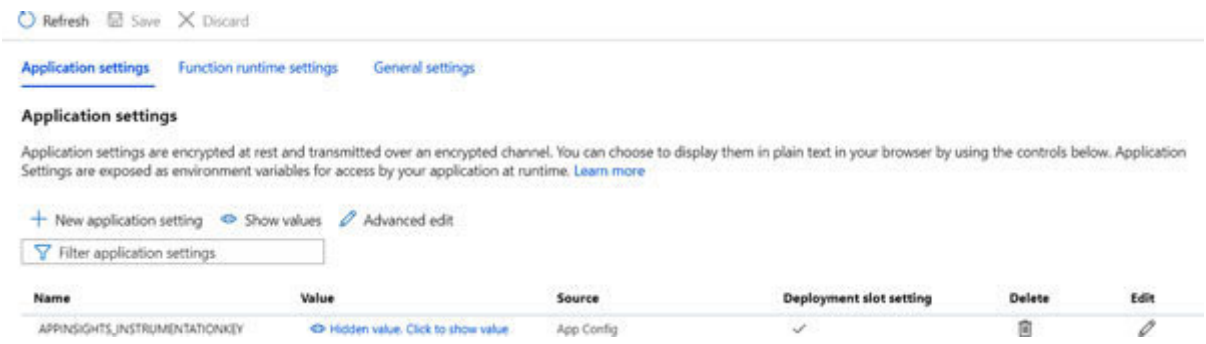


Figure 8.4: *Azure Function Application Settings*

You can configure the data that the functions send to Application Insights in the **host.json** file.

Configuring log level

Every log entry written by function has a log level assigned. A log level is an integer value that defines the severity of the particular log entry. The following is the list of the log level and a high-level description:

description:
description: description: description: description: description: description: description: description: description:
description: description: description: description: description: description: description: description: description:

description: description: description: description: description: description: description: description:

description: description: description: description: description: description: description:

Table 8.1: *Log level descriptions*

The log level for the function app is defined in the **host.json** file. One such example is given as follows:

```
{
"logging": {
"fileLoggingMode": "always",
"logLevel": {
"default": "Information",
"Host.Results": "Error",
"Function": "Error",
"Host.Aggregator": "Trace"
}
}
}
```

Configuring the aggregator

The Azure function runtime aggregates execution logs over 30 seconds period or 1,000 runs, whichever comes first. This setting is configured in the **host.json** file and can be changed according to the application needs. The following is the section example of the same:

```
{
"aggregator": {
"batchSize": 1000,
"flushTimeout": "00:00:30"
}
}
```

Query telemetry data using the Kusto Query language

Application Insights allows you to access the telemetry data represented in the form of tables. You can use the Kusto Query language to analyze or visualize this date. The following is the example of a query to view all the exceptions that have occurred in the last 30 minutes:

```
exceptions
| where timestamp > ago(30m)
```

There are multiple tables created by default in Application Insights to store different type of telemetry data, as follows:

follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows:

Table 8.2: *Application Insights default tables*

Application Insights also allows you to log other data important in the **CustomDimensions** field.

Note: The Kusto Query language is a read-only query language, which processes the data and returns results. You can learn about KQL on Microsoft documentation at https://docs.microsoft.com/en-us/azure/data-explorer/kusto/query/

Troubleshooting Azure Functions

It's really important for any application to be able to troubleshoot in case of any failure. Application Insights integration with Azure functions simplifies this for you.

Viewing exceptions in Application Insights

You can view the exceptions that occurred in your function app by going to the Failures page in your **Application** as shown in *figure*

testfunctionappca

Application Insights

Search (Ctrl+/)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems

Investigate

- Application map
- Smart Detection
- Live metrics
- Transaction search
- Availability
- Failures
- Performance
- Troubleshooting guides (previ...

Figure 8.5: *Application Insights Failures option*

Here you can view Failures that occurred in operations, dependencies, an exception occurred in function, or a role failure, as shown in *figure*

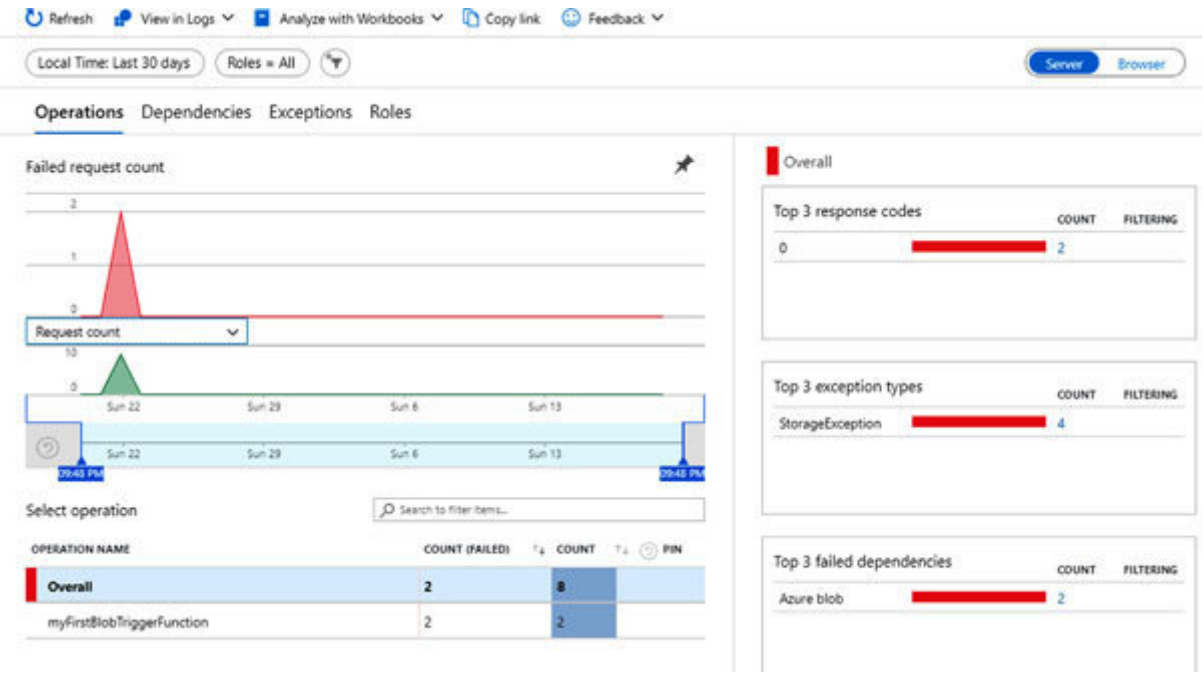


Figure 8.6: *Application Insights Failures page*

Click on **Dependencies,** or **Operations** based on the tab you view the details logs, as shown in *figure*

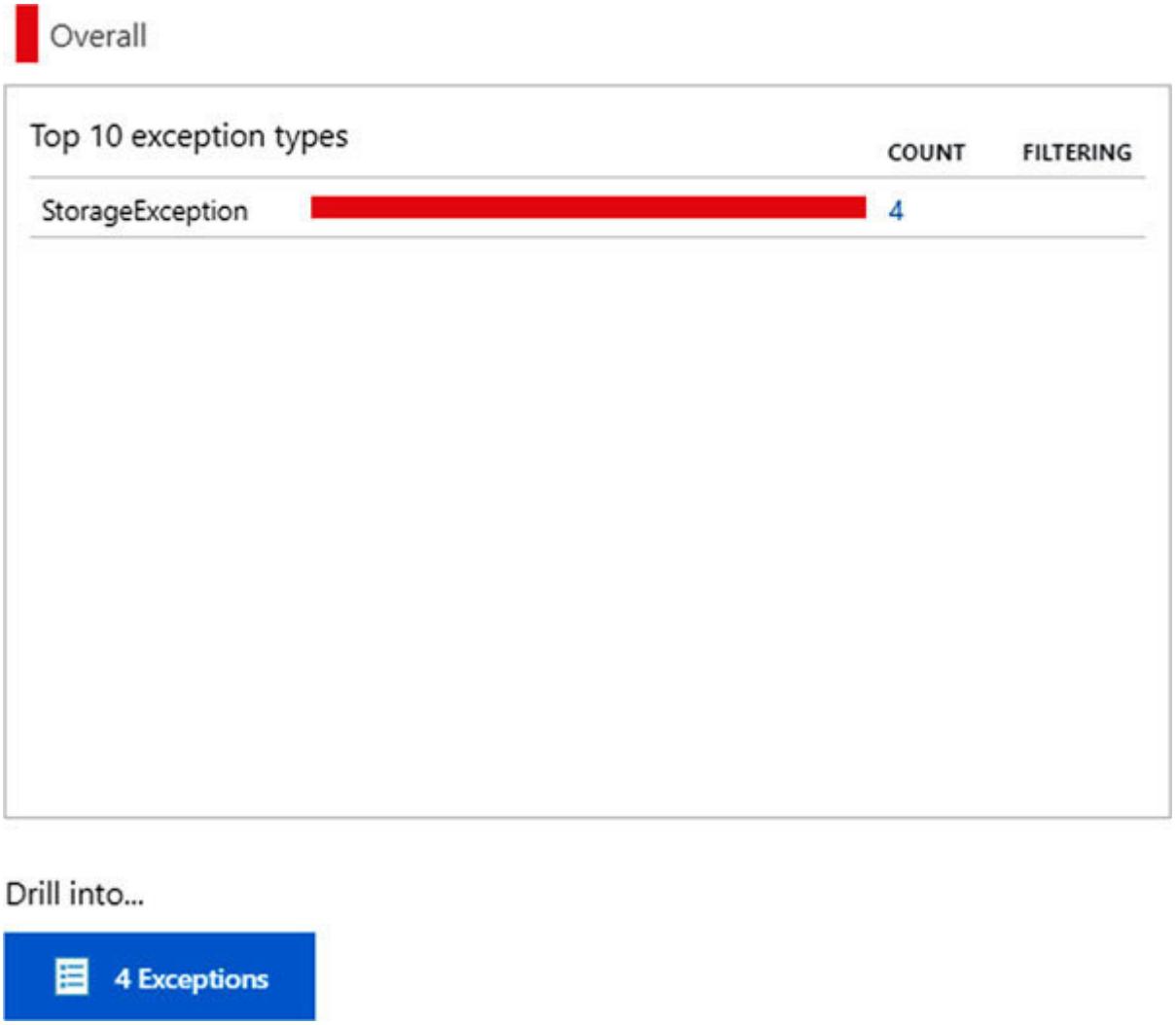


Figure 8.7: *Application Insights Top 10 exceptions section*

Click on the particular exception in the pane. This opens up on the right side of the page, as shown in *figure*

» Select a sample exception

Filtered on client_Type !=...

Suggested

11/22/2020, 11:25:46 PM
Exception while executing function: Functions.myFirstBlobTriggerFunction
Problem Id:
Microsoft.WindowsAzure.Storage.StorageException at Microsoft.WindowsAzure.Storage.Core.Executor.Executor+<ExecuteAsyncInternal>d__4`1.MoveNext

All

Sort by
Relevance

11/22/2020, 11:25:46 PM
Exception while executing function: Functions.myFirstBlobTriggerFunction
Problem Id:
Microsoft.WindowsAzure.Storage.StorageException at Microsoft.WindowsAzure.Storage.Core.Executor.Executor+<ExecuteAsyncInternal>d__4`1.MoveNext

11/22/2020, 11:25:46 PM
Exception while executing function: Functions.myFirstBlobTriggerFunction
Problem Id:
Microsoft.WindowsAzure.Storage.StorageException at Microsoft.WindowsAzure.Stora

ge.Core.Executor.Executor+<ExecuteAsynci

Figure 8.8: *Application Insights Top 10 exceptions details*

This will take you to the end-to-end transaction details of the particular exception, as shown in *figure*

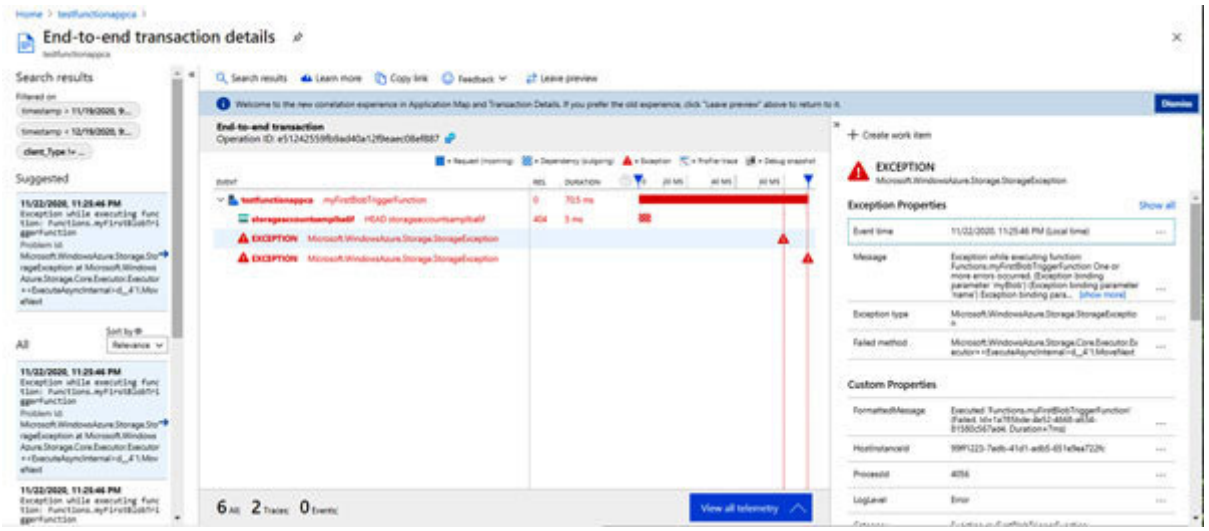


Figure 8.9: *Application Insights End-to-end transaction details page*

This is extremely helpful to understand the various actions performed by the function app, the time taken for each action, and the success or failure of each action. You can track a particular invocation of the function to understand the reason for the failure.

Troubleshooting guides

Although the preceding features help understand most cases, it is helpful to have a troubleshooting guide that can be defined with steps to understand the reasons and actions to resolve. For this, Application Insights has added a cool option of creating troubleshooting guides (In preview at the time of writing this book). With this option, you can select an already existing template created or create your troubleshooting guide from scratch, as shown in figure

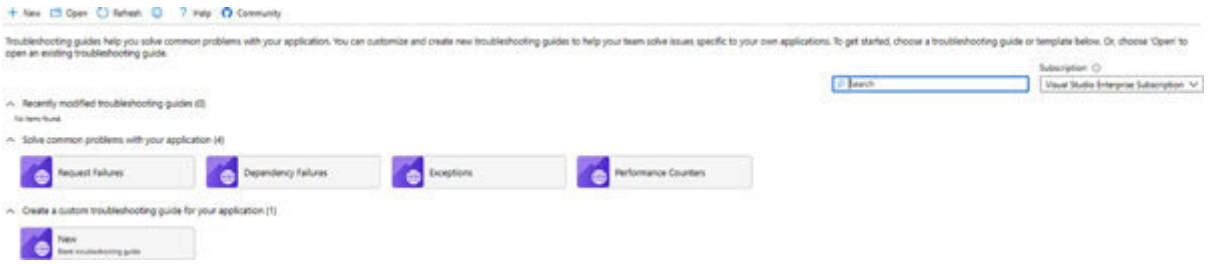


Figure 8.10: *Application Insights troubleshooting guide page*

Figure 8.11 shows the example of the request failures troubleshooting guide:

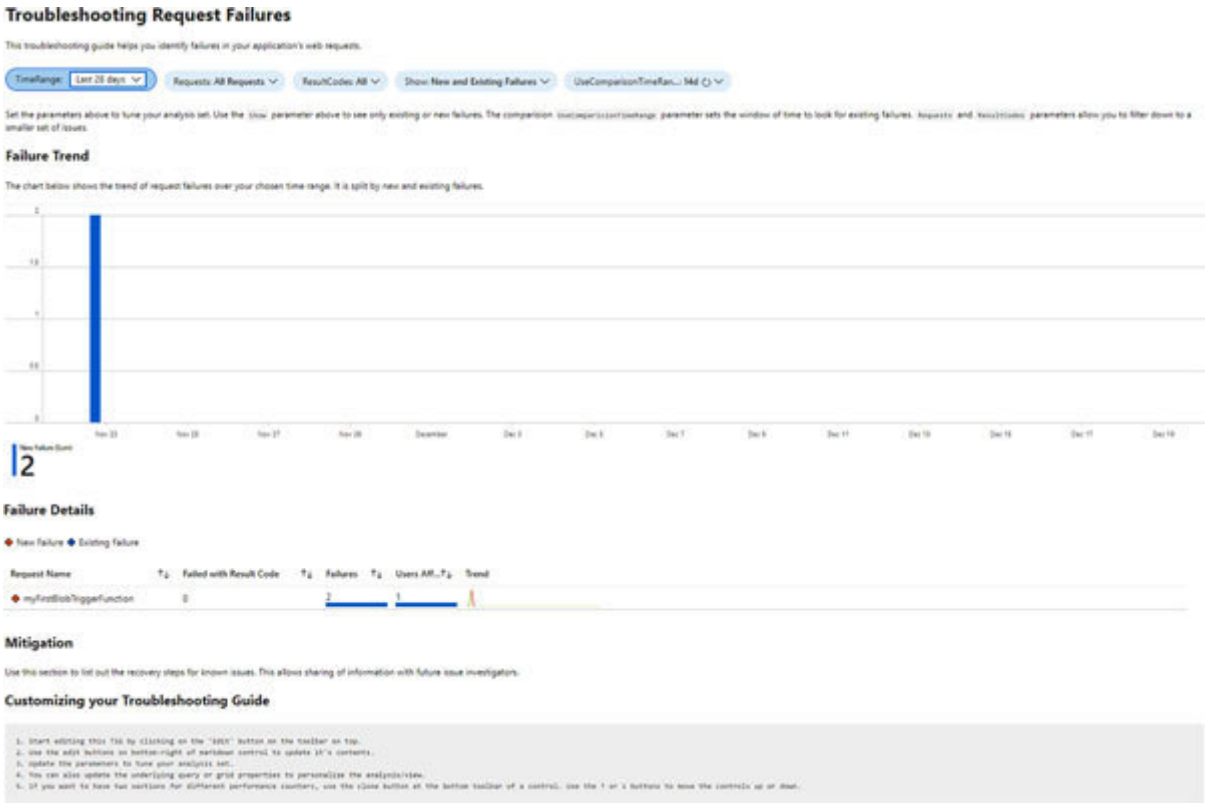


Figure 8.11: *Application Insights troubleshooting guide – request failures*

To open or create troubleshooting guides, click on the option under the section in the left menu of our Application Insights.

Diagnosing and solving problems

Azure Functions diagnose and solve problems in an intelligent and interactive feature that helps troubleshoot the function app. This helps to troubleshoot and solve issues with no extra configuration or cost. It provides you with the right information for quicker and better troubleshooting and resolving the issue.

Starting Azure Functions diagnostics

Follow these steps to start function diagnostics:

Click on **and solve** under the **Overview** section of your function app, as shown in *figure*

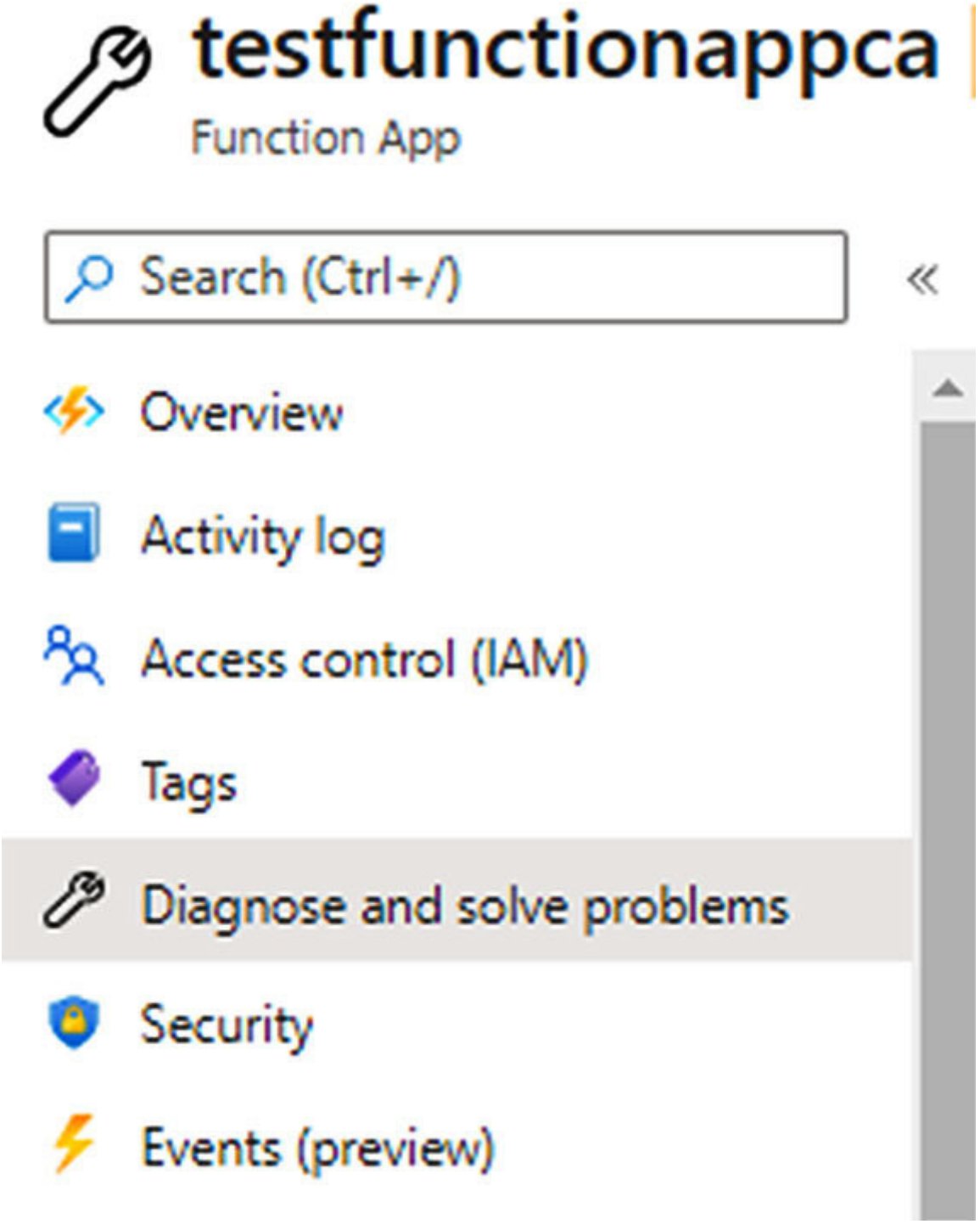


Figure 8.12: *Diagnose and solve problem option in the function app*

On the **Diagnose and solve problems** page, select the troubleshooting category. You can also search for categories or

issues using the search option provided at the top, as shown in *figure*

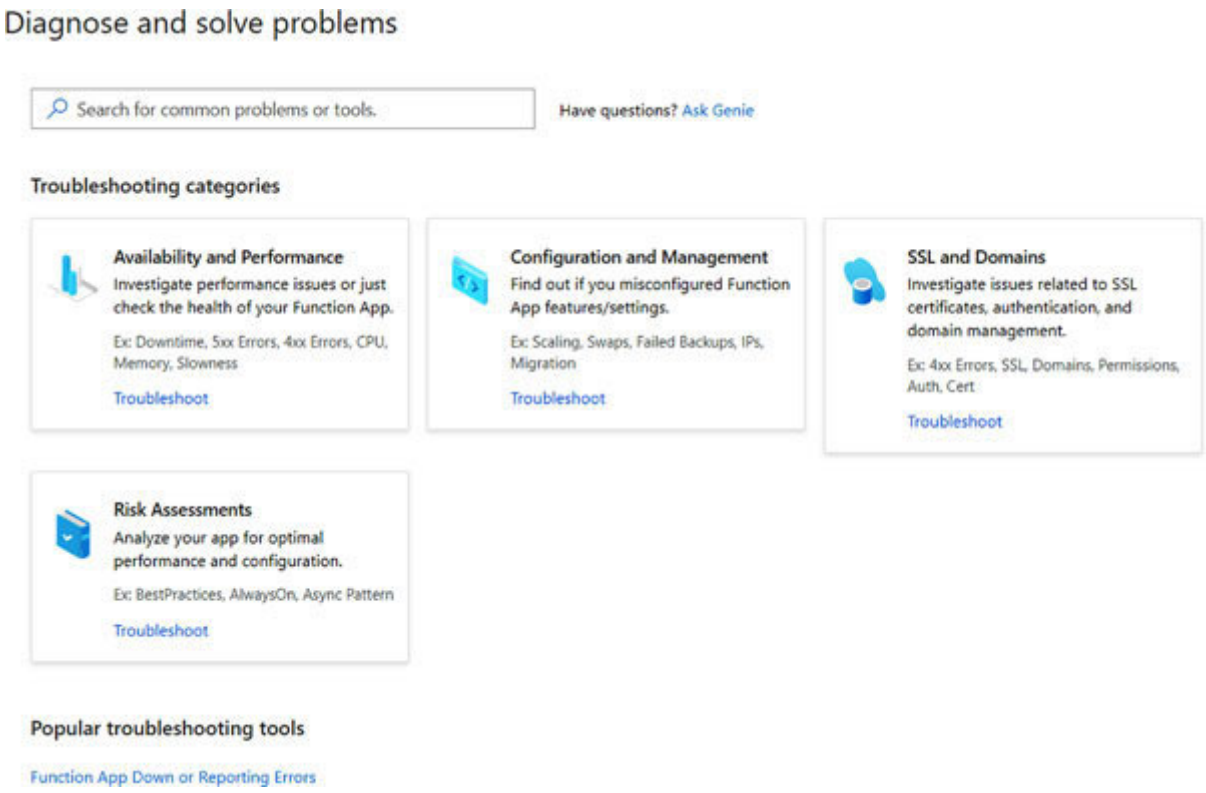


Figure 8.13: *Diagnose and solve problem page*

This will open the diagnostics page for that category with multiple troubleshooting topics in the left menu and the summary of the metrics of your function app for that topic, as shown in *figure*

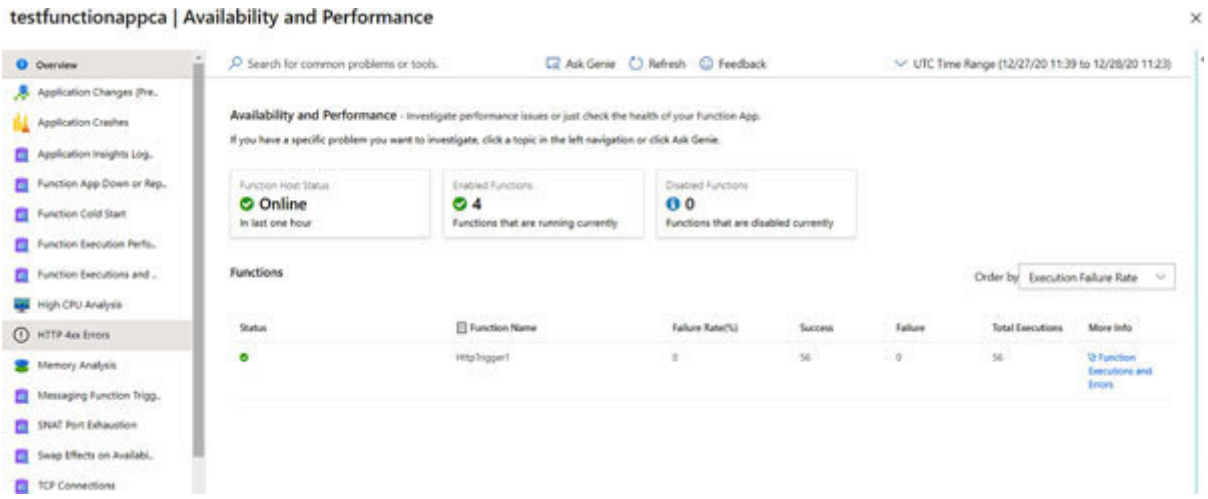


Figure 8.14: *Diagnose and solve problem – availability and performance*

Click the topic you are looking for to troubleshoot your function from the left menu. This will provide a detailed diagnostic report for your function, as shown in *figure* Diagnostic reports use status icons to specific issues and provide detailed descriptions of the issue, recommended actions, related metrics, and help docs.

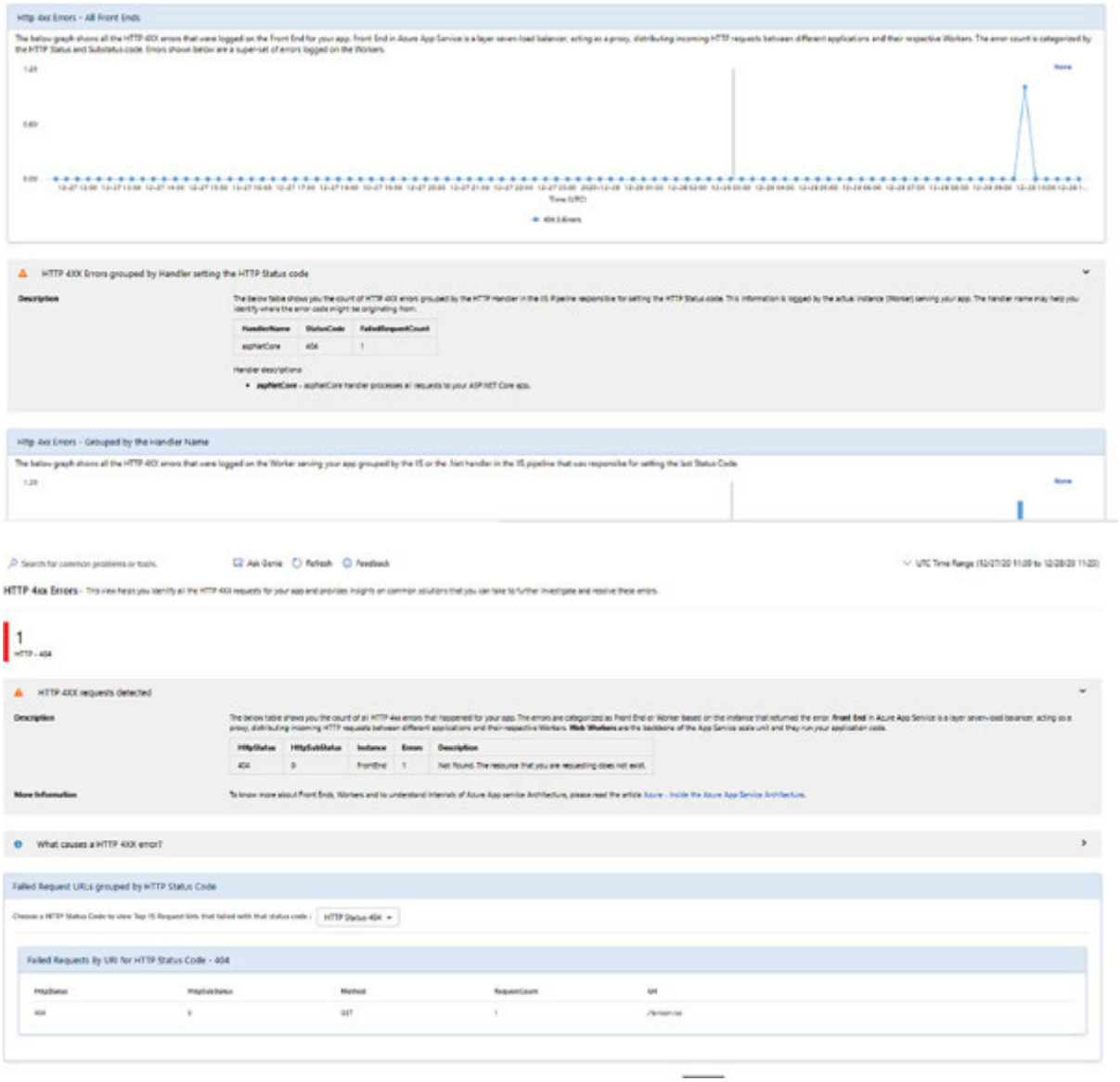


Figure 8.15: *Diagnose and solve a problem – diagnostic reports*

Monitoring Azure Functions

Azure provides multiple and powerful options to monitor your function app, including live log streaming, Azure monitor logs, alerts, and application maps. We will discuss each of these details as follows:

Live streaming

You can use the live metrics stream provided by Application Insights under the Investigate section, the same as other options we have discussed, to monitor and troubleshoot your live web application. It also allows you to view stack traces of the failed requests and exceptions along with profiler and snapshot debugger support. This is a very powerful diagnostic tool for your production functions. Refer to *figure 8.16* for a sample live metric streaming:

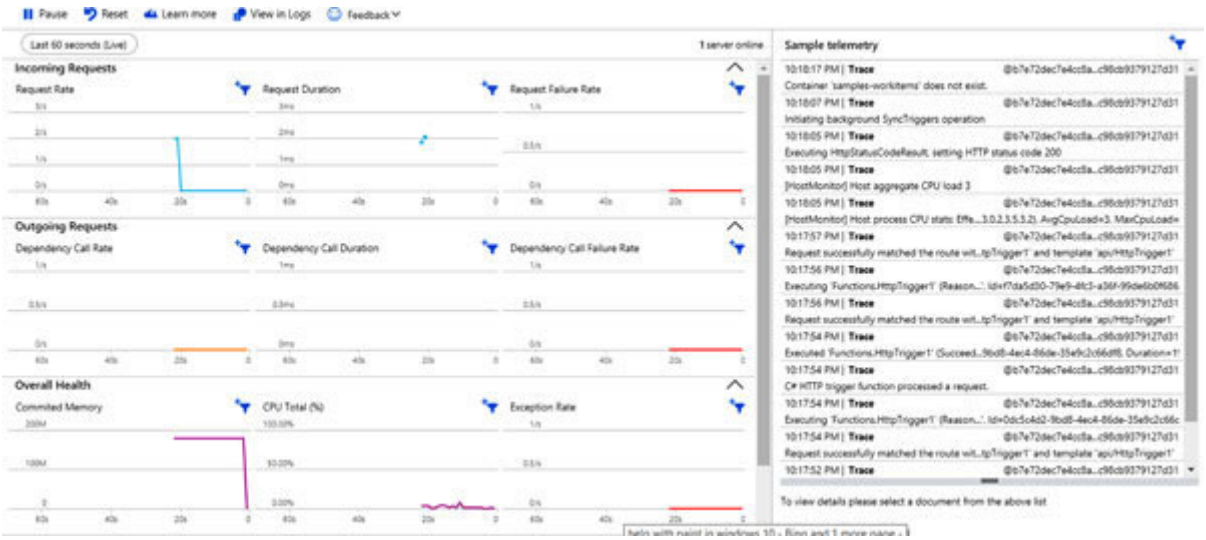


Figure 8.16: *Application Insights live streaming*

To open a live stream, click on the option under the section in the left menu of our Application Insights.

Azure Monitor Logs

Integrate your Azure function with Azure Monitor Logs, which allows you to collect query logs from different resources in a single place. These logs can be easily analyzed and visualized using the powerful Kusto Query language.

Follow these steps to integrate your Azure function with Azure Monitor Logs:

Click on under the **Monitoring** section of your function app, as shown in *figure*

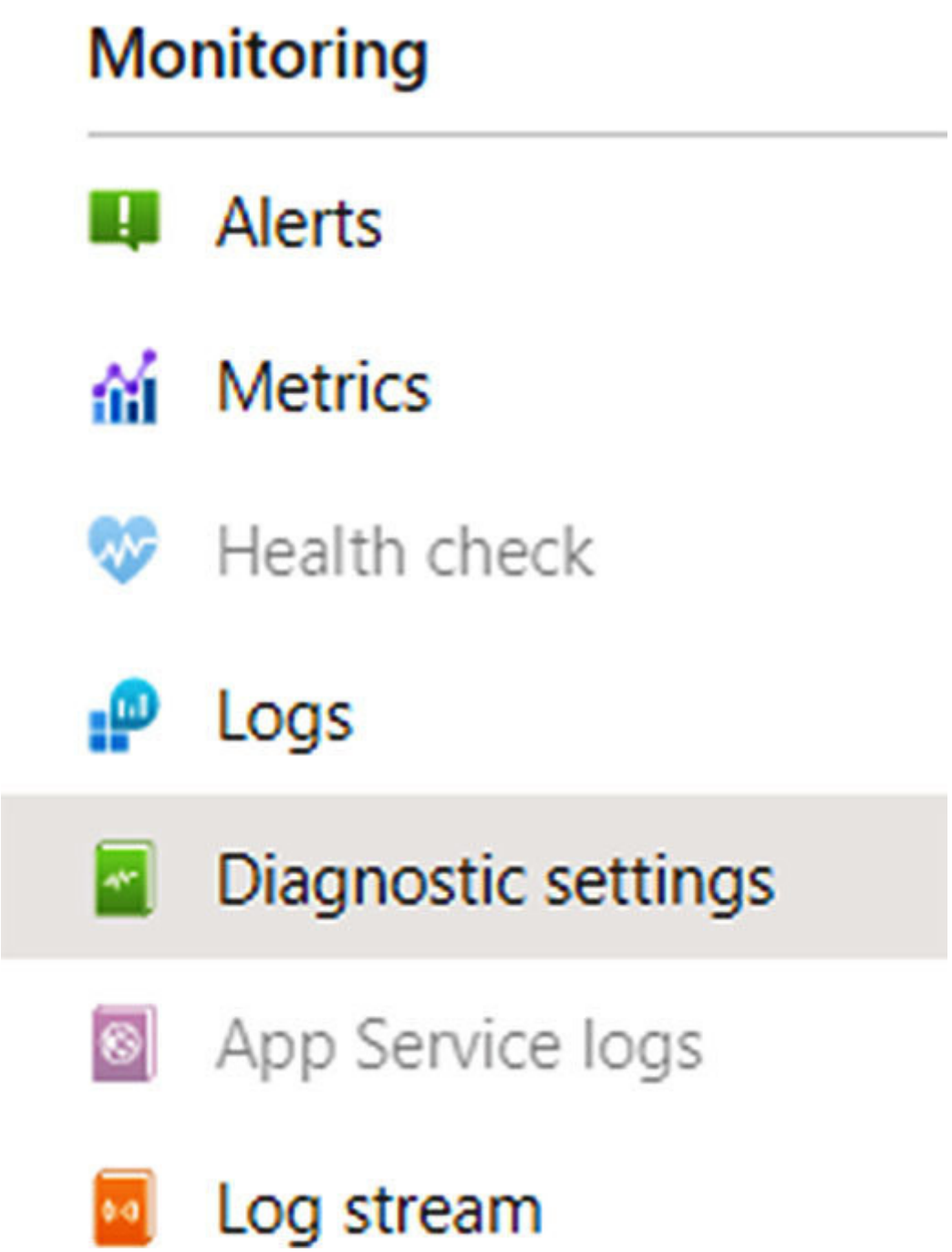


Figure 8.17: *Azure function diagnostics setting menu option*

On the **Diagnostic settings** page, click on **diagnostic** as shown in *figure*

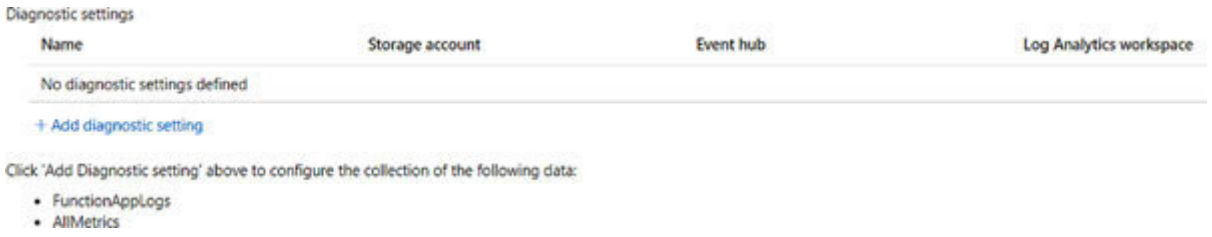


Figure 8.18: *Azure Function adding diagnostics setting*

On the settings page, shown in *figure*

Choose **FunctionAppLogs** under **Category**

Choose **Send to Log Analytics** under **Destination details,** and then select your **Log Analytics**

Enter a diagnostic settings name.

Click on

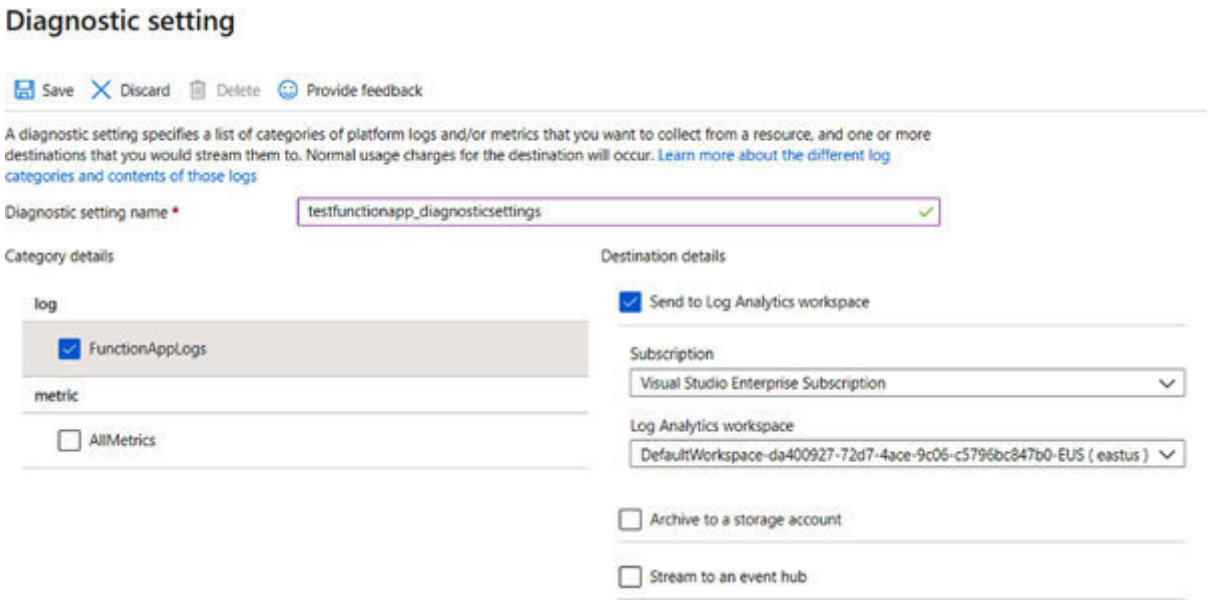


Figure 8.19: *Azure Function diagnostics setting page*

Once your Azure function is integrated with monitoring logs, you can query the logs, as shown in *figure* by clicking on logs under the monitoring section of your Azure function left menu. You can also go to the respective log analytics workspace to query logs.

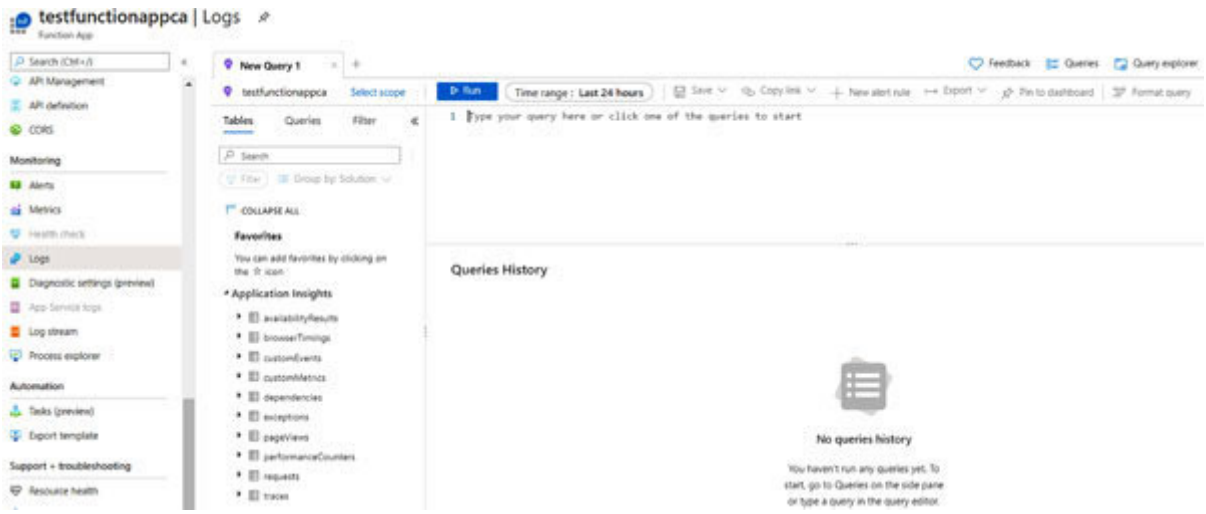


Figure 8.20: *Azure Function logs page*

You can query **FunctionAppLogs** to get the Diagnostics Logs. You can filter in case of the same log analytics workspace used for multiple functions, as done in the following query:

```
FunctionAppLogs
| where FunctionName == "name>"
```

Any exception is logged in the same table with details in the column.

Azure activity log

Activity log on Azure is a platform level log that provides details about the events for the resource or subscription. Activity logs keep track of any changes that happened to your resource, such as resource creation, modification, deletion, etc. For example, the creation of Azure function, update on Azure diagnostics settings, and so on.

Activities are enabled by default, and you don't need to configure anything to get these logs. To view activity logs, click on the option in the overview section of the left menu of your function app. It will automatically filter the logs for your Azure function. You can add more filters by clicking on the option.

You can click on a particular log to view the change history, which will show you all the changes in that event. Change history shows the previous and new values side by side in an easy-to-track manner with changes highlighted in red and green color.

Activity logs are very useful to track when and who made changes to your function app and helps in security audit purposes.

Monitoring alerts

Monitoring alerts are wonderful tools to notify you of any issue or failure in your Azure Function through sms/e-mail or other ways. This helps you to react immediately and resolve the issue on your production systems. These alerts can be created for metric values, log queries, activity logs, function health, and so on.

Alerts are created on Azure by defining an alert rule, consisting of the target resource and the alert criteria. The following are key attributes of an alert rule:

Target resource: A target resource for which alert to be defined. In this case, it will be an Azure Function.

Alert signal: Event to be monitored to generate an alert. The signal types for monitoring vary based on the target and can be metrics, log search queries, or activity logs.

Alert condition: A condition defines the criteria for the signal that generates an alert. The example can be CPU percentage > 80% or total HTTP response code 4xx in last 30 mins > 100, and so on.

Severity: The severity of the particular alert based on business criticality can range from 0 to 4, with 0 being critical and four being verbose.

Action: The specific action needs to be taken when the alert condition is met. Actions are defined in Azure Alerts with the help of Action Group. Multiple alerts can share the same Action Group.

Actions in an Action Group can be executing an Automation Runbook, creating an ITSM ticket, calling another Azure Function, calling a Logic App, or calling a Webhook.

Creating an alert for your function

You can create an alert for your Azure Function using the following steps:

Click on under the **Monitoring** section of your function app, as shown in *figure*

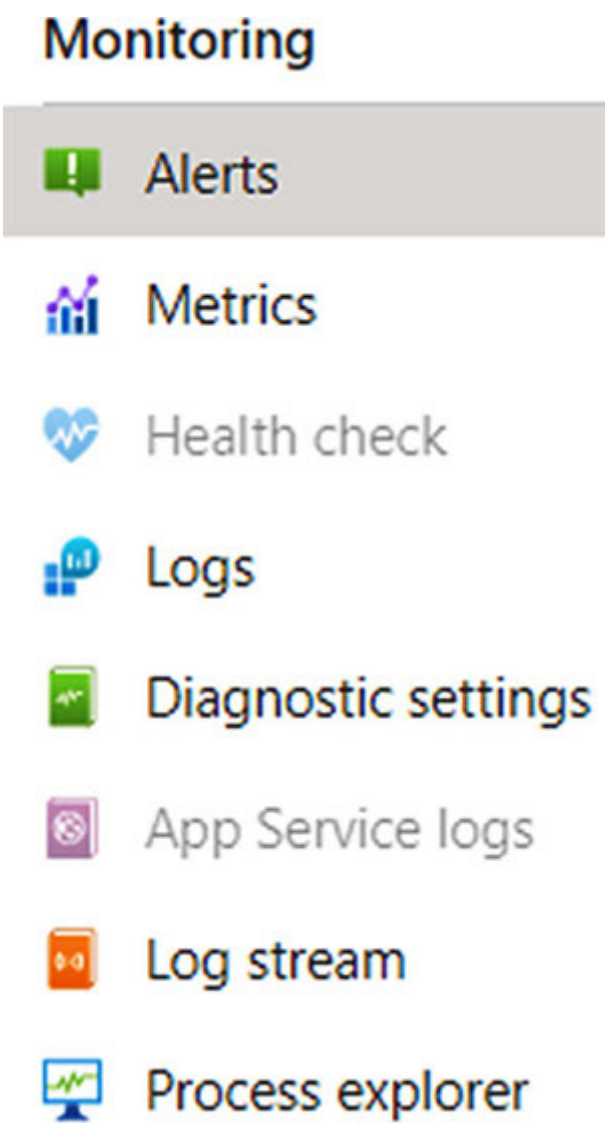


Figure 8.21: *Azure Function alerts menu option*

On the Alert page, click on the **alert** as shown in *figure*

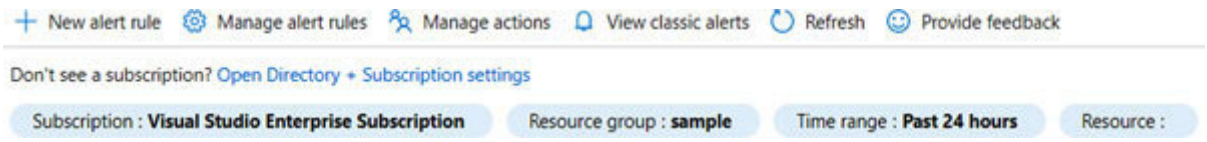


Figure 8.22: *Azure Function alerts page*

This will open the **alert** page with the scope already selected as your function, as shown in *figure*

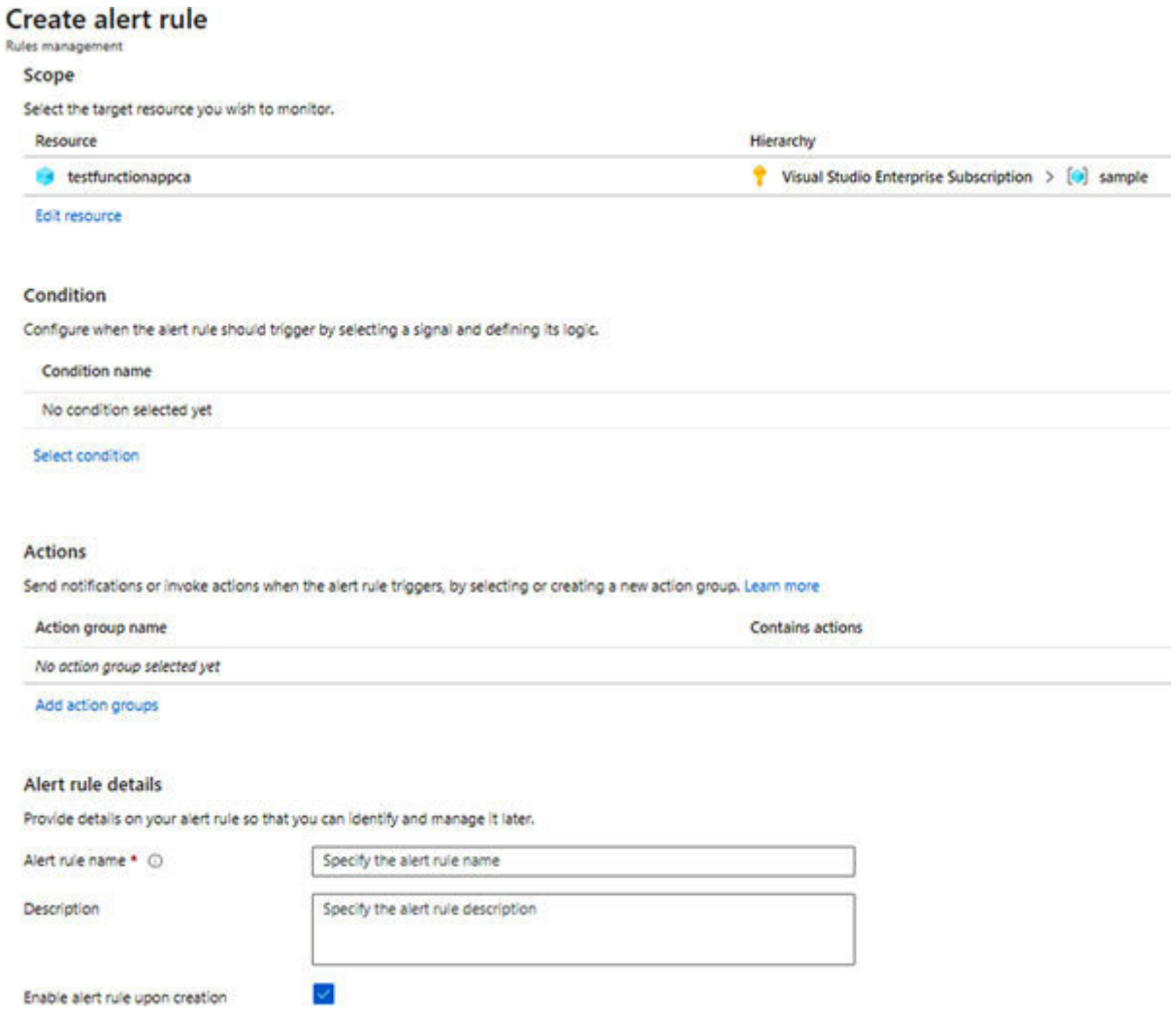


Figure 8.23: *Azure Function create alert rule page*

Follow these to select conditions on the **alert** page.

Click on **Select**

This will open the right pane to configure the signal logic, as shown in *figure*

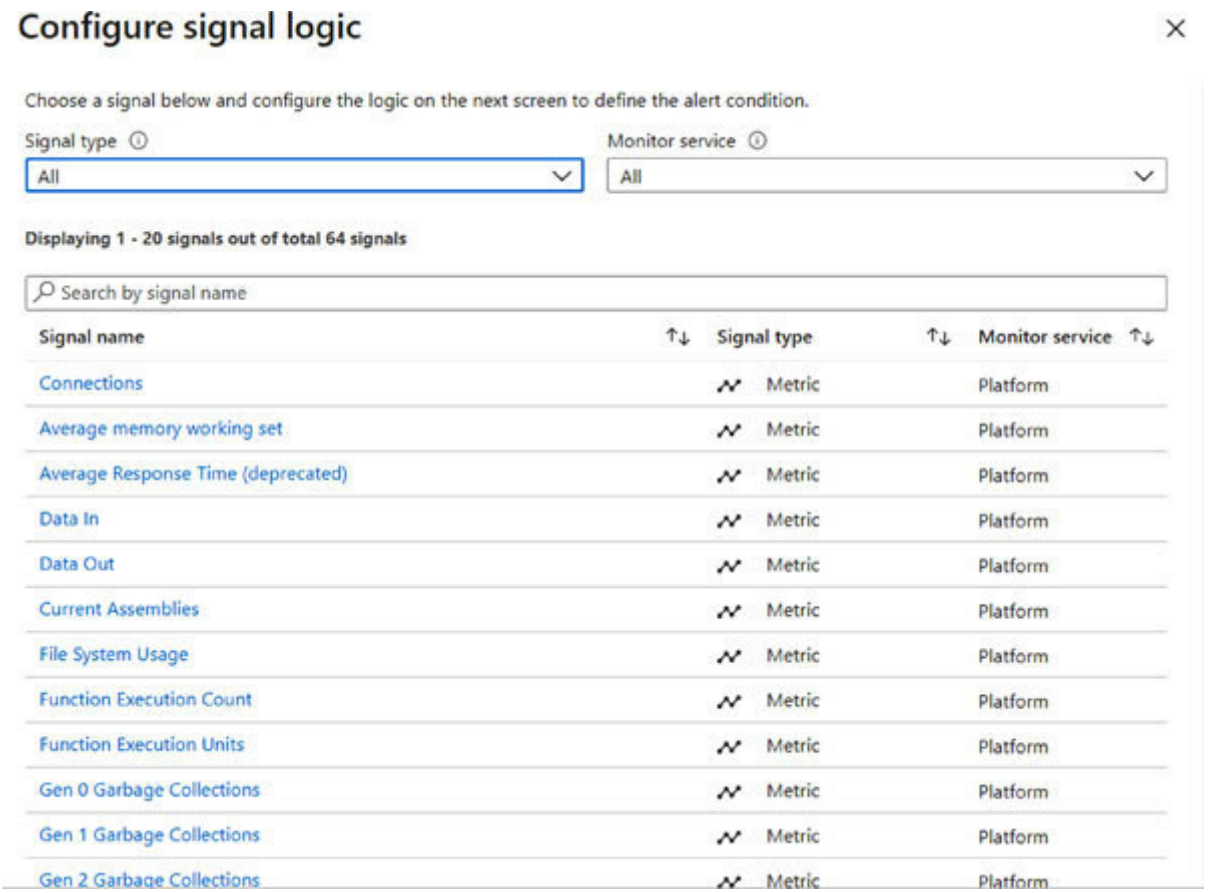


Figure 8.24: *Monitoring alert select signal type*

Select the particular signal you want to generate alerts on, for example, HTTP 404.

This will ask for the alert logic. Select **Aggregation Threshold Evaluation Period,** and **Evaluation frequency** based on your need. Refer to *figure 8.25* for example.

Click on

Configure signal logic

Split by dimensions

Use dimensions to monitor specific time series. If you select more than one dimension value, each time series that results from the combination will trigger its own alert and will be charged separately. About monitoring multiple time series

Dimension name	Operator	Dimension values	
Select dimension	=	0 selected	Add custom value

Monitoring 1 time series ($0.1/time series)

Alert logic

Threshold

Static | Dynamic

Operator	Aggregation type *	Threshold value *	Unit *
Greater than	Total	100	Count

Condition preview

Whenever the total http 404 is greater than 100

Evaluated based on

Aggregation granularity (Period) *	Frequency of evaluation
30 minutes	Every 15 Minutes

Done

Figure 8.25: *Monitoring alert configure condition*

Follow these steps to add an action group on the **alert** page:

Click on **Add Action**

This will open the right pane to configure the signal logic, as shown in *figure*

Figure 8.26: *Monitoring alert action group*

Click on **action** This will open the **Create action group** page, as shown in *figure*

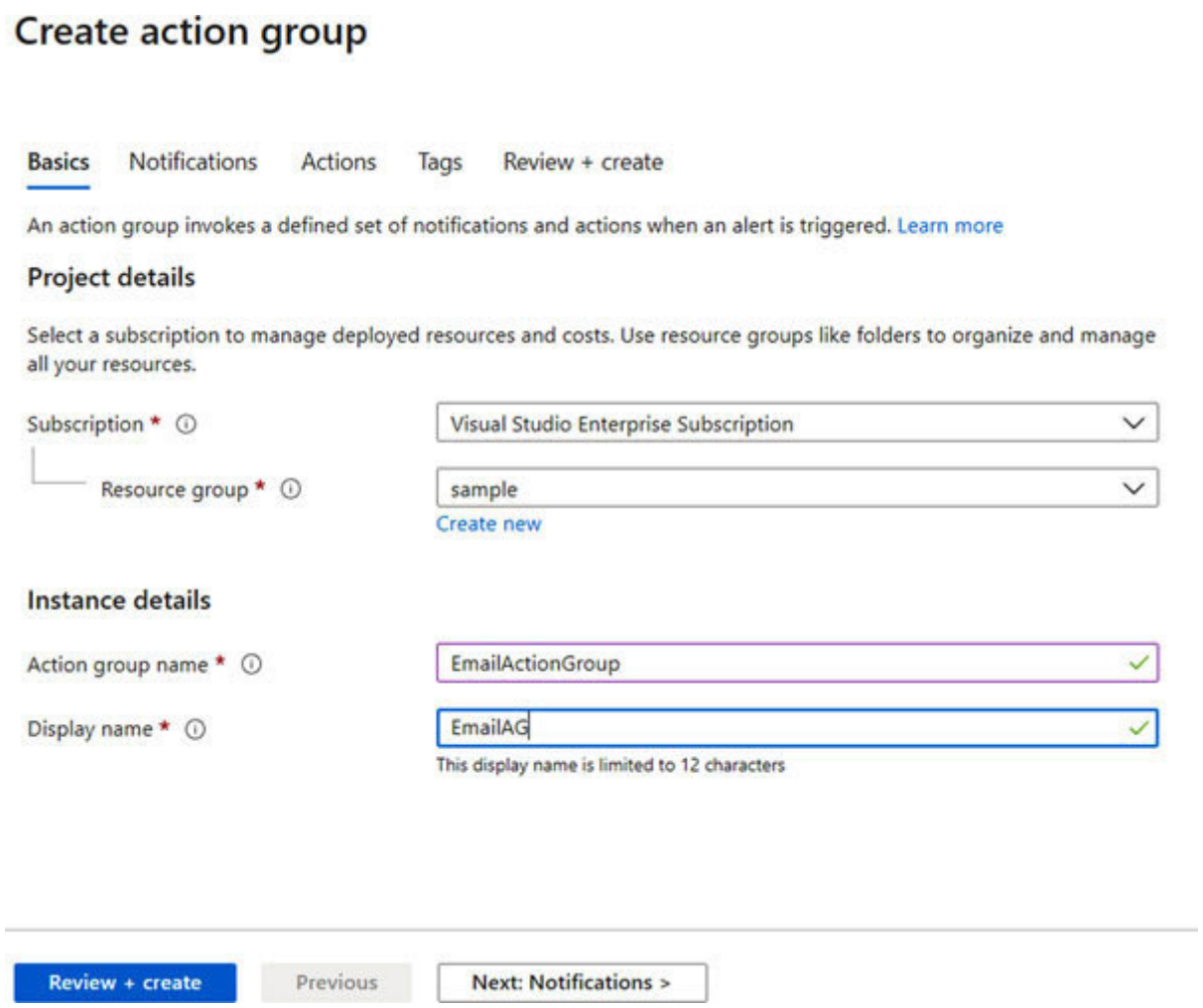


Figure 8.27: *Monitoring alert create new action group – basics page*

Provide the required details and click on

On the Notification page, select **Email/SMS message/Push/Voice** as the notification type and provide details on the right pane, as shown in *figure*

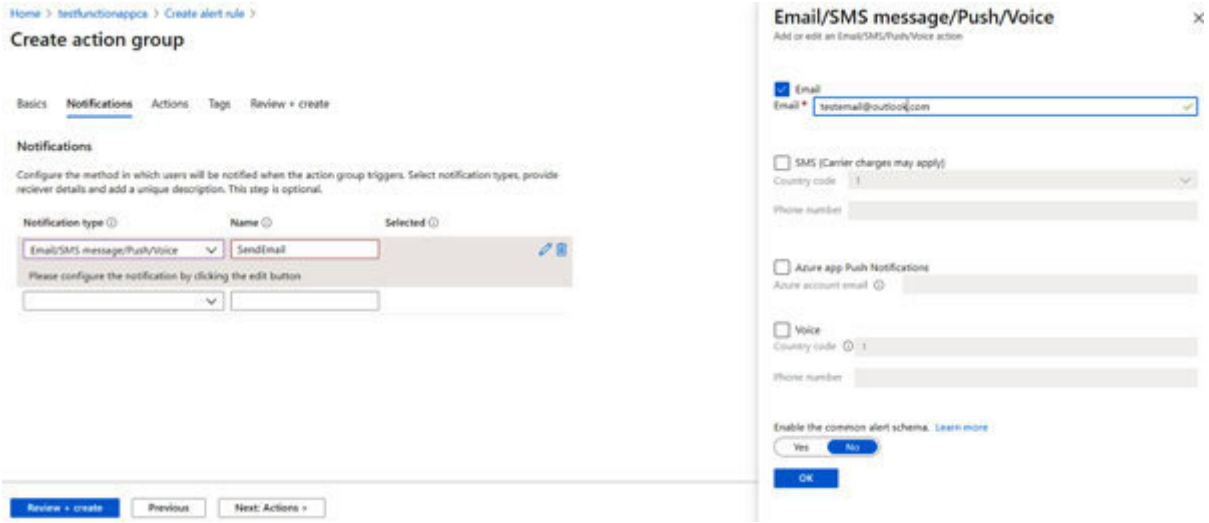


Figure 8.28: *Monitoring alert create new action group – notifications page*

Click on

On the **Actions** page, select the action type from the list of available options, as shown in *figure* If you don't want any action to be taken other than notification, skip this section.

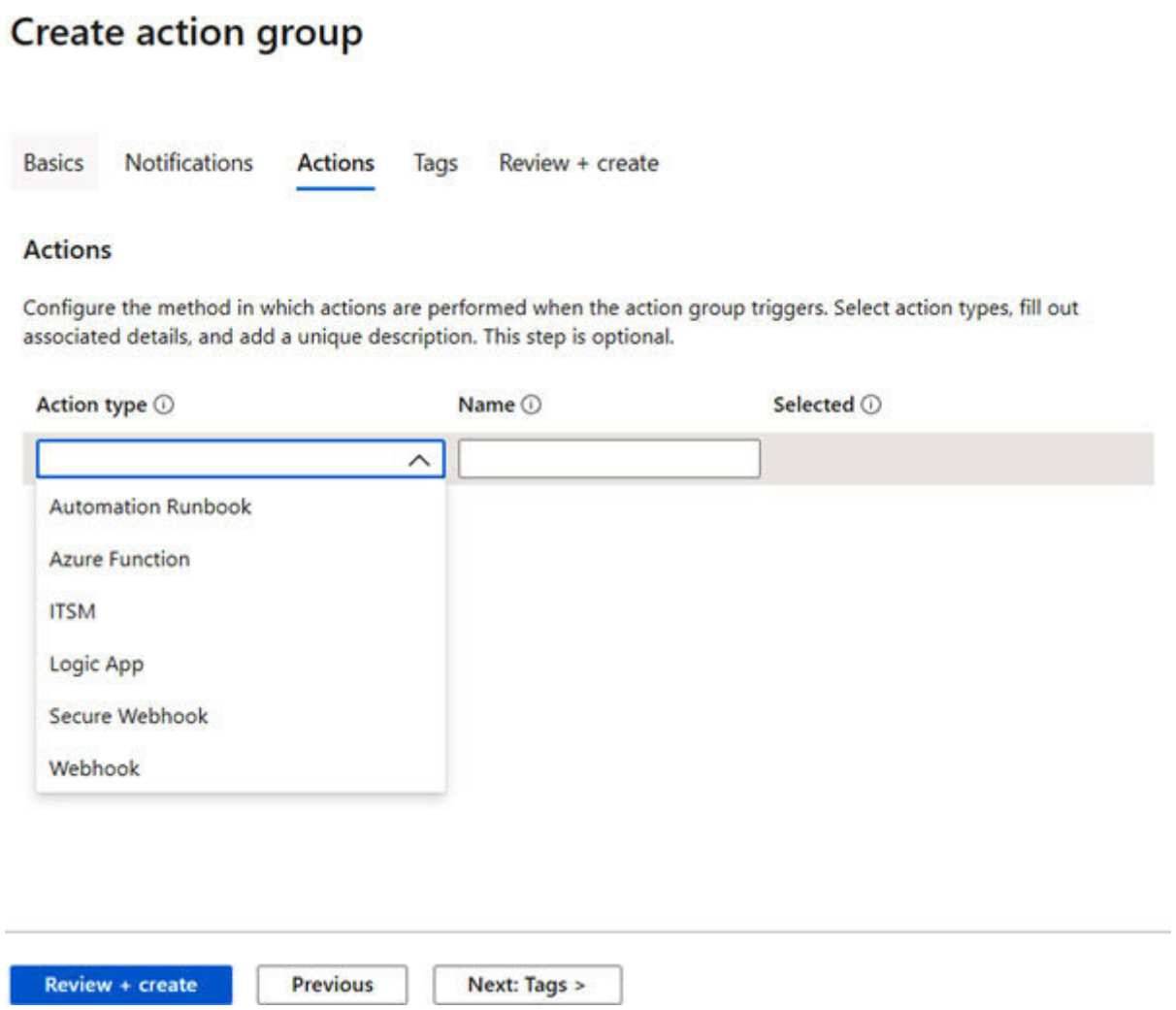


Figure 8.29: *Monitoring alert create new action group – actions page*

Click on **+**

Provide alert name, description, resource group, and severity as shown in *figure*

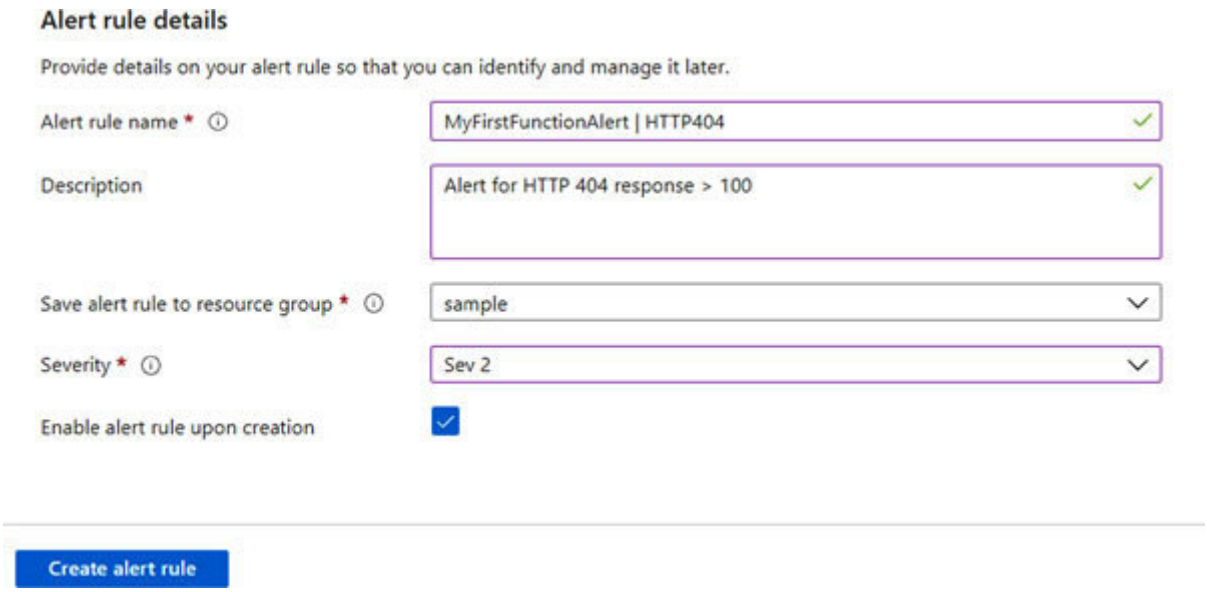


Figure 8.30: *Monitoring alert rule details*

Click on **alert**

Once your alerts are created, you will view those on the alerts page of your function app. You can also edit the alert rule by clicking on the **Alert** option.

Performance monitoring of your function app

Application Insights also helps monitor the performance of your functions by going to the performance page by clicking on the option next to failures, as shown in *figure*

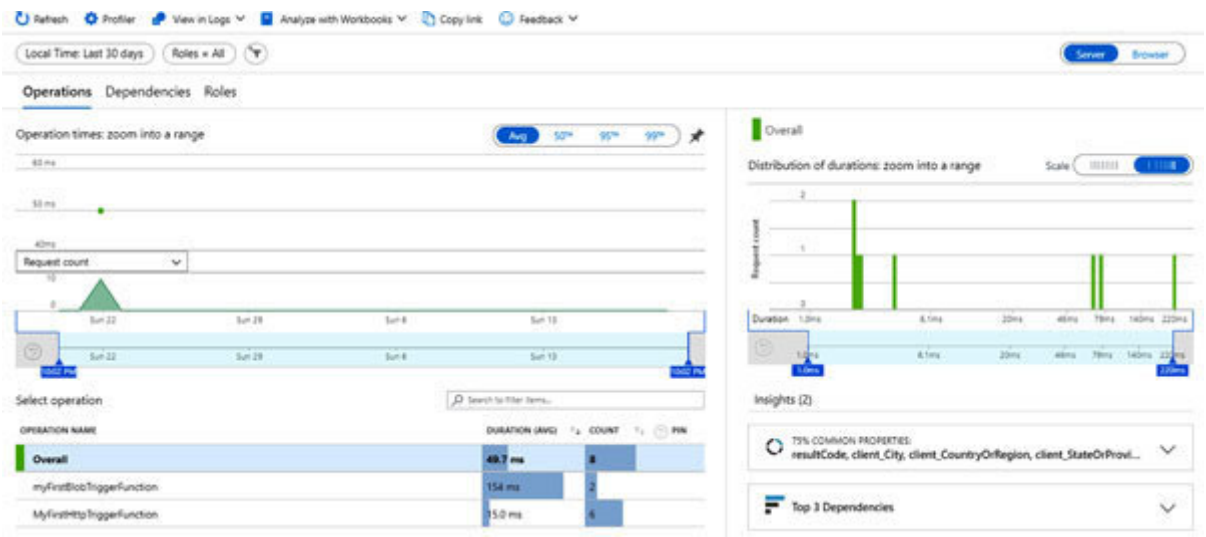


Figure 8.31: *Application Insights performance page*

Here you can view and analyze the operations or dependencies taking more than the usual or expected time.

To open the Performance page, click on the option under the section in the left menu of our Application Insights.

AppDynamics

AppDynamics is a business observability platform that allows you to quickly identify performance bottlenecks and optimize the performance of applications running on Azure. AppDynamics provides the following features:

Full-stack application performance Provides insights into applications written in .NET or .NET Core, deployed as serverless functions, or using other Azure services.

Automatic discovery of applications and their Automatically detects the Azure component used to deploy the application such as Web Apps, WebJobs, and Azure Functions and provides complete observability into the dependencies databases, queues, and so on.

Performance baselining for migration to Automatically baselines application performance before and after the migration to Azure. Provided visualizations of real-time interactions of the dependencies.

Leverage serverless APM for Azure Deploys the latest .NET agent to monitor Azure Functions using AppDynamics Azure Site Extension for .NET and removes dependencies to change the application code for instrumentation.

Prometheus for Azure

Prometheus is one of the popular metric monitoring solutions, a part of Cloud Native Compute Foundation. It's open source and can integrate with Azure Monitor for containers to bring together the best of the two platforms. Getting into the details of Prometheus is out of scope for this book but can be read in detail in Microsoft's documentation - https://docs.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-prometheus-integration and Prometheus official documentation -

Application maps

Another cool feature provided by Application Insights is the application maps. You can view the visualization of requests to the function and its integration metrics to other Azure services, as shown in *figure*

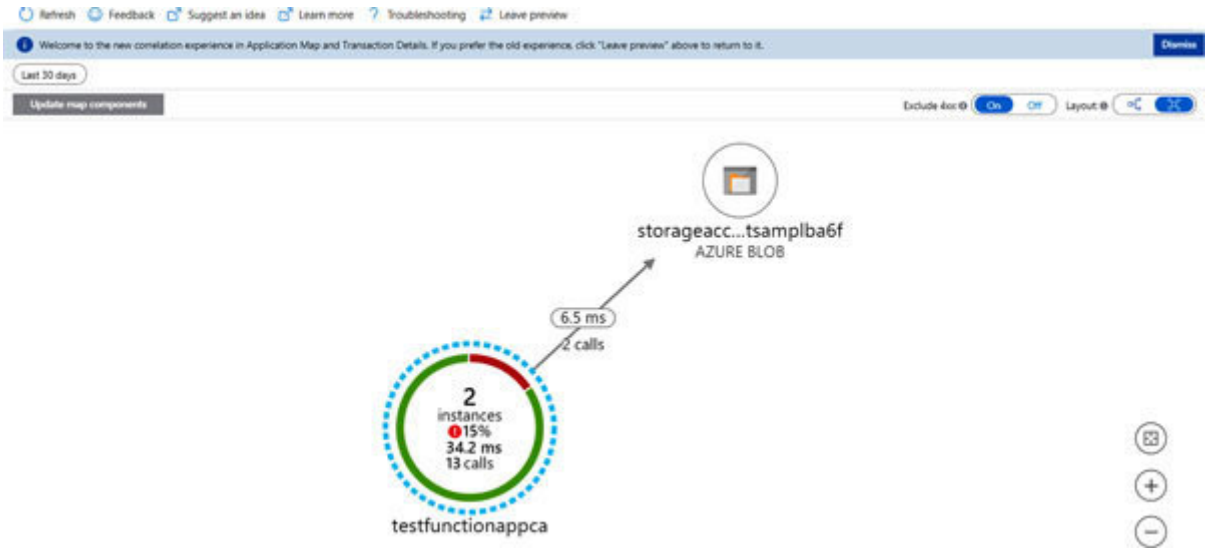


Figure 8.32: *Application Insights application maps*

You can also view the number of instances of your function app, the average time taken for its completion, the number of calls made to their services and their average time, and so on.

To open the application map, click on the option under the section in the left menu of our Application Insights.

Integrating Application Insights with Power BI

The application logs stored in Azure Application Insights provide great help in troubleshooting the issues in applications. However, this data is not limited to troubleshooting but can be used in all sorts of ways to analyze function use and performance. This can provide valuable intelligence for the functions. You can fetch your data feed from Application Insights to Power BI with the help of a rich set of rest APIs available with Application Insights.

Note: Power BI is a business intelligence tool that helps you analyze the data and create interactive dashboards to share insights. You can create these dashboards both on Power BI online or using the Power BI desktop application, which can be downloaded free from

API capabilities

Azure provides several APIs that can be used for a variety of insights using data from Application Insights. These APIs are for three main categories:

REST API allowing to retrieve metric data such as the number of exceptions each hour. You need to specify the metric name, a timespan, aggregation type such as min, max, average, and so on, and the field over which data to be segmented.

REST API allowing access to the event data using OData syntax.

REST API allowing executing the same queries as with Application Insights.

Creating the API Key in the Azure Portal

To connect your PowerBI desktop application to your Application Insights, you need the application ID and API key. Follow these steps to create a key in the Azure portal:

Open your Application Insights in the Azure Portal and click on the option under the **Configure** section of the left menu of your Application Insights resource, as shown in *figure*

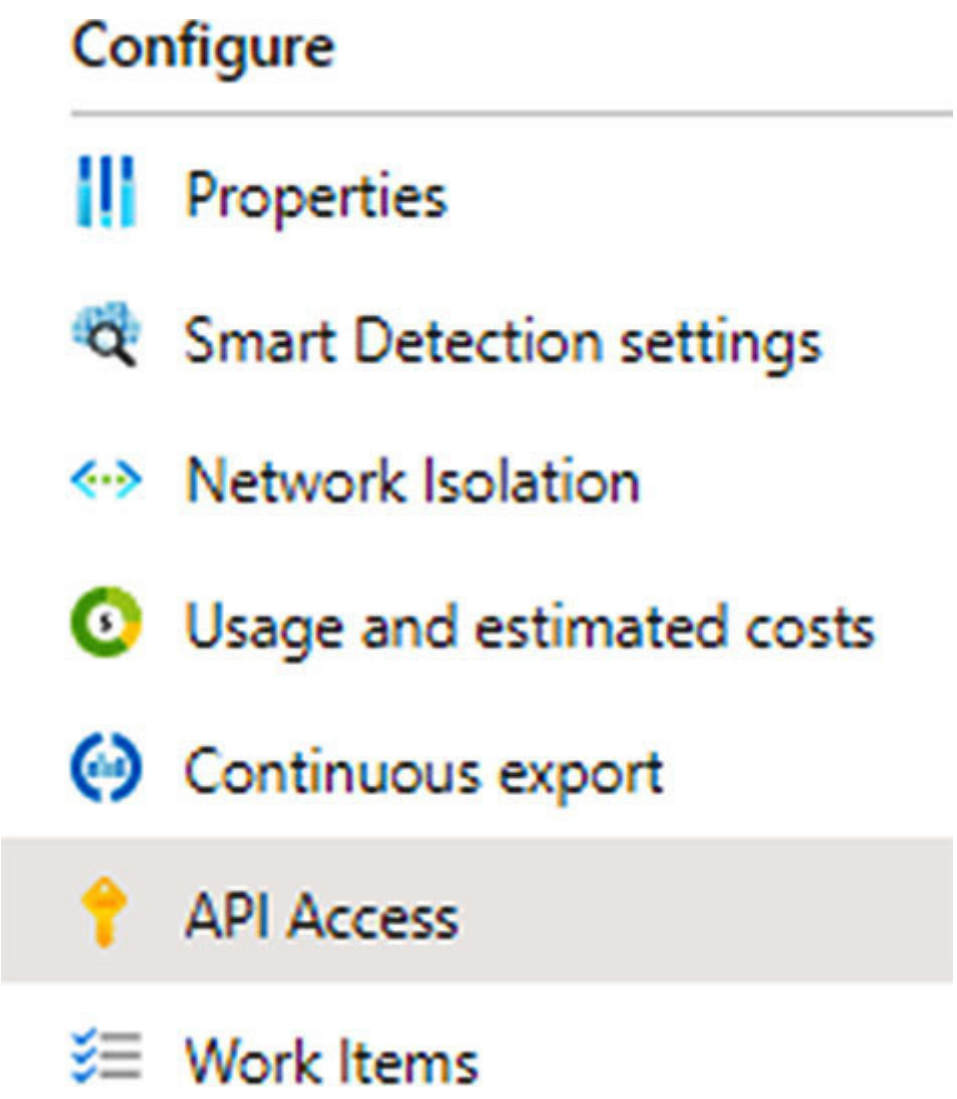


Figure 8.33: *Application Insights API access option*

This will open the API access page as follows, where you will find the **Application** as shown in *figure* Take note of the **Application** We would use the same while connecting through Power BI.

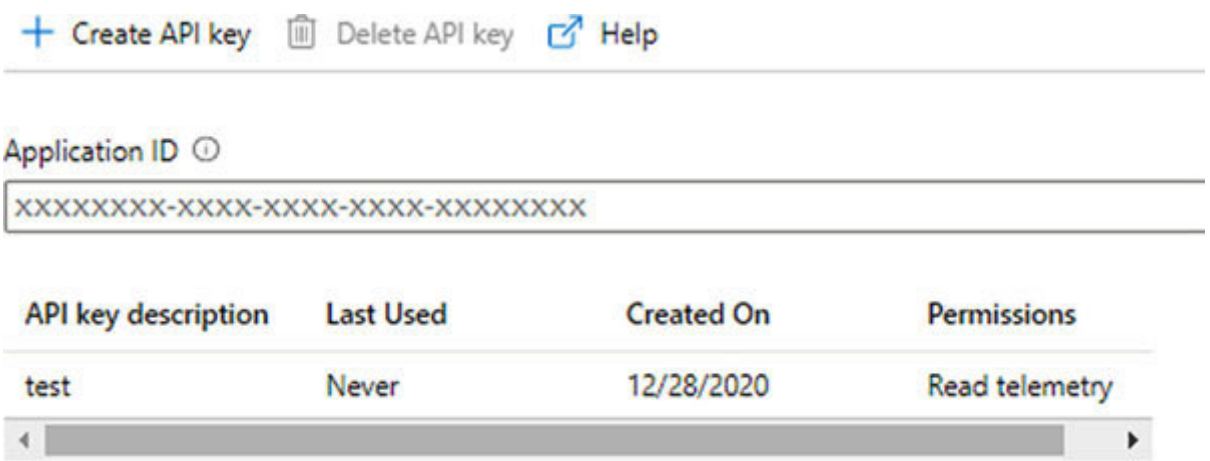


Figure 8.34: *Application Insights API access page*

Click on the **API** option and provide a key name and tick the **Read telemetry** check box, as shown in *figure* This will provide read access of the data to the application using a particular key: Power Bi in our case.

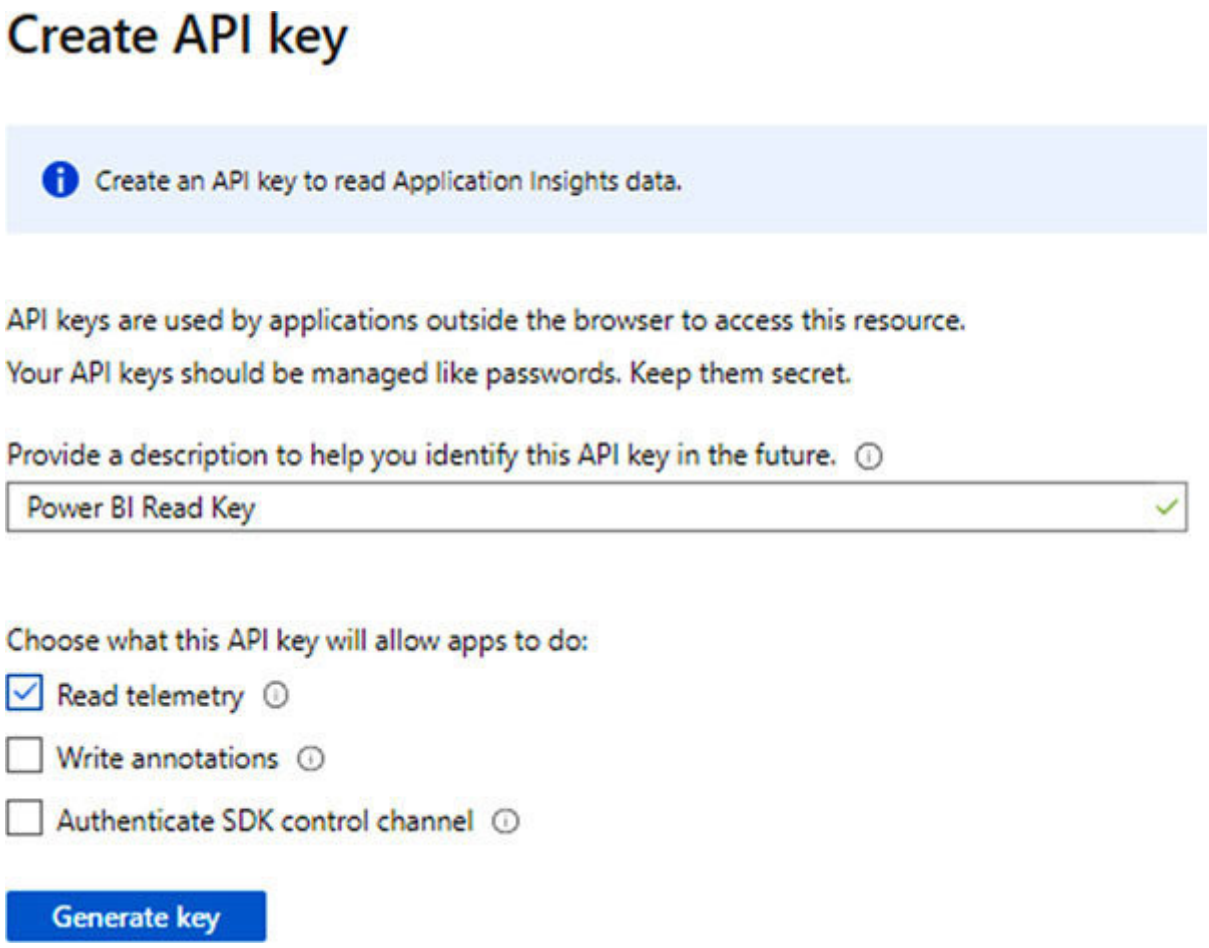


Figure 8.35: *Application Insights create API key*

Click on

This will generate the key for you, as shown in *figure* Please make sure to copy the key, and save it somewhere securely as you only can fetch the same key again. Azure also shows the message as follows to remind you the same:

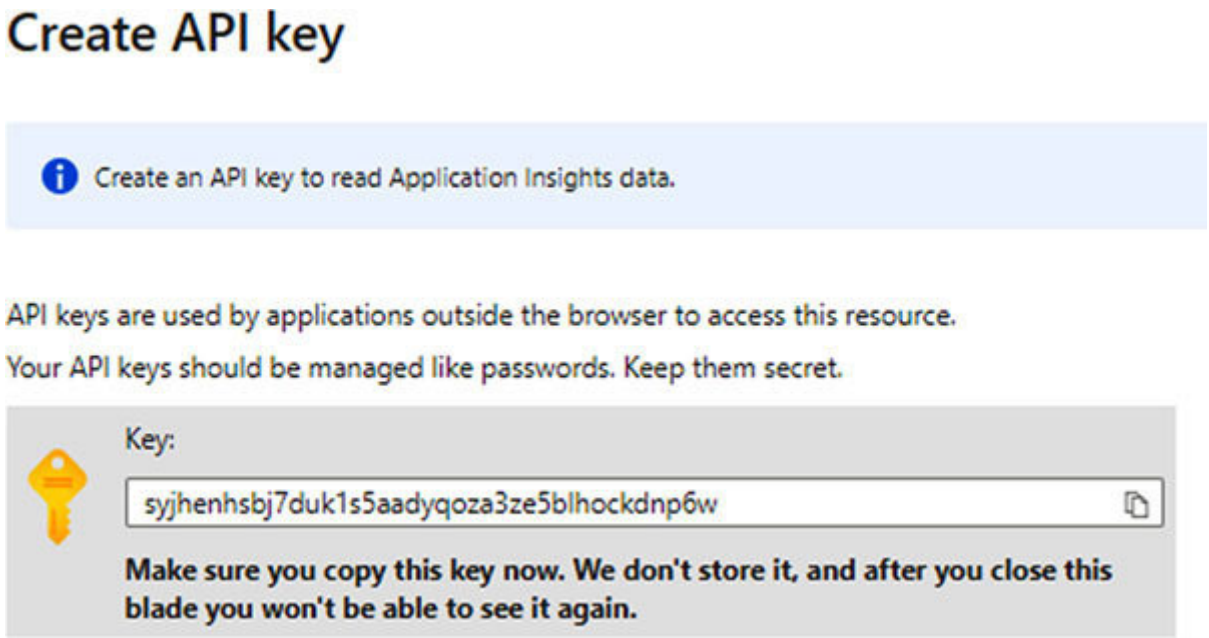


Figure 8.36: *Application Insights API key generated*

Creating the API key using PowerShell and ARM

You can also create the API key using the following PowerShell commands:

```
Connect-ARM

Execute-ARMQuery -SubscriptionId {subscription-id} -HTTPVerb POST -Base "/subscriptions/{subscription-id}/resourcegroups/{resource-group-name}/providers/Microsoft.insights/components/{ai-component-name}/apikeys" -APIVersion "2015-05-01" -Data @{name="Your key name"; linkedReadProperties=@("/subscriptions/{subscription-id}/resourcegroups/{resource-group-name}/providers/Microsoft.insights/components/{ai-component-name}/api")}
```

Connecting to Power BI

Once the key is generated, follow these steps to connect your PowerBI desktop application to your Application Insights:

Open Power BI Desktop and click on under the **Home** section in the ribbon bar, as shown in *figure*

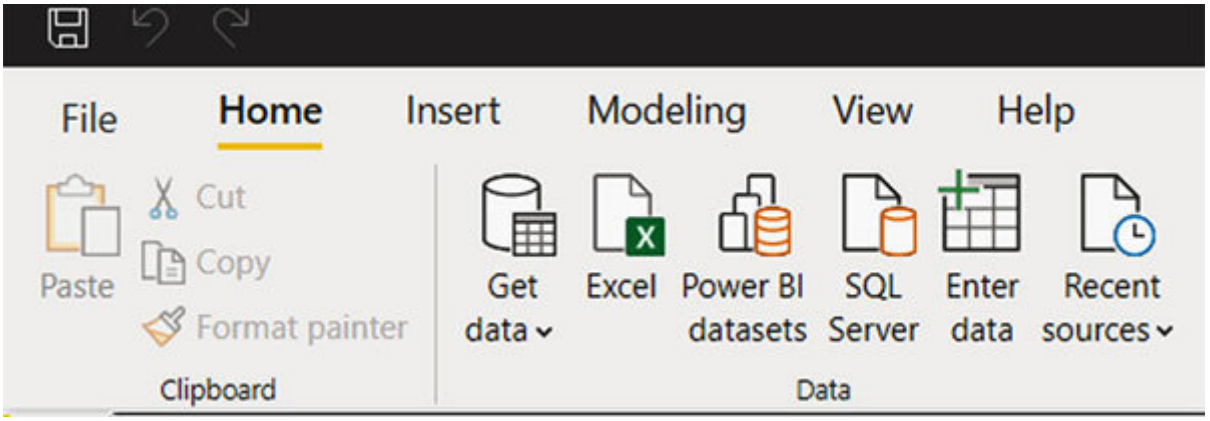


Figure 8.37: *Power BI desktop ribbon bar*

Click on in the drop menu, as shown in *figure*

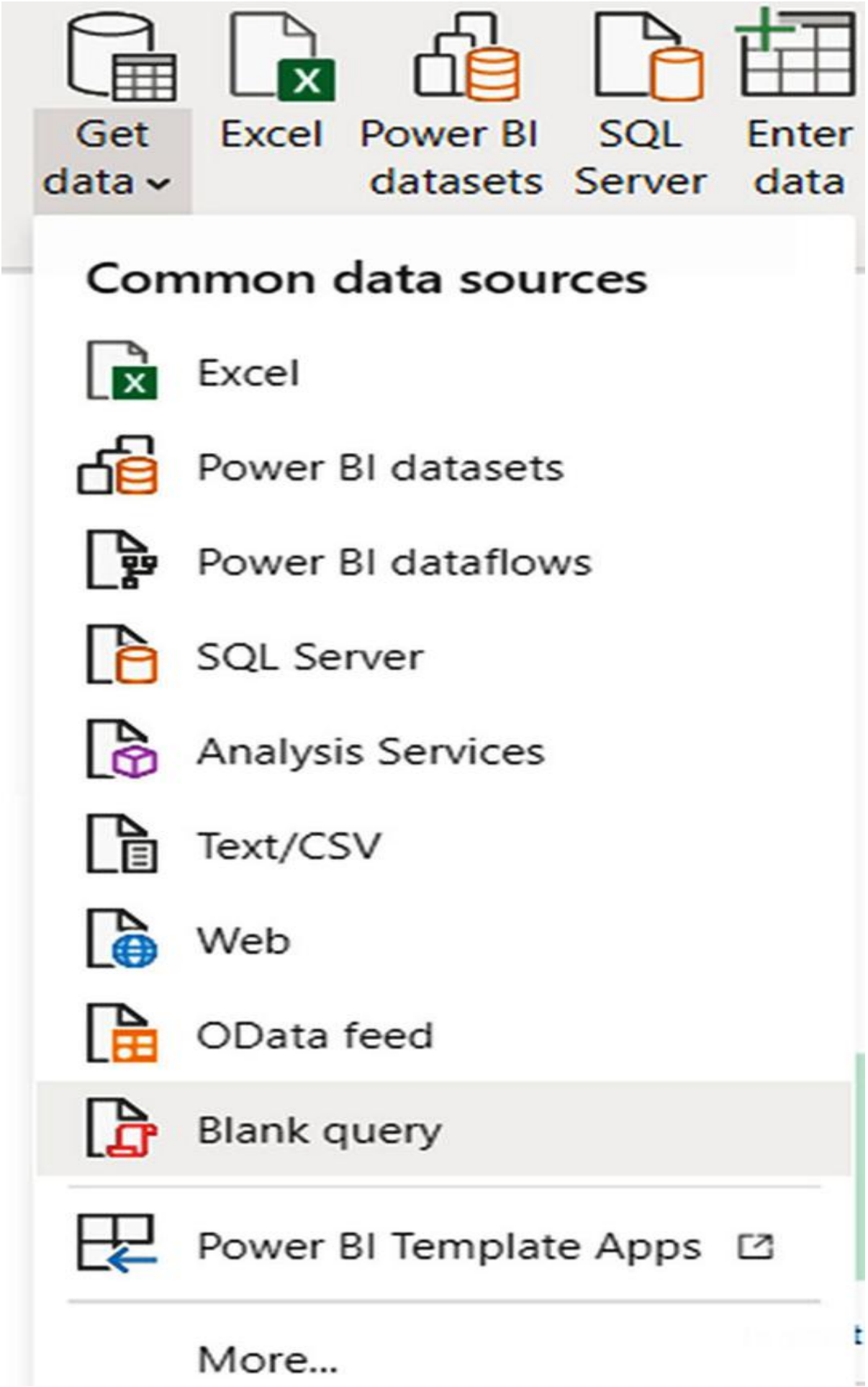

Get data
Excel
Power BI datasets
SQL Server
Enter data
Common data sources
Excel
Power BI datasets
Power BI dataflows
SQL Server
Analysis Services
Text/CSV
Web
OData feed
Blank query
Power BI Template Apps
More...

Figure 8.38: *Power BI Desktop get data drop-down menu*

This will open the **Query Editor** window. Click on the option under the **Home** section of your editor window, as shown in *figure*

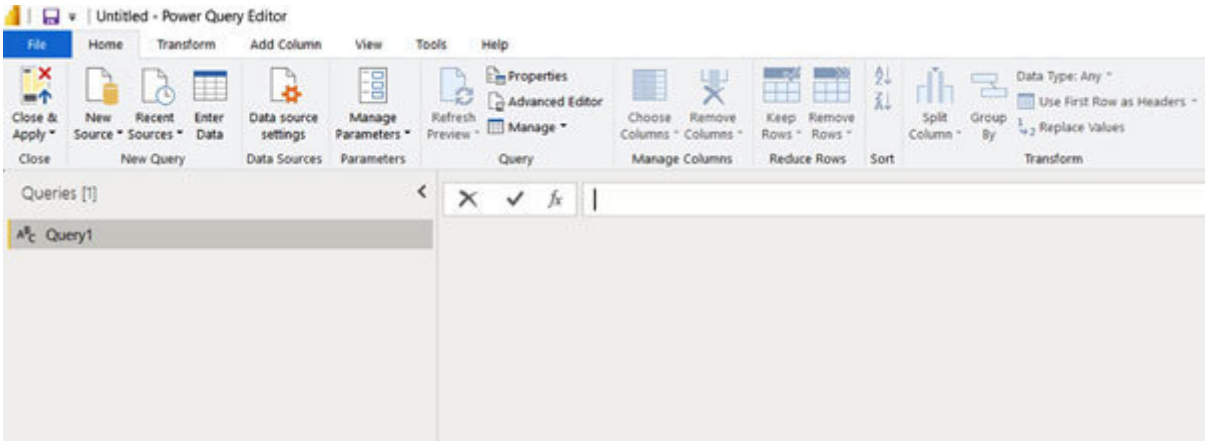


Figure 8.39: *Power BI desktop query editor*

The **Advanced Editor** dialogue (shown in figure defines your Application Insights connection and query. An example for the same can be found as follows. You would need to replace the **ID>** (noted in step 2) with your application ID in the connection URL.

Figure 8.40: *Power BI desktop advanced editor*

Let's take a look at the following code:

```
let

Source =
Json.Document(Web.Contents("https://api.applicationinsights.io/v1/apps/ID>/query?query=traces",
[Query=[#"csl"="requests",#"x-ms-app"="AAPBI"],Timeout=#duration(0,0,4,0)])),
TypeMap = #table(
{"AnalyticsTypes", "Type"},
{
{"Double", Double.Type},
{"Int64", Int64.Type},
{"Int32", Int32.Type},
```

```
{"Int16", Int16.Type},
{"UInt64", Number.Type},
{"UInt32", Number.Type},
{"UInt16", Number.Type},
{"Byte", Byte.Type},
{"Single", Single.Type},
{"Decimal", Decimal.Type},
{"TimeSpan", Duration.Type},
{"DateTime", DateTimeZone.Type},
{"String", Text.Type},
{"Boolean", Logical.Type},
{"SByte", Logical.Type}
}),
DataTable = Source[tables]{0},
Columns = Table.FromRecords(DataTable[columns]),
ColumnsWithType = Table.Join(Columns, {"type"}, TypeMap,
{"AnalyticsTypes"}),

Rows = Table.FromRows(DataTable[rows], Columns[name]),
Table = Table.TransformColumnTypes(Rows,
Table.ToList(ColumnsWithType, (c) => {c{0}, c{3}}))
in
Table
```

Click on

If you are connecting to the particular Application Insights for the first time, it will show the following warning:

Figure 8.41: *Power BI Advance Editor Credential warning*

Click on it will open the Access Dialogue box, as shown in *figure*

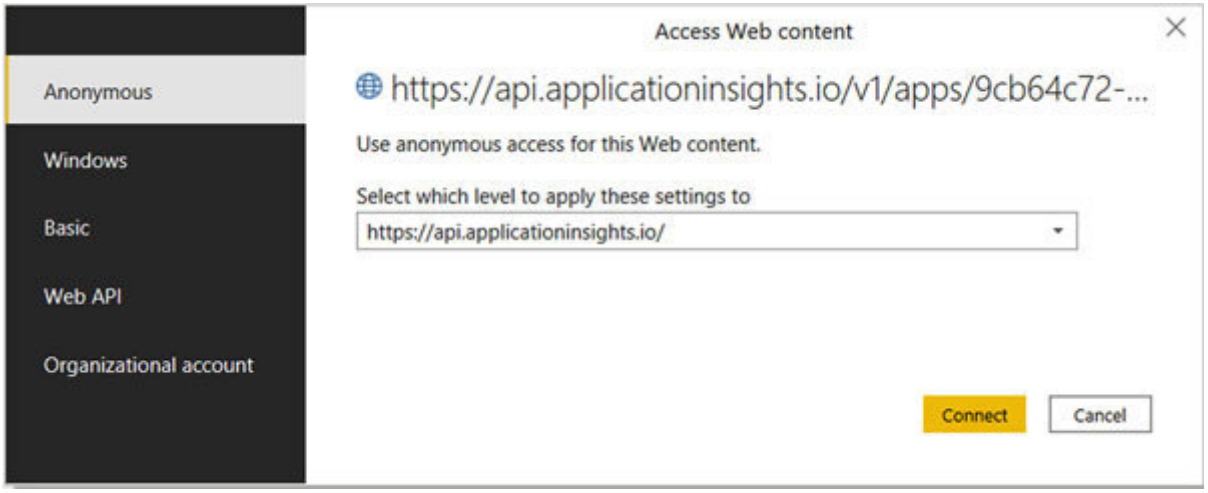


Figure 8.42: *Power BI desktop access dialogue box*

Here you can choose from multiple options such as windows authentication, basic authentication, web API, or your organizational account. For development/testing purposes, it is okay to use an organizational account for authentication. However, this is not recommended as a solution for production reports because if a user accounts password is changed, an account is disabled, or the permissions are revoked. You will need to update the credentials in Power BI manually for each affected report. Therefore, the recommended option is to use the API key. Click on the **Basic** option, as shown in *figure* for the same:

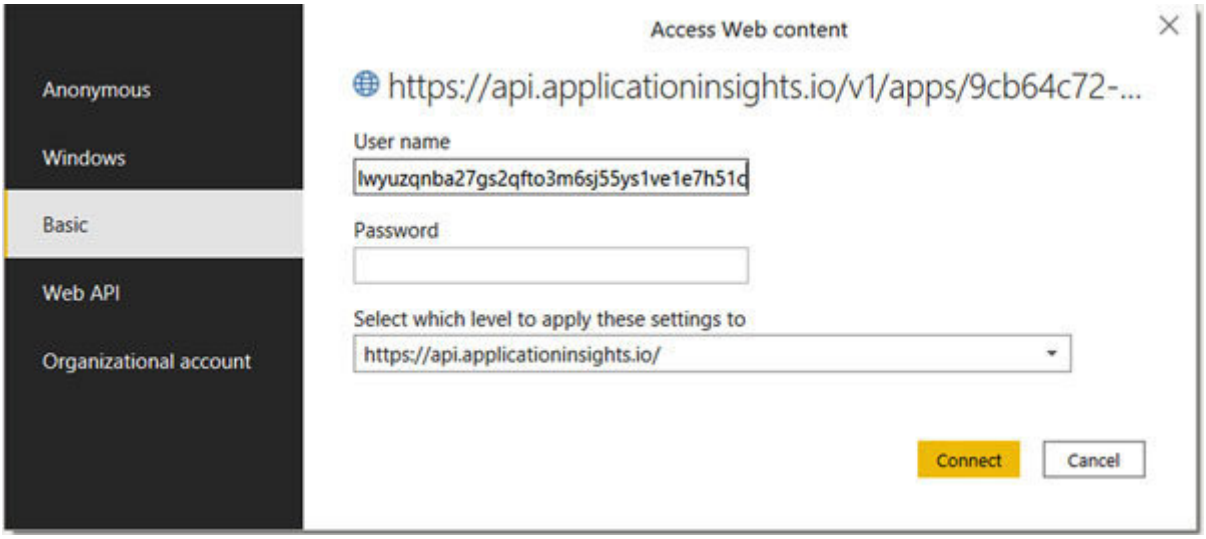


Figure 8.43: *Power BI Desktop access dialogue box-basic authentication*

Paste your API Key in the **User Name** box and click on

Alternatively, you can also modify the query as follows:

```
let
Source =
Json.Document(Web.Contents("https://api.applicationinsights.io/v1/apps/ID>/query?query=traces",
[Query=[#"csl"="requests",#"x-ms-app"="AAPBI"],Timeout=#duration(0,0,4,0),
Headers=[#"x-api-key"="Key>", #"Prefer"="response-v1=true"]])),
```

Replace the Application ID and API Key with the actual values and choose the anonymous option while connecting.

Once connected, you will be able to view data from Application Insights, as in *figure*

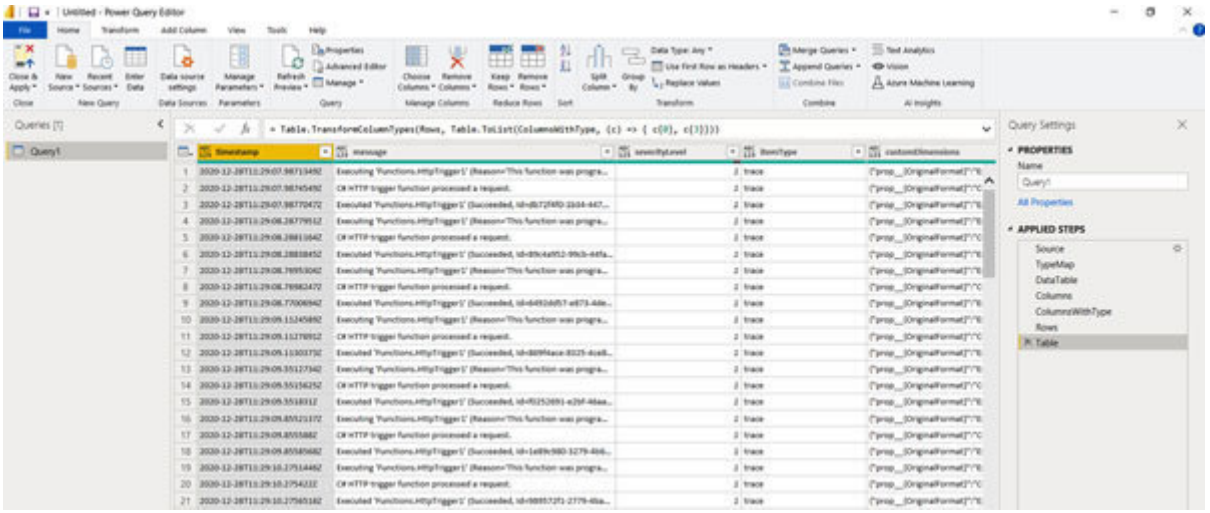


Figure 8.44: *Power BI desktop query result preview*

You may rename the query by right-clicking on the query name in the left menu.

Click on **&**

This will bring you back to the Power BI desktop visualization. You can now create insightful dashboards using the fields from the query created. *Figure 8.45* shows the fields in the right menu:

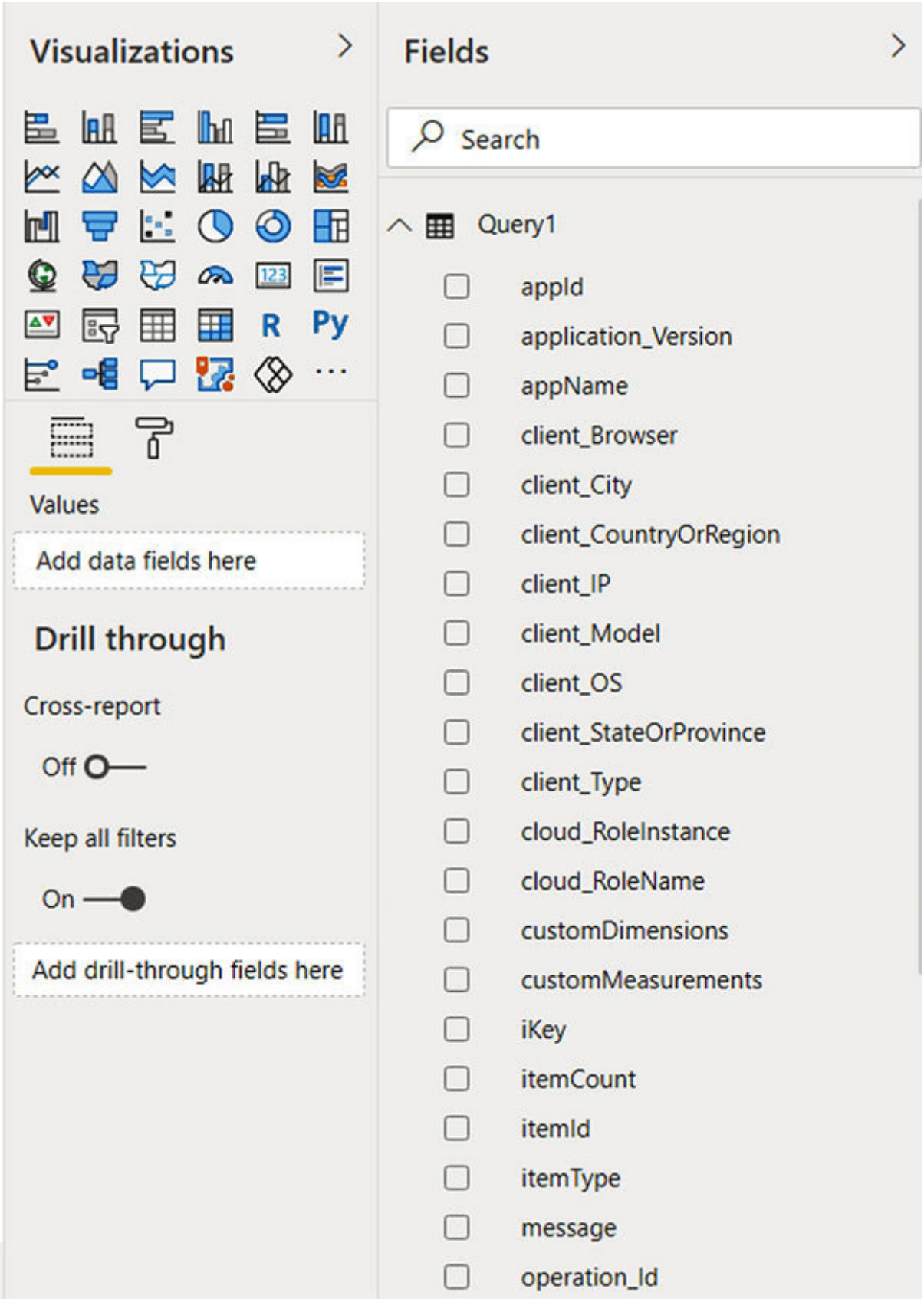


Figure 8.45: *Power BI desktop visualization fields menu*

Note: You can use API Explorer provided by Azure to create and test your API queries. The same can be accessed through the URL,

Conclusion

Azure provides various capabilities and offerings to enable monitoring of the Azure functions. You can integrate your function to Application Insights and monitor your Azure functions' exceptions, availability, and performance. You can also create troubleshooting guides for easy troubleshooting. The Kusto Query language helps you to fetch data from Application Insights and make informed decisions. You can take advantage of the advanced level diagnostics available for Azure functions and configure the monitoring alerts so that you don't miss any failure or issue that requires attention. You can also connect the Azure Application Insights to Power BI for rich and insightful analytics. Overall, with the help of these options, it's easy to monitor and troubleshoot the functions on production.

Points to remember

You can integrate your Azure functions to the Application Insights with the help of built-in integration

You can monitor exceptions, availability, and performance of your functions on Application Insights

You can leverage the diagnose and solve problems feature of Azure functions for advanced-level diagnostics

Azure functions allow you to monitor logs in real-time with the help of live streaming

You can use the Kusto Query language to fetch and analyze logs for your function in Application Insights

You can create monitoring alerts by defining the condition and action group to track any event or failure

You can use AppDynamics for performance monitoring out of the box for your Functions

You can view the visualization of requests to the function and its integration metrics to other Azure services with the help of application maps in Application Insights

You can integrate Application Insights to Power BI for rich and interactive insights into your function performance

A

access restrictions
enabling
Active Directory (AD) 11
AddMessage function
implementing
alert
creating, for Azure Functions
alert rule, key attributes
action 219
alert condition 218
alert signal 218
severity 218
target resource 218
API, categories
events 227
metrics 227
query 227
API management
using, in network security 180
using, in network security
AppDynamics
about 225
features 225
Application Insights
aggregator, configuring 206

enabling 205
log level, configuring 206

used, for integrating Azure Functions 204
Application Insights, with Power BI
API capabilities 227
API key, creating in Azure Portal 228
API key, creating with ARM 229
API key, creating with PowerShell 229
connecting
integrating 227
application maps 226
App Service Editor 37
App Service plans
basic service plan 18
free and shared service plans 17
isolated service plan 18
premium service plan 18
standard service plan 18
ARM template
reference link 74
used, for creating function
Azure
function application, publishing
Azure account
creating 12
Azure active directory authentication
about
advanced settings, configuring
Azure activity log 218
Azure AD
features 11

Azure availability zones 16
Azure blobs
about 9
append blob 9
block blob 9
page blob 9
Azure Blob storage
bindings 93
triggers 93
Azure CLI
about 71
Azure storage account, creating 73
function app, deploying to Azure 74
resource group, creating 72
serverless function app, creating 73
used, for creating function 72
Azure Cloud
about 5
Infrastructure as a service (IaaS) 6
Platform as a Service (PaaS) 6
Software as a Service (SaaS) 6
Azure compute 8
Azure compute, options
App service 8
Azure Functions 8
virtual machines 8
Azure cost analysis
about 19
costs, reviewing 19

view forecasted cost, for service 20

Azure cost analysis, options
accumulated granularity 20
actual cost 20
budget 20
charts 20
cost view 19
forecast 20
Azure databases
about 9
NoSQL database 10
open-source database 10
relational database 10
Azure DevOps
account, setting up 186
build pipeline, creating
project, creating 187
release pipeline, creating
using, in continuous deployment (CS) 185
using, in continuous integration (CI) 185
Azure files 9
Azure function app
configuring 97
development tools 37
folder structure 35
function code 33
function settings 34
runtime versions 36
supported languages 36

working with 33
Azure Functions
alerts, monitoring 218

application maps 226
Azure activity log 218
benefits 28
building, with GitHub actions
cold start process 41
cold start process, preventing 42
CORS, enabling 171
deploying, with GitHub actions
deployment slots 201
deployment slots, creating 201
diagnostics
exceptions, viewing in Application Insights
function app performance, monitoring 225
integrating, with Application Insights 204
integrating, with Azure Monitor Logs
integrating, with Azure virtual network 165
live streaming 215
monitoring 214
overview 40
pre-warmed instances 41
problems, diagnosing 212
problems, solving 212
swap slots 202
troubleshooting 208
troubleshooting guide 211
used, for creating alert

using, to convert monolithic architecture into microservices 88
versus Azure WebJobs 30
Azure Functions Blob
input bindings 93
output bindings 93

Azure Functions, hosting plans
about 28
app service plan 29
consumption plan 29
premium plan 29
Azure Functions Integrations
in Azure portal 90
Azure Functions project
creating
Azure Functions source code
pushing, in Azure Repos 188
Azure networking 10
Azure portal
about 12
Azure Functions Integrations 90
function app, creating
function, creating 53
function, creating in function app
function, running 53
function, testing 52
reference link 44
used, for configuring SendGrid
Azure pricing 17

Azure pricing models, for virtual machine
pay-as-you-go 17
reserved instances 17
spot pricing 17
Azure pricing plans, for functions
app service plan 18
consumption plan 18
premium plan 18

Azure products
overview 7
Azure queues 9
Azure regions 12
Azure resource group
Azure Resource Manager (ARM) 16
Azure serverless compute, services
Event Grid 8
functions 8
Logic Apps 8
Azure services
Azure compute 8
Azure databases 9
Azure networking 10
Azure storage 8
identity and security 11
overview 7
Azure storage 8
Azure Storage Explorer tool
reference link 99
Azure Storage provider

for Durable Functions 117
history table 118
instance table 119
Azure storage types
Azure blobs 9
Azure files 9
Azure queues 9
Azure tables 9
Azure virtual network
about 10

function app, connecting 169
function app, creating in premium plan 166
function proxy, creating to access resources 170
virtual machine, creating without public IP address 167
Azure WebJobs
versus Azure Functions 30

B

bindings
about 85
overview 84

C

cloud computing
key benefits 7
cloud service platforms
evolution 2
Infrastructure as a service (IaaS) 3
on-premises 2
Platform as a Service (PaaS) 4

serverless 4
Command and Query Responsibility Segregation (CQRS) 119
console 39
continuous deployment (CD)
about 185
with Azure DevOps 185
continuous integration (CI)
about 185

with Azure DevOps 185
Cross-Origin Resource Sharing (CORS)
about 170
enabling, for Azure Functions 171
custom HTTP endpoint
configuring, for function 92

D

deployment methods 184
deployment technologies 184
development tools 37
development tools, Azure functions app
App Service Editor 37
console 39
Kudu 39
domain-validated certificate (DV) 159
Durable Functions
Azure Storage provider 117
creating, with Azure portal
email attachment, sending with Blob triggers 140
email attachment, sending with SendGrid triggers 140
exceptions handling 144

locally running 141
locally testing 141
overview 112
publishing, from Visual Studio
records, obtaining from SQL table to export 138
records.txt file, exporting to Azure blob 139
used, for exporting SQL table records file to blob

working with 127
Durable Functions, components
activity function 127
orchestrator client 127
orchestrator function 127
Durable Functions patterns
about 120
aggregator pattern 126
async long-running APIs pattern 123
fanning in pattern 121
fanning out pattern 121
function chaining pattern 120
human interaction pattern 125
monitor pattern 124
Durable Orchestrations
race condition, handling 116
sub-orchestration 117
working 116

E

email notification
sending, with SendGrid 93
event-driven programming 27

F

function
creating, with ARM template
creating, with Azure CLI 72

creating, with Visual Studio Code (VS Code)
function access keys
about 148
existing key, renewing 151
function keys, accessing 149
host keys, accessing 149
host keys, creating 150
master keys, accessing 149
system keys, accessing 149
function access keys, types
function keys 148
host keys 148
master key 148
system key 148
function application
creating, with Visual Studio 2019
publishing, to Azure
Function as a Service (FaaS) 26
functions
about 26
building 27
deploying 27
monitoring 28
operating 28
planning 26

testing 27

H

HTTP bindings 91

HTTP triggers 91

I

Infrastructure as a Service (IaaS) 5
Infrastructure-as-Code (IAC) 16
instance table, Azure Storage provider
control queue 119
work-item queue 119

K

Kudu 38

M

managed identities, for functions
system-assigned identity, enabling 163
user-assigned identity, creating
using 162
microservices
Azure Functions, using to convert monolithic architecture 88
benefits 89
disadvantages 89
monolithic architecture
converting, with Azure Functions into microservices 88

N

network security

about 165

API management, using
Azure Functions, integrating with Azure virtual network 165
NoSQL datastore 9

O

orchestrator functions
about 112
activity function 113
client function 113
entity function 114

P

Platform as a Service (PaaS) 173
private endpoint connections 174
private key certificates (.pfx), options
app service managed certificate, creating 160
buy certificates 158
import app service certificate 159
upload certificate 160
Prometheus
about 226
for Azure 226
protocol settings
configuring 158

Q

Query telemetry data
with Kusto Query language 207

R

replay state 114

resource group
deleting

S

SendEmail function
implementing
SendGrid
about 93
configuring, with Azure portal
used, for sending email notification 93
serverless 4
serverless computing
about 8
benefits 22
overview 21
Software as a Service (SaaS) 5
source control integration
for deployments 185
SQL table records file
exporting, to blob with Durable Functions
system-assigned identity 162

T

Timer bindings
Timer triggers
TLS/SSL bindings
configuring 158
TLS/SSL certificate
auto-renewal settings, configuring 161
for function app 157

private key certificates (.pfx), managing 158
public key certificates (.cer), uploading 161
triggers
about 30
overview 84
types 31
use case 86
triggers, types
Blob trigger 84
Cosmos DB trigger 84
Dapr trigger 85
Event Grid trigger 84
Event Hub trigger 84
HTTP trigger 84
kafka trigger 84
Queue trigger 84
RabbitMQ trigger 85
Service Bus trigger 84
Timer trigger 84
trigger syncing 184

U

user-assigned identity 162

V

virtual network
connection options 10
Visual Studio
Durable Functions, publishing from
Visual Studio 2019

download link 64
used, for creating function application
Visual Studio Code (VS Code)
about 54
used, for creating function