
Hands-on
Kubernetes on Azure
Third Edition
Use Azure Kubernetes Service to automate management, scaling, and deployment of containerized applications
Nills Franssens, Shivakumar Gopalakrishnan, and Gunther Lenz

Hands-on Kubernetes on Azure, Third Edition

Use Azure Kubernetes Service to automate management, scaling, and deployment of containerized applications.

Nills Franssens

Shivakumar Gopalakrishnan

Gunther Lenz

Packt

BIRMINGHAM–MUMBAI

Hands-on Kubernetes on Azure, Third Edition

Authors: Nills Franssens, Shivakumar Gopalakrishnan, and Gunther Lenz

Technical Reviewers: Richard Hooper and Swaminathan Vetri

Managing Editor: Aditya Datar and Siddhant Jain

Acquisitions Editor: Ben Renow-Clarke

Production Editor: Deepak Chavan

Editorial Board: Vishal Bodwani, Ben Renow-Clarke, Arijit Sarkar, and Lucy Wan

First Published: March 2019

Second Published: May 2020

Third Published: April 2021

Production Reference: 3230421

ISBN: 978-1-80107-994-5

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham, B3 2PB, UK.

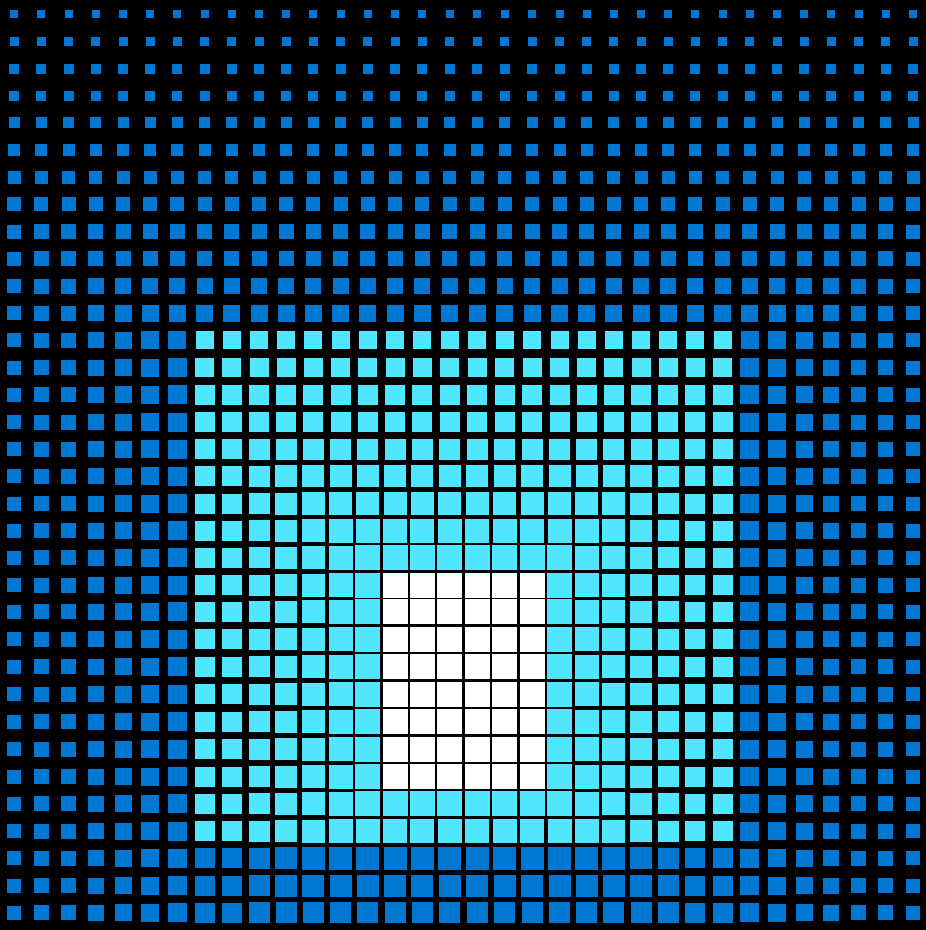
To mama and papa. This book would not have been possible without everything you did for me. I love you both.

To Kelly. I wouldn't be the person I am today without you.

- Nills Franssens

Microsoft Azure

kubernetes



Get started

Kubernetes on Azure

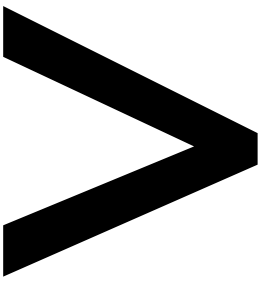
Find out what you can do with a fully managed service for simplifying Kubernetes deployment, management and operations, including:

- Build microservices applications.
- Deploy a Kubernetes cluster.
- Easily monitor and manage Kubernetes.

Create a free account and get started with Kubernetes on Azure. Azure Kubernetes Service (AKS) is one of more than 25 products that are always free with your account. **Start free >**

Then, try these labs to master the basic and advanced tasks required to deploy a multi-container application to Kubernetes on Azure Kubernetes Service (AKS). **Try now >**

Table of Contents



Preface

About

This section briefly introduces the authors and reviewers, the coverage of this book, the technical skills you'll need to get started, and the hardware and software needed to complete all of the topics.

Hands-on Kubernetes on Azure – Third Edition

Containers and Kubernetes containers facilitate cloud deployments and application development by enabling efficient versioning with improved security and portability.

With updated chapters on role-based access control, pod identity, storing secrets, and network security in AKS, this third edition begins by introducing you to containers, Kubernetes, and **Azure Kubernetes Service** (**AKS**), and guides you through deploying an AKS cluster in different ways. You will then delve into the specifics of Kubernetes by deploying a sample guestbook application on AKS and installing complex Kubernetes apps using Helm. With the help of real-world examples, you'll also get to grips with scaling your applications and clusters.

As you advance, you'll learn how to overcome common challenges in AKS and secure your applications with HTTPS. You will also learn how to secure your clusters and applications in a dedicated section on security. In the final section, you'll learn about advanced integrations, which give you the ability to create Azure databases and run serverless functions on AKS as well as the ability to integrate AKS with a continuous integration and continuous delivery pipeline using GitHub Actions.

By the end of this Kubernetes book, you will be proficient in deploying containerized workloads on Microsoft Azure with minimal management overhead.

About the authors

Nills Franssens is a technology enthusiast and a specialist in multiple open-source technologies. He has been working with public cloud technologies since 2013.

In his current position as a Principal Cloud Solutions Architect at Microsoft, he works with Microsoft's strategic customers on their cloud adoption. He has worked with multiple customers in migrating applications to run on Kubernetes on Azure. Nills' areas of expertise are Kubernetes, networking, and storage in Azure.

When he's not working, you can find Nills playing board games with his wife Kelly and friends, or running one of the many trails in San Jose, California.

Shivakumar Gopalakrishnan is DevOps architect at Varian Medical Systems. He has introduced Docker, Kubernetes, and other cloud-native tools to Varian product development to enable "Everything as Code".

He has years of software development experience in a wide variety of fields, including networking, storage, medical imaging, and currently, DevOps. He has worked to develop scalable storage appliances specifically tuned for medical imaging needs and has helped architect cloud-native solutions for delivering modular AngularJS applications backed by microservices. He has spoken at multiple events on incorporating AI and machine learning in DevOps to enable a culture of learning in large enterprises.

He has helped teams in highly regulated large medical enterprises adopt modern agile/DevOps methodologies, including the "You build it, you run it" model. He has defined and leads the implementation of a DevOps roadmap that transforms traditional teams to teams that seamlessly adopt security- and quality-first approaches using CI/CD tools. He holds a bachelor of engineering degree from College of Engineering, Guindy, and a master of science degree from University of Maryland, College Park.

Gunther Lenz is senior director of the technology office at Varian. He is an innovative software R&D leader, architect, MBA, published author, public speaker, and strategic technology visionary with more than 20 years of experience.

He has a proven track record of successfully leading large, innovative, and transformational software development and DevOps teams of more than 50 people, with a focus on continuous improvement. He has defined and lead distributed teams throughout the entire software product lifecycle by leveraging ground-breaking processes, tools, and technologies such as the cloud, DevOps, lean/agile, microservices architecture, digital transformation, software platforms, AI, and distributed machine learning.

He was awarded Microsoft Most Valuable Professional for Software Architecture (2005-2008). Gunther has published two books, .NET – A Complete Development Cycle and Practical Software Factories in .NET.

About the reviewers

Richard Hooper also known as PixelRobots online lives in Newcastle, England, he is a Microsoft MVP for Azure and a Microsoft Certified Trainer (MCT) who works as an Azure architect at a company called Intercept based in the Netherlands. He has more than 15 years of professional experience in the IT industry. He has worked with Microsoft technologies all of his career but also has dabbled with Linux. He is very enthusiastic about Azure and Azure Kubernetes Service (AKS) and has been using them daily. In his spare time, he enjoys sharing knowledge and helping people. He does this by blogging, podcasts, videos, and whatever technology is at hand to share his passion, hoping it will help someone to progress in their Azure journey. Richard has a passion for blogging and learning, which leads him to discover new things every week. When the opportunity arose to be a technical reviewer for a book about AKS, he jumped at the chance! Find him on Twitter at @pixel_robots.

Swaminathan Vetri (Swami) works as an Architect at Maersk Technology Center Bangalore building cloud native applications on Azure using various Azure PaaS offerings and Kubernetes. He has also been recognised as a Microsoft MVP - Developer Technologies since 2016 for his technical contributions to the developer community. In addition to writing technical blogs, he can often be seen speaking at local developer conferences, user group meets, meetups etc., on various topics ranging from .NET, C#, Docker, Kubernetes, Azure DevOps, GitHub Actions to name a few. A continuous learner who is passionate about sharing his little knowledge to the community. You can follow him on Twitter and GitHub at @svswaminathan.

Learning objectives

- Plan, configure, and run containerized applications in production.
- Use Docker to build applications in containers and deploy them on Kubernetes.
- Monitor the AKS cluster and the application.
- Monitor your infrastructure and applications in Kubernetes using Azure Monitor.
- Secure your cluster and applications using azure-native security tools.
- Connect an app to the Azure database.
- Store your container images securely with Azure Container Registry.
- Install complex Kubernetes applications using Helm.
- Integrate Kubernetes with multiple Azure PaaS services, such as databases, Azure Security Center, and Functions.
- Use GitHub Actions to perform continuous integration and continuous delivery to your cluster.

Audience

This book is designed to benefit aspiring DevOps professionals, system administrators, developers, and site reliability engineers who are interested in learning how containers and Kubernetes can benefit them. If you're new to working with containers and orchestration, you'll find this book useful.

Approach

The book focuses on a well-balanced combination of practical experience and theoretical knowledge, accompanied by engaging real-world scenarios that have a direct correlation to how professionals work on the Kubernetes platform. Each chapter has been explicitly designed to enable you to apply what you learn in a practical context with maximum impact.

Hardware and software requirements

Hardware requirements

For the optimal lab experience, we recommend the following hardware configuration:

- Processor: Intel Core i5 or equivalent
- Memory: 4GB RAM (8 GB preferred)
- Storage: 35 GB available space

Software requirements

We also recommend that you have the following software configuration in advance:

- A computer with a Linux, Windows 10, or macOS operating system
- An internet connection and web browser so you can connect to Azure

Conventions

Code words in the text, database names, folder names, filenames, and file extensions are shown as follows.

The `front-end-service-internal.yaml` file contains the configuration to create a Kubernetes service using an Azure internal load balancer. The following code is part of that example:

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: frontend
5     annotations:
6       service.beta.kubernetes.io/azure-load-balancer-internal:
"true"
7     labels:
8       app: guestbook
9       tier: frontend
10  spec:
```

```
11    type: LoadBalancer
12    ports:
13    - port: 80
14    selector:
15      app: guestbook
16      tier: frontend
```

Downloading resources

The code bundle for this book is available at https://github.com/PacktPublishing/Hands-on-Kubernetes-on-Azure-Third-Edition.

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

Foreword

Welcome! By picking up this book, you've shown that you are interested in two things: Azure and Kubernetes, which are both near and dear to my heart. I'm excited that you are joining us on our cloud-native journey. Whether you are new to Azure, new to Kubernetes, or new to both, I'm confident that as you explore Azure Kubernetes Service (AKS), you will find new ways to transform your applications, delight your customers, meet the growing needs of your business, or simply learn new skills that will help you achieve your career goals. Regardless of your reasons for starting this journey, we are eager to help you along the way and see what you can build with Kubernetes and Azure.

The journey of Kubernetes on Azure itself has been an exciting one. Over the last few years, AKS has been the fastest-growing service in the history of Azure. We find ourselves at the inflection point of both hyperscale growth in Azure itself, as well as hockey stick growth in applications running on Kubernetes. Combine the two together, and this has made for an exciting (and busy) few years.

It has been thrilling to see the success that we have been able to deliver for our customers and users. But what is it about Azure and Kubernetes that have enabled customer success? Though it may seem like magic at times, the truth is that there is nothing about either Azure or Kubernetes that is truly magic. The value, success, and transformation that our customers are seeing is related to their needs and how this technology helps make these goals achievable.

We've seen over the past decade, and especially in the last year, that the ability to be agile and adapting as the world changes is a critical capability for all of us. Kubernetes enables this agility by introducing concepts such as containers and container images, as well as higher-level concepts such as services and deployments, which naturally push us toward architectures that are decoupled *microservices*. Although, of course, you can build microservice applications without Kubernetes, the natural tendency of the APIs and design patterns is to push you toward these architectures. Microservices are the *gravity well* of Kubernetes, so to speak. However, it's important to note that microservices are not the only way to run applications on Kubernetes. Many of our customers find great benefits in bringing their legacy applications to Kubernetes and mixing the management of existing applications with the development of new cloud-native implementations.

As more and more people have started to conduct more and more of their lives online, the criticality of all of the services we have built has radically changed. It's no longer acceptable to have *maintenance hours* or *scheduled downtime*. We live in a 24x7 world where applications need to available at all times, even as we build, change, and rearrange them. Here, too, Kubernetes and Azure provide the tools that you need to build reliable applications. Kubernetes has health checks that automatically restart your application if it crashes, infrastructure for zero-downtime rollouts, and autoscaling technology that enables you to automatically grow to sustain a customer's load. On top of these capabilities, Azure provides the infrastructure to perform upgrades to Kubernetes itself without affecting applications running in the cluster, and autoscaling of the cluster itself to provide additional capacity to meet the demands of growing applications and the elasticity to right-size your cluster to the most efficient shape possible.

In addition to these core capabilities, using AKS provides access to broader cloud-native ecosystems. There are countless engineers and projects in the **Cloud Native Compute Foundation** (**CNCF**) ecosystem that can help you build your applications more quickly and reliably. As a leader and a contributor to many of these projects, Azure provides integration and supports access to some of the best open-source software that the world has to offer, including Helm, Gatekeeper, Flux, and more.

But the truth is that building any application on Kubernetes involves much more than just the Kubernetes bits. Microsoft has a unique set of tools that integrate with AKS to provide a seamless, end-to-end experience. Starting with GitHub, where the world comes togeher to develop and collaborate, through to Visual Studio Code, where people build the software itself, and to tools such as Azure Monitor and Azure Security Center to keep your applications healthy and secure, it is truly the combined capabilities of Azure that makes AKS a fantastic place for your applications to thrive. When you combine that with Azure's cloud-leading footprint around the world, which delivers more managed Kubernetes deployments in more locations than anyone else, you can see that AKS enables businesses to rapidly scale and grow to meet their needs from the initial startup phase through to the global enterprise level.

Thank you for choosing Azure and Kubernetes! I'm excited that you're here and I hope you enjoy learning about everything Kubernetes and Azure has to offer.

– Brendan Burns
Co-founder of Kubernetes and Corporate Vice President at Microsoft

Section 1: The Basics

In *Section 1* of this book, we will cover the basic concepts that you need to understand in order to follow the examples in this book.

We will start this section by explaining the basics of these underlying concepts, such as containers and Kubernetes. Then, we will explain how to create a Kubernetes cluster on Azure and deploy an example application.

By the time you have finished this section, you will have a foundational knowledge of containers and Kubernetes and will have a Kubernetes cluster up and running in Azure that will allow you to follow the examples in this book.

This section contains the following chapters:

- *Chapter 1, Introduction to containers and Kubernetes*
- *Chapter 2, Getting started with Azure Kubernetes Service*

1

Introduction to containers and Kubernetes

Kubernetes has become the leading standard in container orchestration. Since its inception in 2014, Kubernetes has gained tremendous popularity. It has been adopted by start-ups as well as major enterprises, with all major public cloud vendors offering a managed Kubernetes service.

Kubernetes builds upon the success of the Docker container revolution. Docker is both a company and the name of a technology. Docker as a technology is the most common way of creating and running software containers, called Docker containers. A container is a way of packaging software that makes it easy to run that software on any platform, ranging from your laptop to a server in a datacenter to a cluster running in the public cloud.

Although the core technology is open source, the Docker company focuses on reducing complexity for developers through a number of commercial offerings.

Kubernetes takes containers to the next level. Kubernetes is a container orchestrator. A container orchestrator is a software platform that makes it easy to run many thousands of containers on top of thousands of machines. It automates a lot of the manual tasks required to deploy, run, and scale applications. The orchestrator takes care of scheduling the right container to run on the right machine. It also takes care of health monitoring and failover, as well as scaling your deployed application.

The container technology Docker uses and Kubernetes are both open-source software projects. Open-source software allows developers from many companies to collaborate on a single piece of software. Kubernetes itself has contributors from companies such as Microsoft, Google, Red Hat, VMware, and many others.

The three major public cloud platforms—Azure, **Amazon Web Services** (**AWS**), and **Google Cloud Platform** (**GCP**)—all offer a managed Kubernetes service. They attract a lot of interest in the market since the virtually unlimited compute power and the ease of use of these managed services make it easy to build and deploy large-scale applications.

Azure Kubernetes Service (**AKS**) is Azure's managed service for Kubernetes. It reduces the complexity of building and managing Kubernetes clusters. In this book, you will learn how to use AKS to run your applications. Each chapter will introduce new concepts, which you will apply through the many examples in this book.

As a user, however, it is still very useful to understand the technologies that underpin AKS. We will explore these foundations in this chapter. You will learn about Linux processes and how they are related to Docker and containers. You will see how various processes fit nicely into containers and how containers fit nicely into Kubernetes.

This chapter introduces fundamental Docker concepts so that you can begin your Kubernetes journey. This chapter also briefly introduces the basics that will help you build containers, implement clusters, perform container orchestration, and troubleshoot applications on AKS. Having cursory knowledge of what's in this chapter will demystify much of the work needed to build your authenticated, encrypted, and highly scalable applications on AKS. Over the next few chapters, you will gradually build scalable and secure applications.

The following topics will be covered in this chapter:

- The software evolution that brought us here
- The fundamentals of containers
- The fundamentals of Kubernetes
- The fundamentals of AKS

The aim of this chapter is to introduce the essentials rather than to provide a thorough information source describing Docker and Kubernetes. To begin with, we'll first take a look at how software has evolved to get us to where we are now.

The software evolution that brought us here

There are two major software development evolutions that enabled the popularity of containers and Kubernetes. One is the adoption of a microservices architectural style. Microservices allow an application to be built from a collection of small services that each serve a specific function. The other evolution that enabled containers and Kubernetes is DevOps. DevOps is a set of cultural practices that allows people, processes, and tools to build and release software faster, more frequently, and more reliably.

Although you can use both containers and Kubernetes without using either microservices or DevOps, the technologies are most widely adopted for deploying microservices using DevOps methodologies.

In this section, we'll discuss both evolutions, starting with microservices.

Microservices

Software development has drastically evolved over time. Initially, software was developed and run on a single system, typically a mainframe. A client could connect to the mainframe through a terminal, and only through that terminal. This changed when computer networks became common when the client-server programming model emerged. A client could connect remotely to a server and even run part of the application on their own system while connecting to the server to retrieve the data the application required.

The client-server programming model has evolved toward distributed systems. Distributed systems are different from the traditional client-server model as they have multiple different applications running on multiple different systems, all interconnected.

Nowadays, a microservices architecture is common when developing distributed systems. A microservices-based application consists of a group of services that work together to form the application, while the individual services themselves can be built, tested, deployed, and scaled independently of each other. The style has many benefits but also has several disadvantages.

A key part of a microservices architecture is the fact that each individual service serves one and only one core function. Each service serves a single-bound business function. Different services work together to form the complete application. Those services work together over network communication, commonly using HTTP REST APIs or gRPC:

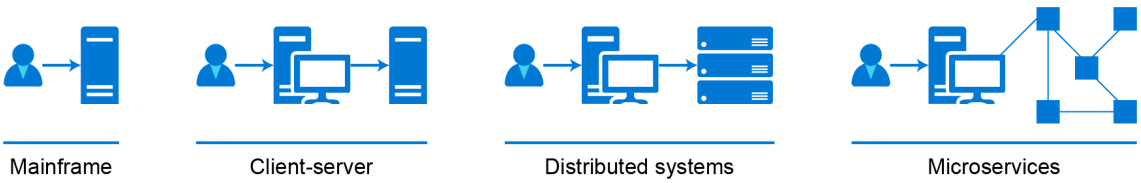


Figure 1.1: A standard microservices architecture

This architectural approach is commonly adopted by applications that run using containers and Kubernetes. Containers are used as the packaging format for the individual services, while Kubernetes is the orchestrator that deploys and manages the different services running together.

Before we dive into container and Kubernetes specifics, let's first explore the benefits and downsides of adopting microservices.

Advantages of running microservices

There are several advantages to running a microservices-based application. The first is the fact that each service is independent of the other services. The services are designed to be small enough (hence micro) to handle the needs of a business domain. As they are small, they can be made self-contained and independently testable, and so are independently releasable.

This leads to the benefit that each microservice is independently scalable as well. If a certain part of the application is getting more demand, that part of the application can be scaled independently from the rest of the application.

The fact that services are independently scalable also means that they are independently deployable. There are multiple deployment strategies when it comes to microservices. The most popular are rolling deployments and blue/green deployments.

With a rolling upgrade, a new version of the service is deployed only to a part of the application. This new version is carefully monitored and gradually gets more traffic if the service remains healthy. If something goes wrong, the previous version is still running, and traffic can easily be cut over.

With a blue/green deployment, you deploy the new version of the service in isolation. Once the new version of the service is deployed and tested, you cut over 100% of the production traffic to the new version. This allows for a clean transition between service versions.

Another benefit of the microservices architecture is that each service can be written in a different programming language. This is described as polyglot—the ability to understand and use multiple languages. For example, the front-end service can be developed in a popular JavaScript framework, the back end can be developed in C#, and the machine learning algorithm can be developed in Python. This allows you to select the right language for the right service and allows developers to use the languages they are most familiar with.

Disadvantages of running microservices

There's a flip side to every coin, and the same is true for microservices. While there are multiple advantages to a microservices-based architecture, this architecture has its downsides as well.

Microservices designs and architectures require a high degree of software development maturity in order to be implemented correctly. Architects who understand the domain very well must ensure that each service is bounded and that different services are cohesive. Since services are independent of each other and versioned independently, the software contract between these different services is important to get right.

Another common issue with a microservices design is the added complexity when it comes to monitoring and troubleshooting such an application. Since different services make up a single application, and those different services run on multiple servers, both logging and tracing such an application is a complicated endeavor.

Linked to the disadvantages mentioned before is that, typically, in microservices, you need to build more fault tolerance into your application. Due to the dynamic nature of the different services in an application, faults are more likely to happen. In order to guarantee application availability, it is important to build fault tolerance into the different microservices that make up an application. Implementing patterns such as retry logic or circuit breakers is critical to avoid a single fault causing application downtime.

In this section, you learned about microservices, their benefits, and their disadvantages. Often linked to microservices, but a separate topic, is the DevOps movement. We will explore what DevOps means in the next section.

DevOps

DevOps literally means the combination of development and operations. More specifically, DevOps is the union of people, processes, and tools to deliver software faster, more frequently, and more reliably. DevOps is more about a set of cultural practices than about any specific tools or implementations. Typically, DevOps spans four areas of software development: planning, developing, releasing, and operating software.

> **Note**
>
> Many definitions of DevOps exist. The authors have adopted this definition, but you as a reader are encouraged to explore different definitions in the literature around DevOps.

The DevOps culture starts with planning. In the planning phase of a DevOps project, the goals of a project are outlined. These goals are outlined both at a high level (called an epic) and at a lower level (as features and tasks). The different work items in a DevOps project are captured in the feature backlog. Typically, DevOps teams use an agile planning methodology working in programming sprints. Kanban boards are often used to represent project status and to track work. As a task changes status from *to do* to *doing* to *done*, it moves from left to right on a Kanban board.

When work is planned, actual development can be done. Development in a DevOps culture isn't only about writing code but also about testing, reviewing, and integrating code with team members. A version control system such as Git is used for different team members to share code with each other. An automated **continuous integration** (**CI**) tool is used to automate most manual tasks such as testing and building code.

When a feature is code-complete, tested, and built, it is ready to be delivered. The next phase in a DevOps project can start delivery. A **continuous delivery** (**CD**) tool is used to automate the deployment of software. Typically, software is deployed to different environments, such as testing, quality assurance, and production. A combination of automated and manual gates is used to ensure quality before moving to the next environment.

Finally, when a piece of software is running in production, the operations phase can start. This phase involves the maintaining, monitoring, and supporting of an application in production. The end goal is to operate an application reliably with as little downtime as possible. Any issues are to be identified as proactively as possible. Bugs in the software will be tracked in the backlog.

The DevOps process is an iterative process. A single team is never in a single phase of the process. The whole team is continuously planning, developing, delivering, and operating software.

Multiple tools exist to implement DevOps practices. There are point solutions for a single phase, such as Jira for planning or Jenkins for CI and CD, as well as complete DevOps platforms, such as GitLab. Microsoft operates two solutions that enable customers to adopt DevOps practices: Azure DevOps and GitHub. Azure DevOps is a suite of services to support all phases of the DevOps process. GitHub is a separate platform that enables DevOps software development. GitHub is known as the leading open-source software development platform, hosting over 40 million open-source projects.

Both microservices and DevOps are commonly used in combination with containers and Kubernetes. Now that we've had this introduction to microservices and DevOps, we'll continue this first chapter with the fundamentals of containers and then the fundamentals of Kubernetes.

Fundamentals of containers

A form of container technology has existed in the Linux kernel since the 1970s. The technology powering today's containers, called **cgroups** (abbreviated from **control groups**), was introduced into the Linux kernel in 2006 by Google. The Docker company popularized the technology in 2013 by introducing an easy developer workflow. Although the name Docker can refer to both the company as well as the technology, most commonly, though, we use Docker to refer to the technology.

> **Note**
>
> Although the Docker technology is a popular way to build and run containers, it is not the only way to build and run them. Many alternatives exist for either building or running containers. One of those alternatives is containerd, which is a container runtime also used by Kubernetes.

Docker as a technology is both a packaging format and a container runtime. Packaging is a process that allows an application to be packaged together with its dependencies, such as binaries and runtime. The runtime points at the actual process of running the container images.

There are three important pieces in Docker's architecture: the client, the daemon, and the registry:

- The Docker client is a client-side tool that you use to interact with the Docker daemon, running locally or remotely.
- The Docker daemon is a long-running process that is responsible for building container images and running containers. The Docker daemon can run on either your local machine or a remote machine.
- A Docker registry is a place to store Docker images. There are public registries such as Docker Hub that contain public images, and there are private registries such as **Azure Container Registry** (**ACR**) that you can use to store your own private images. The Docker daemon can pull images from a registry if images are not available locally:

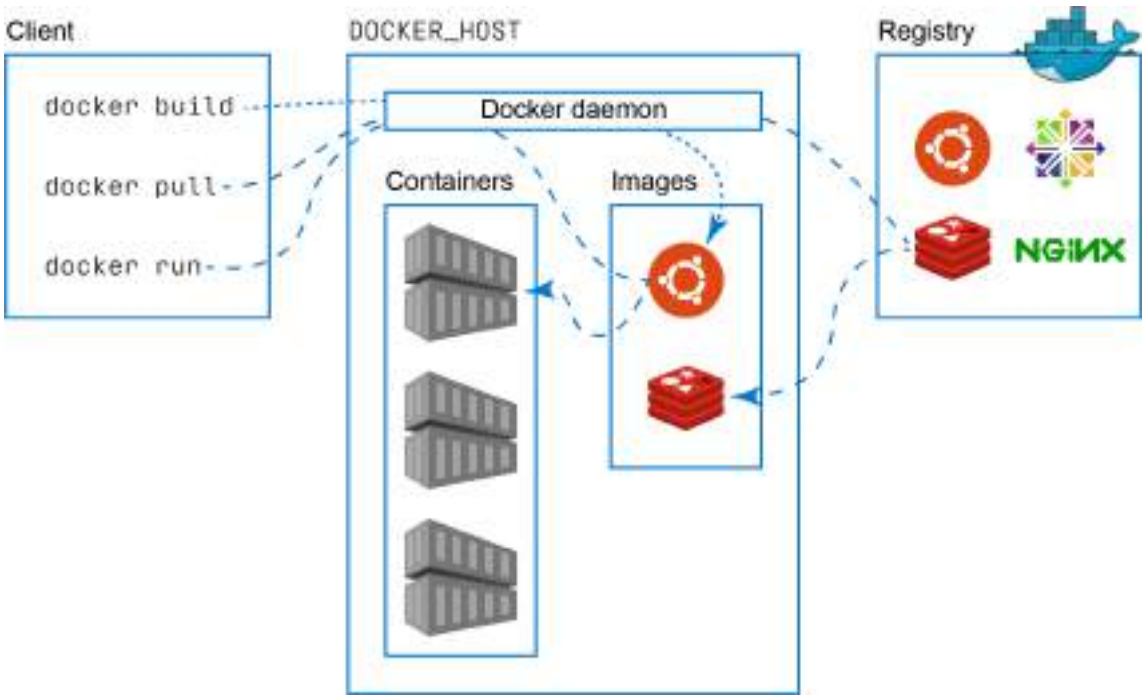


Figure 1.2: Fundamentals of Docker architecture

You can experiment with Docker by creating a free Docker account at Docker Hub (https://hub.docker.com/) and using that login to open Docker Labs (https://labs.play-with-docker.com/). This will give you access to an environment with Docker pre-installed that is valid for 4 hours. We will be using Docker Labs in this section as we build our own container and image.

> **Note**
>
> Although we are using the browser-based Docker Labs in this chapter to introduce Docker, you can also install Docker on your local desktop or server. For workstations, Docker has a product called Docker Desktop (https://www.docker.com/products/docker-desktop) that is available for Windows and Mac to create Docker containers locally. On servers—both Windows and Linux—Docker is also available as a runtime for containers.

Container images

To start a new container, you need an image. An image contains all the software you need to run within your container. Container images can be stored locally on your machine, as well as in a container registry. There are public registries, such as the public Docker Hub (https://hub.docker.com/), or private registries, such as ACR. When you, as a user, don't have an image locally on your PC, you can pull an image from a registry using the `docker pull` command.

In the following example, we will pull an image from the public Docker Hub repository and run the actual container. You can run this example in Docker Labs, which we introduced in the previous section, by following these instructions:

```
#First, we will pull an image
docker pull docker/whalesay
#We can then look at which images are stored locally
docker images
#Then we will run our container
docker run docker/whalesay cowsay boo
```

The output of these commands will look similar to *Figure* 1.3:

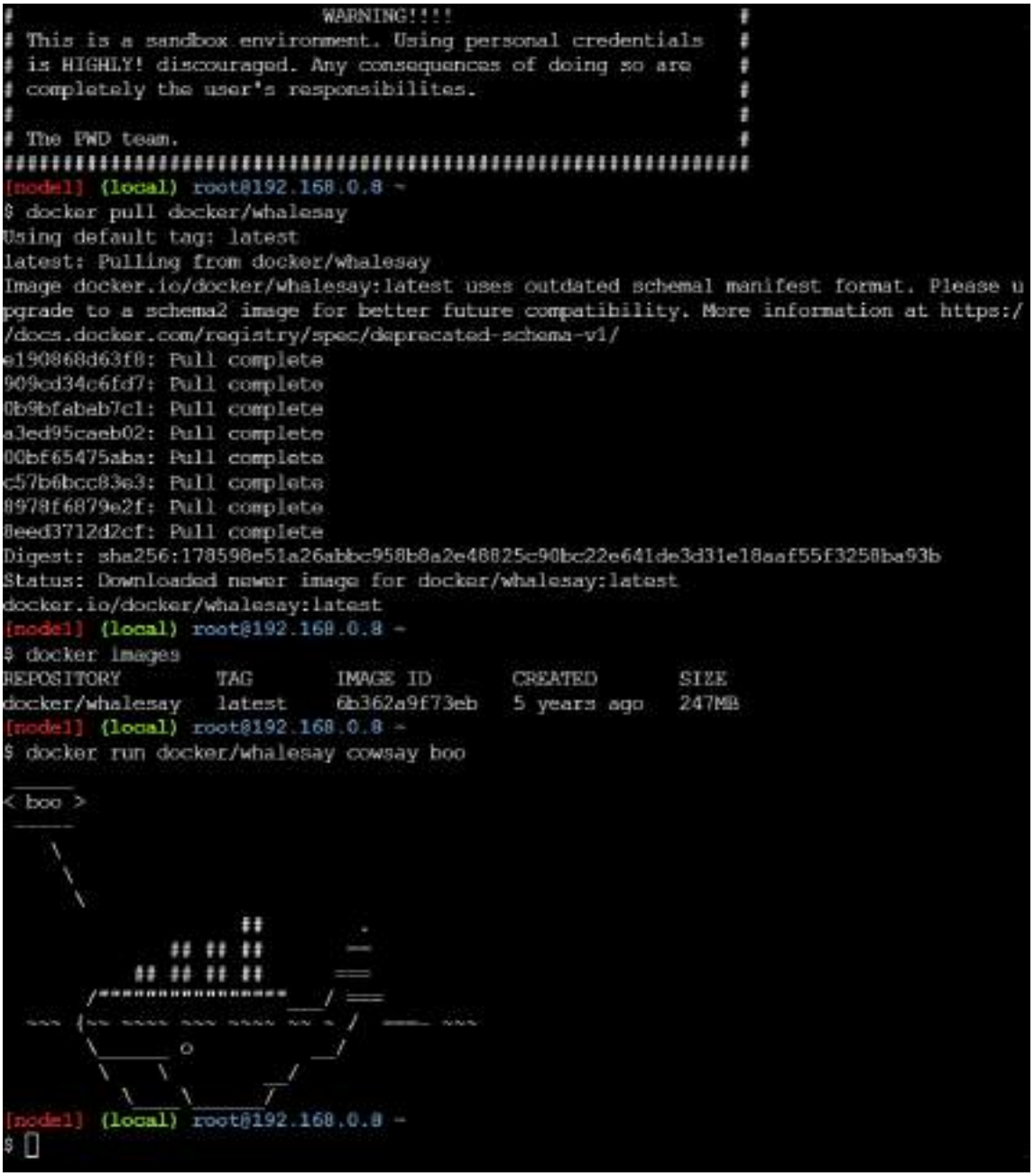


Figure 1.3: An example of running containers in Docker Labs

What happened here is that Docker first pulled your image in multiple parts and stored it locally on the machine it was running on. When you ran the actual application, it used that local image to start a container. If we look at the commands in detail, you will see that `docker pull` took in a single parameter, `docker/whalesay`. If you don't provide a private container registry, Docker will look in the public Docker Hub for images, which is where Docker pulled this image from. The `docker run` command took in a couple of arguments. The first argument was `docker/whalesay`, which is the reference to the image. The next two arguments, `cowsay boo`, are commands that were passed to the running container to execute.

In the previous example, you learned that it is possible to run a container without building an image first. It is, however, very common that you will want to build your own images. To do this, you use a Dockerfile. A Dockerfile contains steps that Docker will follow to start from a base image and build your image. These instructions can range from adding files to installing software or setting up networking.

In the next example, you will build a custom Docker image. This custom image will display inspirational quotes in the whale output. The following Dockerfile will be used to generate this custom image. You will create it in your Docker playground:

```
FROM docker/whalesay:latest
RUN apt-get -y -qq update
RUN apt-get install -qq -y fortunes
CMD /usr/games/fortune -a | cowsay
```

There are four lines in this Dockerfile. The first one will instruct Docker on which image to use as a source image for this new image. The next two steps are commands that are run to add new functionality to our image, in this case, updating your `apt` repository and installing an application called `fortunes`. The `fortunes` application is a small command-line tool that generates inspirational quotes. We will use that to include quotes in the output rather than user input. Finally, the `CMD` command tells Docker which command to execute when a container based on this image is run.

You typically save a Dockerfile in a file called `Dockerfile`, without an extension. To build an image, you need to execute the `docker build` command and point it to the Dockerfile you created. In building the Docker image, the Docker daemon will read the Dockerfile and execute the different steps in the Dockerfile. This command will also output the steps it took to run a container and build your image. Let's walk through a demo of building an image.

In order to create this Dockerfile, open up a text editor via the `vi Dockerfile` command. **vi** is an advanced text editor on the Linux command line. If you are not familiar with it, let's walk through how you would enter the text in there:

1. After you've opened vi, hit the *I* key to enter insert mode.
2. Then, either copy and paste or type the four code lines.
3. Afterward, hit the *Esc* key, and type `:wq!` to write `(w)` your file and quit `(q)` the text editor.

The next step is to execute `docker build` to build the image. We will add a final bit to that command, namely adding a tag to our image so we can call it by a meaningful name. To build the image, you will use the `docker build -t smartwhale.` command (don't forget to add the final period here).

You will now see Docker execute a number of steps—four in this case—to build the image. After the image is built, you can run your application. To run your container, you run `docker run smartwhale`, and you should see an output similar to *Figure 1.4*. However, you will probably see a different smart quote. This is due to the `fortunes` application generating different quotes. If you run the container multiple times, you will see different quotes appear, as shown in *Figure 1.4*:

Figure 1.4: Running a custom container

That concludes our overview and demo of containers. In this section, you started with an existing container image and launched it on Docker Labs. Afterward, you took that a step further and built your own container image, then started containers using that image. You have now learned what it takes to build and run a container. In the next section, we will cover Kubernetes. Kubernetes allows you to run multiple containers at scale.

Kubernetes as a container orchestration platform

Building and running a single container seems easy enough. However, things can get complicated when you need to run multiple containers across multiple servers. This is where a container orchestrator can help. A container orchestrator takes care of scheduling containers to be run on servers, restarting containers when they fail, moving containers to a new host when a host becomes unhealthy, and much more.

The current leading orchestration platform is Kubernetes (https://kubernetes.io/). Kubernetes was inspired by Google's Borg project, which, by itself, was running millions of containers in production.

Kubernetes takes a declarative approach to orchestration; that is, you specify what you need, and Kubernetes takes care of deploying the workload you specified. You don't need to start these containers manually yourself anymore, as Kubernetes will launch the containers you specified.

> **Note**
>
> Although Kubernetes used to support Docker as the container runtime, that support has been deprecated in Kubernetes version 1.20. In AKS, **containerd** has become the default container runtime starting with Kubernetes 1.19.

Throughout the book, you will build multiple examples that run containers in Kubernetes, and you will learn more about the different objects in Kubernetes. In this introductory chapter, you will learn three elementary objects in Kubernetes that you will likely see in every application: a pod, a deployment, and a service.

Pods in Kubernetes

A **pod** in Kubernetes is the essential scheduling element. A pod is a group of one or more containers. This means a pod can contain either a single container or multiple containers. When creating a pod with a single container, you can use the terms container and pod interchangeably. However, the term pod is still preferred and is the term used throughout this book.

When a pod contains multiple containers, these containers share the same file system and the same network namespace. This means that when a container that is part of a pod writes a file, other containers in that same pod can read that file as well. This also means that all containers in a pod can communicate with each other using localhost networking.

In terms of design, you should only put containers that need to be tightly integrated in the same pod. Imagine the following situation: you have an old web application that does not support HTTPS. You want to upgrade that application to support HTTPS. You could create a pod that contains your old web application and includes another container that would do **Transport Layer Security** (**TLS**) offloading for that application, as described in *Figure* 1.5. Users would connect to your application using HTTPS, while the container in the middle converts HTTPS traffic to HTTP:

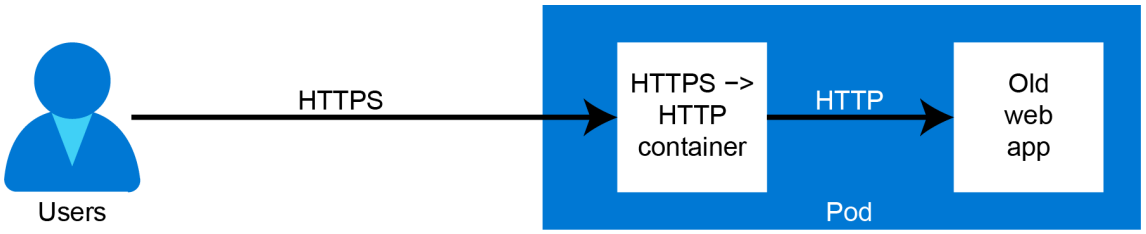


Figure 1.5: An example of a multi-container pod that does HTTPS offloading

> **Note**
>
> This design principle is known as a sidecar. Microsoft has a free e-book available that describes multiple multi-container pod designs and designing distributed systems (https://azure.microsoft.com/resources/designing-distributed-systems/).

A pod, whether it be a single- or multi-container pod, is an ephemeral resource. This means that a pod can be terminated at any point and restarted on another node. When this happens, the state that was stored in that pod will be lost. If you need to store state in your application, you either need to store that state in external storage, such as an external disk or a file share, or store the state outside of Kubernetes in an external database.

Deployments in Kubernetes

A **deployment** in Kubernetes provides a layer of functionality around pods. It allows you to create multiple pods from the same definition and to easily perform updates to your deployed pods. A deployment also helps with scaling your application, and potentially even autoscaling your application.

Under the hood, a deployment creates a **ReplicaSet**, which in turn will create the replica pods you requested. A ReplicaSet is another object in Kubernetes. The purpose of a ReplicaSet is to maintain a stable set of replica pods running at any given time. If you perform updates on your deployment, Kubernetes will create a new ReplicaSet that will contain the updated pods. By default, Kubernetes will do a rolling upgrade to the new version. This means that it will start a few new pods, verify those are running correctly, and if so, then Kubernetes will terminate the old pods and continue this loop until only new pods are running:

Figure1.6: The relationship between deployments, ReplicaSets, and pods

Services in Kubernetes

A **service** in Kubernetes is a network-level abstraction. This allows you to expose multiple pods under a single IP address and a single DNS name.

Each pod in Kubernetes has its own private IP address. You could theoretically connect to your applications using this private IP address. However, as mentioned before, Kubernetes pods are ephemeral, meaning they can be terminated and moved, which would change their IP address. By using a service, you can connect to your applications using a single IP address. When a pod moves from one node to another, the service ensures that traffic is routed to the correct endpoint. If there are multiple pods serving traffic behind one service, that traffic will be load balanced between the different pods.

In this section, we have introduced Kubernetes and three essential objects with Kubernetes. In the next section, we'll introduce AKS.

Azure Kubernetes Service

AKS makes creating and managing Kubernetes clusters easier.

A typical Kubernetes cluster consists of a number of master nodes and a number of worker nodes. A node within Kubernetes is equivalent to a server or a **virtual machine** (**VM**). The master nodes contain the Kubernetes API and a database that contains the cluster state. The worker nodes are the machines that run your actual workload.

AKS makes it easier to create a cluster. When you create an AKS cluster, AKS sets up the Kubernetes master for you. AKS will then create one or more **virtual machine scale sets** (**VMSS**) in your subscription and turn the VMs in these VMSSs into worker nodes of your Kubernetes cluster in your network. In AKS, you have the option to either use a free Kubernetes control plane or pay for a control plane that comes with a financially backed SLA. In either case, you also need to pay for the VMs hosting your worker nodes:

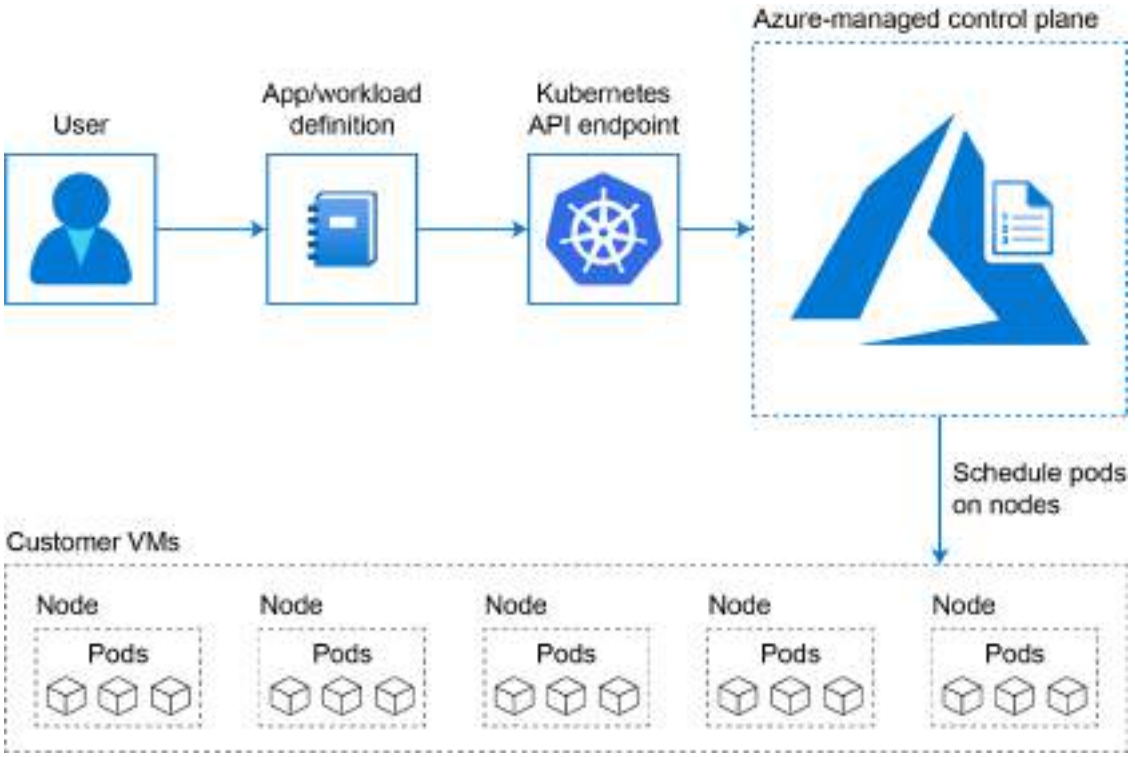


Figure 1.7: Scheduling of pods in AKS

Within AKS, services running on Kubernetes are integrated with Azure Load Balancer and Kubernetes Ingresses can be integrated with Azure Application Gateway. The Azure Load Balancer is a layer-4 network load balancer service; Application Gateway is a layer-7 HTTP-based load balancer. The integration between Kubernetes and both services means that when you create a service or Ingress in Kubernetes, Kubernetes will create a rule in an Azure Load Balancer or Azure Application Gateway respectively. Azure Load Balancer or Application Gateway will then route the traffic to the right node in your cluster that hosts your pod.

Additionally, AKS adds a number of functionalities that make it easier to manage a cluster. AKS contains logic to upgrade clusters to newer Kubernetes versions. It also can easily scale your clusters, by either adding or removing nodes to the cluster.

AKS also comes with integration options that make operations easier. AKS clusters can be configured with integration with **Azure Active Directory** (**Azure AD**) to make managing identities and **role-based access control** (**RBAC**) straightforward. RBAC is the configuration process that defines which users get access to resources and which actions they can take against those resources. AKS can also easily be integrated into Azure Monitor for containers, which makes monitoring and troubleshooting your applications simpler. You will learn about all these capabilities throughout this book.

Summary

In this chapter, you learned about the concepts of containers and Kubernetes. You ran a number of containers, starting with an existing image and then using an image you built yourself. After that demo, you were introduced to three essential Kubernetes objects: the pod, the deployment, and the service.

This provides the context for the remaining chapters, where you will deploy containerized applications using Microsoft AKS. You will see how the AKS offering from Microsoft streamlines deployment by handling many of the management and operational tasks that you would have to do yourself if you managed and operated your own Kubernetes infrastructure.

In the next chapter, you will use the Azure portal to create your first AKS cluster.

2

Getting started with Azure Kubernetes Service

Installing and maintaining Kubernetes clusters correctly and securely is difficult. Thankfully, all the major cloud providers, such as **Azure**, **Amazon Web Services** (**AWS**), and **Google Cloud Platform** (**GCP**), facilitate installing and maintaining clusters. In this chapter, you will navigate through the Azure portal, launch your own cluster, and run a sample application. You will accomplish all of this from your browser.

The following topics will be covered in this chapter:

- Creating a new Azure free account
- Creating and launching your first cluster
- Deploying and inspecting your first demo application

Let's start by looking at different ways to create an **Azure Kubernetes Service** (**AKS**) cluster, and then we will run our sample application.

Different ways to create an AKS cluster

In this chapter, you will use the Azure portal to deploy your AKS cluster. There are, however, multiple ways to create an AKS cluster:

- **Using the portal**: The portal offers a **graphical user interface** (**GUI**) for deploying your cluster through a wizard. This is a great way to deploy your first cluster. For multiple deployments or automated deployments, one of the following methods is recommended.
- **Using the Azure CLI**: The Azure **command-line interface** (**CLI**) is a cross-platform CLI for managing Azure resources. This allows you to script your cluster deployment, which can be integrated into other scripts.
- **Using Azure PowerShell**: Azure PowerShell is a set of PowerShell commands used for managing Azure resources directly from PowerShell. It can also be used to create Kubernetes clusters.
- **Using ARM templates**: **Azure Resource Manager** (**ARM**) templates are an Azure-native way to deploy Azure resources using **Infrastructure as Code** (**IaC**). You can declaratively deploy your cluster, allowing you to create a template that can be reused by multiple teams.
- **Using Terraform for Azure**: Terraform is an open-source IaC tool developed by HashiCorp. The tool is very popular in the open-source community for deploying cloud resources, including AKS. Like ARM templates, Terraform also uses declarative templates for your cluster.

In this chapter, you will create your cluster using the Azure portal. If you are interested in deploying a cluster using either CLI, ARM templates, or Terraform, the following Azure documentation contains steps on how to use these tools to create your own clusters https://docs.microsoft.com/azure/aks.

Getting started with the Azure portal

We will start our initial cluster deployment using the Azure portal. The Azure portal is a web-based management console. It allows you to build, manage, and monitor all your Azure deployments worldwide through a single console.

> **Note**
>
> To follow along with the examples in this book, you will need an Azure account. If you don't have an Azure account, you can create a free account by following the steps at azure.microsoft.com/free. If you plan to run this in an existing subscription, you will need owner rights to the subscription and the ability to create service principals in **Azure Active Directory** (**Azure AD**). All the examples in this book have been verified with a free trial account.

We are going to jump straight in by creating our AKS cluster. By doing so, we are also going to familiarize ourselves with the Azure portal.

Creating your first AKS cluster

To start, browse to the Azure portal on https://portal.azure.com. Enter the keyword aks in the search bar at the top of the Azure portal. Click on **Kubernetes services** under the **Services** category in the search results:

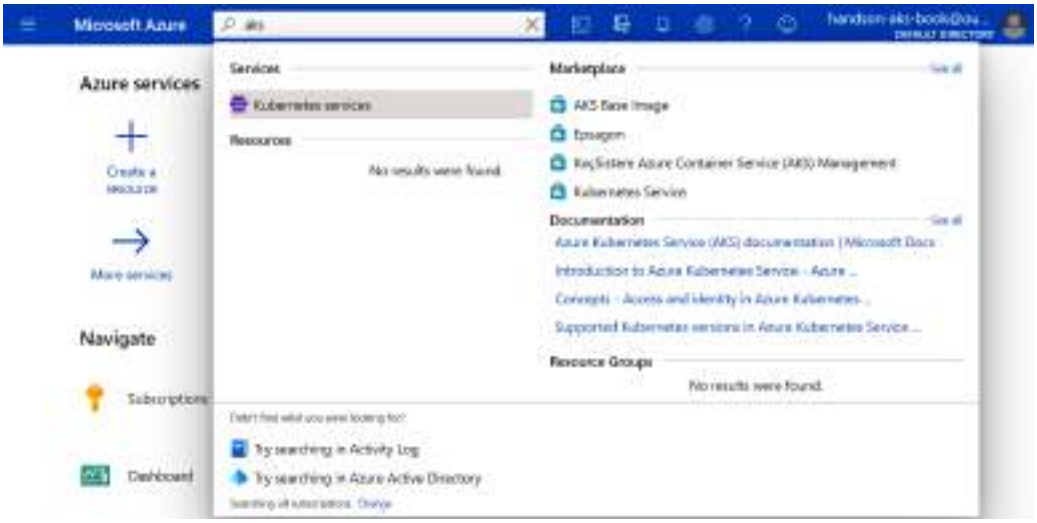


Figure 2.1: Searching for AKS with the search bar

This will take you to the AKS pane in the portal. As you might have expected, you don't have any clusters yet. Go ahead and create a new cluster by hitting the **+ Add** button, and select the **+ Add Kubernetes cluster** option:

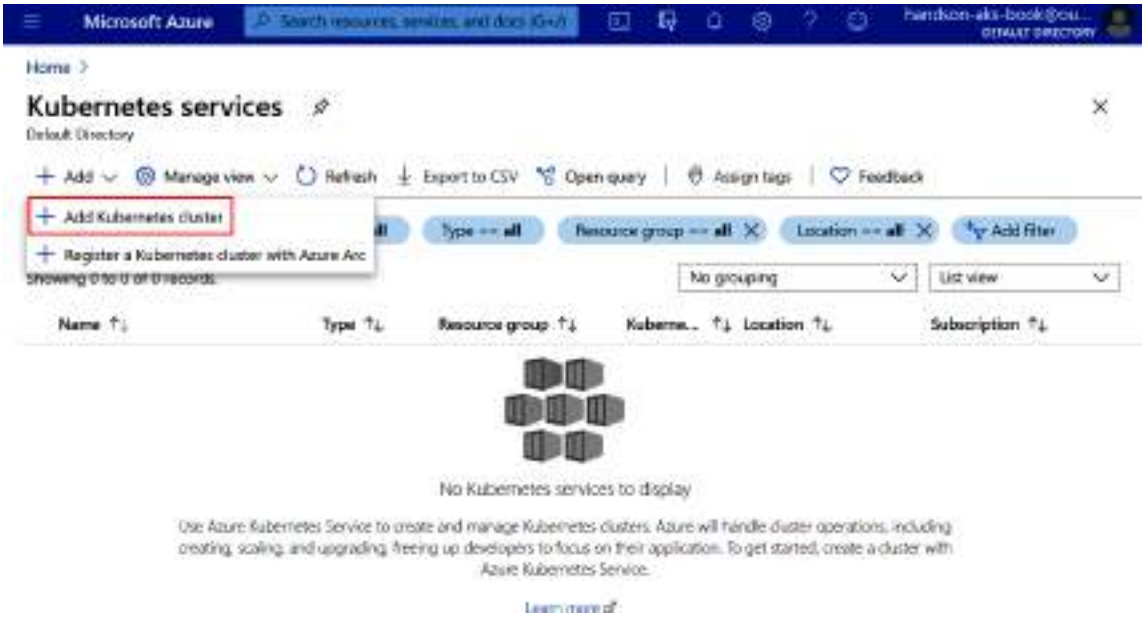


Figure 2.2: Clicking the + Add button and the + Add Kubernetes cluster button to start the cluster creation process

> ### Note
>
> There are a lot of options to configure when you're creating an AKS cluster. For your first cluster, we recommend sticking with the defaults from the portal and following our naming guidelines during this example. The following settings were tested by us to work reliably with a free account.

This will take you to the creation wizard to create your first AKS cluster. The first step here is to create a new resource group. Click **Create new**, give your resource group a name, and hit **OK**. If you want to follow along with the examples in this book, please name the resource group `rg-handsonaks`:

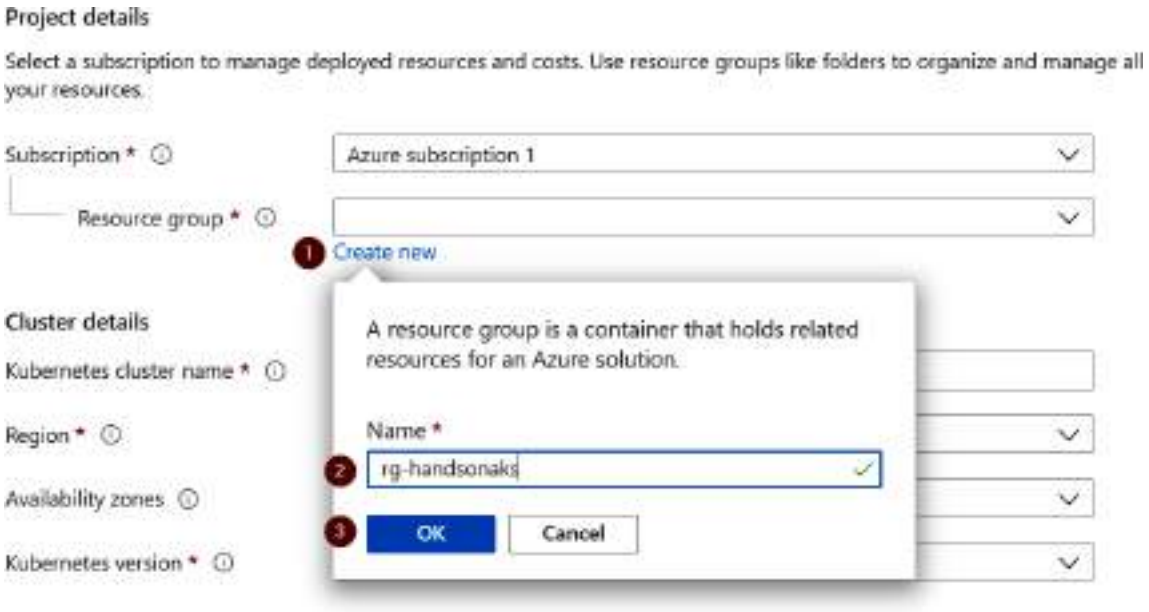


Figure 2.3: Creating a new resource group

Next up, we'll provide the cluster details. Give your cluster a name—if you want to follow the examples in the book, please call it handsonaks. The region we will use in the book is `(US) West US 2`, but you could use any other region of choice close to your location. If the region you selected supports Availability Zones, unselect all the zones.

Select a Kubernetes version—at the time of writing, version 1.19.6 is the latest version that is supported; don't worry if that specific version is not available for you. Kubernetes and AKS evolve very quickly, and new versions are introduced often:

> **Note**
>
> For production environments, deploying a cluster in an Availability Zone is recommended. However, since we are deploying a small cluster, not using Availability Zones works best for the examples in the book.

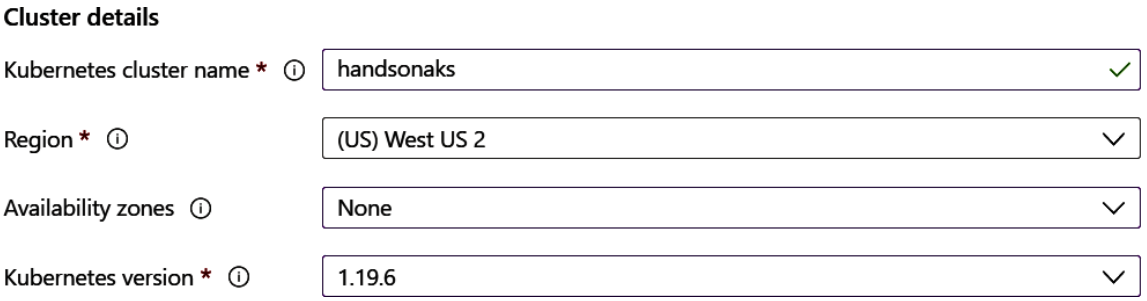


Figure 2.4: Providing the cluster details

Next, change the node count to 2. For the purposes of the demo in this book, the default `Standard DS2 v2` node size is sufficient. This should make your cluster size look similar to that shown in *Figure 2.5*:

Figure 2.5: Updated Node size and Node count

> **Note**
>
> Your free account has a four-core limit that will be breached if you go with the defaults.

The final view of the first pane should look like *Figure* 2.6. There are a number of configuration panes, which you need not change for the demo cluster we'll that you'll use throughout this book. Since you are ready, hit the **Review + create** button to do a final review and create your cluster:

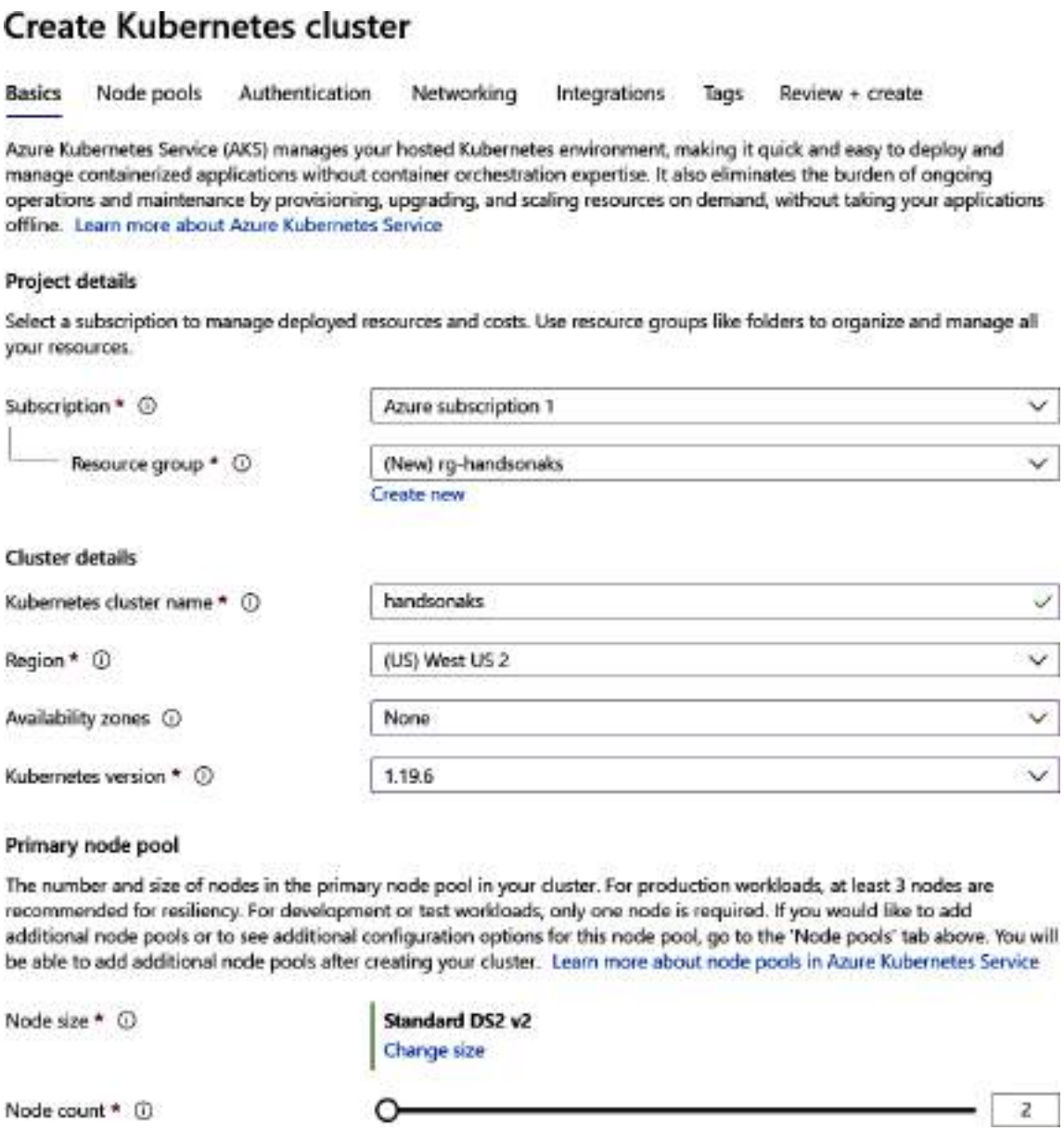


Figure 2.6: Setting the cluster configuration

In the final view, Azure will validate the configuration that was applied to your first cluster. If you get the message **Validation passed**, click **Create**:

Figure 2.7: The final validation of your cluster configuration

Deploying the cluster should take roughly 10 minutes. Once the deployment is complete, you can check the deployment details as shown in *Figure* 2.8:

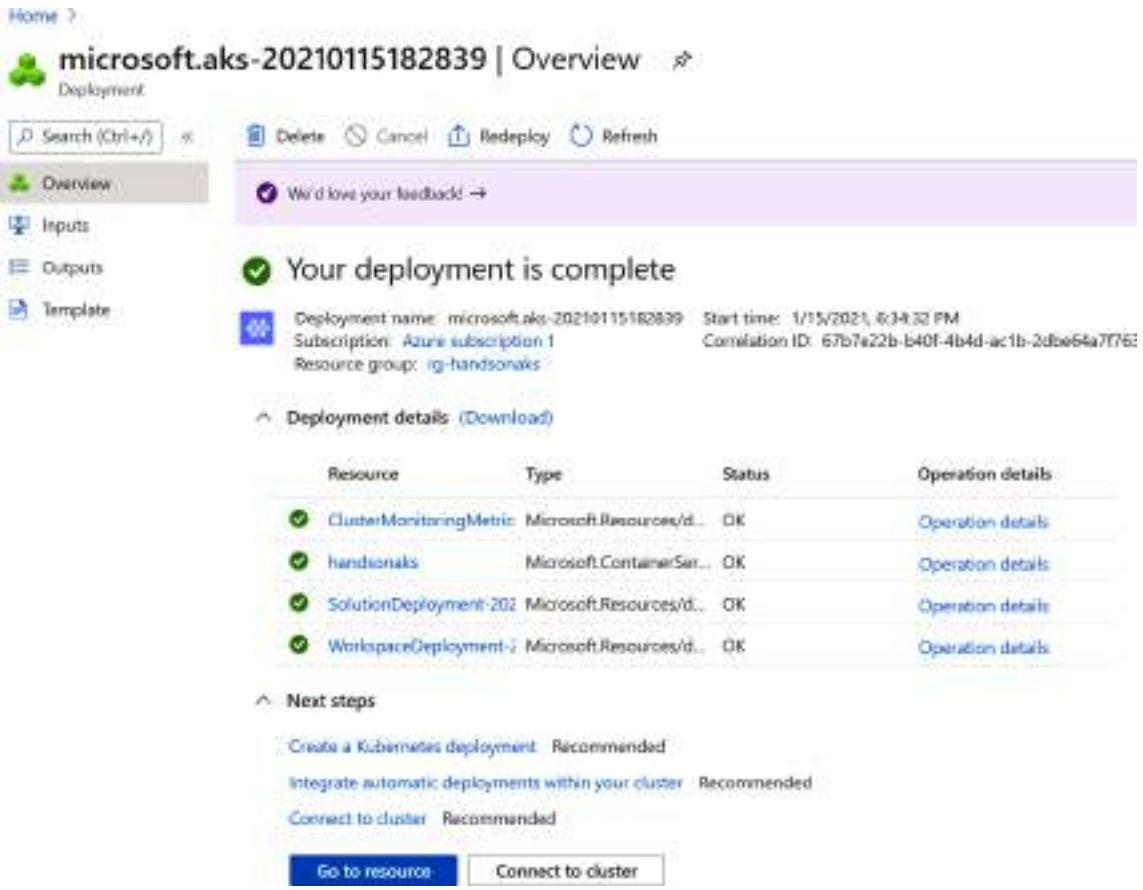


Figure 2.8: Deployment details once the cluster is successfully deployed

If you get a quota limitation error, as shown in *Figure* 2.9, check the settings and try again. Make sure that you select the **Standard DS2_v2** node size and only two nodes:

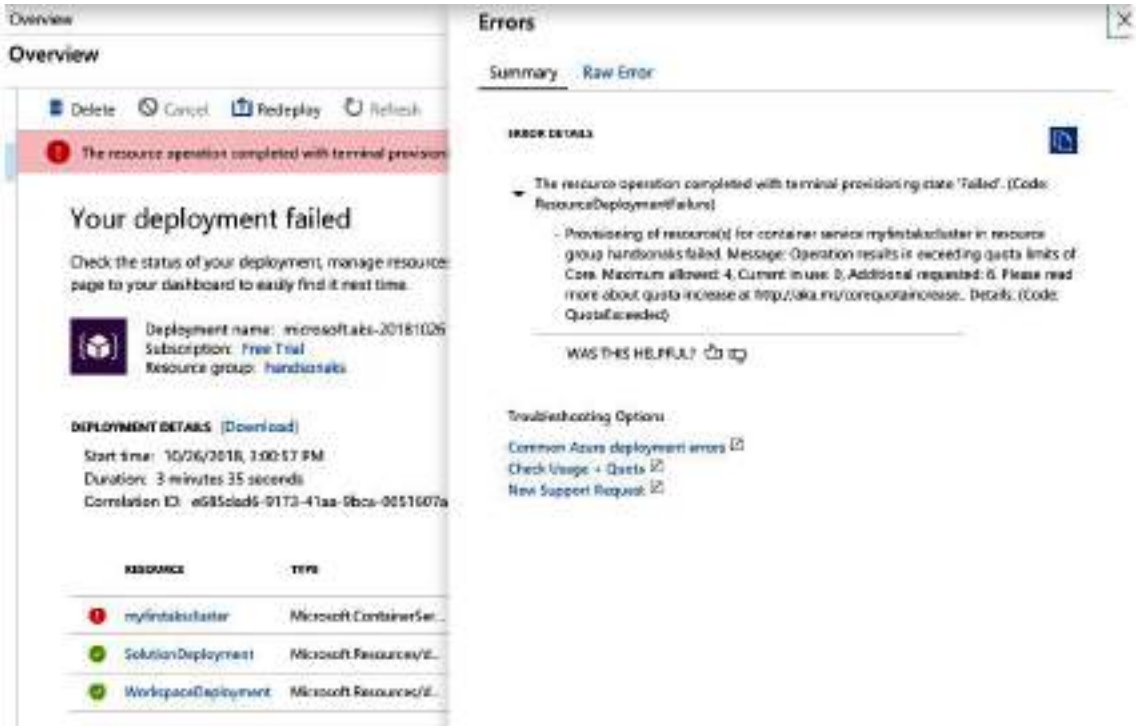


Figure 2.9: Retrying with a smaller cluster size due to a quota limit error

Moving to the next section, we'll take a quick first look at your cluster; hit the **Go to resource** button as seen in *Figure* 2.8. This will take you to the AKS cluster dashboard in the portal.

A quick overview of your cluster in the Azure portal

If you hit the **Go to resource** button in the previous section, you will see the overview of your cluster in the Azure portal:

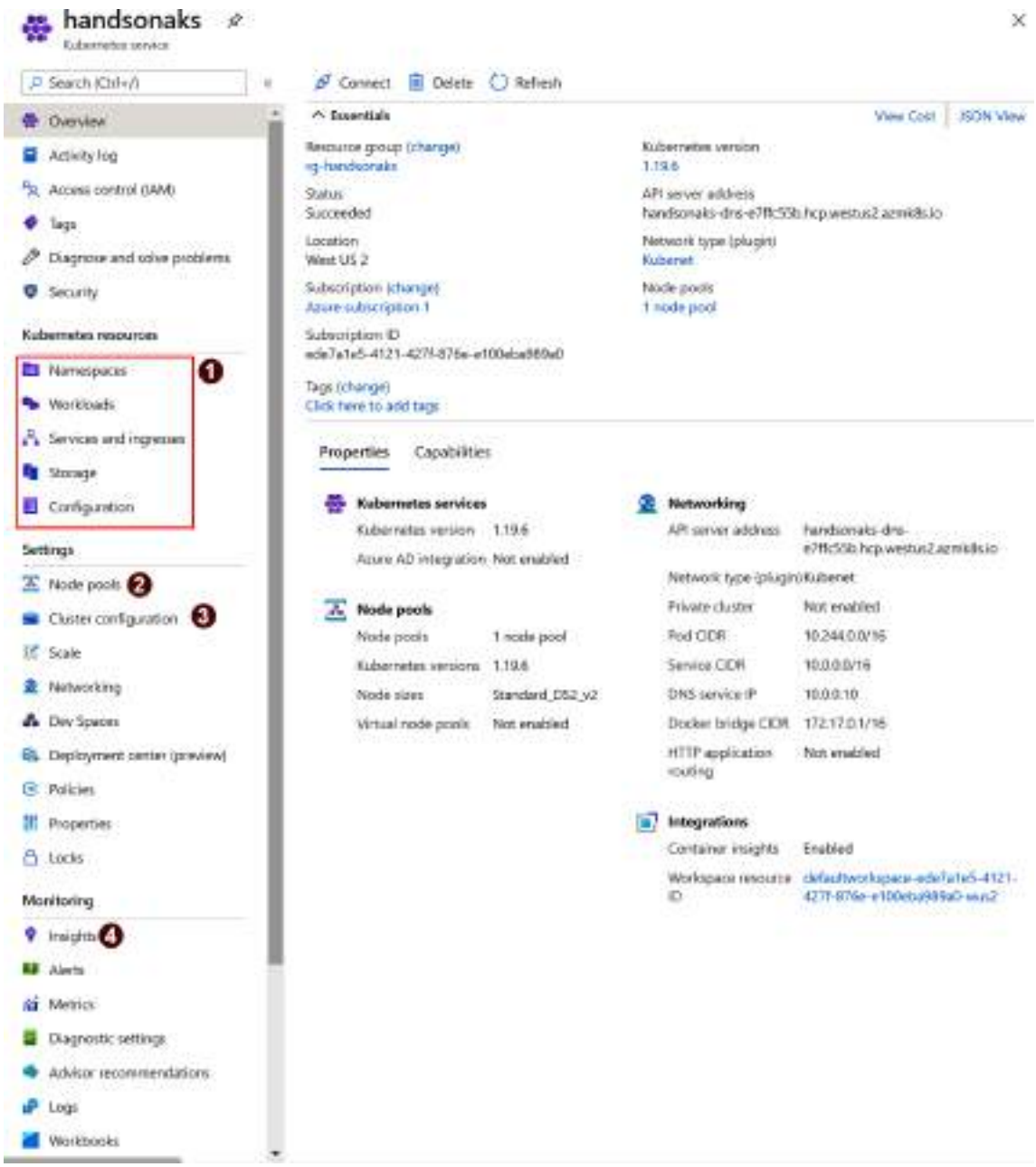


Figure 2.10: The AKS pane in the Azure portal

This is a quick overview of your cluster. It displays the name, the location, and the API server address. The navigation menu on the left provides different options to control and manage your cluster. Let's walk through a couple of interesting options that the has to portal offer.

The **Kubernetes resources** section gives you an insight into the workloads that are running on your cluster. You could, for instance, see running deployments and running pods in your cluster. It also allows you to create new resources on your cluster. We will use this section later in the chapter after you have deployed your first application on AKS.

In the **Node pools** pane, you can scale your existing node pool (meaning the nodes or servers in your cluster) either up or down by adding or removing nodes. You can add a new node pool, potentially with a different virtual machine size, and you can also upgrade your node pools individually. In *Figure 2.11*, you can see the **+ Add node pool** option at the top-left corner, and if you select your node pool, the **Upgrade and Scale** options also become available in the top bar:

Figure 2.11: Adding, scaling, and upgrading node pools

In the **Cluster configuration** pane, you can instruct AKS to upgrade the control plane to a newer version. Typically, in a Kubernetes upgrade, you first upgrade the control plane, and then the individual node pools separately. This pane also allows you to enable **role-based access control** (**RBAC**) (which is enabled by default), and optionally integrate your cluster with Azure AD. You will learn more about Azure AD integration in *Chapter 8*, *Role-based access control in AKS*:

Figure 2.12: Upgrading the Kubernetes version of the API server using the Upgrade pane

Finally, the **Insights** pane allows you to monitor your cluster infrastructure and the workloads running on your cluster. Since your cluster is brand new, there isn't a lot of data to investigate. We will return back to this, in *Chapter 7*, *Monitoring the AKS cluster and the application*:

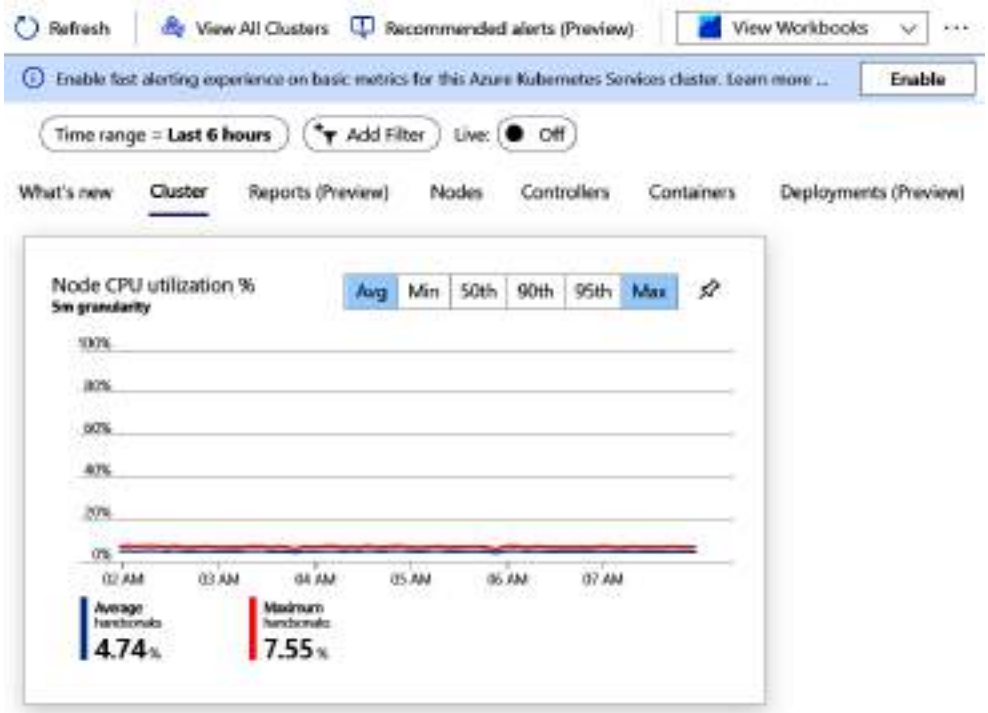


Figure 2.13: Displaying cluster utilization using the Insights pane

This concludes our quick overview of the cluster and some of the interesting configuration options in the Azure portal. In the next section, we'll connect to our AKS cluster using Cloud Shell and then launch a demo application on top of this cluster.

Accessing your cluster using Azure Cloud Shell

Once the deployment is completed successfully, find the small Cloud Shell icon near the search bar, as highlighted in *Figure* 2.14, and click it:

Figure 2.14: Clicking the Cloud Shell icon to open Azure Cloud Shell

The portal will ask you to select either **PowerShell** or **Bash** as your default shell experience. As we will be working mainly with Linux workloads, please select **Bash**:

Figure 2.15: Selecting the Bash option

If this is the first time you have launched Cloud Shell, you will be asked to create a storage account; confirm and create it:

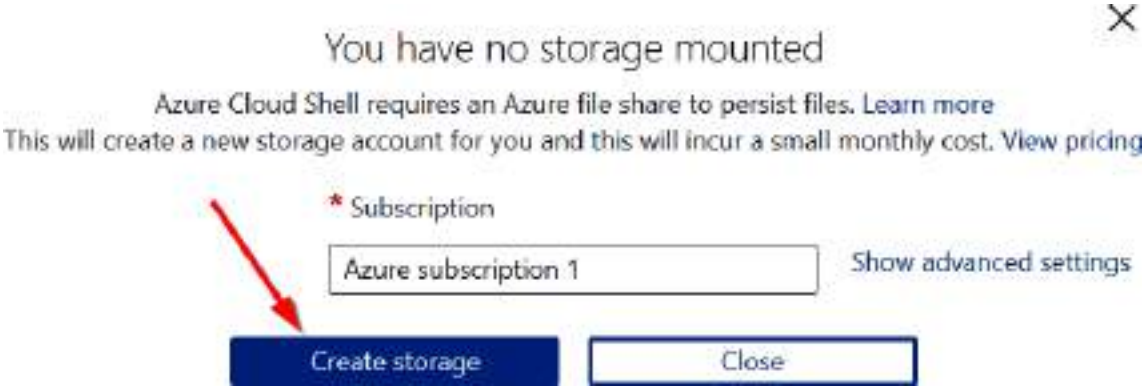


Figure 2.16: Creating a new storage account for Cloud Shell

After creating the storage, you might get an error message that contains a mount storage error. If that occurs, please restart your Cloud Shell:

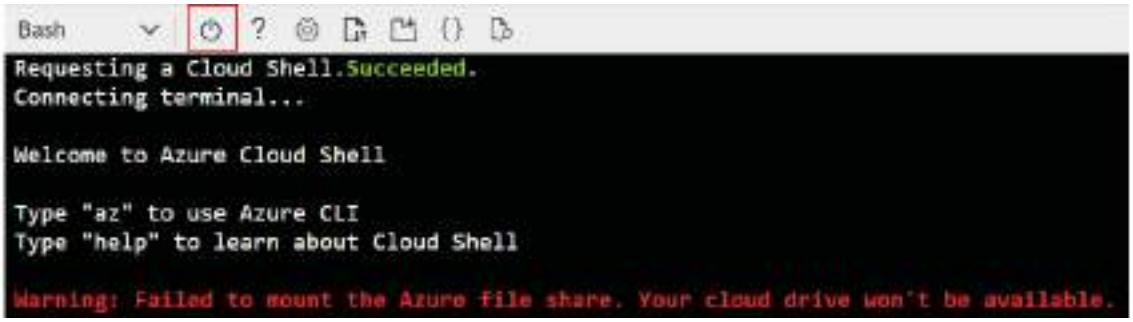


Figure 2.17: Hitting the restart button upon receiving a mount storage error

Click on the power button. It should restart, and you should see something similar to *Figure 2.18*:

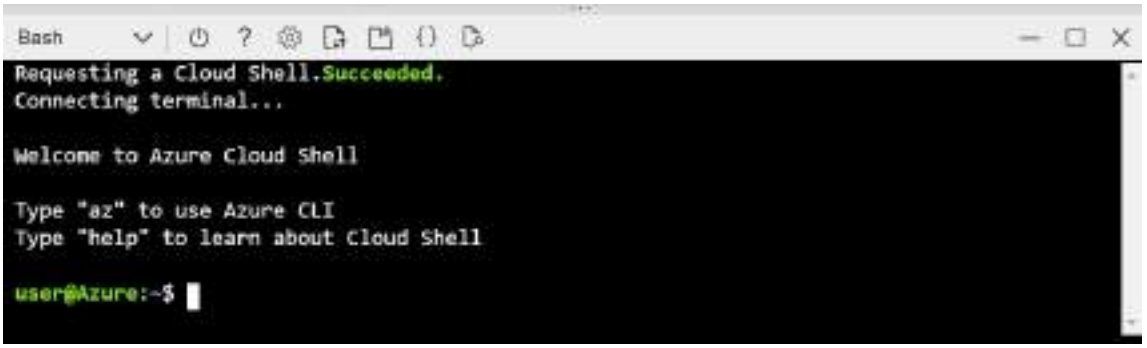


Figure 2.18: Launching Cloud Shell successfully

You can pull the splitter/divider up or down to see more or less of the shell:

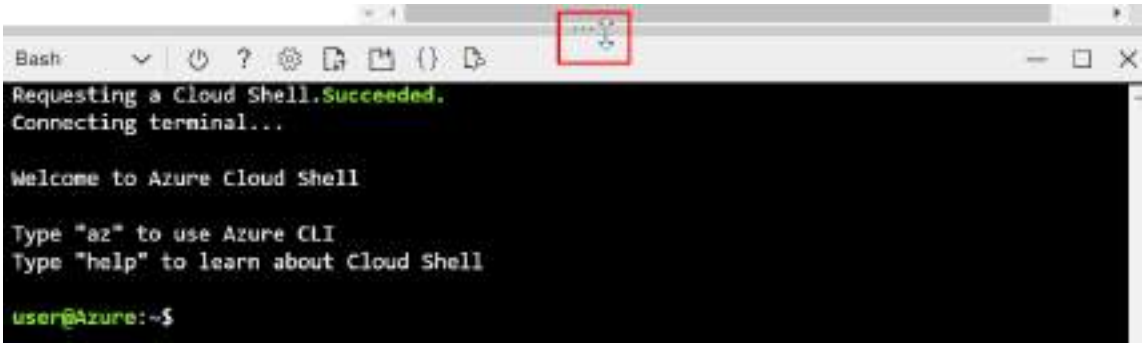


Figure 2.19: Using the divider to make Cloud Shell larger or smaller

The command-line tool that is used to interface with Kubernetes clusters is called `kubectl`. The benefit of using Azure Cloud Shell is that this tool, along with many others, comes preinstalled and is regularly maintained. `kubectl` uses a configuration file stored in `~/.kube/config` to store credentials to access your cluster.

> **Note**
>
> There is some discussion in the Kubernetes community around the correct pronunciation of `kubectl`. The common way to pronounce it is either *kube-c-t-l*, *kube-control*, or *kube-cuddle*.

To get the required credentials to access your cluster, you need to type the following command:

```
az aks get-credentials \
  --resource-group rg-handsonaks \
  --name handsonaks
```

> **Note**
>
> In this book, you will commonly see longer commands spread over multiple lines using the backslash symbol. This helps improve the readability of the commands, while still allowing you to copy and paste them. If you are typing these commands, you can safely ignore the backslash and type the full command in a single line.

To verify that you have access, type the following:

```
kubectl get nodes
```

You should see something like *Figure 2.20*:

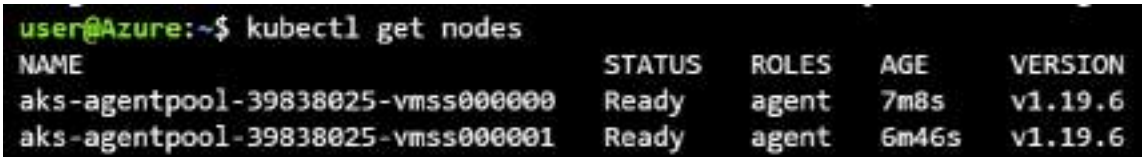


Figure 2.20: Output of the kubectl get nodes command

This command has verified that you can connect to your AKS cluster. In the next section, you'll go ahead and launch your first application.

Deploying and inspecting your first demo application

As you are all connected, let's launch your very first application. In this section, you will deploy your first application and inspect it using `kubectl` and later using the Azure portal. Let's start by deploying the application.

Deploying the demo application

In this section, you will deploy your demo application. For this, you will have to write a bit of code. In Cloud Shell, there are two options to edit code. You can do this either via command-line tools such as `vi` or `nano` or you can use a GUI-based code editor by typing the `code` commands in Cloud Shell. Throughout this book, you will mainly be instructed to use the graphical editor in the examples, but feel free to use any other tool you feel most comfortable with.

For the purpose of this book, all the code examples are hosted in a GitHub repository. You can clone this repository to your Cloud Shell and work with the code examples directly. To clone the GitHub repo into your Cloud Shell, use the following command:

```
git clone https://github.com/PacktPublishing/Hands-on-Kubernetes-on-Azure-Third-Edition.git Hands-On-Kubernetes-on-Azure
```

To access the code examples for this chapter, navigate into the directory of the code examples and go to the `Chapter02` directory:

```
cd Hands-On-Kubernetes-on-Azure/Chapter02/
```

You will use the code directly in the `Chapter02` folder for now. At this point in the book, you will not focus on what is in the code files just yet. The goal of this chapter is to launch a cluster and deploy an application on top of it. In the following chapters, we will dive into how Kubernetes configuration files are built and how you can create your own.

You will create an application based on the definition in the `azure-vote.yaml` file. To open that file in Cloud Shell, you can type the following command:

```
code azure-vote.yaml
```

Here is the code example for your convenience:

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: azure-vote-back
5   spec:
6     replicas: 1
7     selector:
```

```
8         matchLabels:
9           app: azure-vote-back
10    template:
11        metadata:
12          labels:
13               app: azure-vote-back
14        spec:
15          containers:
16          - name: azure-vote-back
17               image: redis
18               resources:
19                 requests:
20                      cpu: 100m
21                      memory: 128Mi
22                 limits:
23                      cpu: 250m
24                      memory: 256Mi
25               ports:
26               - containerPort: 6379
27                 name: redis
28  ---
29  apiVersion: v1
30  kind: Service
31  metadata:
32    name: azure-vote-back
33  spec:
34    ports:
35    - port: 6379
36    selector:
37         app: azure-vote-back
38  ---
39  apiVersion: apps/v1
40  kind: Deployment
41  metadata:
42    name: azure-vote-front
43  spec:
44    replicas: 1
45    selector:
46         matchLabels:
47           app: azure-vote-front
48    template:
49         metadata:
50           labels:
```

```
51              app: azure-vote-front
52        spec:
53          containers:
54          - name: azure-vote-front
55              image: microsoft/azure-vote-front:v1
56              resources:
57                requests:
58                    cpu: 100m
59                    memory: 128Mi
60                limits:
61                    cpu: 250m
62                    memory: 256Mi
63              ports:
64              - containerPort: 80
65              env:
66              - name: REDIS
67                value: "azure-vote-back"
68  ---
69  apiVersion: v1
70  kind: Service
71  metadata:
72    name: azure-vote-front
73  spec:
74    type: LoadBalancer
75    ports:
76    - port: 80
77    selector:
78        app: azure-vote-front
```

You can make changes to files in the Cloud Shell code editor. If you've made changes, you can save them by clicking on the **...** icon in the upper-right corner, and then click **Save** to save the file as highlighted in *Figure 2.21*:

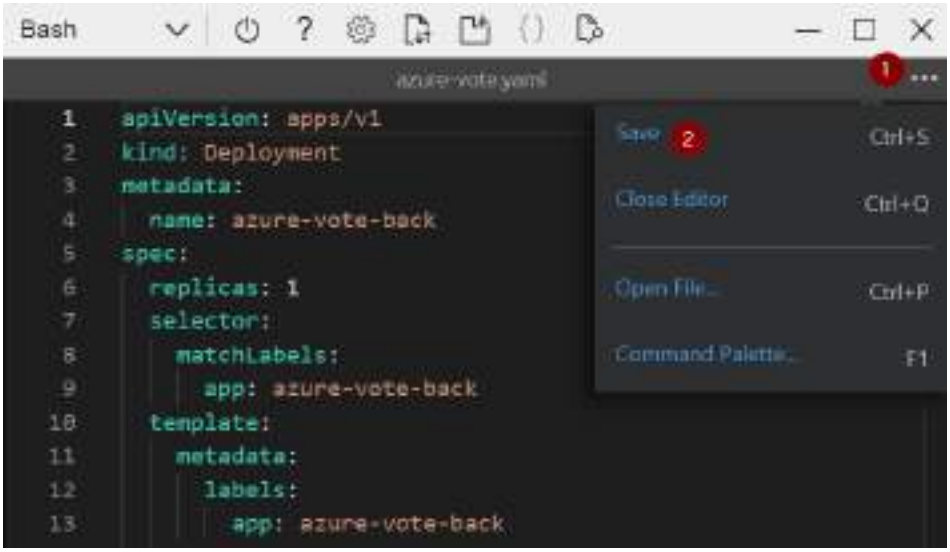


Figure 2.21: Save the azure-vote.yaml file

The file should be saved. You can check this with the following command:

```
cat azure-vote.yaml
```

> **Note:**
>
> Hitting the *Tab* button expands the file name in Linux. In the preceding scenario, if you hit *Tab* after typing `az`, it should expand to `azure-vote.yaml`.

Now, let's launch the application:

```
kubectl create -f azure-vote.yaml
```

You should quickly see the output as shown in *Figure 2.22*, it tells you which resources have been created:

```
deployment.apps/azure-vote-back created
service/azure-vote-back created
deployment.apps/azure-vote-front created
service/azure-vote-front created
```

Figure 2.22: Output of the kubectl create command

You have successfully created your demo application. In the next section, you will inspect all the different objects Kubernetes created for this application and connect to your application.

Exploring the demo application

In the previous section, you deployed a demo application. In this section, you will explore the different objects that Kubernetes created for this application and connect to it.

You can check the progress of the deployment by typing the following command:

```
kubectl get pods
```

If you typed this soon after creating the application, you might have seen that a certain pod was still in the `ContainerCreating` process:

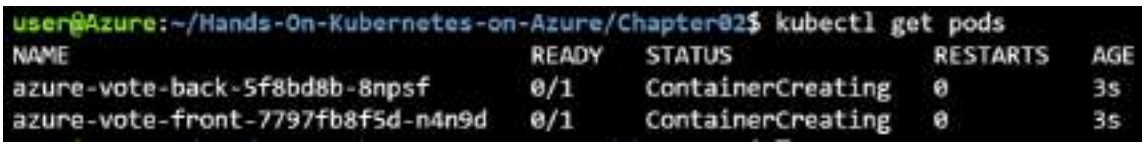


Figure 2.23: Output of the kubectl get pods command

> **Note**
>
> Typing `kubectl` can become tedious. You can use the `alias` command to make your life easier. You can use `k` instead of `kubectl` as the alias with the following command: `alias k=kubectl`. After running the preceding command, you can just use `k get pods`. For instructional purposes in this book, we will continue to use the full `kubectl` command.

Hit the *up arrow* key and press *Enter* to repeat the `kubectl get pods` command until the status of all pods is `Running`. Setting up all the pods takes some time, and you could optionally follow their status using the following command:

```
kubectl get pods --watch
```

To stop following the status of the pods (when they are all in a running state), you can press *Ctrl + C*.

In order to access your application publicly, you need one more thing. You need to know the public IP of the load balancer so that you can access it. If you remember from *Chapter 1*, *Introduction to containers and Kubernetes*, a service in Kubernetes will create an Azure load balancer. This load balancer will get a public IP in your application so you can access it publicly.

Type the following command to get the public IP of the load balancer:

```
kubectl get service azure-vote-front --watch
```

At first, the external IP might show `pending`. Wait for the public IP to appear and then press *Ctrl* + *C* to exit:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter02$ kubectl get service azure-vote-front --watch
NAME               TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
azure-vote-front   LoadBalancer   10.0.220.156   <pending>     80:32248/TCP   3s
azure-vote-front   LoadBalancer   10.0.220.156   20.72.205.222   80:32248/TCP   24s
```

Figure 2.24: Watching the service IP change from pending to the actual IP address

Note the external IP address and type it in a browser. You should see an output similar to *Figure 2.25*:

Figure 2.25: The actual application you just launched

Click on **Cats** or **Dogs** and watch the count go up.

To see all the objects in Kubernetes that were created for your application, you can use the `kubectl get all` command. This will show an output similar to *Figure 2.26*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter02$ kubectl get all
NAME                                    READY   STATUS    RESTARTS   AGE
pod/azure-vote-back-5f8bd8b-4wblf       1/1     Running   0          5m54s
pod/azure-vote-front-7797fb8f5d-f2q7t   1/1     Running   0          5m54s

NAME                         TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
service/azure-vote-back      ClusterIP      10.0.221.93    <none>          6379/TCP       5m54s
service/azure-vote-front     LoadBalancer   10.0.220.156   20.72.205.222   80:32248/TCP   5m54s
service/kubernetes           ClusterIP      10.0.0.1       <none>          443/TCP        13h

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/azure-vote-back     1/1     1            1           5m54s
deployment.apps/azure-vote-front    1/1     1            1           5m54s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/azure-vote-back-5f8bd8b       1         1         1       5m54s
replicaset.apps/azure-vote-front-7797fb8f5d   1         1         1       5m54s
```

Figure 2.26: Exploring all the Kubernetes objects created for your application

As you can see, a number of objects were created:

- Pods: You will see two pods, one for the back end and one for the front end.
- Services: You will also see two services, one for the back end of type `ClusterIP` and one for the front end of type LoadBalancer. What these types mean will be explored in *Chapter 3*, *Application deployment on AKS*.
- Deployments: You will also see two deployments.
- ReplicaSets: And finally you'll see two ReplicaSets.

You can also view these objects from the Azure portal. To see, for example, the two deployments, you can click on **Workloads** in the left-hand navigation menu of the AKS pane, and you will see all the deployments in your cluster as shown in *Figure 2.27*. This figure shows you all the deployments in your cluster, including the system deployments. At the bottom of the list, you can see your own deployments. As you can also see in this figure, you can explore other objects such as pods and ReplicaSets using the top menu:

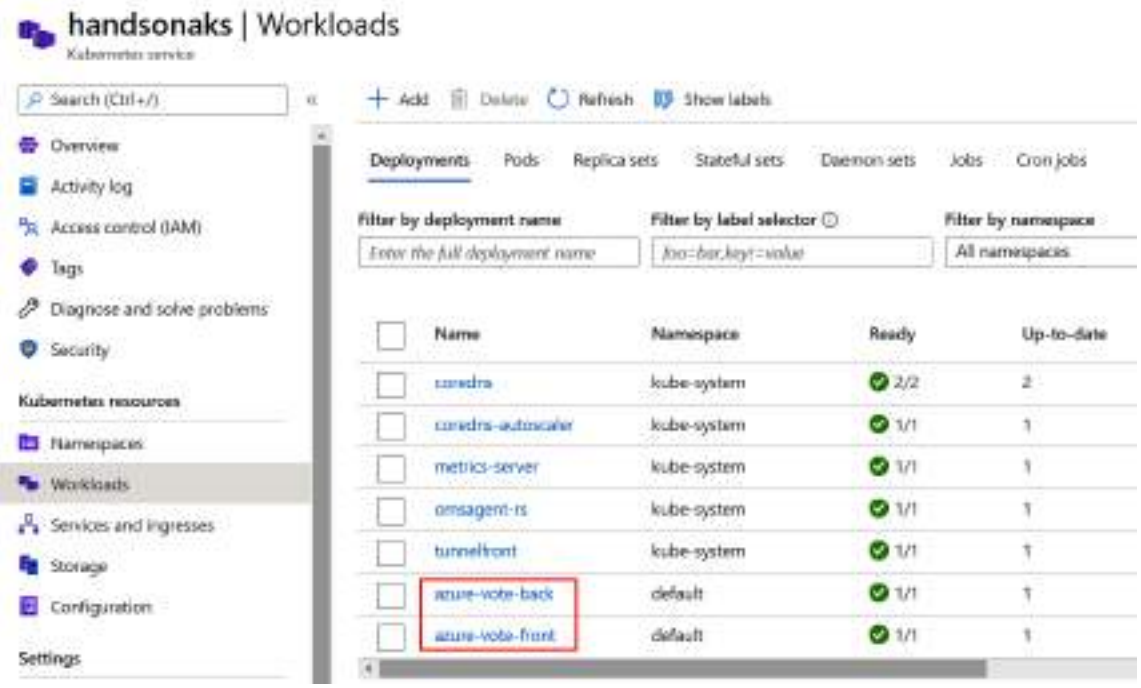


Figure 2.27: Exploring the two deployments part of your application in the Azure portal

You have now launched your own cluster and your first Kubernetes application. Note that Kubernetes took care of tasks such as connecting the front end and the back end, and exposing them to the outside world, as well as providing storage for the services.

Before moving on to the next chapter, let's clean up your deployment. Since you created everything from a file, you can also delete everything by pointing Kubernetes to that file. Type `kubectl delete -f azure-vote.yaml` and watch all your objects get deleted:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter02$ kubectl delete -f azure-vote.yaml
deployment.apps "azure-vote-back" deleted
service "azure-vote-back" deleted
deployment.apps "azure-vote-front" deleted
service "azure-vote-front" deleted
```

Figure 2.28: Cleaning up the application

In this section, you have connected to your AKS cluster using Cloud Shell, successfully launched and connected to a demo application, explored the objects created using Cloud Shell and the Azure portal, and finally, cleaned up the resources that were created.

Summary

Having completed this chapter, you will now be able to access and navigate the Azure portal to perform all the functions required to deploy an AKS cluster. We used the free trial on Azure to our advantage to learn the ins and outs of AKS. We also launched our own AKS cluster with the ability to customize configurations if required using the Azure portal.

We also used Cloud Shell without installing anything on the computer. This is important for all the upcoming sections, where you will be doing more than just launching simple applications. Finally, we launched a publicly accessible service. The skeleton of this application is the same as for complex applications that we will cover in the later chapters.

In the next chapter, we will take an in-depth look at different deployment options to deploy applications onto AKS.

Section 2: Deploying on AKS

At this point in the book, you have learned the basics of containers and Kubernetes and set up a Kubernetes cluster on Azure. In this section, you will learn how to deploy applications on top of that Kubernetes cluster.

Throughout this section, you will progressively build and deploy different applications on top of AKS. You will start by deploying a simple application, and later introduce concepts such as scaling, monitoring, and authentication. By the end of the section, you should feel comfortable deploying applications to AKS.

This section contains the following chapters:

- *Chapter 3, Application deployment on AKS*
- *Chapter 4, Building scalable applications*
- *Chapter 5, Handling common failures in AKS*
- *Chapter 6, Securing your application with HTTPS*
- *Chapter 7, Monitoring the AKS cluster and the application*

Let's start this section by exploring application deployment on AKS in *Chapter 3, Application deployment on AKS*.

3

Application deployment on AKS

In this chapter, you will deploy two applications on **Azure Kubernetes Service** (**AKS**). An application consists of multiple parts, and you will build the applications one step at a time while the conceptual model behind them is explained. You will be able to easily adapt the steps in this chapter to deploy any other application on AKS.

To deploy the applications and make changes to them, you will be using **YAML** files. YAML is a recursive acronym for **YAML Ain't Markup Language**. YAML is a language that is used to create configuration files to deploy to Kubernetes. Although you can use either JSON or YAML files to deploy applications to Kubernetes, YAML is the most commonly used language to do so. YAML became popular because it is easier for a human to read when compared to JSON or XML. You will see multiple examples of YAML files throughout this chapter and throughout the book.

During the deployment of the sample guestbook application, you will see Kubernetes concepts in action. You will see how a **deployment** is linked to a **ReplicaSet**, and how that is linked to the **pods** that are deployed. A deployment is an object in Kubernetes that is used to define the desired state of an application. A **deployment** will create a ReplicaSet. A **ReplicaSet** is an object in Kubernetes that guarantees that a certain number of **pods** will always be available. Hence, a ReplicaSet will create one or more pods. A pod is an object in Kubernetes that is a group of one or more containers. Let's revisit the relationship between deployments, ReplicaSets, and pods:

Figure 3.1: Relationship between a deployment, a ReplicaSet, and pods

While deploying the sample applications, you will use the **service** object to connect to the application. A service in Kubernetes is an object that is used to provide a static IP address and DNS name to an application. Since a pod can be killed and moved to different nodes in the cluster, a service ensures you can connect to a static endpoint for your application.

You will also edit the sample applications to provide configuration details using a **ConfigMap**. A ConfigMap is an object that is used to provide configuration details to pods. It allows you to keep configuration settings outside of the actual container. You can then provide these configuration details to your application by connecting the ConfigMap to your deployment.

Finally, you will be introduced to Helm. Helm is a package manager for Kubernetes that helps to streamline the deployment process. You will deploy a WordPress site using Helm and gain an understanding of the value Helm brings to Kubernetes. This WordPress installation makes use of persistent storage in Kubernetes and you will learn how persistent storage in AKS is set up.

The following topics will be covered in this chapter:

- Deploying the sample guestbook application step by step
- Full deployment of the sample guestbook application
- Using Helm to install complex Kubernetes applications

We'll begin with the sample guestbook application.

Deploying the sample guestbook application step by step

In this chapter, you will deploy the classic guestbook sample Kubernetes application. You will be mostly following the steps from https://kubernetes.io/docs/tutorials/stateless-application/guestbook/ with some modifications. You will employ these modifications to show additional concepts, such as ConfigMaps, that are not present in the original sample.

The sample guestbook application is a simple, multi-tier web application. The different tiers in this application will have multiple instances. This is beneficial for both high availability and scalability. The guestbook's front end is a stateless application because the front end doesn't store any state. The Redis cluster in the back end is stateful as it stores all the guestbook entries.

You will be using this application as the basis for testing out the scaling of the back end and the front end, independently, in the next chapter.

Before we get started, let's consider the application that we'll be deploying.

Introducing the application

The application stores and displays guestbook entries. You can use it to record the opinion of all the people who visit your hotel or restaurant, for example.

Figure 3.2 shows you a high-level overview of the application. The application uses PHP as a front end. The front end will be deployed using multiple replicas. The application uses Redis for its data storage. Redis is an in-memory key-value database. Redis is most often used as a cache.

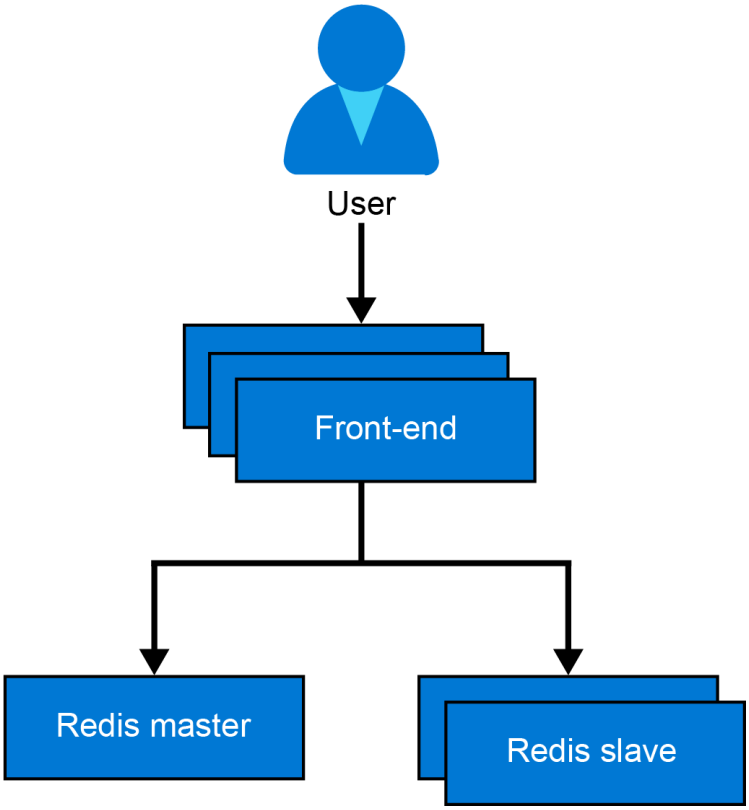


Figure 3.2: High-level overview of the guestbook application

We will begin deploying this application by deploying the Redis master.

Deploying the Redis master

In this section, you are going to deploy the Redis master. You will learn about the YAML syntax that is required for this deployment. In the next section, you will make changes to this YAML. Before making changes, let's start by deploying the Redis master.

Perform the following steps to complete the task:

1. Open your friendly Azure Cloud Shell, as highlighted in *Figure* 3.3:

Figure 3.3: Opening the Cloud Shell

2. If you have not cloned the GitHub repository for this book, please do so now by using the following command:

```
git clone https://github.com/PacktPublishing/Hands-on-Kubernetes-on-Azure-Third-Edition/
```

3. Change into the directory for Chapter 3 using the following command:

```
cd Hands-On-Kubernetes-on-Azure/Chapter03/
```

4. Enter the following command to deploy the master:

```
kubectl apply -f redis-master-deployment.yaml
```

It will take some time for the application to download and start running. While you wait, let's understand the command you just typed and executed. Let's start by exploring the content of the YAML file that was used (the line numbers are used for explaining key elements from the code snippets):

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: redis-master
5     labels:
6       app: redis
7   spec:
8     selector:
9       matchLabels:
10        app: redis
11        role: master
12        tier: backend
13    replicas: 1
14    template:
15      metadata:
16        labels:
17          app: redis
18          role: master
19          tier: backend
20      spec:
21        containers:
22        - name: master
```

```
23          image: k8s.gcr.io/redis:e2e
24          resources:
25            requests:
26              cpu: 100m
27              memory: 100Mi
28            limits:
29              cpu: 250m
30              memory: 1024Mi
31          ports:
32          - containerPort: 6379
```

Let's dive deeper into the code line by line to understand the provided parameters:

- **Line 2**: This states that we are creating a deployment. As explained in *Chapter 1*, *Introduction to containers and Kubernetes*, a deployment is a wrapper around pods that makes it easy to update and scale pods.
- **Lines 4-6**: Here, the `deployment` is given a name, which is `redis-master`.
- **Lines 7-12**: These lines let us specify the containers that this deployment will manage. In this example, the deployment will select and manage all containers for which labels match (`app: redis, role: master, and tier: backend`). The preceding label exactly matches the labels provided in lines 14-19.
- **Line 13**: This line tells Kubernetes that we need exactly one copy of the running Redis master. This is a key aspect of the declarative nature of Kubernetes. You provide a description of the containers your applications need to run (in this case, only one replica of the Redis master), and Kubernetes takes care of it.
- **Line 14-19**: These lines add labels to the running instance so that it can be grouped and connected to other pods. We will discuss them later to see how they are used.
- **Line 22**: This line gives the single container in the pod a name, which is `master`. In the case of a multi-container pod, each container in a pod requires a unique name.

- **Line 23**: This line indicates the container image that will be run. In this case, it is the `redis` image tagged with `e2e` (the latest Redis image that successfully passed its end-to-end [`e2e`] tests).
- **Lines 24-30**: These lines set the `cpu/memory` resources requested for the container. A request in Kubernetes is a reservation of resources that cannot be used by other pods. If those resources are not available in the cluster, the pod will not start. In this case, the request is 0.1 CPU, which is equal to `100m` and is also often referred to as 100 millicores. The memory requested is `100Mi`, or 104,857,600 bytes, which is equal to ~105 MB. CPU and memory limits are set in a similar way. Limits are caps on what a container can use. If your pod hits the CPU limit, it'll get throttled, whereas if it hits the memory limits, it'll get restarted. Setting requests and limits is a best practice in Kubernetes. For more info, refer to https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/.
- **Lines 31-32**: These two lines indicate that the container is going to listen on port `6379`.

As you can see, the YAML definition for the deployment contains several settings and parameters that Kubernetes will use to deploy and configure your application.

> **Note**
>
> The Kubernetes YAML definition is similar to the arguments given to Docker to run a particular container image. If you had to run this manually, you would define this example in the following way:
>
> ```
> # Run a container named master, listening on port 6379, with 100M memory and 100m CPU using the redis:e2e image.
> docker run --name master -p 6379:6379 -m 100M -c 100m -d k8s.gcr.io/redis:e2e
> ```

In this section, you have deployed the Redis master and learned about the syntax of the YAML file that was used to create this deployment. In the next section, you will examine the deployment and learn about the different elements that were created.

Examining the deployment

The `redis-master` deployment should be complete by now. Continue in the Azure Cloud Shell that you opened in the previous section and type the following:

```
kubectl get all
```

You should get an output similar to the one displayed in *Figure* 3.4. In your case, the name of the pod and the ReplicaSet might contain different IDs at the end of the name. If you do not see a pod, a deployment, and a ReplicaSet, please run the code as explained in step 4 in the previous section again.

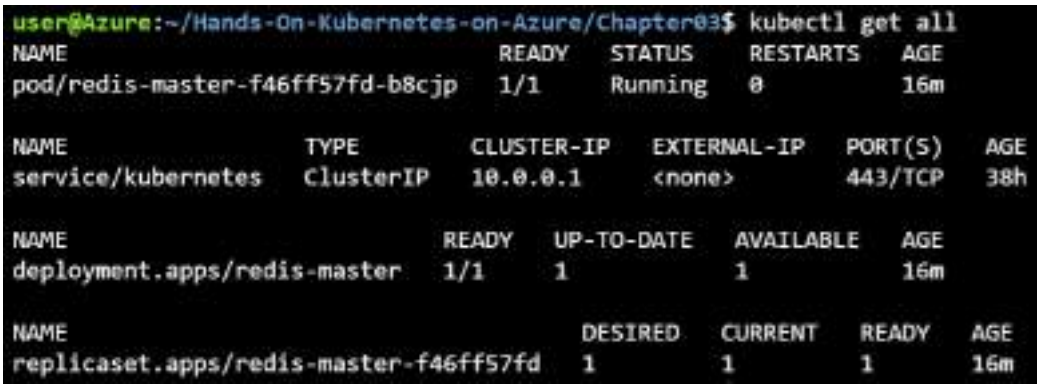


Figure 3.4: Objects that were created by your deployment

You can see that you created a deployment named `redis-master`. It controls a ReplicaSet named `redis-master-f46ff57fd`. On further examination, you will also find that the ReplicaSet is controlling a pod, `redis- master-f46ff57fd-b8cjp`. *Figure 3.1* has a graphical representation of this relationship.

More details can be obtained by executing the `kubectl describe <object> <instance-name>` command, as follows:

```
kubectl describe deployment/redis-master
```

This will generate an output as follows:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl describe deployment/redis-master
Name:                   redis-master
Namespace:              default
CreationTimestamp:      Sun, 17 Jan 2021 16:42:15 +0000
Labels:                 app=redis
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=redis,role=master,tier=backend
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=redis
           role=master
           tier=backend
  Containers:
   master:
    Image:      k8s.gcr.io/redis:e2e
    Port:       6379/TCP
    Host Port:  0/TCP
    Requests:
      cpu:        100m
      memory:     100Mi
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   redis-master-f46ff57fd (1/1 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  18m   deployment-controller  Scaled up replica set redis-master-f46ff57fd to 1
```

Figure 3.5: Description of the deployment

You have now launched a Redis master with the default configuration. Typically, you would launch an application with an environment-specific configuration.

In the next section, you will get acquainted with a new concept called ConfigMaps and then recreate the Redis master. So, before proceeding, clean up the current version, which you can do by running the following command:

```
kubectl delete deployment/redis-master
```

Executing this command will produce the following output:

```
deployment.apps "redis-master" deleted
```

In this section, you examined the Redis master deployment you created. You saw how a deployment relates to a ReplicaSet and how a ReplicaSet relates to pods. In the following section, you will recreate this Redis master with an environment-specific configuration provided via a ConfigMap.

Redis master with a ConfigMap

There was nothing wrong with the previous deployment. In practical use cases, it would be rare that you would launch an application without some configuration settings. In this case, you are going to set the configuration settings for `redis-master` using a ConfigMap.

A ConfigMap is a portable way of configuring containers without having specialized images for each environment. It has a key-value pair for data that needs to be set on a container. A ConfigMap is used for non-sensitive configuration. Kubernetes has a separate object called a **Secret**. A Secret is used for configurations that contain critical data such as passwords. This will be explored in detail in *Chapter 10, Storing Secrets in AKS* of this book.

In this example, you are going to create a ConfigMap. In this ConfigMap, you will configure `redis-config` as the key and the value will be the following two lines:

```
maxmemory 2mb
maxmemory-policy allkeys-lru
```

Now, let's create this ConfigMap. There are two ways to create a ConfigMap:

- Creating a ConfigMap from a file
- Creating a ConfigMap from a YAML file

In the following two sections, you'll explore both.

Creating a ConfigMap from a file

The following steps will help us create a ConfigMap from a file:

1. Open the Azure Cloud Shell code editor by typing `code redis-config` in the terminal. Copy and paste the following two lines and save the file as `redis-config`:

```
maxmemory 2mb
maxmemory-policy allkeys-lru
```

2. Now you can create the ConfigMap using the following code:

```
kubectl create configmap \
  example-redis-config --from-file=redis-config
```

   You should get an output as follows:

```
configmap/example-redis-config created
```

3. You can use the same command to describe this ConfigMap:

```
kubectl describe configmap/example-redis-config
```

   The output will be as shown in *Figure* 3.6:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl describe configmap/example-redis-config
Name:         example-redis-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
redis-config:
----
maxmemory 2mb
maxmemory-policy allkeys-lru
Events:  <none>
```

Figure 3.6: Description of the ConfigMap

In this example, you created the ConfigMap by referring to a file on disk. A different way to deploy ConfigMaps is by creating them from a YAML file. Let's have a look at how this can be done in the following section.

Creating a ConfigMap from a YAML file

In this section, you will recreate the ConfigMap from the previous section using a YAML file:

1. To start, delete the previously created ConfigMap:

```
kubectl delete configmap/example-redis-config
```

2. Copy and paste the following lines into a file named `example-redis-config.yaml`, and then save the file:

```
1  apiVersion: v1
2  data:
3    redis-config: |-
4      maxmemory 2mb
5      maxmemory-policy allkeys-lru
6  kind: ConfigMap
7  metadata:
8    name: example-redis-config
```

3. You can now create your ConfigMap via the following command:

```
kubectl create -f example-redis-config.yaml
```

   You should get an output as follows:

```
configmap/example-redis-config created
```

4. Next, run the following command:

```
kubectl describe configmap/example-redis-config
```

   This command returns the same output as the previous one, as shown in *Figure 3.6*.

As you can see, using a YAML file, you were able to create the same ConfigMap.

> **Note**
>
> `kubectl get` has the useful `-o` option, which can be used to get the output of an object in either YAML or JSON. This is very useful in cases where you have made manual changes to a system and want to see the resulting object in YAML format. You can get the current ConfigMap in YAML using the following command:
> `kubectl get -o yaml configmap/example-redis-config`

Now that you have the ConfigMap defined, let's use it.

Using a ConfigMap to read in configuration data

In this section, you will reconfigure the `redis-master` deployment to read configuration from the ConfigMap:

1. To start, modify `redis-master-deployment.yaml` to use the ConfigMap as follows. The changes you need to make will be explained after the source code:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: redis-master
5    labels:
6      app: redis
7  spec:
8    selector:
9      matchLabels:
10       app: redis
11       role: master
12       tier: backend
13   replicas: 1
14   template:
15     metadata:
```

```
16      labels:
17        app: redis
18        role: master
19        tier: backend
20    spec:
21      containers:
22      - name: master
23        image: k8s.gcr.io/redis:e2e
24        command:
25        - redis-server
26        - "/redis-master/redis.conf"
27        env:
28        - name: MASTER
29          value: "true"
30        volumeMounts:
31        - mountPath: /redis-master
32          name: config
33        resources:
34          requests:
35            cpu: 100m
36            memory: 100Mi
37        ports:
38        - containerPort: 6379
39      volumes:
40        - name: config
41          configMap:
42            name: example-redis-config
43            items:
44            - key: redis-config
45              path: redis.conf
```

> **Note**
>
> If you downloaded the source code accompanying this book, there is a file in *Chapter 3, Application deployment on AKS*, called `redis-master-deployment_Modified.yaml`, that has the necessary changes applied to it.

Let's dive deeper into the code to understand the different sections:

- **Lines 24-26**: These lines introduce a command that will be executed when your pod starts. In this case, this will start the `redis-server` pointing to a specific configuration file.
- **Lines 27-29**: These lines show how to pass configuration data to your running container. This method uses environment variables. In Docker form, this would be equivalent to `docker run -e "MASTER=true". --name master -p 6379:6379 -m 100M -c 100m -d Kubernetes / redis:v1`. This sets the environment variable `MASTER` to `true`.
  Your application can read the environment variable settings for its configuration.
- **Lines 30-32**: These lines mount the volume called `config` (this volume is defined in lines 39-45) on the `/redis-master` path on the running container. It will hide whatever exists on `/redis-master` on the original container.
- In Docker terms, it would be equivalent to `docker run -v config:/ redis-master. -e "MASTER=TRUE" --name master -p 6379:6379 -m 100M -c 100m -d Kubernetes /redis:v1`.
- **Line 40**: This gives the volume the name `config`. This name will be used within the context of this pod.
- **Lines 41-42**: This declares that this volume should be loaded from the `example-redis-config` ConfigMap. This ConfigMap should already exist in the system. You have already defined this, so you are good.
- **Lines 43-45**: Here, you are loading the value of the `redis-config` key (the two-line `maxmemory` settings) as a `redis.conf` file.

By adding the ConfigMap as a volume and mounting the volume, you are able to load dynamic configuration.

1. Let's create this updated deployment:

   ```
   kubectl create -f redis-master-deployment_Modified.yaml
   ```

   This should output the following:

   ```
   deployment.apps/redis-master created
   ```

2. Let's now make sure that the configuration was successfully applied. First, get the pod's name:

   ```
   kubectl get pods
   ```

   This should return an output similar to *Figure* 3.7:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
redis-master-7f8dc96bd7-tdp75  1/1     Running   0          29s
```

Figure 3.7: Details of the pod

3. Then `exec` into the pod and verify that the settings were applied:

   ```
   kubectl exec -it redis-master-<pod-id> -- redis-cli
   ```

   This open a `redis-cli` session with the running pod. Now you can get the `maxmemory` configuration:

   ```
   CONFIG GET maxmemory
   ```

   And then you can get the `maxmemory-policy` configuration:

   ```
   CONFIG GET maxmemory-policy
   ```

   This should give you an output similar to *Figure* 3.8:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl exec -it redis-master-766c5cf9c8-pw7qg -- redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
127.0.0.1:6379> exit
```

Figure 3.8: Verifying the Redis configuration in the pod

4. To leave the Redis shell, type the `exit` command.

To summarize, you have just performed an important part of configuring cloud-native applications, namely providing dynamic configuration data to an application. You will have also noticed that the apps have to be configured to read config dynamically. After you set up your app with configuration, you accessed a running container to verify the running configuration. You will use this methodology frequently throughout this book to verify the functionality of running applications.

> **Note**
>
> Connecting to a running container by using the `kubectl exec` command is useful for troubleshooting and doing diagnostics. Due to the ephemeral nature of containers, you should never connect to a container to do additional configuration or installation. This should either be part of your container image or configuration you provide via Kubernetes (as you just did).

In this section, you configured the Redis master to load configuration data from a ConfigMap. In the next section, we will deploy the end-to-end application.

Complete deployment of the sample guestbook application

Having taken a detour to understand the dynamic configuration of applications using a ConfigMap, you will now return to the deployment of the rest of the guestbook application. You will once again come across the concepts of deployment, ReplicaSets, and pods. Apart from this, you will also be introduced to another key concept, called a service.

To start the complete deployment, we are going to create a service to expose the Redis master service.

Exposing the Redis master service

When exposing a port in plain Docker, the exposed port is constrained to the host it is running on. With Kubernetes networking, there is network connectivity between different pods in the cluster. However, pods themselves are ephemeral in nature, meaning they can be shut down, restarted, or even moved to other hosts without maintaining their IP address. If you were to connect to the IP of a pod directly, you might lose connectivity if that pod was moved to a new host.

Kubernetes provides the `service` object, which handles this exact problem. Using label-matching selectors, it sends traffic to the right pods. If there are multiple pods serving traffic to a service, it will also do load balancing. In this case, the master has only one pod, so it just ensures that the traffic is directed to the pod independent of the node the pod runs on. To create the service, run the following command:

```
kubectl apply -f redis-master-service.yaml
```

The `redis-master-service.yaml` file has the following content:

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: redis-master
5     labels:
6       app: redis
7       role: master
8       tier: backend
9   spec:
10   ports:
11   - port: 6379
12     targetPort: 6379
13    selector:
14      app: redis
15      role: master
16      tier: backend
```

Let's now see what you have created using the preceding code:

- **Lines 1-8**: These lines tell Kubernetes that we want a service called `redis-master`, which has the same labels as our `redis-master` server pod.
- **Lines 10-12**: These lines indicate that the service should handle traffic arriving at port `6379` and forward it to port `6379` of the pods that match the selector defined between lines 13 and 16.
- **Lines 13-16**: These lines are used to find the pods to which the incoming traffic needs to be sent. So, any pod with labels matching (`app: redis`, `role: master` and `tier: backend`) is expected to handle port `6379` traffic. If you look back at the previous example, those are the exact labels we applied to that deployment.

You can check the properties of the service by running the following command:

```
kubectl get service
```

This will give you an output as shown in *Figure* 3.9:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl create -f redis-master-service.yaml
service/redis-master created
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get service
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
kubernetes     ClusterIP   10.0.0.1       <none>        443/TCP    43h
redis-master   ClusterIP   10.0.106.207   <none>        6379/TCP   7s
```

Figure 3.9: Properties of the created service

You see that a new service, named `redis-master`, has been created. It has a Cluster-IP of `10.0.106.207` (in your case, the IP will likely be different). Note that this IP will work only within the cluster (hence the `ClusterIP` type).

> **Note**
>
> You are now creating a service of type `ClusterIP`. There are other types of service as well, which will be introduced later in this chapter.

A service also introduces a **Domain Name Server** (**DNS**) name for that service. The DNS name is of the form `<service-name>.<namespace>.svc.cluster.local`; in this case, it would be `redis-master.default.svc.cluster.local`. To see this in action, we'll do a name resolution on our `redis-master` pod. The default image doesn't have `nslookup` installed, so we'll bypass that by running a `ping` command. Don't worry if that traffic doesn't return; this is because you didn't expose `ping` on your service, only the `redis` port. The command is, however, useful to see the full DNS name and the name resolution work. Let's have a look:

```
kubectl get pods
#note the name of your redis-master pod
kubectl exec -it redis-master-<pod-id> -- bash
ping redis-master
```

This should output the resulting name resolution, showing you the **Fully Qualified Domain Name** (**FQDN**) of your service and the IP address that showed up earlier. You can stop the ping command from running by pressing *Ctrl+C*. You can exit the pod via the `exit` command, as shown in *Figure 3.10*:

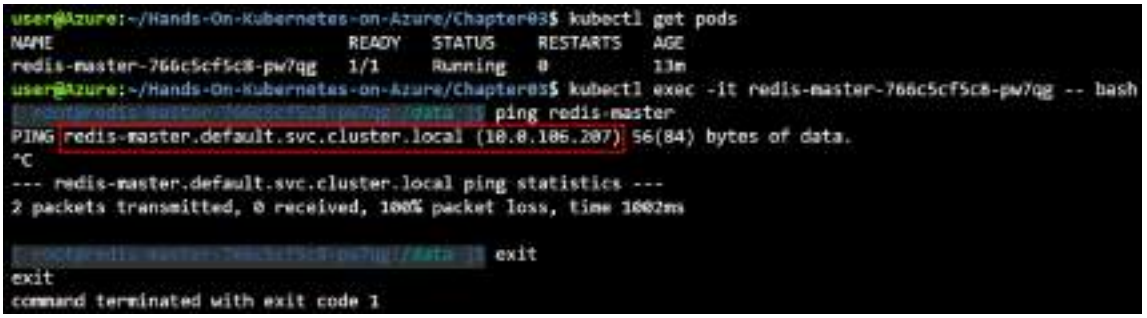


Figure 3.10: Using a ping command to view the FQDN of your service

In this section, you exposed the Redis master using a service. This ensures that even if a pod moves to a different host, it can be reached through the service's IP address. In the next section, you will deploy the Redis replicas, which help to handle more read traffic.

Deploying the Redis replicas

Running a single back end on the cloud is not recommended. You can configure Redis in a leader-follower (master-slave) setup. This means that you can have a master that will serve write traffic and multiple replicas that can handle read traffic. It is useful for handling increased read traffic and high availability.

Let's set this up:

1. Create the deployment by running the following command:

```
kubectl apply -f redis-replica-deployment.yaml
```

2. Let's check all the resources that have been created now:

```
kubectl get all
```

The output would be as shown in *Figure 3.11*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get all
NAME                                 READY   STATUS    RESTARTS   AGE
pod/redis-master-766c5cf5c8-pw7qg    1/1     Running   0          32m
pod/redis-replica-57c8c66cc4-42hnk   1/1     Running   0          2m12s
pod/redis-replica-57c8c66cc4-dfvbv   1/1     Running   0          2m12s

NAME                    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/kubernetes      ClusterIP   10.0.0.1       <none>        443/TCP    43h
service/redis-master    ClusterIP   10.0.106.207   <none>        6379/TCP   21m

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/redis-master       1/1     1            1           32m
deployment.apps/redis-replica      2/2     2            2           2m12s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/redis-master-766c5cf5c8       1         1         1       32m
replicaset.apps/redis-replica-57c8c66cc4      2         2         2       2m12s
```

Figure 3.11: Deploying the Redis replicas creates a number of new objects

3. Based on the preceding output, you can see that you created two replicas of the `redis-replica` pods. This can be confirmed by examining the `redis-replica- deployment.yaml` file:

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: redis-replica
5     labels:
6       app: redis
7   spec:
8     selector:
9       matchLabels:
10       app: redis
11       role: replica
12       tier: backend
13    replicas: 2
14    template:
15      metadata:
16        labels:
17          app: redis
18          role: replica
19          tier: backend
20      spec:
21        containers:
22        - name: replica
23          image: gcr.io/google-samples/gb-redis-follower:v1
24          resources:
25            requests:
26              cpu: 100m
27              memory: 100Mi
28          env:
29          - name: GET_HOSTS_FROM
30            value: dns
31          ports:
32          - containerPort: 6379
```

Everything is the same except for the following:

- **Line 13**: The number of replicas is 2.
- **Line 23**: You are now using a specific replica (follower) image.
- **Lines 29-30**: Setting `GET_HOSTS_FROM` to `dns`. This is a setting that specifies that Redis should get the hostname of the master using DNS.

As you can see, this is similar to the Redis master you created earlier.

4. Like the master service, you need to expose the replica service by running the following:

```
kubectl apply -f redis-replica-service.yaml
```

The only difference between this service and the `redis-master` service is that this service proxies traffic to pods that have the `role:replica` label.

5. Check the `redis-replica` service by running the following command:

```
kubectl get service
```

This should give you the output shown in *Figure 3.12*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl create -f redis-replica-service.yaml
service/redis-replica created
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get service
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
kubernetes      ClusterIP   10.0.0.1       <none>        443/TCP    43h
redis-master    ClusterIP   10.0.106.207   <none>        6379/TCP   23m
redis-replica   ClusterIP   10.0.133.171   <none>        6379/TCP   5s
```

Figure 3.12: Redis-master and redis-replica service

You now have a Redis cluster up and running, with a single master and two replicas. In the next section, you will deploy and expose the front end.

Deploying and exposing the front end

Up to now, you have focused on the Redis back end. Now you are ready to deploy the front end. This will add a graphical web page to your application that you'll be able to interact with.

You can create the front end using the following command:

```
kubectl apply -f frontend-deployment.yaml
```

To verify the deployment, run this command:

```
kubectl get pods
```

This will display the output shown in *Figure* 3.13:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
frontend-6c6d6dfd4d-8dl5v         1/1     Running   0          54s
frontend-6c6d6dfd4d-gz59t         1/1     Running   0          55s
frontend-6c6d6dfd4d-mghz2         1/1     Running   0          54s
redis-master-766c5cf5c8-pw7qg     1/1     Running   0          37m
redis-replica-57c8c66cc4-42hnk    1/1     Running   0          6m27s
redis-replica-57c8c66cc4-dfvbv    1/1     Running   0          6m27s
```

Figure 3.13: Verifying the front end deployment

You will notice that this deployment specifies 3 replicas. The deployment has the usual aspects with minor changes, as shown in the following code:

```
1  apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: frontend
5     labels:
6       app: guestbook
7   spec:
8     selector:
9       matchLabels:
10        app: guestbook
11        tier: frontend
12    replicas: 3
13    template:
14      metadata:
15        labels:
16          app: guestbook
17          tier: frontend
18      spec:
19        containers:
20        - name: php-redis
21          image: gcr.io/google-samples/gb-frontend:v4
```

```
22          resources:
23            requests:
24              cpu: 100m
25              memory: 100Mi
26          env:
27          - name: GET_HOSTS_FROM
28            value: env
29          - name: REDIS_SLAVE_SERVICE_HOST
30            value: redis-replica
31          ports:
32          - containerPort: 80
```

Let's see these changes:

- **Line 11**: The replica count is set to 3.
- **Line 8-10 and 14-16**: The labels are set to `app: guestbook` and `tier: frontend`.
- **Line 20**: `gb-frontend:v4` is used as the image.

You have now created the front-end deployment. You now need to expose it as a service.

Exposing the front-end service

There are multiple ways to define a Kubernetes service. The two Redis services we created were of the type `ClusterIP`. This means they are exposed on an IP that is reachable only from the cluster, as shown in *Figure 3.14*:

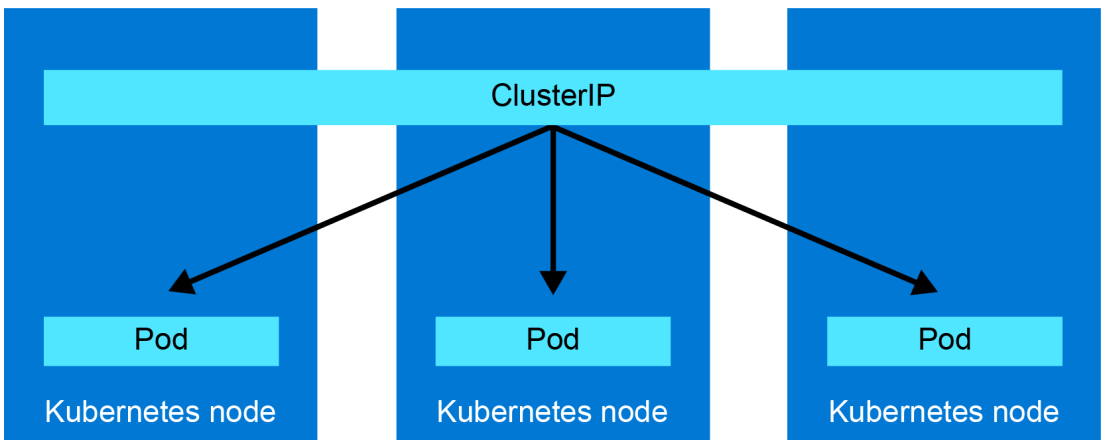


Figure 3.14: Kubernetes service of type ClusterIP

Another type of service is the type `NodePort`. A service of type NodePort is accessible from outside the cluster, by connecting to the IP of a node and the specified port. This service is exposed on a static port on each node as shown in *Figure 3.15*:

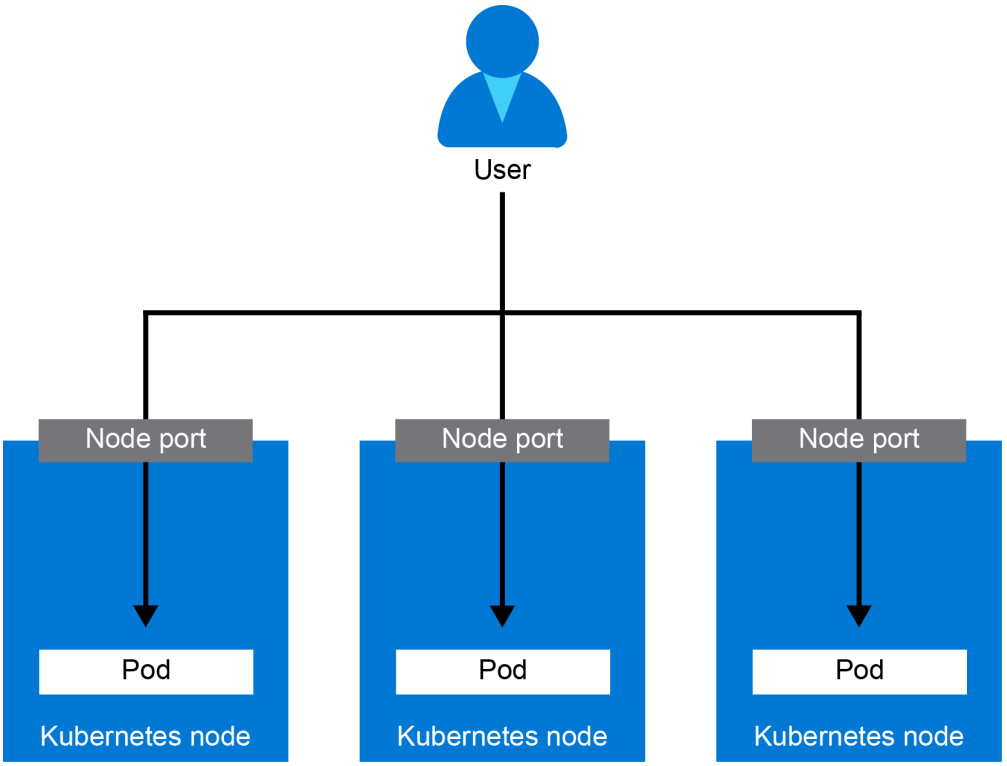


Figure 3.15: Kubernetes service of type NodePort

A final type – which will be used in this example – is the `LoadBalancer` type. This will create an **Azure Load Balancer** that will get a public IP that you can use to connect to, as shown in *Figure 3.16*:

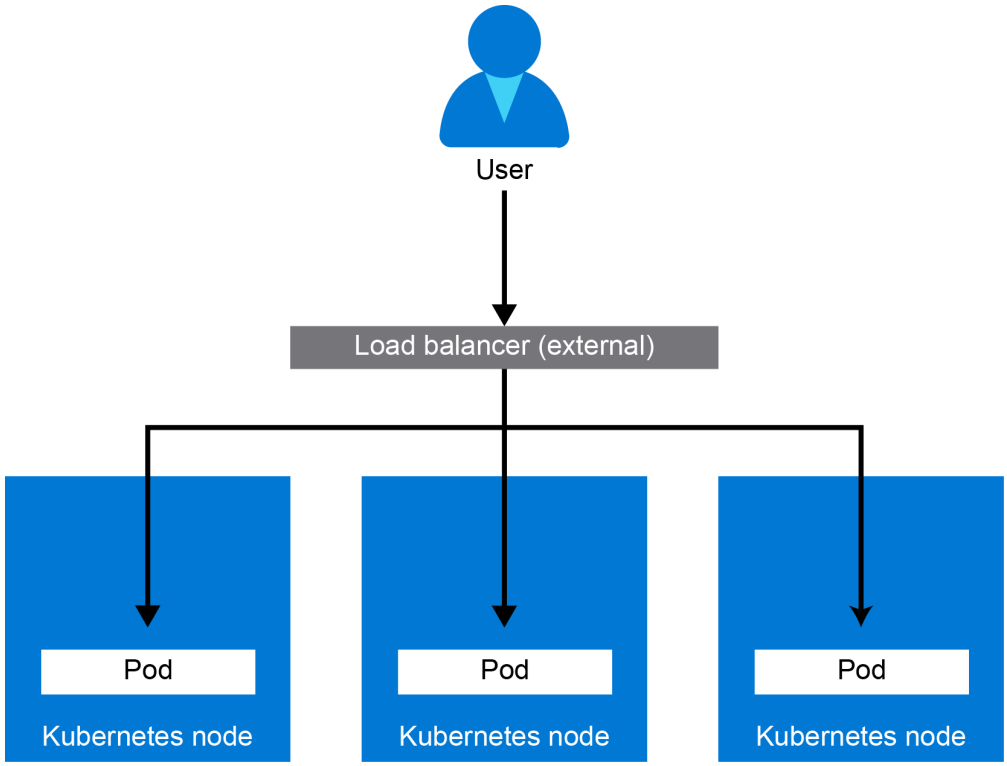


Figure 3.16: Kubernetes service of type LoadBalancer

The following code will help you to understand how the frontend service is exposed:

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: frontend
5     labels:
6       app: guestbook
7       tier: frontend
8   spec:
9     type: LoadBalancer # line uncommented
10    ports:
11    - port: 80
12    selector:
13      app: guestbook
14      tier: frontend
```

This definition is similar to the services you created earlier, except that in *line* 9 you defined `type: Load Balancer`. This will create a service of that type, which will cause AKS to add rules to the Azure load balancer.

Now that you have seen how a front-end service is exposed, let's make the guestbook application ready for use with the following steps:

1. To create the service, run the following command:

   ```
   kubectl create -f frontend-service.yaml
   ```

   This step takes some time to execute when you run it for the first time. In the background, Azure must perform a couple of actions to make it seamless. It has to create an Azure load balancer and a public IP and set the port-forwarding rules to forward traffic on port `80` to internal ports of the cluster.

2. Run the following until there is a value in the `EXTERNAL-IP` column:

```
kubectl get service -w
```

This should display the output shown in *Figure 3.17*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get service
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
frontend       LoadBalancer   10.0.119.181   52.143.73.223   80:30991/TCP   41s
kubernetes     ClusterIP      10.0.0.1       <none>          443/TCP        43h
redis-master   ClusterIP      10.0.106.207   <none>          6379/TCP       32m
redis-replica  ClusterIP      10.0.133.171   <none>          6379/TCP       9m24s
```

Figure 3.17: External IP value

3. In the Azure portal, if you click on **All Resources** and filter on **Load balancer**, you will see a **kubernetes Load balancer**. Clicking on it shows you something similar to *Figure 3.18*. The highlighted sections show you that there is a load balancing rule accepting traffic on `port 80` and you have two public IP addresses:

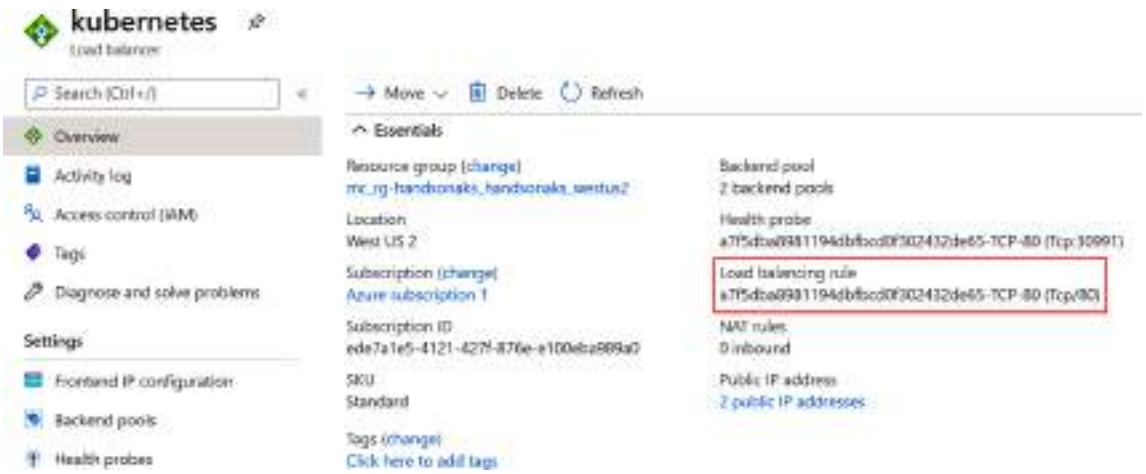


Figure 3.18: kubernetes Load balancer in the Azure portal

If you click through on the two public IP addresses, you'll see both IP addresses linked to your cluster. One of those will be the IP address of your actual front-end service; the other one is used by AKS to make outbound connections.

> **Note**
>
> Azure has two types of load balancers: basic and standard.
> Virtual machines behind a basic load balancer can make outbound connections without any specific configuration. Virtual machines behind a standard load balancer (which is the default for AKS now) need an outbound rule on the load balancer to make outbound connections. This is why you see a second IP address configured.

You're finally ready to see your guestbook app in action!

The guestbook application in action

Type the public IP of the service in your favorite browser. You should get the output shown in *Figure 3.19*:

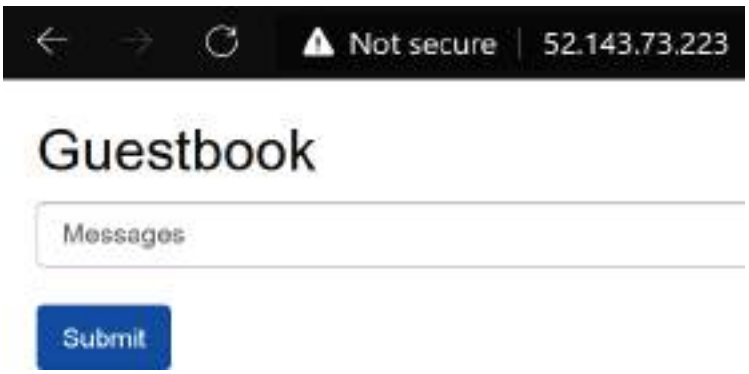


Figure 3.19: The guestbook application in action

Go ahead and record your messages. They will be saved. Open another browser and type the same IP; you will see all the messages you typed.

Congratulations – you have completed your first fully deployed, multi-tier, cloud-native Kubernetes application!

To conserve resources on your free-trial virtual machines, it is better to delete the created deployments to run the next round of the deployments by using the following commands:

```
kubectl delete deployment frontend redis-master redis-replica
kubectl delete service frontend redis-master redis-replica
```

Over the course of the preceding sections, you have deployed a Redis cluster and deployed a publicly accessible web application. You have learned how deployments, ReplicaSets, and pods are linked, and you have learned how Kubernetes uses the `service` object to route network traffic. In the next section of this chapter, you will use Helm to deploy a more complex application on top of Kubernetes.

Installing complex Kubernetes applications using Helm

In the previous section, you used static YAML files to deploy an application. When deploying more complicated applications, across multiple environments (such as dev/test/prod), it can become cumbersome to manually edit YAML files for each environment. This is where the Helm tool comes in.

Helm is the package manager for Kubernetes. Helm helps you deploy, update, and manage Kubernetes applications at scale. For this, you write something called Helm Charts.

You can think of Helm Charts as parameterized Kubernetes YAML files. If you think about the Kubernetes YAML files we wrote in the previous section, those files were static. You would need to go into the files and edit them to make changes.

Helm Charts allow you to write YAML files with certain parameters in them, which you can dynamically set. This setting of the parameters can be done through a values file or as a command-line variable when you deploy the chart.

Finally, with Helm, you don't necessarily have to write Helm Charts yourself; you can also use a rich library of pre-written Helm Charts and install popular software in your cluster through a simple command such as `helm install --name my-release stable/mysql`.

This is exactly what you are going to do in the next section. You will install WordPress on your cluster by issuing only two commands. In the next chapters, you'll also dive into custom Helm Charts that you'll edit.

> **Note**
>
> On November 13, 2019, the first stable release of Helm v3 was released. We will be using Helm v3 in the following examples. The biggest difference between Helm v2 and Helm v3 is that Helm v3 is a fully client-side tool that no longer requires the server-side tool called **Tiller**.

Let's start by installing WordPress on your cluster using Helm. In this section, you'll also learn about persistent storage in Kubernetes.

Installing WordPress using Helm

As mentioned in the introduction, Helm has a rich library of pre-written Helm Charts. To access this library, you'll have to add a repo to your Helm client:

1. Add the repo that contains the stable Helm Charts using the following command:

```
helm repo add bitnami \
  https://charts.bitnami.com/bitnami
```

2. To install WordPress, run the following command:

```
helm install handsonakswp bitnami/wordpress
```

   This execution will cause Helm to install the chart detailed at https://github.com/bitnami/charts/tree/master/bitnami/wordpress.

It takes some time for Helm to install and the site to come up. Let's look at a key concept, `PersistentVolumeClaims`, while the site is loading. After covering this, we'll go back and look at your site that got created.

PersistentVolumeClaims

A typical process requires compute, memory, network, and storage. In the guestbook example, we saw how Kubernetes helps us abstract the compute, memory, and network. The same YAML files work across all cloud providers, including a cloud-specific setup of public-facing load balancers. The WordPress example shows how the last piece, namely storage, is abstracted from the underlying cloud provider.

In this case, the WordPress Helm Chart depends on the MariaDB helm chart (https://github.com/bitnami/charts/tree/master/bitnami/mariadb) for its database installation.

Unlike stateless applications, such as our front ends, MariaDB requires careful handling of storage. To make Kubernetes handle stateful workloads, it has a specific object called a **StatefulSet**. A StatefulSet (https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/) is like a deployment with the additional capability of ordering, and the uniqueness of the pods. This means that Kubernetes will ensure that the pod and its storage are kept together. Another way that StatefulSets help is with the consistent naming of pods in a StatefulSet. The pods are named `<pod-name>-#`, where `#` starts from `0` for the first pod, and `1` for the second pod.

Running the following command, you can see that MariaDB has a predictable number attached to it, whereas the WordPress deployment has a random number attached to the end:

```
kubectl get pods
```

This will generate the output shown in *Figure 3.20*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
handsonakswp-mariadb-0                  1/1     Running   0          3m4s
handsonakswp-wordpress-856d56c5b5-fwdjz 1/1     Running   0          3m4s
```

Figure 3.20: Numbers attached to MariaDB and WordPress pods

The numbering reinforces the ephemeral nature of the deployment pods versus the StatefulSet pods.

Another difference is how pod deletion is handled. When a deployment pod is deleted, Kubernetes will launch it again anywhere it can, whereas when a StatefulSet pod is deleted, Kubernetes will relaunch it only on the node it was running on. It will relocate the pod only if the node is removed from the Kubernetes cluster.

Often, you will want to attach storage to a StatefulSet. To achieve this, a StatefulSet requires a **PersistentVolume** (**PV**). This volume can be backed by many mechanisms (including blocks, such as Azure Blob, EBS, and iSCSI, and network filesystems, such as AFS, NFS, and GlusterFS). StatefulSets require either a pre-provisioned volume or a dynamically provisioned volume handled by a **PersistentVolumeClaim** (**PVC**). A PVC allows a user to dynamically request storage, which will result in a PV being created.

Please refer to https://kubernetes.io/docs/concepts/storage/persistent-volumes/ for more detailed information.

In this WordPress example, you are using a PVC. A PVC provides an abstraction over the underlying storage mechanism. Let's look at what the MariaDB Helm Chart did by running the following:

```
kubectl get statefulset -o yaml > mariadbss.yaml
code mariadbss.yaml
```

In the preceding command, you got the YAML definition of the StatefulSet that was created and stored it in a file called `mariadbss.yaml`. Let's look at the most relevant parts of that YAML file. The code has been truncated to only show the most relevant parts:

```
1   apiVersion: v1
2   items:
3   - apiVersion: apps/v1
4     kind: StatefulSet
...
285           volumeMounts:
286           - mountPath: /bitnami/mariadb
287             name: data
...
306 volumeClaimTemplates:
307 - apiVersion: v1
308   kind: PersistentVolumeClaim
```

```
309   metadata:
310     creationTimestamp: null
311     labels:
312       app.kubernetes.io/component: primary
313       app.kubernetes.io/instance: handsonakswp
314       app.kubernetes.io/name: mariadb
315     name: data
316   spec:
317     accessModes:
318     - ReadWriteOnce
319     resources:
320       requests:
321         storage: 8Gi
322     volumeMode: Filesystem
...
```

Most of the elements of the preceding code have been covered earlier in the deployment. In the following points, we will highlight the key differences, to take a look at just the PVC:

> **Note**
>
> PVC can be used by any pod, not just StatefulSet pods.

Let's discuss the different elements of the preceding code in detail:

- **Line 4**: This line indicates the `StatefulSet` declaration.
- **Lines 285-287**: These lines mount the volume defined as `data` and mount it under the `/bitnami/mariadb` path.
- **Lines 306-322**: These lines declare the PVC. Note specifically:
    - **Line 315**: This line gives it the name `data`, which is reused at *line 285*.
    - **Line 318**: This line gives the access mode `ReadWriteOnce`, which will create block storage, which on Azure is a disk. There are other access modes as well, namely `ReadOnlyMany` and `ReadWriteMany`. As the name suggests, a `ReadWriteOnce` volume can only be attached to a single pod, while a `ReadOnlyMany` or `ReadWriteMany` volume can be attached to multiple pods at the same time. These last two types require a different underlying storage mechanism such as Azure Files or Azure Blob.
    - **Line 321**: This line defines the size of the disk.

Based on the preceding information, Kubernetes dynamically requests and binds an 8 GiB volume to this pod. In this case, the default dynamic-storage provisioner backed by the Azure disk is used. The dynamic provisioner was set up by Azure when you created the cluster. To see the storage classes available on your cluster, you can run the following command:

```
kubectl get storageclass
```

This will show you an output similar to *Figure 3.21*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get storageclass
NAME                PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
azurefile           kubernetes.io/azure-file   Delete          Immediate              true                   44h
azurefile-premium   kubernetes.io/azure-file   Delete          Immediate              true                   44h
default (default)   kubernetes.io/azure-disk   Delete          Immediate              true                   44h
managed-premium     kubernetes.io/azure-disk   Delete          Immediate              true                   44h
```

Figure 3.21: Different storage classes in your cluster

We can get more details about the PVC by running the following:

```
kubectl get pvc
```

The output generated is displayed in *Figure 3.22*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get pvc
NAME                          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-handsonakswp-mariadb-0   Bound    pvc-c88da151-777c-4efa-ac72-c05dd0b33801   8Gi        RWO            default        13m
handsonakswp-wordpress        Bound    pvc-102a8500-5f0b-411d-8ef0-518f2759ca36   10Gi       RWO            default        13m
```

Figure 3.22: Different PVCs in the cluster

When we asked for storage in the StatefulSet description (*lines 128-143*), Kubernetes performed Azure-disk-specific operations to get the Azure disk with 8 GiB of storage. If you copy the name of the PVC and paste that in the Azure search bar, you should find the disk that was created:

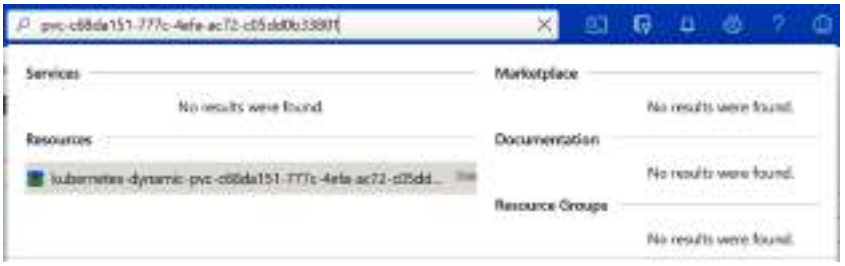


Figure 3.23: Getting the disk linked to a PVC

The concept of a PVC abstracts cloud provider storage specifics. This allows the same Helm template to work across Azure, AWS, or GCP. On AWS, it will be backed by **Elastic Block Store** (**EBS**), and on GCP it will be backed by Persistent Disk.

Also, note that PVCs can be deployed without using Helm.

In this section, the concept of storage in Kubernetes using **PersistentVolumeClaim** (**PVC**) was introduced. You saw how they were created by the WordPress Helm deployment, and how Kubernetes created an Azure disk to support the PVC used by MariaDB. In the next section, you will explore the WordPress application on Kubernetes in more detail.

Checking the WordPress deployment

After our analysis of the PVCs, let's check back in with the Helm deployment. You can check the status of the deployment using:

```
helm ls
```

This should return the output shown in *Figure 3.24*:

Figure 3.24: WordPress application deployment status

We can get more info from our deployment in Helm using the following command:

```
helm status handsonakswp
```

This will return the output shown in *Figure* 3.25:

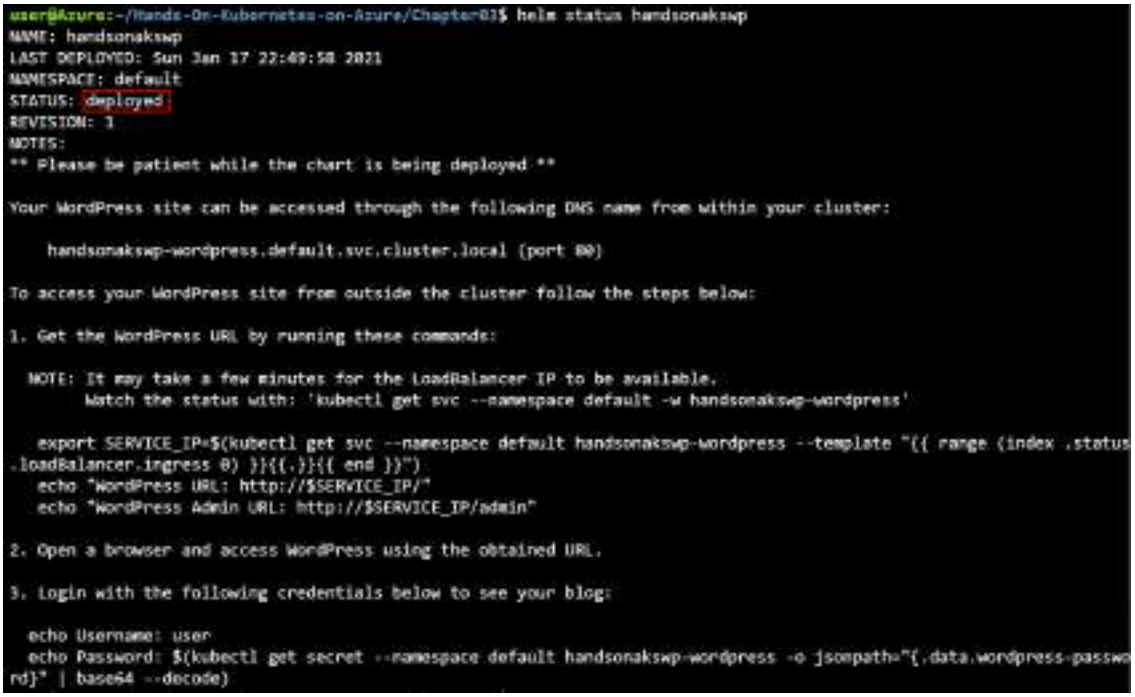


Figure 3.25: Getting more details about the deployment

This shows you that your chart was deployed successfully. It also shows more info on how you can connect to your site. You won't be using these steps for now; you will revisit these steps in *Chapter* 5, *Handling common failures in AKS*, in the section where we cover fixing storage mount issues. For now, let's look into everything that Helm created for you:

```
kubectl get all
```

This will generate an output similar to *Figure* 3.26:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get all
NAME                                        READY   STATUS    RESTARTS   AGE
pod/handsonakswp-mariadb-0                  1/1     Running   0          20m
pod/handsonakswp-wordpress-856d56c5b5-fwdjz 1/1     Running   0          20m

NAME                              TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)                      AGE
service/handsonakswp-mariadb      ClusterIP      10.0.105.160   <none>          3306/TCP                     20m
service/handsonakswp-wordpress    LoadBalancer   10.0.255.15    20.69.187.228   80:30104/TCP,443:32279/TCP   20m
service/kubernetes                ClusterIP      10.0.0.1       <none>          443/TCP                      44h

NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/handsonakswp-wordpress 1/1     1            1           20m

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/handsonakswp-wordpress-856d56c5b5 1         1         1       20m

NAME                                 READY   AGE
statefulset.apps/handsonakswp-mariadb 1/1     20m
```

Figure 3.26: List of objects created by Helm

If you don't have an external IP yet, wait for a couple of minutes and retry the command.

You can then go ahead and connect to your external IP and access your WordPress site. *Figure* 3.27 is the resulting output:

Figure 3.27: WordPress site being displayed on connection with the external IP

To make sure you don't run into issues in the following chapters, let's delete the WordPress site. This can be done in the following way:

```
helm delete handsonakswp
```

By design, the PVCs won't be deleted. This ensures persistent data is kept. As you don't have any persistent data, you can safely delete the PVCs as well:

```
kubectl delete pvc --all
```

> **Note**
>
> Be very careful when executing `kubectl delete <object> --all` as it will delete all the objects in a namespace. This is not recommended on a production cluster.

In this section, you have deployed a full WordPress site using Helm. You also learned how Kubernetes handles persistent storage using PVCs.

Summary

In this chapter, you deployed two applications. You started the chapter by deploying the guestbook application. During that deployment, the details of pods, ReplicaSets, and deployments were explored. You also used dynamic configuration using ConfigMaps. Finally, you looked into how services are used to route traffic to the deployed applications.

The second application you deployed was a WordPress application. You deployed it via the Helm package manager. As part of this deployment, PVCs were used, and you explored how they were used in the system and how they were linked to disks on Azure.

In *Chapter 4*, *Building scalable applications*, you will look into scaling applications and the cluster itself. You will first learn about the manual and automatic scaling of the application, and afterward, you'll learn about the manual and automatic scaling of the cluster itself. Finally, different ways in which applications can be updated on Kubernetes will be explained.

4

Building scalable applications

When running an application efficiently, the ability to scale and upgrade your application is critical. Scaling allows your application to handle additional load. While upgrading, scaling is needed to keep your application up to date and to introduce new functionality.

Scaling on demand is one of the key benefits of using cloud-native applications. It also helps optimize resources for your application. If the front end component encounters heavy load, you can scale the front end alone, while keeping the same number of back end instances. You can increase or reduce the number of **virtual machines** (**VMs**) required depending on your workload and peak demand hours. This chapter will cover the scale dimensions of the application and its infrastructure in detail.

In this chapter, you will learn how to scale the sample guestbook application that was introduced in *Chapter 3*, *Application deployment on* AKS. You will first scale this application using manual commands, and afterward you'll learn how to autoscale it using the **Horizontal Pod Autoscaler (HPA)**. The goal is to make you comfortable with `kubectl`, which is an important tool for managing applications running on top of **Azure Kubernetes Service** (**AKS**). After scaling the application, you will also scale the cluster. You will first scale the cluster manually, and then use the **cluster autoscaler** to automatically scale the cluster. In addition, you will get a brief introduction on how you can upgrade applications running on top of AKS.

In this chapter, we will cover the following topics:

- Scaling your application
- Scaling your cluster
- Upgrading your application

Let's begin this chapter by discussing the different dimensions of scaling applications on top of AKS.

Scaling your application

There are two scale dimensions for applications running on top of AKS. The first scale dimension is the number of pods a deployment has, while the second scale dimension in AKS is the number of nodes in the cluster.

By adding new pods to a deployment, also known as scaling out, you can add additional compute power to the deployed application. You can either scale out your applications manually or have Kubernetes take care of this automatically via HPA. HPA can monitor metrics such as the CPU to determine whether pods need to be added to your deployment.

The second scale dimension in AKS is the number of nodes in the cluster. The number of nodes in a cluster defines how much CPU and memory are available for all the applications running on that cluster. You can scale your cluster manually by changing the number of nodes, or you can use the cluster autoscaler to automatically scale out your cluster. The cluster autoscaler watches the cluster

for pods that cannot be scheduled due to resource constraints. If pods cannot be scheduled, it will add nodes to the cluster to ensure that your applications can run.

Both scale dimensions will be covered in this chapter. In this section, you will learn how you can scale your application. First, you will scale your application manually, and then later, you will scale your application automatically.

Manually scaling your application

To demonstrate manual scaling, let's use the guestbook example that we used in the previous chapter. Follow these steps to learn how to implement manual scaling:

> **Note**
>
> In the previous chapter, we cloned the example files in Cloud Shell. If you didn't do this back then, we recommend doing that now:
> `git clone https://github.com/PacktPublishing/Hands-On-Kubernetes-on-Azure-third-edition`
> For this chapter, navigate to the `Chapter04` directory:
> `cd Chapter04`

1. Set up the guestbook by running the `kubectl create` command in the Azure command line:

```
kubectl create -f guestbook-all-in-one.yaml
```

2. After you have entered the preceding command, you should see something similar to what is shown in *Figure 4.1* in your command-line output:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl create -f guestbook-all-in-one.yaml
service/redis-master created
deployment.apps/redis-master created
service/redis-replica created
deployment.apps/redis-replica created
service/frontend created
deployment.apps/frontend created
```

Figure 4.1: Launching the guestbook application

3. Right now, none of the services are publicly accessible. We can verify this by running the following command:

```
kubectl get service
```

4. As seen in *Figure* 4.2, none of the services have an external IP:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get service
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
frontend        ClusterIP   10.0.118.101   <none>        80/TCP     76s
kubernetes      ClusterIP   10.0.0.1       <none>        443/TCP    3d22h
redis-master    ClusterIP   10.0.181.124   <none>        6379/TCP   77s
redis-replica   ClusterIP   10.0.138.136   <none>        6379/TCP   77s
```

Figure 4.2: Output confirming that none of the services have a public IP

5. To test the application, you will need to expose it publicly. For this, let's introduce a new command that will allow you to edit the service in Kubernetes without having to change the file on your file system. To start the edit, execute the following command:

```
kubectl edit service frontend
```

6. This will open a `vi` environment. Use the down arrow key to navigate to the line that says `type: ClusterIP` and change that to `type: LoadBalancer`, as shown in *Figure* 4.3. To make that change, hit the *I* button, change `type` to `LoadBalancer`, hit the *Esc* button, type `:wq!`, and then hit *Enter* to save the changes:

```
spec:
  clusterIP: 10.0.118.101
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: guestbook
    tier: frontend
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer: {}
```

Figure 4.3: Changing this line to type: LoadBalancer

7. Once the changes are saved, you can watch the service object until the public IP becomes available. To do this, type the following:

```
kubectl get service -w
```

8. It will take a couple of minutes to show you the updated IP. Once you see the correct public IP, you can exit the `watch` command by hitting *Ctrl* + *C*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get service -w
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
frontend       LoadBalancer   10.0.118.101   <pending>     80:30009/TCP   3m55s
kubernetes     ClusterIP      10.0.0.1       <none>        443/TCP        3d22h
redis-master   ClusterIP      10.0.181.124   <none>        6379/TCP       3m56s
redis-replica  ClusterIP      10.0.138.136   <none>        6379/TCP       3m56s
frontend       LoadBalancer   10.0.118.101   52.149.17.246   80:30009/TCP   4m7s
```

Figure 4.4: Output showing the front-end service getting a public IP

9. Type the IP address from the preceding output into your browser navigation bar as follows: `http://<EXTERNAL-IP>/`. The result of this is shown in *Figure 4.5*:

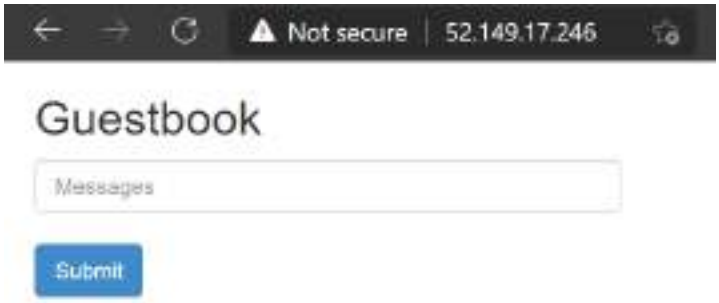


Figure 4.5: Browse to the guestbook application

The familiar guestbook sample should be visible. This shows that you have successfully publicly accessed the guestbook.

Now that you have the guestbook application deployed, you can start scaling the different components of the application.

Scaling the guestbook front-end component

Kubernetes gives us the ability to scale each component of an application dynamically. In this section, we will show you how to scale the front end of the guestbook application. Right now, the front-end deployment is deployed with three replicas. You can confirm by using the following command:

```
kubectl get pods
```

This should return an output as shown in *Figure 4.6*:

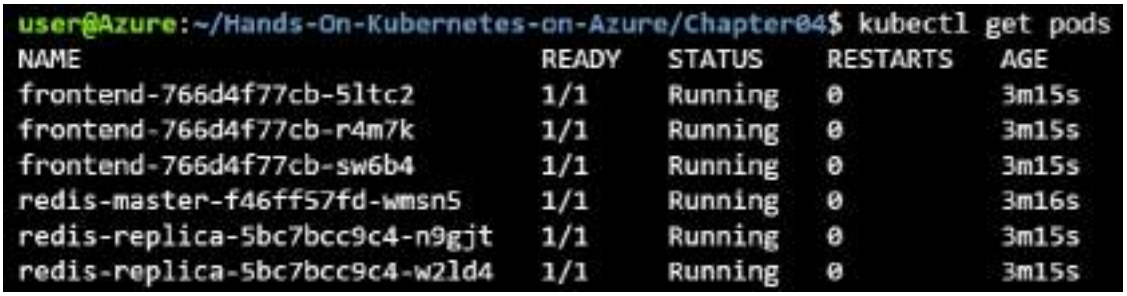


Figure 4.6: Confirming the three replicas in the front-end deployment

To scale the front-end deployment, you can execute the following command:

```
kubectl scale deployment/frontend --replicas=6
```

This will cause Kubernetes to add additional pods to the deployment. You can set the number of replicas you want, and Kubernetes takes care of the rest. You can even scale it down to zero (one of the tricks used to reload the configuration when the application doesn't support the dynamic reload of configuration). To verify that the overall scaling worked correctly, you can use the following command:

```
kubectl get pods
```

This should give you the output shown in *Figure* 4.7:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-5ltc2           1/1     Running   0          3m57s
frontend-766d4f77cb-6xwvz           1/1     Running   0          5s
frontend-766d4f77cb-gmd5p           1/1     Running   0          5s
frontend-766d4f77cb-r4m7k           1/1     Running   0          3m57s
frontend-766d4f77cb-sw6b4           1/1     Running   0          3m57s
frontend-766d4f77cb-vz726           1/1     Running   0          5s
redis-master-f46ff57fd-wmsn5        1/1     Running   0          3m58s
redis-replica-5bc7bcc9c4-n9gjt      1/1     Running   0          3m57s
redis-replica-5bc7bcc9c4-w2ld4      1/1     Running   0          3m57s
```

Figure 4.7: Different pods running in the guestbook application after scaling out

As you can see, the front-end service scaled to six pods. Kubernetes also spread these pods across multiple nodes in the cluster. You can see the nodes that this is running on with the following command:

```
kubectl get pods -o wide
```

This will generate the following output:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE     IP            NODE
frontend-766d4f77cb-5ltc2           1/1     Running   0          4m22s   10.244.1.6    aks-agentpool-35858025-vmss000001
frontend-766d4f77cb-6xwvz           1/1     Running   0          30s     10.244.1.8    aks-agentpool-35858025-vmss000001
frontend-766d4f77cb-gmd5p           1/1     Running   0          30s     10.244.0.11   aks-agentpool-35858025-vmss000000
frontend-766d4f77cb-r4m7k           1/1     Running   0          4m22s   10.244.0.8    aks-agentpool-35858025-vmss000000
frontend-766d4f77cb-sw6b4           1/1     Running   0          4m22s   10.244.1.7    aks-agentpool-35858025-vmss000001
frontend-766d4f77cb-vz726           1/1     Running   0          30s     10.244.0.10   aks-agentpool-35858025-vmss000000
redis-master-f46ff57fd-wmsn5        1/1     Running   0          4m23s   10.244.1.4    aks-agentpool-35858025-vmss000001
redis-replica-5bc7bcc9c4-n9gjt      1/1     Running   0          4m22s   10.244.1.5    aks-agentpool-35858025-vmss000001
redis-replica-5bc7bcc9c4-w2ld4      1/1     Running   0          4m22s   10.244.0.9    aks-agentpool-35858025-vmss000000
```

Figure 4.8: Showing which nodes the pods are running on

In this section, you have seen how easy it is to scale pods with Kubernetes. This capability provides a very powerful tool for you to not only dynamically adjust your application components but also provide resilient applications with failover capabilities enabled by running multiple instances of components at the same time. However, you won't always want to manually scale your application. In the next section, you will learn how you can automatically scale your application in and out by automatically adding and removing pods in a deployment.

Using the HPA

Scaling manually is useful when you're working on your cluster. For example, if you know your load is going to increase, you can manually scale out your application. In most cases, however, you will want some sort of autoscaling to happen on your application. In Kubernetes, you can configure autoscaling of your deployment using an object called the **Horizontal Pod Autoscaler** (**HPA**).

HPA monitors Kubernetes metrics at regular intervals and, based on the rules you define, it automatically scales your deployment. For example, you can configure the HPA to add additional pods to your deployment once the CPU utilization of your application is above 50%.

In this section, you will configure the HPA to scale the front-end of the application automatically:

1. To start the configuration, let's first manually scale down our deployment to one instance:

```
kubectl scale deployment/frontend --replicas=1
```

2. Next up, we'll create an HPA. Open up the code editor in Cloud Shell by typing `code hpa.yaml` and enter the following code:

```
1   apiVersion: autoscaling/v1
2   kind: HorizontalPodAutoscaler
3   metadata:
4     name: frontend-scaler
5   spec:
6     scaleTargetRef:
7       apiVersion: apps/v1
8       kind: Deployment
9       name: frontend
10    minReplicas: 1
11    maxReplicas: 10
12    targetCPUUtilizationPercentage: 50
```

Let's investigate what is configured in this file:

- **Line 2**: Here, we define that we need `HorizontalPodAutoscaler`.
- **Lines 6-9**: These lines define the deployment that we want to autoscale.
- **Lines 10-11**: Here, we configure the minimum and maximum pods in our deployment.
- **Lines 12**: Here, we define the target CPU utilization percentage for our deployment.

3. Save this file, and create the HPA using the following command:

```
kubectl create -f hpa.yaml
```

This will create our autoscaler. You can see your autoscaler with the following command:

```
kubectl get hpa
```

This will initially output something as shown in *Figure 4.9*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get hpa
NAME              REFERENCE              TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
frontend-scaler   Deployment/frontend   <unknown>/50%   1         10        0          2s
```

Figure 4.9: The target unknown shows that the HPA isn't ready yet

It takes a couple of seconds for the HPA to read the metrics. Wait for the return from the HPA to look something similar to the output shown in *Figure 4.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get hpa -w
NAME              REFERENCE              TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
frontend-scaler   Deployment/frontend   <unknown>/50%   1         10        0          1s
frontend-scaler   Deployment/frontend   <unknown>/50%   1         10        1          15s
frontend-scaler   Deployment/frontend   10%/50%         1         10        1          31s
```

Figure 4.10: Once the target shows a percentage, the HPA is ready

4. You will now go ahead and do two things: first, you will watch the pods to see whether new pods are created. Then, you will create a new shell, and create some load for the system. Let's start with the first task–watching our pods:

```
kubectl get pods -w
```

This will continuously monitor the pods that get created or terminated.

Let's now create some load in a new shell. In Cloud Shell, hit the **open new session** icon to open a new shell:

Figure 4.11: Use this button to open a new Cloud Shell

This will open a new tab in your browser with a new session in Cloud Shell. You will generate load for the application from this tab.

5. Next, you will use a program called `hey` to generate this load. `hey` is a tiny program that sends loads to a web application. You can install and run `hey` using the following commands:

```
export GOPATH=~/go
export PATH=$GOPATH/bin:$PATH
go get -u github.com/rakyll/hey
hey -z 20m http://<external-ip>
```

The `hey` program will now try to create up to 20 million connections to the front-end. This will generate CPU loads on the system, which will trigger the HPA to start scaling the deployment. It will take a couple of minutes for this to trigger a scale action, but at a certain point, you should see multiple pods being created to handle the additional load, as shown in *Figure 4.12*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods -w
NAME                              READY   STATUS              RESTARTS   AGE
frontend-766d4f77cb-r4m7k         1/1     Running             0          5m40s
redis-master-f46ff57fd-wmsn5      1/1     Running             0          5m41s
redis-replica-5bc7bcc9c4-n9gjt    1/1     Running             0          5m40s
redis-replica-5bc7bcc9c4-w2ld4    1/1     Running             0          5m40s
frontend-766d4f77cb-kvd24         0/1     Pending             0          0s
frontend-766d4f77cb-kvd24         0/1     Pending             0          0s
frontend-766d4f77cb-25bjj         0/1     Pending             0          0s
frontend-766d4f77cb-z855p         0/1     Pending             0          0s
frontend-766d4f77cb-z855p         0/1     Pending             0          0s
frontend-766d4f77cb-25bjj         0/1     Pending             0          0s
frontend-766d4f77cb-z855p         0/1     ContainerCreating   0          0s
frontend-766d4f77cb-kvd24         0/1     ContainerCreating   0          0s
frontend-766d4f77cb-25bjj         0/1     ContainerCreating   0          0s
frontend-766d4f77cb-z855p         1/1     Running             0          1s
frontend-766d4f77cb-25bjj         1/1     Running             0          1s
frontend-766d4f77cb-kvd24         1/1     Running             0          2s
```

Figure 4.12: New pods get started by the HPA

At this point, you can go ahead and kill the hey program by hitting *Ctrl* + *C*.

6. Let's have a closer look at what the HPA did by running the following command:

   ```
   kubectl describe hpa
   ```

   We can see a few interesting points in the `describe` operation, as shown in *Figure 4.13*:

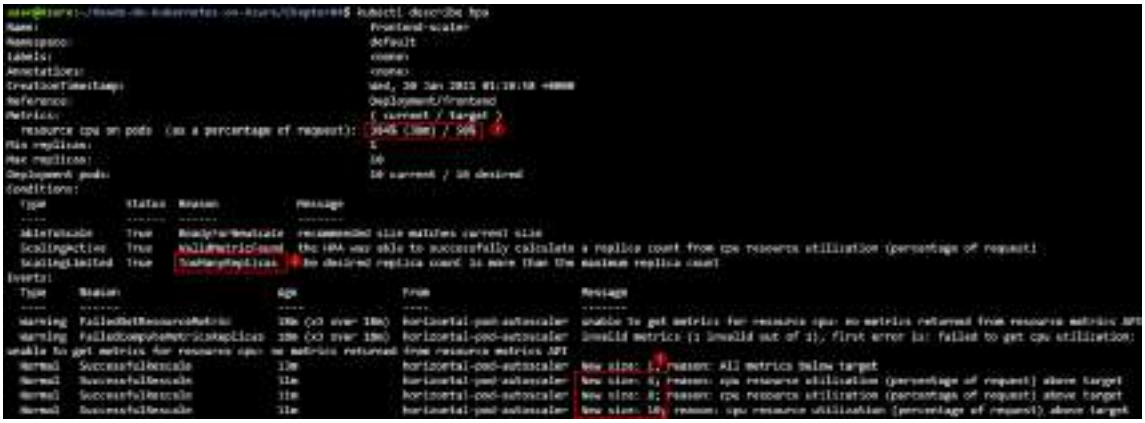


Figure 4.13: Detailed view of the HPA

The annotations in *Figure 4.13* are explained as follows:

- This shows you the current CPU utilization (`384%`) versus the desired (`50%`). The current CPU utilization will likely be different in your situation.
- This shows you that the current desired replica count is higher than the actual maximum you had configured. This ensures that a single deployment doesn't consume all resources in the cluster.
- This shows you the scaling actions that the HPA took. It first scaled to 4, then to 8, and then to 10 pods in the deployment.

7. If you wait for a couple of minutes, the HPA should start to scale down. You can track this scale-down operation using the following command:

```
kubectl get hpa -w
```

This will track the HPA and show you the gradual scaling down of the deployment, as displayed in *Figure 4.14*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get hpa -w
NAME              REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
frontend-scaler   Deployment/frontend      10%/50%   1         10        10         21m
frontend-scaler   Deployment/frontend      10%/50%   1         10        10         25m
frontend-scaler   Deployment/frontend      10%/50%   1         10        2          26m
frontend-scaler   Deployment/frontend      10%/50%   1         10        2          30m
frontend-scaler   Deployment/frontend      10%/50%   1         10        1          31m
```

Figure 4.14: Watching the HPA scale down

8. Before we move on to the next section, let's clean up the resources we created in this section:

```
kubectl delete -f hpa.yaml
kubectl delete -f guestbook-all-in-one.yaml
```

In this section, you first manually and then automatically scaled an application. However, the infrastructure supporting the application was static; you ran this on a two-node cluster. In many cases, you might also run out of resources on the cluster. In the next section, you will deal with this issue and learn how you can scale the AKS cluster yourself.

Scaling your cluster

In the previous section, you dealt with scaling the application running on top of a cluster. In this section, you'll learn how you can scale the actual cluster you are running. First, you will manually scale your cluster to one node. Then, you'll configure the cluster autoscaler. The cluster autoscaler will monitor your cluster and scale out when there are pods that cannot be scheduled on the cluster.

Manually scaling your cluster

You can manually scale your AKS cluster by setting a static number of nodes for the cluster. The scaling of your cluster can be done either via the Azure portal or the command line.

In this section, you'll learn how you can manually scale your cluster by scaling it down to one node. This will cause Azure to remove one of the nodes from your cluster. First, the workload on the node that is about to be removed will be rescheduled onto the other node. Once the workload is safely rescheduled, the node will be removed from your cluster, and then the VM will be deleted from Azure.

To scale your cluster, follow these steps:

1. Open the Azure portal and go to your cluster. Once there, go to **Node pools** and click on the number below **Node count**, as shown in *Figure* 4.15:

Figure 4.15: Manually scaling the cluster

2. This will open a pop-up window that will give the option to scale your cluster. For our example, we will scale down our cluster to one node, as shown in *Figure 4.16*:

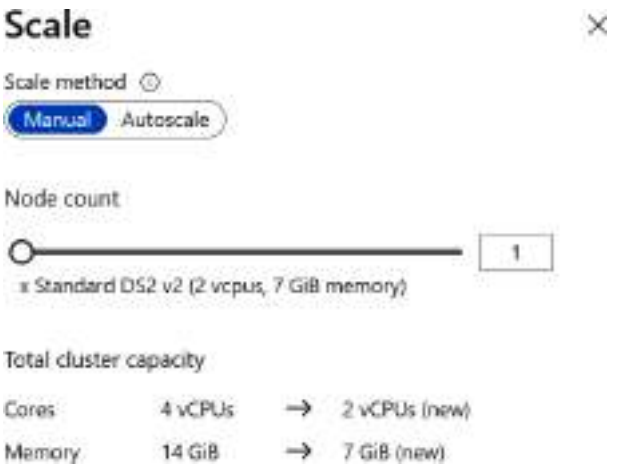


Figure 4.16: Pop-up window confirming the new cluster size

3. Hit the **Apply** button at the bottom of the screen to save these settings. This will cause Azure to remove a node from your cluster. This process will take about 5 minutes to complete. You can follow the progress by clicking on the notification icon at the top of the Azure portal as follows:

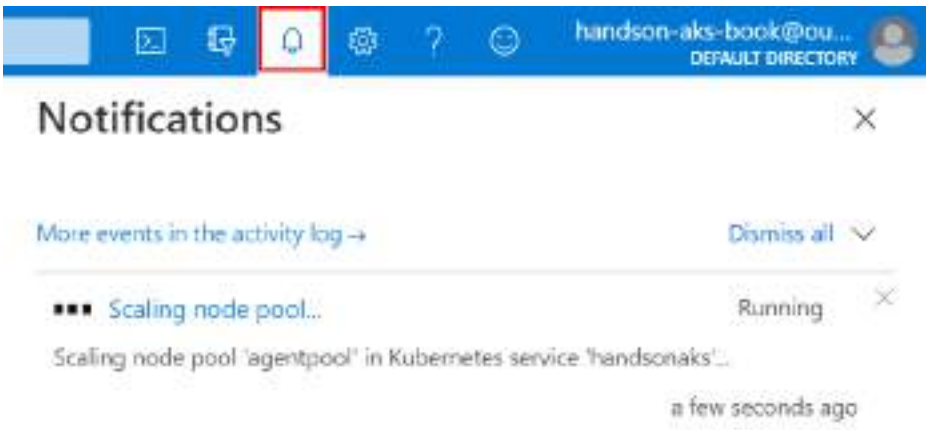


Figure 4.17: Cluster scaling can be followed using the notifications in the Azure portal

Once this scale-down operation has completed, relaunch the guestbook application on this small cluster:

```
kubectl create -f guestbook-all-in-one.yaml
```

In the next section, you will scale out the guestbook so that it can no longer run on this small cluster. You will then configure the cluster autoscaler to scale out the cluster.

Scaling your cluster using the cluster autoscaler

In this section, you will explore the cluster autoscaler. The cluster autoscaler will monitor the deployments in your cluster and scale your cluster to meet your application requirements. The cluster autoscaler watches the number of pods in your cluster that cannot be scheduled due to insufficient resources. You will first force your deployment to have pods that cannot be scheduled, and then configure the cluster autoscaler to automatically scale your cluster.

To force your cluster to be out of resources, you will—manually—scale out the `redis-replica` deployment. To do this, use the following command:

```
kubectl scale deployment redis-replica --replicas 5
```

You can verify that this command was successful by looking at the pods in our cluster:

```
kubectl get pods
```

This should show you something similar to the output shown in *Figure 4.18*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
NAME                             READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-578vs        1/1     Running   0          30m
frontend-766d4f77cb-p64vw        1/1     Running   0          30m
frontend-766d4f77cb-wjwj4        1/1     Running   0          30m
redis-master-f46ff57fd-b9518     1/1     Running   0          30m
redis-replica-5bc7bcc9c4-btkzp   1/1     Running   0          30m
redis-replica-5bc7bcc9c4-ckvz2   1/1     Running   0          30m
redis-replica-5bc7bcc9c4-mwmcm   1/1     Running   0          30m
redis-replica-5bc7bcc9c4-vdxcl   0/1     Pending   0          30m
redis-replica-5bc7bcc9c4-vjrg5   0/1     Pending   0          30m
```

Figure 4.18: Four out of five pods are pending, meaning they cannot be scheduled

As you can see, you now have two pods in a `Pending` state. The `Pending` state in Kubernetes means that that pod cannot be scheduled onto a node. In this case, this is due to the cluster being out of resources.

> **Note**
>
> If your cluster is running on a larger VM size than the DS2v2, you might not notice pods in a `Pending` state now. In that case, increase the number of replicas to a higher number until you see pods in a pending state.

You will now configure the cluster autoscaler to automatically scale the cluster. Similar to manual scaling in the previous section, there are two ways you can configure the cluster autoscaler. You can configure it either via the Azure portal—similar to how we did the manual scaling—or you can configure it using the **command-line interface (CLI)**. In this example, you will use CLI to enable the cluster autoscaler. The following command will configure the cluster autoscaler for your cluster:

```
az aks nodepool update --enable-cluster-autoscaler \
  -g rg-handsonaks --cluster-name handsonaks \
  --name agentpool --min-count 1 --max-count 2
```

This command configures the cluster autoscaler on the node pool you have in the cluster. It configures it to have a minimum of one node and a maximum of two nodes. This will take a couple of minutes to configure.

Once the cluster autoscaler is configured, you can see it in action by using the following command to watch the number of nodes in the cluster:

```
kubectl get nodes -w
```

It will take about 5 minutes for the new node to show up and become `Ready` in the cluster. Once the new node is `Ready`, you can stop watching the nodes by hitting *Ctrl + C*. You should see an output similar to what you see in *Figure 4.19*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get nodes -w
NAME                                STATUS     ROLES    AGE   VERSION
aks-agentpool-39838025-vmss000000   Ready      agent    58m   v1.19.6
aks-agentpool-39838025-vmss000002   NotReady   <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002   NotReady   <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002   NotReady   <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002   NotReady   <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002   NotReady   <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002   NotReady   <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002   Ready      <none>   10s   v1.19.6
aks-agentpool-39838025-vmss000002   Ready      <none>   10s   v1.19.6
aks-agentpool-39838025-vmss000002   Ready      <none>   10s   v1.19.6
aks-agentpool-39838025-vmss000002   Ready      <none>   11s   v1.19.6
aks-agentpool-39838025-vmss000002   Ready      <none>   30s   v1.19.6
aks-agentpool-39838025-vmss000002   Ready      <none>   43s   v1.19.6
aks-agentpool-39838025-vmss000002   Ready      agent    46s   v1.19.6
```

Figure 4.19: The new node joins the cluster

The new node should ensure that your cluster has sufficient resources to schedule the scaled-out `redis- replica` deployment. To verify this, run the following command to check the status of the pods:

```
kubectl get pods
```

This should show you all the pods in a `Running` state as follows:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-578vs       1/1     Running   0          37m
frontend-766d4f77cb-p64vw       1/1     Running   0          37m
frontend-766d4f77cb-wjwj4       1/1     Running   0          37m
redis-master-f46ff57fd-b9518    1/1     Running   0          37m
redis-replica-5bc7bcc9c4-btkzp  1/1     Running   0          37m
redis-replica-5bc7bcc9c4-ckvz2  1/1     Running   0          37m
redis-replica-5bc7bcc9c4-mwmcm  1/1     Running   0          37m
redis-replica-5bc7bcc9c4-vdxcl  1/1     Running   0          37m
redis-replica-5bc7bcc9c4-vjrg5  1/1     Running   0          37m
```

Figure 4.20: All pods are now in a Running state

Now clean up the resources you created, disable the cluster autoscaler, and ensure that your cluster has two nodes for the next example. To do this, use the following commands:

```
kubectl delete -f guestbook-all-in-one.yaml
az aks nodepool update --disable-cluster-autoscaler \
  -g rg-handsonaks --cluster-name handsonaks --name agentpool
az aks nodepool scale --node-count 2 -g rg-handsonaks \
  --cluster-name handsonaks --name agentpool
```

> **Note**
>
> The last command from the previous example will show you an error message, `The new node count is the same as the current node count.`, if the cluster already has two nodes. You can safely ignore this error.

In this section, you first manually scaled down your cluster and then used the cluster autoscaler to scale out your cluster. You used the Azure portal to scale down the cluster manually and then used the Azure CLI to configure the cluster autoscaler. In the next section, you will look into how you can upgrade applications running on AKS.

Upgrading your application

Using deployments in Kubernetes makes upgrading an application a straightforward operation. As with any upgrade, you should have good failbacks in case something goes wrong. Most of the issues you will run into will happen during upgrades. Cloud-native applications are supposed to make dealing with this relatively easy, which is possible if you have a very strong development team that embraces DevOps principles.

The State of DevOps report (https://puppet.com/resources/report/2020-state-of-devops-report/) has reported for multiple years that companies that have high software deployment frequency rates have higher availability and stability in their applications as well. This might seem counterintuitive, as doing software deployments heightens the risk of issues. However, by deploying more frequently and deploying using automated DevOps practices, you can limit the impact of software deployment.

There are multiple ways you can make updates to applications running in a Kubernetes cluster. In this section, you will explore the following ways to update Kubernetes resources:

- Upgrading by changing YAML files: This method is useful when you have access to the full YAML file required to make the update. This can be done either from your command line or from an automated system.
- Upgrading using `kubectl edit`: This method is mostly used for minor changes on a cluster. It is a quick way to update your configuration live on a cluster.
- Upgrading using `kubectl patch`: This method is useful when you need to script a particular small update to a Kubernetes but don't have access to the full YAML file. It can be done either from a command line or an automated system. If you have access to the original YAML files, it is typically better to edit the YAML file and use `kubectl apply` to apply the updates.
- Upgrading using Helm: This method is used when your application is deployed through Helm.

The methods described in the following sections work great if you have stateless applications. If you have a state stored anywhere, make sure to back up that state before you try upgrading your application.

Let's start this section by doing the first type of upgrade by changing YAML files.

Upgrading by changing YAML files

In order to upgrade a Kubernetes service or deployment, you can update the actual YAML definition file and apply that to the currently deployed application. Typically, we use `kubectl create` to create resources. Similarly, we can use `kubectl apply` to make changes to the resources.

The deployment detects the changes (if any) and matches the running state to the desired state. Let's see how this is done:

1. Start with our guestbook application to explore this example:

```
kubectl apply -f guestbook-all-in-one.yaml
```

2. After a few minutes, all the pods should be running. Let's perform the first upgrade by changing the service from `ClusterIP` to `LoadBalancer`, as you did earlier in the chapter. However, now you will edit the YAML file rather than using `kubectl edit`. Edit the YAML file using the following command:

```
code guestbook-all-in-one.yaml
```

   Uncomment line 102 in this file to set the `type` to `LoadBalancer`, and save the file, as shown in *Figure 4.21*:

```
100   spec:
101     # uncomment the line below to create a Load Balanced service
102     type: LoadBalancer
103     ports:
104     - port: 80
105     selector:
106       app: guestbook
107       tier: frontend
```

Figure 4.21: Setting the type to LoadBalancer in the guestbook-all-in-one YAML file

3. Apply the change as shown in the following code:

```
kubectl apply -f guestbook-all-in-one.yaml
```

   You should see an output similar to *Figure 4.22*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl apply -f guestbook-all-in-one.yaml
service/redis-master unchanged
deployment.apps/redis-master unchanged
service/redis-replica unchanged
deployment.apps/redis-replica unchanged
service/frontend configured
deployment.apps/frontend unchanged
```

Figure 4.22: The service's front-end is updated

   As you can see in *Figure 4.22*, only the object that was updated in the YAML file, which is the service in this case, was updated on Kubernetes, and the other objects remained unchanged.

4. You can now get the public IP of the service using the following command:

```
kubectl get service
```

Give it a few minutes, and you should be shown the IP, as displayed in *Figure 4.23*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get service
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)        AGE
frontend       LoadBalancer   10.0.119.74    40.64.105.32   80:32287/TCP   2m43s
kubernetes     ClusterIP      10.0.0.1       <none>         443/TCP        4d7h
redis-master   ClusterIP      10.0.75.94     <none>         6379/TCP       2m43s
redis-replica  ClusterIP      10.0.1.20      <none>         6379/TCP       2m43s
```

Figure 4.23: Output displaying a public IP

5. You will now make another change. You'll downgrade the front-end image on line 127 from `image: gcr.io/google-samples/gb-frontend:v4` to the following:

   ```
   image: gcr.io/google-samples/gb-frontend:v3
   ```

   This change can be made by opening the guestbook application in the editor by using this familiar command:

   ```
   code guestbook-all-in-one.yaml
   ```

6. Run the following command to perform the update and watch the pods change:

   ```
   kubectl apply -f guestbook-all-in-one.yaml && kubectl get pods -w
   ```

   This will generate an output similar to *Figure 4.24*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl apply -f guestbook-all-in-one.yaml
&& kubectl get pods -w
service/redis-master unchanged
deployment.apps/redis-master unchanged
service/redis-replica unchanged
deployment.apps/redis-replica unchanged
service/frontend unchanged
deployment.apps/frontend configured
NAME                            READY   STATUS              RESTARTS   AGE
frontend-666b4455f5-bv77x       1/1     Running             0          29s
frontend-666b4455f5-mn4x7       1/1     Running             0          27s
frontend-666b4455f5-ws2mn       1/1     Running             0          30s
frontend-74f5779d98-bb58v       0/1     ContainerCreating   0          0s
redis-master-f46ff57fd-jg6zx    1/1     Running             0          6m11s
redis-replica-5bc7bcc9c4-4f2tj  1/1     Running             0          6m11s
redis-replica-5bc7bcc9c4-d9mhn  1/1     Running             0          6m11s
frontend-74f5779d98-bb58v       1/1     Running             0          1s
frontend-666b4455f5-mn4x7       1/1     Terminating         0          28s
frontend-74f5779d98-qllgn       0/1     Pending             0          0s
frontend-74f5779d98-qllgn       0/1     Pending             0          0s
frontend-74f5779d98-qllgn       0/1     ContainerCreating   0          0s
frontend-666b4455f5-mn4x7       0/1     Terminating         0          29s
```

Figure 4.24: Pods from a new ReplicaSet are created

What you can see here is that a new version of the pod gets created (based on a new ReplicaSet). Once the new pod is running and ready, one of the old pods is terminated. This create-terminate loop is repeated until only new pods are running. In *Chapter 5, Handling common failures in AKS*, you'll see an example of such an upgrade gone wrong and you'll see that Kubernetes will not continue with the upgrade process until the new pods are healthy.

7. Running `kubectl get events | grep ReplicaSet` will show the rolling update strategy that the deployment uses to update the front-end images:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get events | grep ReplicaSet
54m         Normal   ScalingReplicaSet   deployment/frontend   Scaled up replica set frontend-666b4455f5 to 1
5m1s        Normal   ScalingReplicaSet   deployment/frontend   Scaled up replica set frontend-666b4455f5 to 3
6m5s        Normal   ScalingReplicaSet   deployment/frontend   Scaled up replica set frontend-74f5779d98 to 1
4m33s       Normal   ScalingReplicaSet   deployment/frontend   Scaled down replica set frontend-666b4455f5 to 2
```

Figure 4.25: Monitoring Kubernetes events and filtering to only see ReplicaSet-related events

> **Note**
>
> In the preceding example, you are making use of a pipe—shown by the | sign—and the `grep` command. A pipe in Linux is used to send the output of one command to the input of another command. In this case, you sent the output of `kubectl get events` to the `grep` command. Linux uses the `grep` command to filter text. In this case, you used the `grep` command to only show lines that contain the word ReplicaSet.

You can see here that the new ReplicaSet gets scaled up, while the old one gets scaled down. You will also see two ReplicaSets for the front-end, the new one replacing the other one pod at a time:

```
kubectl get replicaset
```

This will display the output shown in *Figure 4.26*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get replicaset
NAME                       DESIRED   CURRENT   READY   AGE
frontend-666b4455f5        0         0         0       12m
frontend-74f5779d98        3         3         3       8m11s
redis-master-f46ff57fd     1         1         1       12m
redis-replica-5bc7bcc9c4   2         2         2       12m
```

Figure 4.26: Two different ReplicaSets

8. Kubernetes will also keep a history of your rollout. You can see the rollout history using this command:

```
kubectl rollout history deployment frontend
```

This will generate the output shown in *Figure 4.27*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl rollout history deployment frontend
deployment.apps/frontend
REVISION  CHANGE-CAUSE
3         <none>
4         <none>
```

Figure 4.27: Deployment history of the application

9. Since Kubernetes keeps a history of the rollout, this also enables rollback. Let's do a rollback of your deployment:

```
kubectl rollout undo deployment frontend
```

This will trigger a rollback. This means that the new ReplicaSet will be scaled down to zero instances, and the old one will be scaled up to three instances again. You can verify this using the following command:

```
kubectl get replicaset
```

The resultant output is as shown in *Figure 4.28*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get replicaset
NAME                       DESIRED   CURRENT   READY   AGE
frontend-666b4455f5        3         3         3       14m
frontend-74f5779d98        0         0         0       10m
redis-master-f46ff57fd     1         1         1       14m
redis-replica-5bc7bcc9c4   2         2         2       14m
```

Figure 4.28: The old ReplicaSet now has three pods, and the new one is scaled down to zero

This shows you, as expected, that the old ReplicaSet is scaled back to three instances and the new one is scaled down to zero instances.

10. Finally, let's clean up again by running the `kubectl delete` command:

```
kubectl delete -f guestbook-all-in-one.yaml
```

Congratulations! You have completed the upgrade of an application and a rollback to a previous version.

In this example, you have used `kubectl apply` to make changes to your application. You can similarly also use `kubectl edit` to make changes, which will be explored in the next section.

Upgrading an application using kubectl edit

You can also make changes to your application running on top of Kubernetes by using `kubectl edit`. You used this previously in this chapter, in the *Manually scaling your application* section. When running `kubectl edit`, the `vi` editor will be opened for you, which will allow you to make changes directly against the object in Kubernetes.

Let's redeploy the guestbook application without a public load balancer and use `kubectl` to create the load balancer:

1. Undo the changes you made in the previous step. You can do this by using the following command:

```
git reset --hard
```

2. You will then deploy the guestbook application:

```
kubectl create -f guestbook-all-in-one.yaml
```

3. To start the edit, execute the following command:

```
kubectl edit service frontend
```

4. This will open a `vi` environment. Navigate to the line that now says `type: ClusterIP` (line 27) and change that to `type: LoadBalancer`, as shown in *Figure 4.29*. To make that change, hit the *I* button, type your changes, hit the *Esc* button, type `:wq!`, and then hit *Enter* to save the changes:

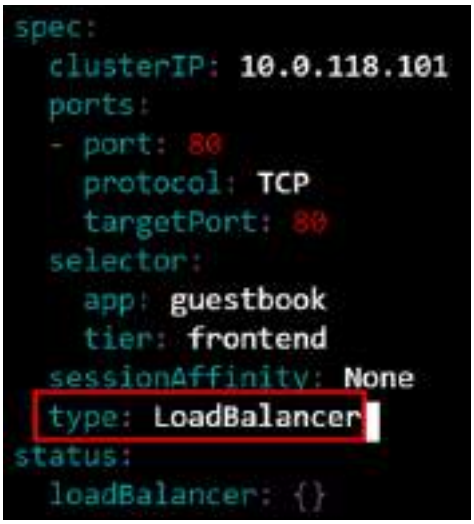


Figure 4.29: Changing this line to type: LoadBalancer

5. Once the changes are saved, you can watch the service object until the public IP becomes available. To do this, type the following:

```
kubectl get svc -w
```

6. It will take a couple of minutes to show you the updated IP. Once you see the right public IP, you can exit the `watch` command by hitting *Ctrl* + *C*.

This is an example of using `kubectl edit` to make changes to a Kubernetes object. This command will open up a text editor to interactively make changes. This means that you need to interact with the text editor to make the changes. This will not work in an automated environment. To make automated changes, you can use the `kubectl patch` command.

Upgrading an application using kubectl patch

In the previous example, you used a text editor to make the changes to Kubernetes. In this example, you will use the `kubectl patch` command to make changes to resources on Kubernetes. The `patch` command is particularly useful in automated systems when you don't have access to the original YAML file that is deployed on a cluster. It can be used, for example, in a script or in a continuous integration/continuous deployment system.

There are two main ways in which to use `kubectl patch`: either by creating a file containing your changes (called a patch file) or by providing the changes inline. Both approaches will be explained here. First, in this example, you'll change the image of the front-end from `v4` to `v3` using a patch file:

1. Start this example by creating a file called `frontend-image-patch.yaml`:

```
code frontend-image-patch.yaml
```

2. Use the following text as a patch in that file:

```
spec:
  template:
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v3
```

   This patch file uses the same YAML layout as a typical YAML file. The main thing about a patch file is that it only has to contain the changes and doesn't have to be capable of deploying the whole resource.

3. To apply the patch, use the following command:

```
kubectl patch deployment frontend \
  --patch "$(cat frontend-image-patch.yaml)"
```

   This command does two things: first, it reads the `frontend-image-patch.yaml` file using the `cat` command, and then it passes that to the `kubectl patch` command to execute the change.

4. You can verify the changes by describing the front-end deployment and looking for the `Image` section:

```
kubectl describe deployment frontend
```

This will display an output as follows:

```
Pod Template:
  Labels:  app=guestbook
           tier=frontend
  Containers:
   php-redis:
    Image:      gcr.io/google-samples/gb-frontend:v4
    Port:       80/TCP
    Host Port:  0/TCP
    Requests:
      cpu:     10m
      memory:  10Mi
```

Figure 4.30: After the patch, we are running the old image

This was an example of using the `patch` command using a patch file. You can also apply a patch directly on the command line without creating a YAML file. In this case, you would describe the change in JSON rather than in YAML.

Let's run through an example in which we will revert the image change to `v4`:

5. Run the following command to patch the image back to `v4`:

```
kubectl patch deployment frontend \
--patch='
{
    "spec": {
        "template": {
            "spec": {
                "containers": [{
                    "name": "php-redis",
                    "image": "gcr.io/google-samples/gb-frontend:v4"
                }]
            }
        }
    }
}
'
```

6. You can verify this change by describing the deployment and looking for the `Image` section:

```
kubectl describe deployment frontend
```

This will display the output shown in *Figure 4.31*:

```
Pod Template:
  Labels:  app=guestbook
           tier=frontend
  Containers:
   php-redis:
    Image:      gcr.io/google-samples/gb-frontend:v3
    Port:       80/TCP
    Host Port:  0/TCP
    Requests:
      cpu:     10m
      memory:  10Mi
```

Figure 4.31: After another patch, we are running the new version again

Before moving on to the next example, let's remove the guestbook application from the cluster:

```
kubectl delete -f guestbook-all-in-one.yaml
```

So far, you have explored three ways of upgrading Kubernetes applications. First, you made changes to the actual YAML file and applied them using `kubectl apply`. Afterward, you used `kubectl edit` and `kubectl patch` to make more changes. In the final section of this chapter, you will use Helm to upgrade an application.

Upgrading applications using Helm

This section will explain how to perform upgrades using Helm operators:

1. Run the following command:

```
helm install wp bitnami/wordpress
```

You will force an update of the image of the MariaDB container. Let's first check the version of the current image:

```
kubectl describe statefulset wp-mariadb | grep Image
```

At the time of writing, the image version is `10.5.8-debian-10-r46` as follows:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl describe statefulset wp-mariadb | grep Image
    Image:      docker.io/bitnami/mariadb:10.5.8-debian-10-r46
```

Figure 4.32: Getting the current image of the StatefulSet

Let's look at the tags from https://hub.docker.com/r/bitnami/mariadb/tags and select another tag. For example, you could select the `10.5.8-debian-10-r44` tag to update your StatefulSet.

However, in order to update the MariaDB container image, you need to get the root password for the server and the password for the database. This is because the WordPress application is configured to use these passwords to connect to the database. By default, the update using Helm on the WordPress deployment would generate new passwords. In this case, you'll be providing the existing passwords, to ensure the application remains functional.

The passwords are stored in a Kubernetes Secret object. Secrets will be explained in more depth in *Chapter 10, Storing secrets in AKS*. You can get the MariaDB passwords in the following way:

```
kubectl get secret wp-mariadb -o yaml
```

This will generate the output shown in *Figure 4.33*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get secret wp-mariadb -o yaml
apiVersion: v1
data:
  mariadb-password: OHpveVdWUmRUWA==
  mariadb-root-password: NTg4TUJrVUk2dA==
kind: Secret
metadata:
  annotations:
    meta.helm.sh/release-name: wp
    meta.helm.sh/release-namespace: default
  creationTimestamp: "2021-01-20T03:05:01Z"
  labels:
    app.kubernetes.io/instance: wp
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: mariadb
    helm.sh/chart: mariadb-9.2.2
```

Figure 4.33: The encrypted secrets that MariaDB uses

In order to get the decoded password, use the following command:

```
echo "<password>" | base64 -d
```

This will show us the decoded root password and the decoded database password, as shown in *Figure 4.34*:

Figure 4.34: The decoded root and database passwords

You also need the WordPress password. You can get that by getting the `wp-wordpress` secret and using the same decoding process:

```
kubectl get secret wp-wordpress -o yaml
echo "<WordPress password>" | base64 -d
```

2. You can update the image tag with Helm and then watch the pods change using the following command:

```
helm upgrade wp bitnami/wordpress \
--set mariadb.image.tag=10.5.8-debian-10-r44\
--set mariadb.auth.password="<decoded password>" \
--set mariadb.auth.rootPassword="<decoded password>" \
--set wordpressPassword="<decoded password>" \
&& kubectl get pods -w
```

This will update the image of MariaDB and make a new pod start. You should see an output similar to *Figure 4.35*, where you can see the previous version of the database pod being terminated, and a new one start:

```
NAME                          READY   STATUS              RESTARTS   AGE
wp-mariadb-0                  1/1     Terminating         0          3m37s
wp-wordpress-6f7c4f85b5-t9dlf 1/1     Running             0          3m37s
wp-mariadb-0                  0/1     Terminating         0          3m39s
wp-mariadb-0                  0/1     Terminating         0          3m40s
wp-mariadb-0                  0/1     Terminating         0          3m40s
wp-mariadb-0                  0/1     Pending             0          0s
wp-mariadb-0                  0/1     Pending             0          0s
wp-mariadb-0                  0/1     ContainerCreating   0          0s
wp-mariadb-0                  0/1     Running             0          27s
wp-wordpress-6f7c4f85b5-t9dlf 0/1     Running             0          4m29s
wp-wordpress-6f7c4f85b5-t9dlf 0/1     Running             1          4m39s
wp-mariadb-0                  1/1     Running             0          63s
wp-wordpress-6f7c4f85b5-t9dlf 1/1     Running             1          5m9s
```

Figure 4.35: The previous MariaDB pod gets terminated and a new one starts

Running `describe` on the new pod and grepping for `Image` will show us the new image version:

```
kubectl describe pod wp-mariadb-0 | grep Image
```

This will generate an output as shown in *Figure 4.36*:

```
user@Azure:~$ kubectl describe pod wp-mariadb-0 | grep Image
    Image:          docker.io/bitnami/mariadb:10.5.8-debian-10-r44
    Image ID:       docker.io/bitnami/mariadb@sha256:02ea62312a3b05
```

Figure 4.36: Showing the new image

3. Finally, clean up by running the following command:

```
helm delete wp
kubectl delete pvc --all
kubectl delete pv --all
```

You have now learned how to upgrade an application using Helm. As you have seen in this example, upgrading using Helm can be done by using the `--set` operator. This makes performing upgrades and multiple deployments using Helm efficient.

Summary

This a chapter covered a plethora of information on building scalable applications. The goal was to show you how to scale deployments with Kubernetes, which was achieved by creating multiple instances of your application.

We started the chapter by looking at how to define the use of a load balancer and leverage the deployment scale feature in Kubernetes to achieve scalability. With this type of scalability, you can also achieve failover by using a load balancer and multiple instances of the software for stateless applications. We also looked into using the HPA to automatically scale your deployment based on load.

After that, we looked at how you can scale the cluster itself. First, we manually scaled the cluster, and afterward we used a cluster autoscaler to scale the cluster based on application demand.

We finished the chapter by looking into different ways to upgrade a deployed application: first, by exploring updating YAML files manually, and then by learning two additional `kubectl` commands (`edit` and `patch`) that can be used to make changes. Finally, we learned how Helm can be used to perform these upgrades.

In the next chapter, we will look at a couple of common failures that you may face while deploying applications to AKS and how to fix them.

5

Handling common failures in AKS

Kubernetes is a distributed system with many working parts. AKS abstracts most of it for you, but it is still your responsibility to know where to look and how to respond when bad things happen. Much of the failure handling is done automatically by Kubernetes; however, you will encounter situations where manual intervention is required.

There are two areas where things can go wrong in an application that is deployed on top of AKS. Either the cluster itself has issues, or the application deployed on top of the cluster has issues. This chapter focuses specifically on cluster issues. There are several things that can go wrong with a cluster.

The first thing that can go wrong is a node in the cluster can become unavailable. This can happen either due to an Azure infrastructure outage or due to an issue with the virtual machine itself, such as an operating system crash. Either way, Kubernetes monitors the cluster for node failures and will recover automatically. You will see this process in action in this chapter.

A second common issue in a Kubernetes cluster is out-of-resource failures. This means that the workload you are trying to deploy requires more resources than are available on your cluster. You will learn how to monitor these signals and how you can solve them.

Another common issue is problems with mounting storage, which happens when a node becomes unavailable. When a node in Kubernetes becomes unavailable, Kubernetes will not detach the disks attached to this failed node. This means that those disks cannot be used by workloads on other nodes. You will see a practical example of this and learn how to recover from this failure.

We will look into the following topics in depth in this chapter:

- Handling node failures
- Solving out-of-resource failures
- Handling storage mount issues

In this chapter, you will learn about common failure scenarios, as well as solutions to those scenarios. To start, we will introduce node failures.

> **Note:**
>
> Refer to Kubernetes the Hard Way (https://github.com/kelseyhightower/kubernetes-the-hard-way), an excellent tutorial, to get an idea about the blocks on which Kubernetes is built. For the Azure version, refer to Kubernetes the Hard Way – Azure Translation (https://github.com/ivanfioravanti/kubernetes-the-hard-way-on-azure).

Handling node failures

Intentionally (to save costs) or unintentionally, nodes can go down. When that happens, you don't want to get the proverbial 3 a.m. call that your system is down. Kubernetes can handle moving workloads on failed nodes automatically for you instead. In this exercise, you are going to deploy the guestbook application and bring a node down in your cluster to see what Kubernetes does in response:

1. Ensure that your cluster has at least two nodes:

```
kubectl get nodes
```

This should generate an output as shown in *Figure 5.1*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-agentpool-39838025-vmss000000   Ready    agent   82m   v1.19.6
aks-agentpool-39838025-vmss000002   Ready    agent   22m   v1.19.6
```

Figure 5.1: List of nodes in the cluster

If you don't have two nodes in your cluster, look for your cluster in the Azure portal, navigate to **Node pools**, select the pool you wish to scale, and click on **Scale**. You can then scale **Node count** to **2** nodes as shown in *Figure 5.2*:

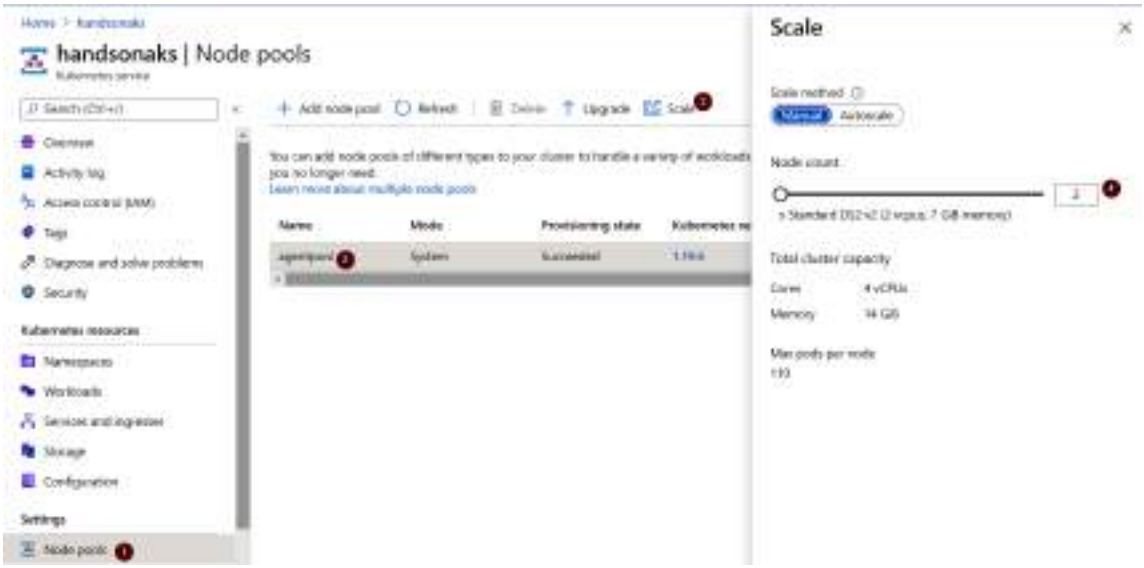


Figure 5.2: Scaling the cluster

2. As an example application in this section, deploy the guestbook application. The YAML file to deploy this has been provided in the source code for this chapter (`guestbook-all-in-one.yaml`). To deploy the guestbook application, use the following command:

```
kubectl create -f guestbook-all-in-one.yaml
```

3. Watch the `service` object until the public IP becomes available. To do this, type the following:

```
kubectl get service -w
```

> **Note**
>
> You can also get services in Kubernetes by using `kubectl get svc` rather than the full `kubectl get service`.

4. This will take a couple of seconds to show you the updated external IP. *Figure* 5.3 shows the service's public IP. Once you see the public IP appear (**20.72.244.113** in this case), you can exit the watch command by hitting *Ctrl* + C:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get service -w
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
frontend       LoadBalancer   10.0.23.47     <pending>       80:30619/TCP   4s
kubernetes     ClusterIP      10.0.0.1       <none>          443/TCP        27h
redis-master   ClusterIP      10.0.184.142   <none>          6379/TCP       4s
redis-replica  ClusterIP      10.0.218.85    <none>          6379/TCP       4s
frontend       LoadBalancer   10.0.23.47     20.72.244.113   80:30619/TCP   5s
```

Figure 5.3: The external IP of the frontend service changes from <pending> to an actual IP address

5. Go to `http://<EXTERNAL-IP>` (`http://20.72.244.113` in this case) as shown in *Figure* 5.4:

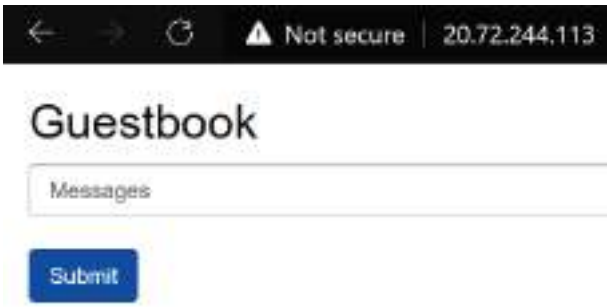


Figure 5.4: Browsing to the guestbook application

6. Let's see where the pods are currently running using the following command:

```
kubectl get pods -o wide
```

This will generate an output as shown in *Figure* 5.5:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE   IP            NODE
frontend-766d4f77cb-9w9t2         1/1     Running   0          42s   10.244.0.54   aks-agentpool-39858825-vmss000002
frontend-766d4f77cb-vkc2l         1/1     Running   0          42s   10.244.0.53   aks-agentpool-39858825-vmss000002
frontend-766d4f77cb-z7s54         1/1     Running   0          42s   10.244.1.79   aks-agentpool-39858825-vmss000000
redis-master-f46ff57fd-hmwr4      1/1     Running   0          42s   10.244.0.55   aks-agentpool-39858825-vmss000002
redis-replica-786bd64556-hf4kd    1/1     Running   0          42s   10.244.0.56   aks-agentpool-39858825-vmss000002
redis-replica-786bd64556-l72z7    1/1     Running   0          42s   10.244.1.80   aks-agentpool-39858825-vmss000000
```

Figure 5.5: The pods are spread between node 0 and node 2

This shows you that you should have the workload spread between node 0 and node 2.

> **Note**
>
> In the example shown in *Figure 5.5*, the workload is spread between nodes 0 and 2. You might notice that node 1 is missing here. If you followed the example in *Chapter 4, Building scalable applications*, your cluster should be in a similar state. The reason for this is that as Azure removes old nodes and adds new nodes to a cluster (as you did in *Chapter 4, Building scalable applications*), it keeps incrementing the node counter.

7. Before introducing the node failures, there are two optional steps you can take to verify whether your application can continue to run. You can run the following command to hit the guestbook front end every 5 seconds and get the HTML. It's recommended to open this in a new Cloud Shell window:

```
while true; do
  curl -m 1 http://<EXTERNAl-IP>/;
  sleep 5;
done
```

> **Note**
>
> The preceding command will keep calling your application till you press *Ctrl* + *C*. There might be intermittent times where you don't get a reply, which is to be expected as Kubernetes takes a couple of minutes to rebalance the system.

You can also add some guestbook entries to see what happens to them when you cause the node to shut down. This will display an output as shown in *Figure* 5.6:

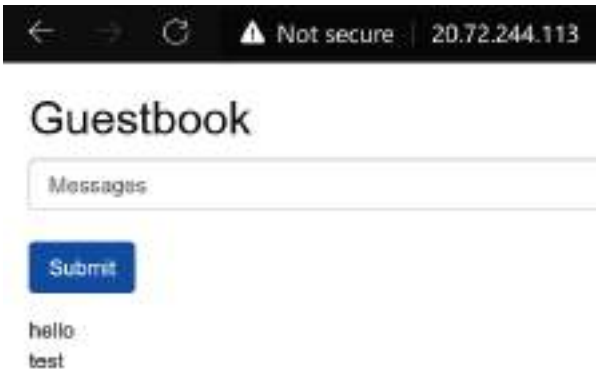


Figure 5.6: Writing a couple of messages in the guestbook

8. In this example, you are exploring how Kubernetes handles a node failure. To demonstrate this, shut down a node in the cluster. You can shut down either node, although for maximum impact it is recommended you shut down the node from *step* 6 that hosted the most pods. In the case of the example shown, node 2 will be shut down.

   To shut down this node, look for **VMSS** (**virtual machine scale sets**) in the Azure search bar, and select the scale set used by your cluster, as shown in *Figure* 5.7. If you have multiple scale sets in your subscription, select the one whose name corresponds to the node names shown in *Figure* 5.5:

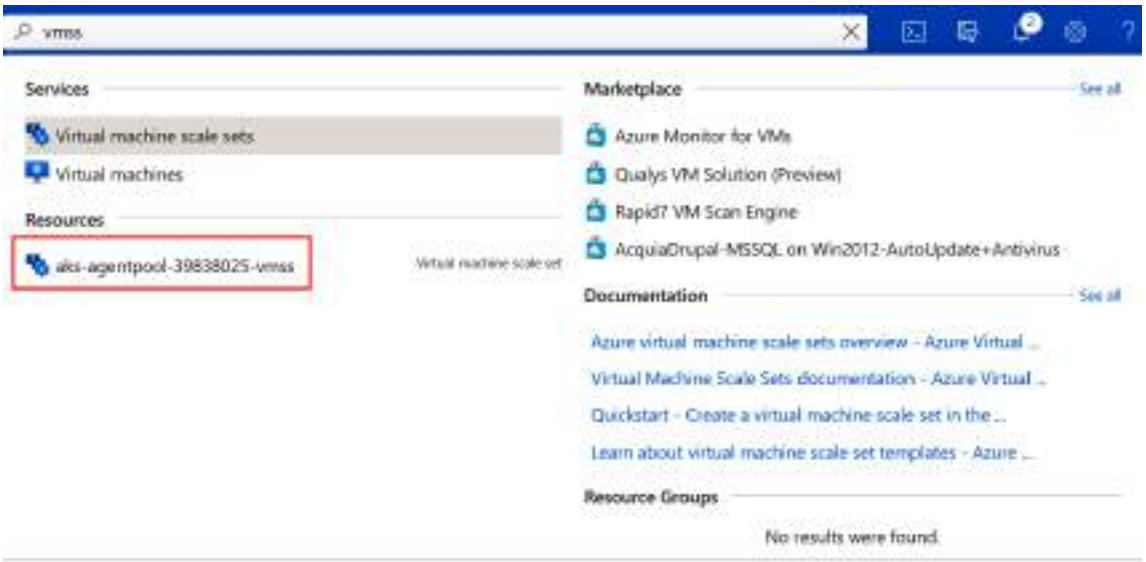


Figure 5.7: Looking for the scale set hosting your cluster

After navigating to the pane of the scale set, go to the **Instances** view, select the instance you want to shut down, and then hit the **Stop** button, as shown in *Figure 5.8*:

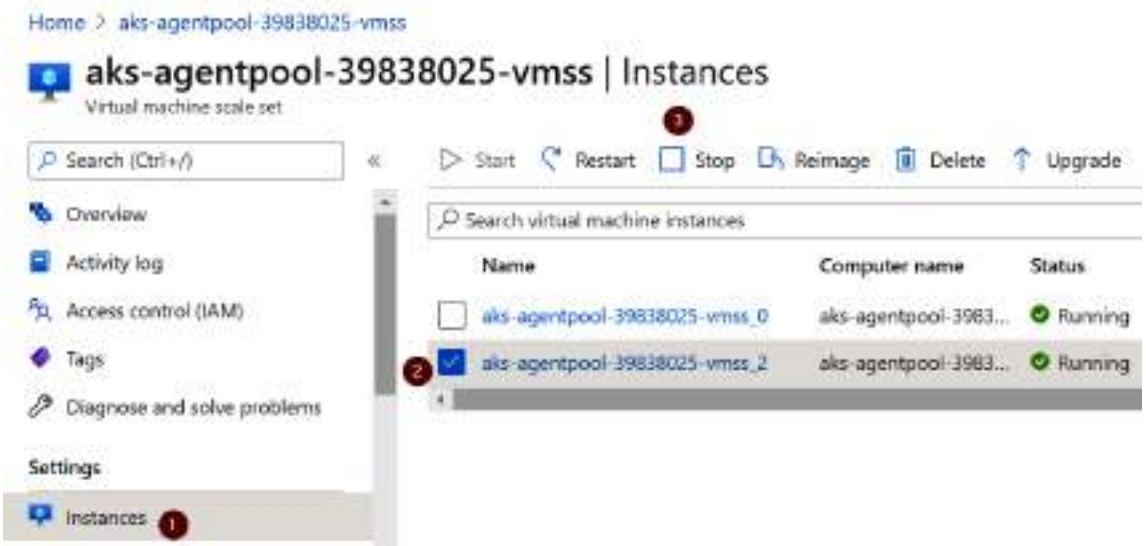


Figure 5.8: Shutting down node 2

This will shut down the node. To see how Kubernetes will react with your pods, you can watch the pods in your cluster via the following command:

```
kubectl get pods -o wide -w
```

After a while, you should notice additional output, showing you that the pods got rescheduled on the healthy host, as shown in *Figure* 5.9:

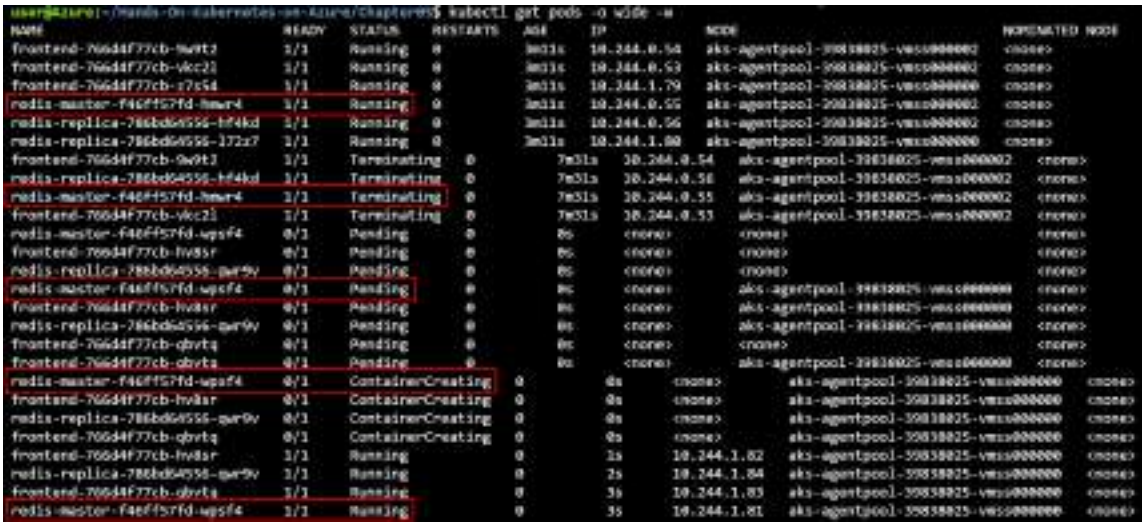


Figure 5.9: The pods from the failed node getting recreated on a healthy node

What you see here is the following:

- The Redis master pod running on **node 2** got terminated as the host became unhealthy.
- A new Redis master pod got created, on host **0**. This went through the stages **Pending**, **ContainerCreating**, and then **Running**.

> **Note**
>
> In the preceding example, Kubernetes picked up that the host was unhealthy before it rescheduled the pods. If you were to do `kubectl get nodes`, you would see node **2** is in a **NotReady** state. There is a configuration in Kubernetes called `pod-eviction-timeout` that defines how long the system will wait to reschedule pods on a healthy host. The default is 5 minutes.

9. If you recorded a number of messages in the guestbook during *step* 7, browse back to the guestbook application on its public IP. What you can see is that all your precious messages are gone! This shows the importance of having **PersistentVolumeClaims** (**PVCs**) for any data that you want to survive in the case of a node failure, which is not the case in our application here. You will see an example of this in the last section of this chapter.

In this section, you learned how Kubernetes automatically handles node failures by recreating pods on healthy nodes. In the next section, you will learn how you can diagnose and solve out-of-resource issues.

Solving out-of-resource failures

Another common issue that can come up with Kubernetes clusters is the cluster running out of resources. When the cluster doesn't have enough CPU power or memory to schedule additional pods, pods will become stuck in a `Pending` state. You have seen this behavior in *Chapter* 4, *Building scalable applications*, as well.

Kubernetes uses requests to calculate how much CPU power or memory a certain pod requires. The guestbook application has requests defined for all the deployments. If you open the `guestbook-all-in-one.yaml` file in the folder `Chapter05`, you'll see the following for the `redis-replica` deployment:

```
63  kind: Deployment
64  metadata:
65    name: redis-replica
...
83          resources:
84            requests:
85              cpu: 200m
86              memory: 100Mi
```

This section explains that every pod for the `redis-replica` deployment requires `200m` of a CPU core (`200` milli or `20%`) and `100MiB` (Mebibyte) of memory. In your 2 CPU clusters (with node 1 shut down), scaling this to 10 pods will cause issues with the available resources. Let's look into this:

> **Note**
>
> In Kubernetes, you can use either the binary prefix notation or the base 10 notation to specify memory and storage. Binary prefix notation means using KiB (kibibyte) to represent 1,024 bytes, MiB (mebibyte) to represent 1,024 KiB, and Gib (gibibyte) to represent 1,024 MiB. Base 10 notation means using kB (kilobyte) to represent 1,000 bytes, MB (megabyte) to represent 1,000 kB, and GB (gigabyte) represents 1,000 MB.

1. Let's start by scaling the `redis-replica` deployment to 10 pods:

```
kubectl scale deployment/redis-replica --replicas=10
```

2. This will cause a couple of new pods to be created. We can check our pods using the following:

```
kubectl get pods
```

This will generate an output as shown in *Figure 5.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl scale deployment/redis-replica --replicas=10
deployment.apps/redis-replica scaled
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods
NAME                             READY   STATUS        RESTARTS   AGE
frontend-766d4f77cb-9w9t2        1/1     Terminating   0          9m55s
frontend-766d4f77cb-hv8sr        1/1     Running       0          2m24s
frontend-766d4f77cb-qbvtq        1/1     Running       0          2m24s
frontend-766d4f77cb-vkc2l        1/1     Terminating   0          9m55s
frontend-766d4f77cb-z7s54        1/1     Running       0          9m55s
redis-master-f46ff57fd-hmwr4     1/1     Terminating   0          9m55s
redis-master-f46ff57fd-wpsf4     1/1     Running       0          2m24s
redis-replica-786bd64556-2t8ms   1/1     Running       0          5s
redis-replica-786bd64556-7k7cn   0/1     Pending       0          5s
redis-replica-786bd64556-7shvm   0/1     Pending       0          4s
redis-replica-786bd64556-dv7qv   0/1     Pending       0          5s
redis-replica-786bd64556-hf4kd   1/1     Terminating   0          9m55s
redis-replica-786bd64556-jdfxj   0/1     Pending       0          5s
redis-replica-786bd64556-l72z7   1/1     Running       0          9m55s
redis-replica-786bd64556-qwr9v   1/1     Running       0          2m24s
redis-replica-786bd64556-r8j6b   1/1     Running       0          5s
redis-replica-786bd64556-xk82s   1/1     Running       0          5s
redis-replica-786bd64556-zgfjq   0/1     Pending       0          5s
```

Figure 5.10: Some pods are in the Pending state

Highlighted here is one of the pods that are in the **Pending** state. This occurs if the cluster is out of resources.

3. We can get more information about these pending pods using the following command:

```
kubectl describe pod redis-replica-<pod-id>
```

   This will show you more details. At the bottom of the `describe` command, you should see something like what's shown in *Figure 5.11*:

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  104s  default-scheduler  0/2 nodes are available: 1 Insufficient cpu
, 1 node(s) had taint {node.kubernetes.io/unreachable: }, that the pod didn't tolerate.
  Warning  FailedScheduling  104s  default-scheduler  0/2 nodes are available: 1 Insufficient cpu
, 1 node(s) had taint {node.kubernetes.io/unreachable: }, that the pod didn't tolerate.
```

Figure 5.11: Kubernetes is unable to schedule this pod

   It explains two things:

   - One of the nodes is out of CPU resources.
   - One of the nodes has a taint (node.kubernetes.io/unreachable) that the pod didn't tolerate. This means that the node that is `NotReady` can't accept pods.

4. We can solve this capacity issue by starting up node **2** as shown in *Figure 5.12*. This can be done in a way similar to the shutdown process:

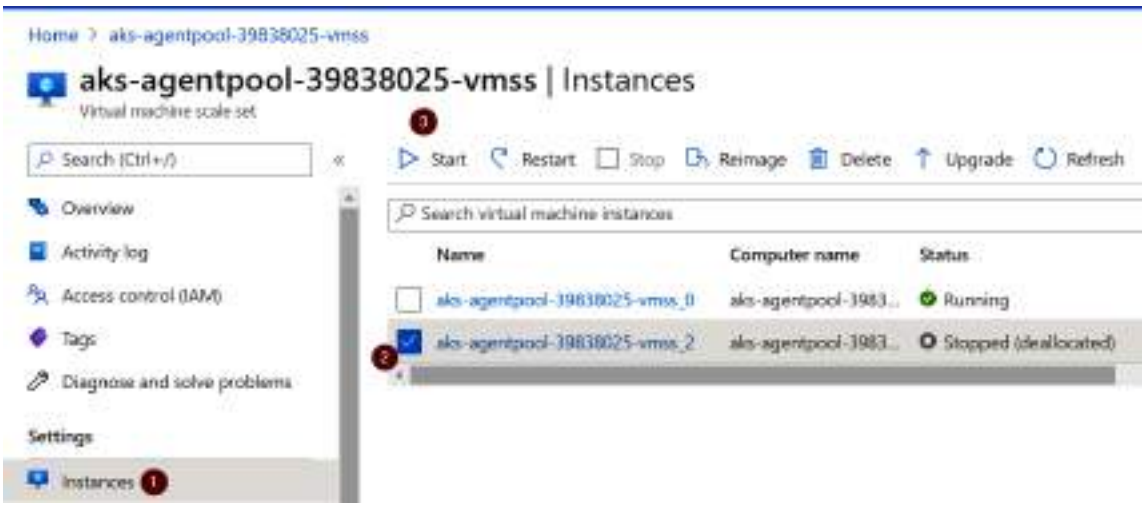


Figure 5.12: Start node 2 again

5. It will take a couple of minutes for the other node to become available again in Kubernetes. You can monitor the progress on the pods by executing the following command:

```
kubectl get pods -w
```

This will show you an output after a couple of minutes similar to *Figure* 5.13:

```
redis-replica-786bd64556-7k7cn   0/1     Pending             0          2m29s
redis-replica-786bd64556-dv7qv   0/1     Pending             0          2m29s
redis-replica-786bd64556-jdfxj   0/1     Pending             0          2m29s
redis-replica-786bd64556-7shvm   0/1     Pending             0          2m28s
redis-replica-786bd64556-zgfjq   0/1     Pending             0          2m29s
redis-replica-786bd64556-7k7cn   0/1     ContainerCreating   0          2m29s
redis-replica-786bd64556-dv7qv   0/1     ContainerCreating   0          2m29s
redis-replica-786bd64556-jdfxj   0/1     ContainerCreating   0          2m29s
redis-replica-786bd64556-7shvm   0/1     ContainerCreating   0          2m28s
redis-replica-786bd64556-zgfjq   0/1     ContainerCreating   0          2m30s
redis-replica-786bd64556-7k7cn   1/1     Running             0          2m30s
redis-replica-786bd64556-zgfjq   1/1     Running             0          2m31s
redis-replica-786bd64556-dv7qv   1/1     Running             0          2m31s
redis-replica-786bd64556-jdfxj   1/1     Running             0          2m31s
redis-replica-786bd64556-7shvm   1/1     Running             0          2m31s
```

Figure 5.13: Pods move from a Pending state to ContainerCreating to Running

Here again, you see the container status change from **Pending**, to **ContainerCreating**, to finally **Running**.

6. If you re-execute the `describe` command on the previous pod, you'll see an output like what's shown in *Figure* 5.14:

```
Events:
  Type     Reason            Age    From               Message
  ----     ------            ----   ----               -------
  Warning  FailedScheduling  4m48s  default-scheduler  0/2 nodes are available: 1 Insufficient cpu, 1 node(s) had taint
 {node.kubernetes.io/unreachable: }, that the pod didn't tolerate.
  Warning  FailedScheduling  4m48s  default-scheduler  0/2 nodes are available: 1 Insufficient cpu, 1 node(s) had taint
 {node.kubernetes.io/unreachable: }, that the pod didn't tolerate.
  Normal   Scheduled         2m20s  default-scheduler  Successfully assigned default/redis-replica-786bd64556-7k7cn to
aks-agentpool-39838025-vmss000002
  Normal   Pulled            2m20s  kubelet            Container image "gcr.io/google_samples/gb-redis-follower:v1" alr
eady present on machine
  Normal   Created           2m19s  kubelet            Created container replica
  Normal   Started           2m19s  kubelet            Started container replica
```

Figure 5.14: When the node is available again, the Pending pods are assigned to that node

This shows that after node 2 became available, Kubernetes scheduled the pod on that node, and then started the container.

In this section, you learned how to diagnose out-of-resource errors. You were able to solve the error by adding another node to the cluster. Before moving on to the final failure mode, clean up the guestbook deployment.

> **Note**
>
> In *Chapter 4, Building scalable applications*, the **cluster autoscaler** was introduced. The cluster autoscaler will monitor out-of-resource errors and add new nodes to the cluster automatically.

Let's clean up the guestbook deployment by running the following `delete` command:

```
kubectl delete -f guestbook-all-in-one.yaml
```

It is now also safe to close the other Cloud Shell window you opened earlier.

So far, you have learned how to recover from two failure modes for nodes in a Kubernetes cluster. First, you saw how Kubernetes handles a node going offline and how the system reschedules pods to a working node. After that, you saw how Kubernetes uses requests to manage the scheduling of pods on a node, and what happens when a cluster is out of resources. In the next section, you'll learn about another failure mode in Kubernetes, namely what happens when Kubernetes encounters storage mounting issues.

Fixing storage mount issues

Earlier in this chapter, you noticed how the guestbook application lost data when the Redis master was moved to another node. This happened because that sample application didn't use any persistent storage. In this section, you'll see an example of how PVCs can be used to prevent data loss when Kubernetes moves a pod to another node. You will see a common error that occurs when Kubernetes moves pods with PVCs attached, and you'll learn how to fix this.

For this, you will reuse the WordPress example from the previous chapter. Before starting, let's make sure that the cluster is in a clean state:

```
kubectl get all
```

This should show you just the one Kubernetes service, as in *Figure 5.15*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.0.0.1     <none>        443/TCP   87m
```

Figure 5.15: You should only have the one Kubernetes service running for now

Let's also ensure that both nodes are running and **Ready**:

```
kubectl get nodes
```

This should show us both nodes in a **Ready** state, as in *Figure 5.16*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-agentpool-39838025-vmss000000   Ready    agent   86m   v1.19.6
aks-agentpool-39838025-vmss000002   Ready    agent   26m   v1.19.6
```

Figure 5.16: You should have two nodes available in your cluster

In the previous example, under the *Handling node failures* section, you saw that the messages stored in `redis-master` are lost if the pod gets restarted. The reason for this is that `redis-master` stores all data in its container, and whenever it is restarted, it uses the clean image without the data. In order to survive reboots, the data has to be stored outside. Kubernetes uses PVCs to abstract the underlying storage provider to provide this external storage.

To start this example, set up the WordPress installation.

Starting the WordPress installation

Let's start by installing WordPress. We will demonstrate how it works and then verify that storage is still present after a reboot.

If you have not done so yet in a previous chapter, add the Helm repository for Bitnami:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Begin reinstallation by using the following command:

```
helm install wp bitnami/wordpress
```

This will take a couple of minutes to process. You can follow the status of this installation by executing the following command:

```
kubectl get pods -w
```

After a couple of minutes, this should show you two pods with a status of **Running** and with a ready status of **1/1** for both pods, as shown in *Figure 5.17*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -w
NAME                          READY   STATUS              RESTARTS   AGE
wp-mariadb-0                  0/1     Pending             0          3s
wp-wordpress-6f7c4f85b5-4p264 0/1     Pending             0          3s
wp-wordpress-6f7c4f85b5-4p264 0/1     Pending             0          15s
wp-wordpress-6f7c4f85b5-4p264 0/1     ContainerCreating   0          15s
wp-mariadb-0                  0/1     Pending             0          19s
wp-mariadb-0                  0/1     ContainerCreating   0          19s
wp-wordpress-6f7c4f85b5-4p264 0/1     Running             0          95s
wp-mariadb-0                  0/1     Running             0          110s
wp-wordpress-6f7c4f85b5-4p264 0/1     Error               0          2m27s
wp-wordpress-6f7c4f85b5-4p264 0/1     Running             1          2m28s
wp-mariadb-0                  1/1     Running             0          2m29s
wp-wordpress-6f7c4f85b5-4p264 1/1     Running             1          3m4s
```

Figure 5.17: All pods will have the status of Running after a couple of minutes

You might notice that the `wp-wordpress` pod went through an Error status and was restarted afterward. This is because the `wp-mariadb` pod was not ready in time, and `wp-wordpress` went through a restart. You will learn more about readiness and how this can influence pod restarts in *Chapter 7, Monitoring the AKS cluster and the application*.

In this section, you saw how to install WordPress. Now, you will see how to avoid data loss using persistent volumes.

Using persistent volumes to avoid data loss

A **persistent volume** (**PV**) is the way to store persistent data in the cluster with Kubernetes. PVs were discussed in more detail in *Chapter 3*, *Application deployment on AKS*. Let's explore the PVs created for the WordPress deployment:

1. You can get the PersistentVolumeClaims using the following command:

   ```
   kubectl get pvc
   ```

   This will generate an output as shown in *Figure 5.18*:

Figure 5.18: Two PVCs are created by the WordPress deployment

   A PersistentVolumeClaim will result in the creation of a PersistentVolume. The PersistentVolume is the link to the physical resource created, which is an Azure disk in this case. The following command shows the actual PVs that are created:

   ```
   kubectl get pv
   ```

   This will show you the two PersistentVolumes:

Figure 5.19: Two PVs are created to store the data of the PVCs

   You can get more details about the specific PersistentVolumes that were created. Copy the name of one of the PVs, and run the following command:

   ```
   kubectl describe pv <pv name>
   ```

This will show you the details of that volume, as in *Figure* 5.20:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl describe pv pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997
Name:              pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997
Labels:            failure-domain.beta.kubernetes.io/region=westus2
Annotations:       pv.kubernetes.io/bound-by-controller: yes
                   pv.kubernetes.io/provisioned-by: kubernetes.io/azure-disk
                   volumehelper.VolumeDynamicallyCreatedByKey: azure-disk-dynamic-provisioner
Finalizers:        [kubernetes.io/pv-protection]
StorageClass:      default
Status:            Bound
Claim:             default/wp-wordpress
Reclaim Policy:    Delete
Access Modes:      RWO
VolumeMode:        Filesystem
Capacity:          10Gi
Node Affinity:
  Required Terms:
    Term 0:        failure-domain.beta.kubernetes.io/region in [westus2]
Message:
Source:
    Type:          AzureDisk (an Azure Data Disk mount on the host and bind mount to the pod)
    DiskName:      kubernetes-dynamic-pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997
    DiskURI:       /subscriptions/ede7a1e5-4121-427f-876e-e100eba9890/resourceGroups/mc_rg-handsonaks_handsonaks_
westus2/providers/Microsoft.Compute/disks/kubernetes-dynamic-pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997
    Kind:          Managed
    FSType:
    CachingMode:   ReadOnly
    ReadOnly:      false
Events:            <none>
```

Figure 5.20: The details of one of the PVs

Here, you can see which PVC has claimed this volume and what the **DiskName** is in Azure.

2. Verify that your site is working:

```
kubectl get service
```

This will show us the public IP of our WordPress site, as seen in *Figure* 5.21:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get service
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)                      AGE
kubernetes     ClusterIP      10.0.0.1       <none>         443/TCP                      7d14h
wp-mariadb     ClusterIP      10.0.204.3     <none>         3306/TCP                     17m
wp-wordpress   LoadBalancer   10.0.189.10    20.72.222.87   80:32239/TCP,443:30828/TCP   17m
```

Figure 5.21: Public IP of the WordPress site

3. If you remember from *Chapter 3, Application deployment of AKS*, Helm showed you the commands you need to get the admin credentials for our WordPress site. Let's grab those commands and execute them to log on to the site as follows:

```
helm status wp
echo Username: user
echo Password: $(kubectl get secret --namespace default wp-wordpress
-o jsonpath="{.data.wordpress-password}" | base64 -d)
```

This will show you the **username** and **password**, as displayed in *Figure 5.22*:

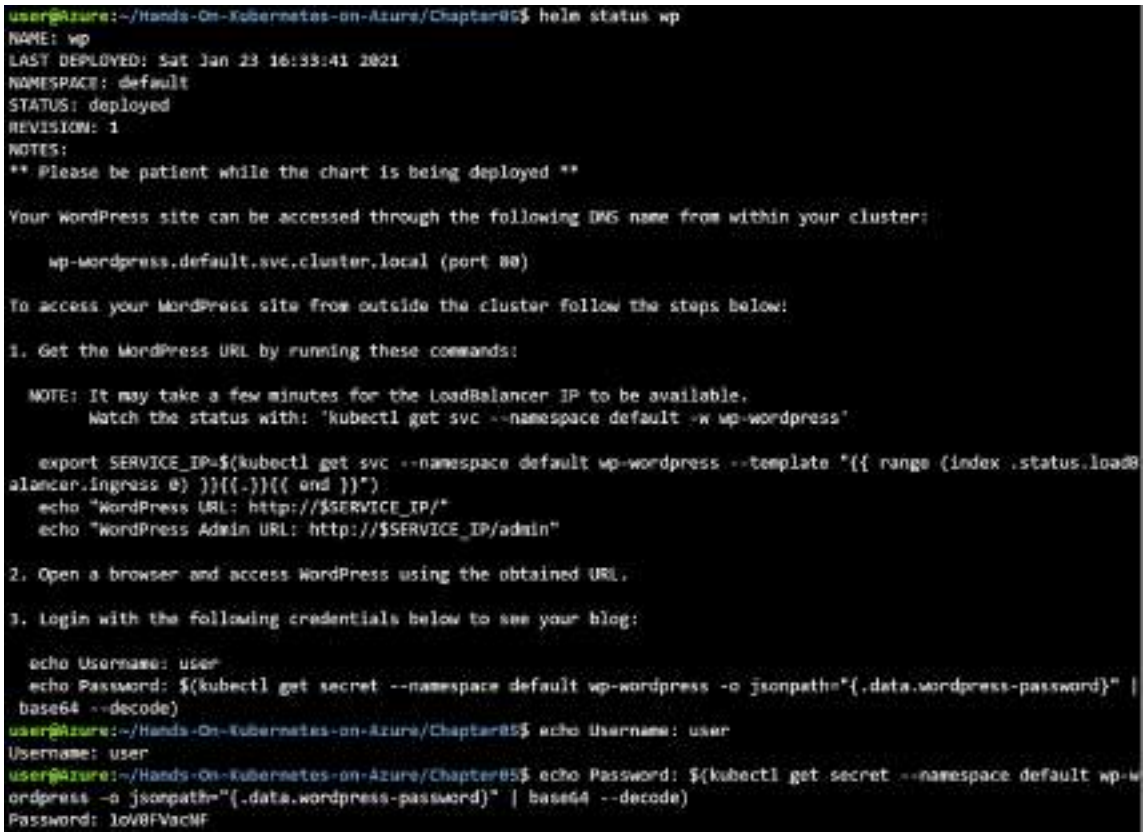


Figure 5.22: Getting the username and password for the WordPress application

You can log in to our site via the following address: `http://<external-ip>/admin`. Log in here with the credentials from the previous step. Then you can go ahead and add a post to your website. Click the **Write your first blog post** button, and then create a short post, as shown in *Figure* 5.23:

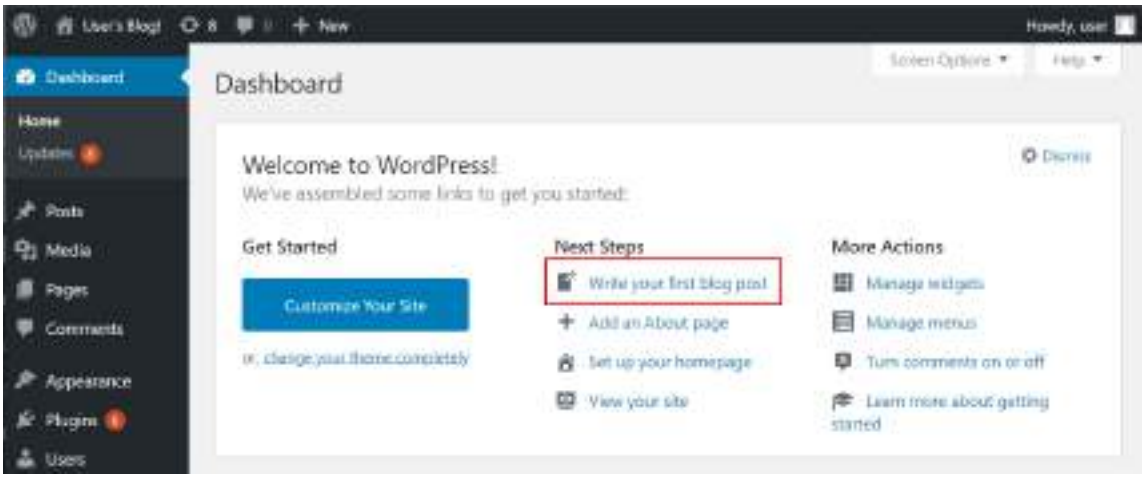


Figure 5.23: Writing your first blog post

Type some text now and hit the **Publish** button, as shown in *Figure* 5.24. The text itself isn't important; you are writing this to verify that data is indeed persisted to disk:

Figure 5.24: Publishing a post with random text

If you now head over to the main page of your website at `http://<external-ip>`, you'll see your test post as shown in *Figure* 5.25:

Figure 5.25: The published blog post appears on the home page

In this section, you deployed a WordPress site, you logged in to your WordPress site, and you created a post. You will verify whether this post survives a node failure in the next section.

Handling pod failure with PVC involvement

The first test you'll do with the PVCs is to kill the pods and verify whether the data has indeed persisted. To do this, let's do two things:

1. **Watch the pods in your application**: To do this, use the current Cloud Shell and execute the following command:

   ```
   kubectl get pods -w
   ```

2. **Kill the two pods that have the PVC mounted**: To do this, create a new Cloud Shell window by clicking on the icon shown in *Figure* 5.26:

Figure 5.26: Opening a new Cloud Shell instance

Once you open a new Cloud Shell, execute the following command:

```
kubectl delete pod --all
```

In the original Cloud Shell, follow along with the `watch` command that you executed earlier. You should see an output like what's shown in *Figure 5.27*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -w
NAME                            READY   STATUS              RESTARTS   AGE
wp-mariadb-0                    1/1     Running             0          2m27s
wp-wordpress-6f7c4f85b5-gq27t   1/1     Running             0          2m40s
wp-mariadb-0                    1/1     Terminating         0          2m29s
wp-wordpress-6f7c4f85b5-gq27t   1/1     Terminating         0          2m42s
wp-wordpress-6f7c4f85b5-xm5bb   0/1     Pending             0          0s
wp-wordpress-6f7c4f85b5-xm5bb   0/1     Pending             0          0s
wp-wordpress-6f7c4f85b5-xm5bb   0/1     ContainerCreating   0          0s
wp-mariadb-0                    0/1     Terminating         0          2m30s
wp-wordpress-6f7c4f85b5-gq27t   0/1     Terminating         0          2m43s
wp-wordpress-6f7c4f85b5-gq27t   0/1     Terminating         0          2m48s
wp-wordpress-6f7c4f85b5-gq27t   0/1     Terminating         0          2m48s
wp-mariadb-0                    0/1     Terminating         0          2m40s
wp-mariadb-0                    0/1     Terminating         0          2m40s
wp-mariadb-0                    0/1     Pending             0          0s
wp-mariadb-0                    0/1     Pending             0          0s
wp-mariadb-0                    0/1     ContainerCreating   0          0s
wp-wordpress-6f7c4f85b5-xm5bb   0/1     Running             0          66s
wp-mariadb-0                    0/1     Running             0          66s
wp-mariadb-0                    1/1     Running             0          98s
wp-wordpress-6f7c4f85b5-xm5bb   1/1     Running             0          111s
```

Figure 5.27: After deleting the pods, Kubernetes will automatically recreate both pods

As you can see, the two original pods went into a **Terminating** state. Kubernetes quickly started creating new pods to recover from the pod outage. The pods went through a similar life cycle as the original ones, going from **Pending** to **ContainerCreating** to **Running**.

3. If you head on over to your website, you should see that your demo post has been persisted. This is how PVCs can help you prevent data loss, as they persist data that would not have been persisted in the pod itself.

In this section, you've learned how PVCs can help when pods get recreated on the same node. In the next section, you'll see how PVCs are used when a node has a failure.

Handling node failure with PVC involvement

In the previous example, you saw how Kubernetes can handle pod failures when those pods have a PV attached. In this example, you'll learn how Kubernetes handles node failures when a volume is attached:

1. Let's first check which node is hosting your application, using the following command:

   ```
   kubectl get pods -o wide
   ```

   In the example shown in *Figure* 5.28, node **2** was hosting **MariaDB**, and node **0** was hosting the **WordPress** site:

Figure 5.28: Check which node hosts the WordPress site

2. Introduce a failure and stop the node that is hosting the **WordPress** pod using the Azure portal. You can do this in the same way as in the earlier example. First, look for the scale set backing your cluster, as shown in *Figure* 5.29:

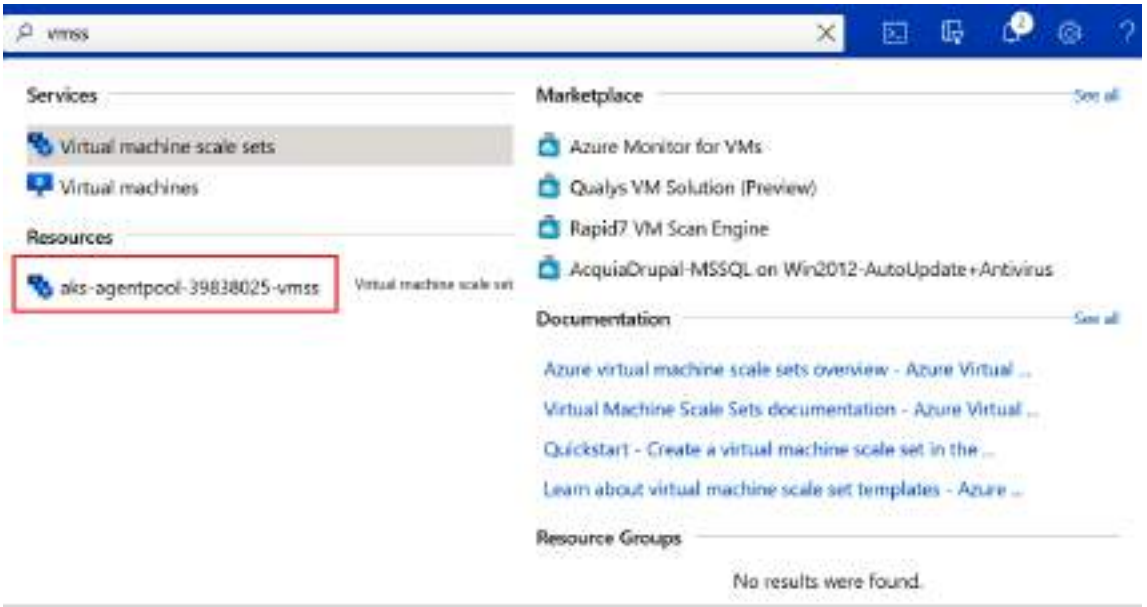


Figure 5.29: Looking for the scale set hosting your cluster

3. Then shut down the node, by clicking on **Instances** in the left-hand menu, then selecting the node you need to shut down and clicking the **Stop** button, as shown in *Figure* 5.30:

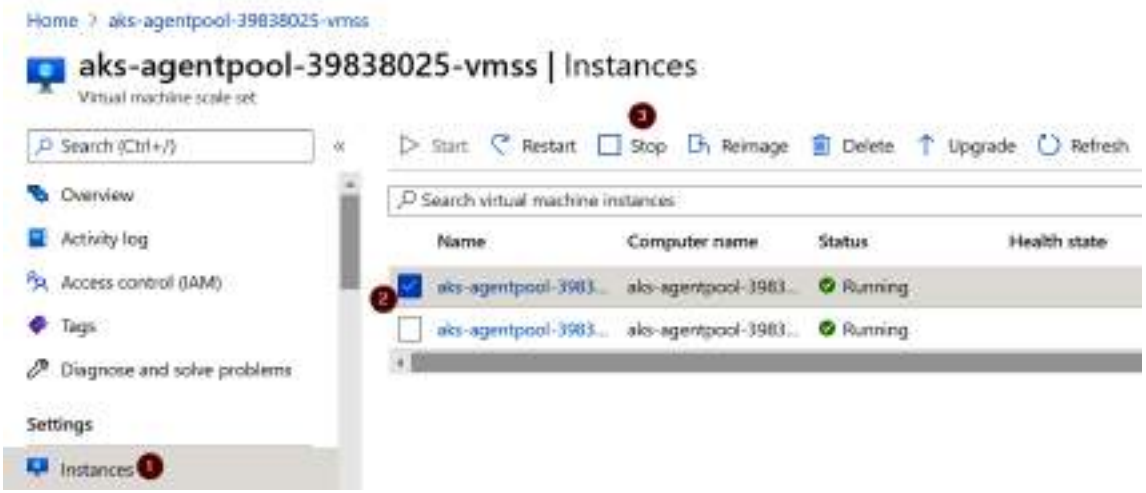


Figure 5.30: Shutting down the node

4. After this action, once again, watch the pods to see what is happening in the cluster:

```
kubectl get pods -o wide -w
```

   As in the previous example, it is going to take 5 minutes before Kubernetes will start taking action against the failed node. You can see that happening in *Figure* 5.31:

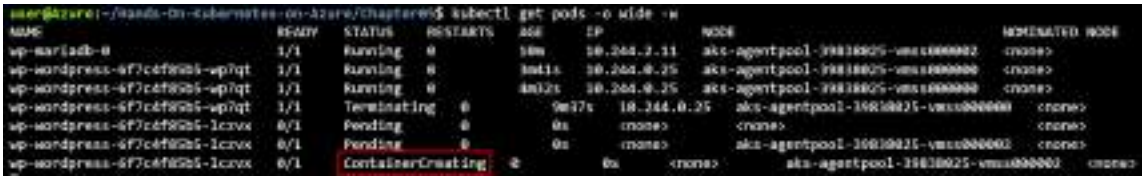


Figure 5.31: A pod in a ContainerCreating state

5. You are seeing a new issue here. The new pod is stuck in a **ContainerCreating** state. Let's figure out what is happening here. First, describe that pod:

```
kubectl describe pods/wp-wordpress-<pod-id>
```

You will get an output as shown in *Figure* 5.32:

```
Events:
  Type     Reason              Age    From                     Message
  ----     ------              ----   ----                     -------
  Normal   Scheduled           3m31s  default-scheduler        Successfully assigned default/wp-wordpress-6f7c4f85b5-lczvx
to aks-agentpool-39838025-vmss000002
  Warning  FailedAttachVolume  3m31s  attachdetach-controller  Multi-Attach error for volume "pvc-848e0fc5-ed4b-4765-aa0d-5
44f014d6997" Volume is already used by pod(s) wp-wordpress-6f7c4f85b5-wp7qt
  Warning  FailedMount         88s    kubelet                  Unable to attach or mount volumes: unmounted volumes=[wordpr
ess-data], unattached volumes=[wordpress-data default-token-ktl66]: timed out waiting for the condition
```

Figure 5.32: Output explaining why the pod is in a ContainerCreating state

This tells you that there is a problem with the volume. You see two errors related to that volume: the `FailedAttachVolume` error explains that the volume is already used by another pod, and `FailedMount` explains that the current pod cannot mount the volume. You can solve this by manually forcefully removing the old pod stuck in the `Terminating` state.

> **Note**
>
> The behavior of the pod stuck in the `Terminating` state is not a bug. This is default Kubernetes behavior. The Kubernetes documentation states the following: *"Kubernetes (versions 1.5 or newer) will not delete pods just because a Node is unreachable. The pods running on an unreachable Node enter the Terminating or Unknown state after a timeout. Pods may also enter these states when the user attempts the graceful deletion of a pod on an unreachable Node."* You can read more at https://kubernetes.io/docs/tasks/run-application/force-delete-stateful-set-pod/.

6. To forcefully remove the terminating pod from the cluster, get the full pod name using the following command:

```
kubectl get pods
```

This will show you an output similar to *Figure* 5.33:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods
NAME                              READY   STATUS              RESTARTS   AGE
wp-mariadb-0                      1/1     Running             0          23m
wp-wordpress-6f7c4f85b5-lczvx     0/1     ContainerCreating   0          6m43s
wp-wordpress-6f7c4f85b5-wp7qt     1/1     Terminating         0          16m
```

Figure 5.33: Getting the name of the pod stuck in the Terminating state

7. Use the pod's name to force the deletion of this pod:

```
kubectl delete pod wordpress-wp-<pod-id> --force
```

8. After the pod has been deleted, it will take a couple of minutes for the other pod to enter a Running state. You can monitor the state of the pod using the following command:

```
kubectl get pods -w
```

This will return an output similar to *Figure* 5.34:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -w
NAME                            READY   STATUS              RESTARTS   AGE
wp-mariadb-0                    1/1     Running             0          30m
wp-wordpress-6f7c4f85b5-lczvx   0/1     ContainerCreating   0          14m
wp-wordpress-6f7c4f85b5-lczvx   0/1     Running             0          18m
wp-wordpress-6f7c4f85b5-lczvx   1/1     Running             0          18m
```

Figure 5.34: The new WordPress pod returning to a Running state

9. As you can see, this brought the new pod to a healthy state. It did take a couple of minutes for the system to pick up the changes and then mount the volume to the new pod. Let's get the details of the pod again using the following command:

```
kubectl describe pod wp-wordpress-<pod-id>
```

This will generate an output as follows:

```
Events:
  Type     Reason                  Age                From                     Message
  ----     ------                  ----               ----                     -------
  Normal   Scheduled               21m                default-scheduler        Successfully assigned default/wp-wordpress
-6f7c4f85b5-lczvx to aks-agentpool-39838025-vmss000002
  Warning  FailedAttachVolume      21m                attachdetach-controller  Multi-Attach error for volume "pvc-848e0fc
5-ed4b-4765-aa0d-544f014d6997" Volume is already used by pod(s) wp-wordpress-6f7c4f85b5-wp7qt
  Warning  FailedMount             6m2s (x3 over 15m)  kubelet                 Unable to attach or mount volumes: unmount
ed volumes=[wordpress-data], unattached volumes=[default-token-ktlGG wordpress-data]: timed out waiting for the condition
  Normal   SuccessfulAttachVolume  5m2s               attachdetach-controller  AttachVolume.Attach succeeded for volume "
pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997"
  Warning  FailedMount             3m46s (x5 over 19m)  kubelet                 Unable to attach or mount volumes: unmount
ed volumes=[wordpress-data], unattached volumes=[wordpress-data default-token-ktlGG]: timed out waiting for the condition
  Normal   Pulled                  3m11s              kubelet                  Container image "docker.io/bitnami/wordpre
ss:5.6.0-debian-10-r30" already present on machine
  Normal   Created                 3m11s              kubelet                  Created container wordpress
  Normal   Started                 3m11s              kubelet                  Started container wordpress
```

Figure 5.35: The new pod is now attaching the volume and pulling the container image

10. This shows you that the new pod successfully got the volume attached and that the container image got pulled. This also made your WordPress website available again, which you can verify by browsing to the public IP. Before continuing to the next chapter, clean up the application using the following command:

```
helm delete wp
kubectl delete pvc --all
kubectl delete pv --all
```

11. Let's also start the node that was shut down: go back to the scale set pane in the Azure portal, click **Instances** in the left-hand menu, select the node you need to start, and click on the **Start** button, as shown in *Figure* 5.36:

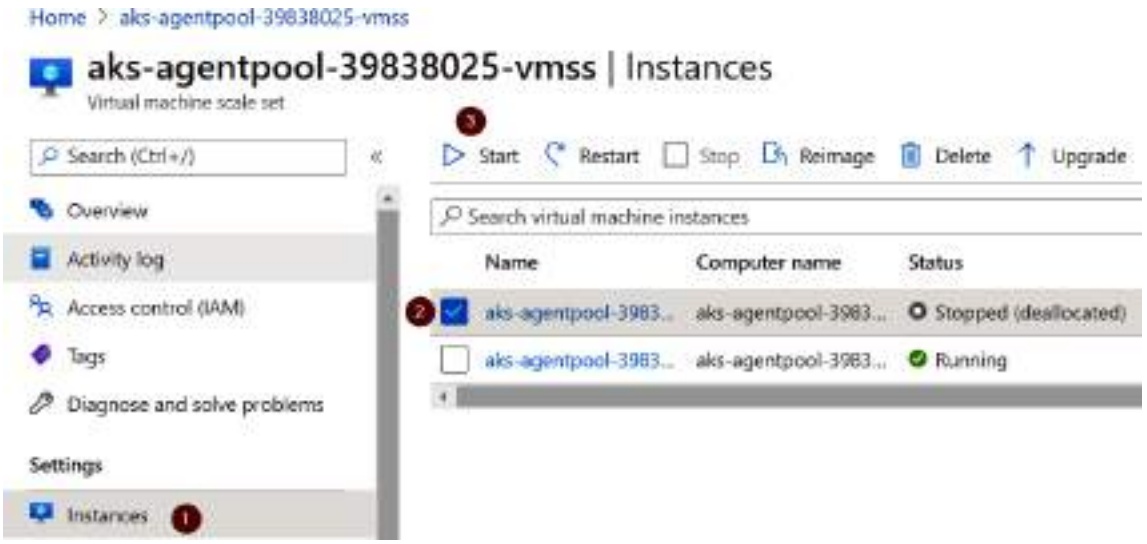


Figure 5.36: Starting node 0 again

In this section, you learned how you can recover from a node failure when PVCs aren't mounting to new pods. All you needed to do was forcefully delete the pod that was stuck in the `Terminating` state.

Summary

In this chapter, you learned about common Kubernetes failure modes and how you can recover from them. This chapter started with an example of how Kubernetes automatically detects node failures and how it will start new pods to recover the workload. After that, you scaled out your workload and had your cluster run out of resources. You recovered from that situation by starting the failed node again to add new resources to the cluster.

Next, you saw how PVs are useful to store data outside of a pod. You deleted all pods on the cluster and saw how the PV ensured that no data was lost in your application. In the final example in this chapter, you saw how you can recover from a node failure when PVs are attached. You were able to recover the workload by forcefully deleting the terminating pod. This brought your workload back to a healthy state.

This chapter has explained common failure modes in Kubernetes. In the next chapter, we will introduce HTTPS support to our services and introduce authentication with Azure Active Directory.

6

Securing your application with HTTPS

HTTPS has become a necessity for any public-facing website. Not only does it improve the security of your website, but it is also becoming a requirement for new browser functionalities. HTTPS is a secure version of the HTTP protocol. HTTPS makes use of **Transport Layer Security** (**TLS**) certificates to encrypt traffic between an end user and a server, or between two servers. TLS is the successor to the **Secure Sockets Layer** (**SSL**). The terms *TLS* and *SSL* are often used interchangeably.

In the past, you needed to buy certificates from a **certificate authority** (**CA**), then set them up on your web server and renew them periodically. While that is still possible today, the **Let's Encrypt** service and helpers in Kubernetes make it very easy to set up verified TLS certificates in your cluster. Let's Encrypt is a non-profit organization run by the **Internet Security Research Group** and backed by multiple companies. It is a free service that offers verified TLS certificates in an automated manner. Automation is a key benefit of the Let's Encrypt service.

In terms of Kubernetes helpers, you will learn about a new object called an **Ingress** and use a Kubernetes add-on called **cert-manager**. An ingress is an object within Kubernetes that manages external access to services, commonly used for HTTP services. An ingress adds additional functionality on top of the service object we explained in *Chapter 3*, *Application deployment on AKS*. It can be configured to handle HTTPS traffic. It can also be configured to route traffic to different back-end services based on the hostname, which is assigned by the **Domain Name System** (**DNS**) that is used to connect.

`cert-manager` is a Kubernetes add-on that helps in automating the creation of TLS certificates. It also helps in the rotation of certificates when they are close to expiring. `cert-manager` can interface with Let's Encrypt to request certificates automatically.

In this chapter, you will see how to set up Azure Application Gateway as a Kubernetes ingress, and `cert-manager` to interface with Let's Encrypt.

The following topics will be covered in this chapter:

- Setting up Azure Application Gateway as a Kubernetes ingress
- Setting up an ingress in front of a service
- Adding TLS support to an ingress

Let's start with setting up Azure Application Gateway as an ingress for AKS.

Setting up Azure Application Gateway as a Kubernetes ingress

An ingress in Kubernetes is an object that is used to route HTTP and HTTPS traffic from outside the cluster to services in a cluster. Exposing services using an ingress rather than exposing them directly, as you've done up to this point—has a number of advantages. These advantages include the ability to route multiple hostnames to the same public IP address and offloading TLS termination from the actual application to the ingress.

To create an ingress in Kubernetes, you need to install an ingress controller. An ingress controller is software that can create, configure, and manage ingresses in Kubernetes. Kubernetes does not come with a preinstalled ingress controller. There are multiple implementations of ingress controllers, and a full list is available at this URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

In Azure, application gateway is a Layer 7 load balancer, which can be used as an ingress for Kubernetes by using the **Application Gateway Ingress Controller (AGIC)**. A layer 7 load balancer is a load balancer that works at the application layer, which is the seventh and highest layer in the OSI networking reference model. Azure Application Gateway has a number of advanced features such as autoscaling and **Web Application Firewall (WAF)**.

There are two ways of configuring the AGIC, either using Helm or as an **Azure Kubernetes Service** (**AKS**) add-on. Installing AGIC using the AKS add-on functionality will result in a Microsoft-supported configuration. Additionally, the add-on method of deployment will be automatically updated by Microsoft, ensuring that your environment is always up to date.

In this section, you will create a new application gateway instance, set up AGIC using the add-on method, and finally, deploy an ingress resource to expose an application. Later in this chapter, you will extend this setup to also include TSL using a Let's Encrypt certificate.

Creating a new application gateway

In this section, you will use the Azure CLI to create a new application gateway. You will then use this application gateway in the next section to integrate with AGIC. The different steps in this section are summarized in the code samples for this chapter in the `setup-appgw.sh` file that is part of the code samples that come with this book.

1. To organize the resources created in this chapter, it is recommended that you create a new resource group. Make sure to create the new resource group in the same location you deployed your AKS cluster in. You can do this using the following command in the Azure CLI:

```
az group create -n agic -l westus2
```

2. Next, you will need to create the networking components required for your application gateway. These are a public IP with a DNS name and a new virtual network. You can do this using the following commands:

```
az network public-ip create -n agic-pip \
   -g agic --allocation-method Static --sku Standard \
   --dns-name "<your unique DNS name>"
az network vnet create -n agic-vnet -g agic \
  --address-prefix 192.168.0.0/24 --subnet-name agic-subnet \
  --subnet-prefix 192.168.0.0/24
```

> **Note**
>
> The az network public-ip create command might show you a warning message `[Coming breaking change] In the coming release, the default behavior will be changed as follows when sku is Standard and zone is not provided: For zonal regions, you will get a zone-redundant IP indicated by zones:["1","2","3"]; For non-zonal regions, you will get a non zone-redundant IP indicated by zones:[].`

3. Finally, you can create the application gateway. This command will take a few minutes to execute

```
az network application-gateway create -n agic -l westus2 \
  -g agic --sku Standard_v2 --public-ip-address agic-pip \
  --vnet-name agic-vnet --subnet agic-subnet
```

4. It will take a couple of minutes for the application gateway to deploy. Once it is created, you can see the resource in the Azure portal. To find this, look for `agic` (or the name you gave your application gateway) in the Azure search bar, and select your application gateway.

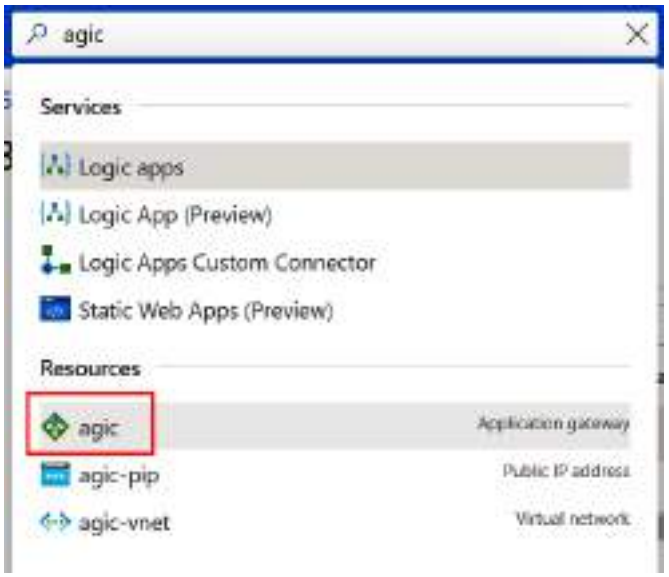


Figure 6.1: Looking for the application gateway in the Azure search bar

5. This will show you your application gateway in the Azure portal, as shown in *Figure 6.2*:

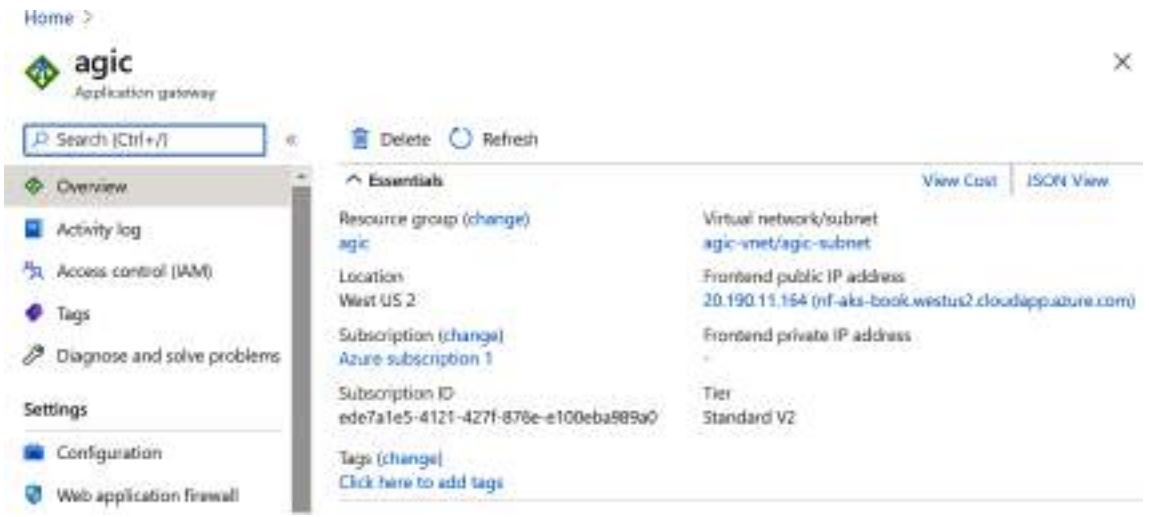


Figure 6.2: The application gateway in the Azure portal

6. To verify that it has been created successfully, browse to the DNS name you configured for the public IP address. This will show you an output similar to *Figure 6.3*. Note that the error message shown is expected since you haven't configured any applications yet behind the application gateway. You will configure applications behind the application gateway using AGIC in the *Adding an ingress rule for the guestbook application* section.

Figure 6.3: Verify that you can connect to the application gateway

Now that you've created a new application gateway and were able to connect to it, we will move on to integrating this application gateway with your existing Kubernetes cluster.

Setting up the AGIC

In this section, you will integrate the application gateway with your Kubernetes cluster using the AGIC AKS add-on. You will also set up virtual network peering so the application gateway can send traffic to your Kubernetes cluster.

1. To enable integration between your cluster and your application gateway, use the following command:

```
appgwId=$(az network application-gateway \
  show -n agic -g agic -o tsv --query "id")
az aks enable-addons -n handsonaks \
  -g rg-handsonaks -a ingress-appgw \
  --appgw-id $appgwId
```

2. Next, you will need to peer the application gateway network with the AKS network. To peer both networks, you can use the following code:

```
nodeResourceGroup=$(az aks show -n handsonaks \
  -g rg-handsonaks -o tsv --query "nodeResourceGroup")
aksVnetName=$(az network vnet list \
  -g $nodeResourceGroup -o tsv --query "[0].name")

aksVnetId=$(az network vnet show -n $aksVnetName \
  -g $nodeResourceGroup -o tsv --query "id")
az network vnet peering create \
  -n AppGWtoAKSVnetPeering -g agic \
  --vnet-name agic-vnet --remote-vnet $aksVnetId \
  --allow-vnet-access

appGWVnetId=$(az network vnet show -n agic-vnet \
  -g agic -o tsv --query "id")
az network vnet peering create \
  -n AKStoAppGWVnetPeering -g $nodeResourceGroup \
  --vnet-name $aksVnetName --remote-vnet $appGWVnetId --allow-vnet-
access
```

This concludes the integration between the application gateway and your AKS cluster. You've enabled the AGIC add-on, and connected both the networks together. In the next section, you will use this AGIC integration to create an ingress for a demo application.

Adding an ingress rule for the guestbook application

Up to this point, you have created a new application gateway and integrated it with your Kubernetes cluster. In this section, you will deploy the guestbook application and then expose it using an ingress.

1. To launch the guestbook application, type in the following command:

```
kubectl create -f guestbook-all-in-one.yaml
```

This will create the guestbook application you've used in the previous chapters. You should see the objects being created as shown in *Figure 6.4*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl create -f guestbook-all-in-one.yaml
service/redis-master created
deployment.apps/redis-master created
service/redis-replica created
deployment.apps/redis-replica created
service/frontend created
deployment.apps/frontend created
```

Figure 6.4: Creating the guestbook application

2. You can then use the following YAML file to expose the front-end service via the ingress. This is provided as `simple-frontend-ingress.yaml` in the source code for this chapter:

```
1   apiVersion: networking.k8s.io/v1
2   kind: Ingress
3   metadata:
4     name: simple-frontend-ingress
5     annotations:
6       kubernetes.io/ingress.class: azure/application-gateway
7   spec:
8     rules:
9     - http:
10        paths:
11        - path: /
12          pathType: Prefix
13          backend:
14            service:
15              name: frontend
16              port:
17                number: 80
```

Let's have a look at what is defined in this YAML file:

- **Line 1**: You specify the Kubernetes API version for the object you are creating.
- **Line 2**: You define that you are creating an `Ingress` object.
- **Lines 5-6**: Here, you're telling Kubernetes that you want to create an ingress of the class `azure/application-gateway`.

The following lines define the actual ingress:

- **Lines 8-12**: Here, you define the path this ingress is listening on. In our case, this is the top-level path. In more advanced cases, you can have different paths pointing to different services.
- **Lines 13-17**: These lines define the actual service this traffic should be pointed to.

You can use the following command to create this ingress:

```
kubectl apply -f simple-frontend-ingress.yaml
```

3. If you now go to http://dns-name/, which you created in the *Creating a new application gateway* section, you should get an output as shown in *Figure* 6.5:

Figure 6.5: Accessing the guestbook application via the ingress

> **Note**
>
> You didn't have to publicly expose the front-end service as you have done in the preceding chapters. You have added the ingress as the exposed service, and the front-end service remains private to the cluster.

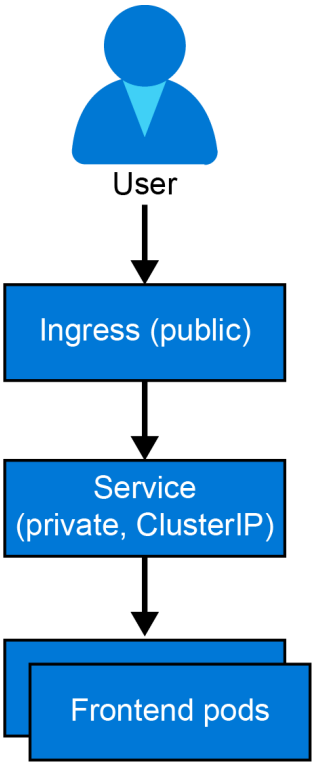


Figure 6.6: Flowchart displaying publicly accessible ingress

4. You can verify this by running the following command:

```
kubectl get service
```

5. This should show you that you have no public services, as seen by the lack of `EXTERNAL-IP` in *Figure 6.7*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl get svc
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
frontend        ClusterIP   10.0.42.112    <none>        80/TCP     11m
kubernetes      ClusterIP   10.0.0.1       <none>        443/TCP    5h57m
redis-master    ClusterIP   10.0.36.111    <none>        6379/TCP   11m
redis-replica   ClusterIP   10.0.230.112   <none>        6379/TCP   11m
```

Figure 6.7: Output shows that you have no public services

In this section, you launched an instance of the guestbook application. You then exposed it publicly by creating an ingress, which in turn configured the application gateway that you created earlier. Only the ingress was publicly accessible.

Next, you'll extend the functionality of AGIC and learn how to secure traffic using a Certificate from Let's Encrypt.

Adding TLS to an ingress

You will now add HTTPS support to your application. To do this, you need a TLS certificate. You will be using the `cert-manager` Kubernetes add-on to request a certificate from Let's Encrypt.

> **Note**
>
> Although this section focuses on using an automated service such as Let's Encrypt, you can still pursue the traditional path of buying a certificate from an existing CA and importing it into Kubernetes. Please refer to the Kubernetes documentation for more information on how to do this: https://kubernetes.io/docs/concepts/services-networking/ingress/#tls

There are a couple of steps involved. The process of adding HTTPS to the application involves the following:

1. Install `cert-manager`, which interfaces with the Let's Encrypt API to request a certificate for the domain name you specify.
2. Install the certificate issuer, which will get the certificate from Let's Encrypt.
3. Create an SSL certificate for a given **Fully Qualified Domain Name** (**FQDN**). An FQDN is a fully qualified DNS record that includes the top-level domain name (such as .org or .com). You created an FQDN linked to your public IP in *step 2* in the section *Creating a new application gateway*.
4. Secure the front-end service by creating an ingress to the service with the certificate created in *step 3*. In the example in this section, you will not be executing this step as an individual step. You will, however, reconfigure the ingress to automatically pick up the certificate created in *step 3*.

Let's start with the first step by installing `cert-manager` in the cluster.

Installing cert-manager

`cert-manager` (https://github.com/jetstack/cert-manager) is a Kubernetes add-on that automates the management and issuance of TLS certificates from various issuing sources. It is responsible for renewing certificates and ensuring they are updated periodically.

> **Note**
>
> The `cert-manager` project is not managed or maintained by Microsoft. It is an open-source solution previously managed by the company **Jetstack**, which recently donated it to the Cloud Native Computing Foundation.

The following commands install `cert-manager` in your cluster:

```
kubectl apply -f https://github.com/jetstack/cert-manager/releases/
download/v1.2.0/cert-manager.yaml
```

This will install a number of components in your cluster as shown in *Figure 6.8*. A detailed explanation of these components can be found in the `cert-manager` documentation at https://cert-manager.io/docs/installation/kubernetes/.

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl apply -f https://github.com/jetstack
/cert-manager/releases/download/v1.1.0/cert-manager.yaml
customresourcedefinition.apiextensions.k8s.io/certificaterequests.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/certificates.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/challenges.acme.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/clusterissuers.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/issuers.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/orders.acme.cert-manager.io created
namespace/cert-manager created
serviceaccount/cert-manager-cainjector created
serviceaccount/cert-manager created
serviceaccount/cert-manager-webhook created
clusterrole.rbac.authorization.k8s.io/cert-manager-cainjector created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-issuers created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-clusterissuers created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-certificates created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-orders created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-challenges created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-ingress-shim created
clusterrole.rbac.authorization.k8s.io/cert-manager-view created
clusterrole.rbac.authorization.k8s.io/cert-manager-edit created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-cainjector created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-issuers created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-clusterissuers created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-certificates created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-orders created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-challenges created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-ingress-shim created
role.rbac.authorization.k8s.io/cert-manager-cainjector:leaderelection created
role.rbac.authorization.k8s.io/cert-manager:leaderelection created
role.rbac.authorization.k8s.io/cert-manager-webhook:dynamic-serving created
rolebinding.rbac.authorization.k8s.io/cert-manager-cainjector:leaderelection created
rolebinding.rbac.authorization.k8s.io/cert-manager:leaderelection created
rolebinding.rbac.authorization.k8s.io/cert-manager-webhook:dynamic-serving created
service/cert-manager created
service/cert-manager-webhook created
deployment.apps/cert-manager-cainjector created
deployment.apps/cert-manager created
deployment.apps/cert-manager-webhook created
mutatingwebhookconfiguration.admissionregistration.k8s.io/cert-manager-webhook created
validatingwebhookconfiguration.admissionregistration.k8s.io/cert-manager-webhook created
```

Figure 6.8: Installing cert-manager in your cluster

`cert-manager` makes use of a Kubernetes functionality called **CustomResourceDefinition** (**CRD**). CRD is a functionality used to extend the Kubernetes API server to create custom resources. In the case of `cert-manager`, there are six CRDs that are created, some of which you will use later in this chapter.

Now that you have installed `cert-manager`, you can move on to the next step: setting up a certificate issuer.

Installing the certificate issuer

In this section, you will install the Let's Encrypt staging certificate issuer. A certificate can be issued by multiple issuers. `letsencrypt-staging`, for example, is for testing purposes. As you are building tests, you'll use the staging server. The code for the certificate issuer has been provided in the source code for this chapter in the `certificate-issuer.yaml` file. As usual, use `kubectl create -f certificate-issuer.yaml`; the YAML file has the following contents:

```
1   apiVersion: cert-manager.io/v1
2   kind: Issuer
3   metadata:
4     name: letsencrypt-staging
5   spec:
6     acme:
7       server: https://acme-staging-v02.api.letsencrypt.org/directory
8       email: <your e-mail address>
9       privateKeySecretRef:
10        name: letsencrypt-staging
11      solvers:
12      - http01:
13          ingress:
14            class: azure/application-gateway
```

Let's look at what we have defined here:

- **Lines 1-2**: Here, you point to one of the CRDs that `cert-manager` created. In this case, specifically, you point to the `Issuer` object. An issuer is a link between your Kubernetes cluster and the actual certificate authority creating the certificate, which is Let's Encrypt in this case.
- **Lines 6-10**: Here you provide the configuration for Let's Encrypt and point to the staging server.
- **Lines 11-14**: This is additional configuration for the ACME client to certify domain ownership. You point Let's Encrypt to the Azure Application Gateway ingress to verify that you own the domain you will request a certificate for later.

With the certificate issuer installed, you can now move on to the next step: creating the TLS certificate on the ingress.

Creating the TLS certificate and securing the ingress

In this section, you will create a TLS certificate. There are two ways you can configure `cert-manager` to create certificates. You can either manually create a certificate and link it to the ingress, or you can configure your ingress controller, so `cert-manager` automatically creates the certificate.

In this example, you will configure your ingress using the latter method.

1. To start, edit the ingress to look like the following YAML code. This file is present in the source code on GitHub as `ingress-with-tls.yaml`:

```
1   apiVersion: networking.k8s.io/v1
2   kind: Ingress
3   metadata:
4     name: simple-frontend-ingress
5     annotations:
6       kubernetes.io/ingress.class: azure/application-gateway
7       cert-manager.io/issuer: letsencrypt-staging
8       cert-manager.io/acme-challenge-type: http01
9   spec:
10    rules:
```

```
11    - http:
12        paths:
13        - path: /
14          pathType: Prefix
15          backend:
16            service:
17              name: frontend
18              port:
19                number: 80
20      host: <your dns-name>.<your azure region>.cloudapp.azure.com
21    tls:
22      - hosts:
23        - <your dns-name>.<your azure region>.cloudapp.azure.com
24        secretName: frontend-tls
```

You should make the following changes to the original ingress:

- **Lines 7-8**: You add two additional annotations to the ingress that points to a certificate issuer and `acme-challenge` to prove domain ownership.
- **Line 20**: The domain name for the ingress is added here. This is required because Let's Encrypt only issues certificates for domains.
- **Line 21-24**: This is the TLS configuration of the ingress. It contains the hostname as well as the name of the secret that will be created to store the certificate.

2. You can update the ingress you created earlier with the following command:

```
kubectl apply -f ingress-with-tls.yaml
```

It takes `cert-manager` about a minute to request a certificate and configure the ingress to use that certificate. While you are waiting for that, let's have a look at the intermediate resources that `cert-manager` created on your behalf.

3. First off, `cert-manager` created a `certificate` object for you. You can look at the status of that object using the following:

```
kubectl get certificate
```

This command will generate an output as shown in *Figure 6.9*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl get certificate
NAME           READY   SECRET         AGE
frontend-tls   False   frontend-tls   3s
```

Figure 6.9: The status of the certificate object

4. As you can see, the certificate isn't ready yet. There is another object that `cert-manager` created to actually get the certificate. This object is `certificaterequest`. You can get its status by using the following command:

```
kubectl get certificaterequest
```

This will generate the output shown in *Figure 6.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl get certificaterequest
NAME                 READY   AGE
frontend-tls-p528r   False   4s
```

Figure 6.10: The status of the certificaterequest obiect

You can also get more details about the request by issuing a `describe` command against the `certificaterequest` object:

```
kubectl describe certificaterequest
```

While you're waiting for the certificate to be issued, the status will look similar to *Figure 6.11*:

```
Status:
  Conditions:
    Last Transition Time:  2021-01-24T22:57:12Z
    Message:               Waiting on certificate issuance from order default/frontend-tls-p528r-
3330258237: "pending"
    Reason:                Pending
    Status:                False
    Type:                  Ready
Events:
  Type    Reason        Age   From          Message
  ----    ------        ----  ----          -------
  Normal  OrderCreated  10s   cert-manager  Created Order resource default/frontend-tls-p528r-333
0258237
  Normal  OrderPending  10s   cert-manager  Waiting on certificate issuance from order default/fr
ontend-tls-p528r-3330258237: ""
```

Figure 6.11: Using the kubectl describe command to obtain details of the certificaterequest object

As you can see, the `certificaterequest` object shows you that the order has been created and that it is pending.

5. After a couple of additional seconds, the `describe` command should return a successful certificate creation message. Run the following command to get the updated status:

```
kubectl describe certificaterequest
```

The output of this command is shown in *Figure 6.12*:

```
Events:
  Type    Reason             Age    From          Message
  ----    ------             ----   ----          -------
  Normal  OrderCreated       6m39s  cert-manager  Created Order resource default/frontend-tls-p52
8r-3330258237
  Normal  OrderPending       6m39s  cert-manager  Waiting on certificate issuance from order defa
ult/frontend-tls-p528r-3330258237: ""
  Normal  CertificateIssued  5m54s  cert-manager  Certificate fetched from issuer successfully
```

Figure 6.12: The issued certificate

This should now enable the front-end ingress to be served over HTTPS.

6. Let's try this out in a browser by browsing to the DNS name you created in the *Creating a new application gateway* section. Depending on your browser's cache, you might need to add `https://` in front of the URL.
7. Once you reach the ingress, it will indicate an error in the browser, showing you that the certificate isn't valid, similar to *Figure 6.13*. This is to be expected since you are using the Let's Encrypt staging server:

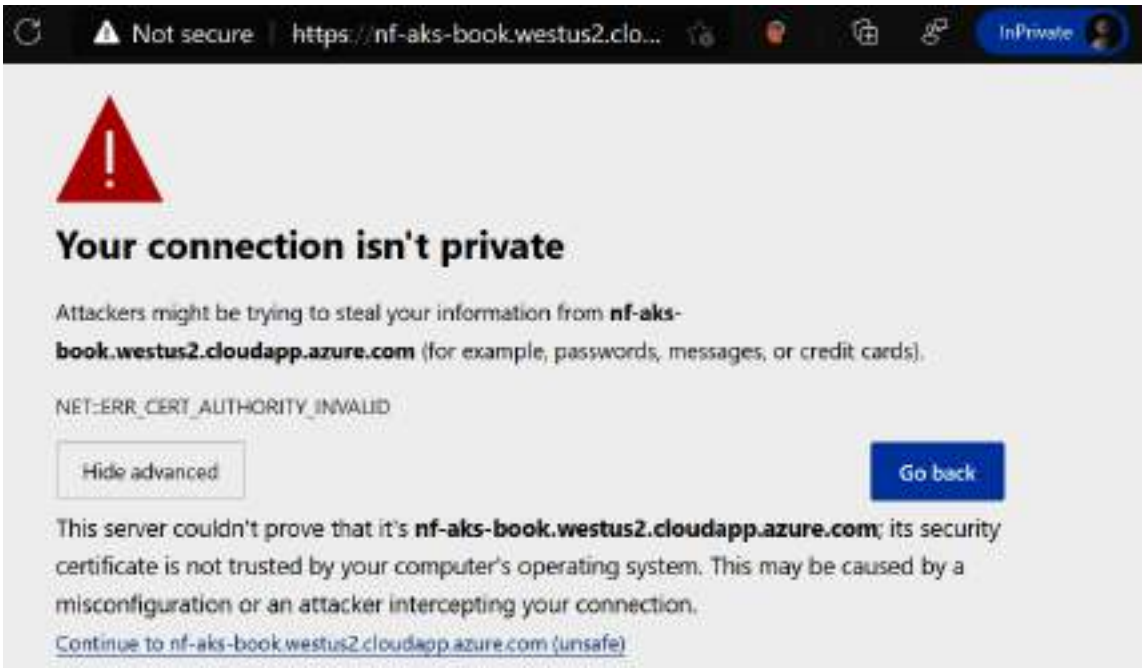


Figure 6.13: Using the Let's Encrypt staging server, the certificate isn't trusted by default

You can browse to your application by clicking **Advanced** and selecting **Continue**.

In this section, you successfully added a TLS certificate to your ingress to secure traffic to it. Since you were able to complete the test with the staging certificate, you can now move on to a production system.

Switching from staging to production

In this section, you will switch from a staging certificate to a production-level certificate. To do this, you can redo the previous exercise by creating a new issuer in your cluster, like the following (provided in `certificate-issuer-prod.yaml` as part of the code samples with this book). Don't forget to change your email address in the file. The following code is contained in that file:

```
apiVersion: cert-manager.io/v1alpha2
kind: Issuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: <your e-mail>
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
    - http01:
        ingress:
          class: azure/application-gateway
```

Then, replace the reference to the issuer in the `ingress-with-tls.yaml` file with `letsencrypt-prod` as shown (provided in the `ingress-with-tls-prod.yaml` file):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-frontend-ingress
  annotations:
    kubernetes.io/ingress.class: azure/application-gateway
    cert-manager.io/issuer: letsencrypt-prod
    cert-manager.io/acme-challenge-type: http01
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
```

```
17              name: frontend
18              port:
19                number: 80
20      host: <your dns-name>.<your azure region>.cloudapp.azure.com
21    tls:
22      - hosts:
23        - <your dns-name>.<your azure region>.cloudapp.azure.com
24        secretName: frontend-prod-tls
```

To apply these changes, execute the following commands:

```
kubectl create -f certificate-issuer-prod.yaml
kubectl apply -f ingress-with-tls-prod.yaml
```

It will again take about a minute for the certificate to become active. Once the new certificate is issued, you can browse to your DNS name again and shouldn't see any more warnings regarding invalid certificates. If you click the padlock icon in the browser, you should see that your connection is secure and uses a valid certificate:

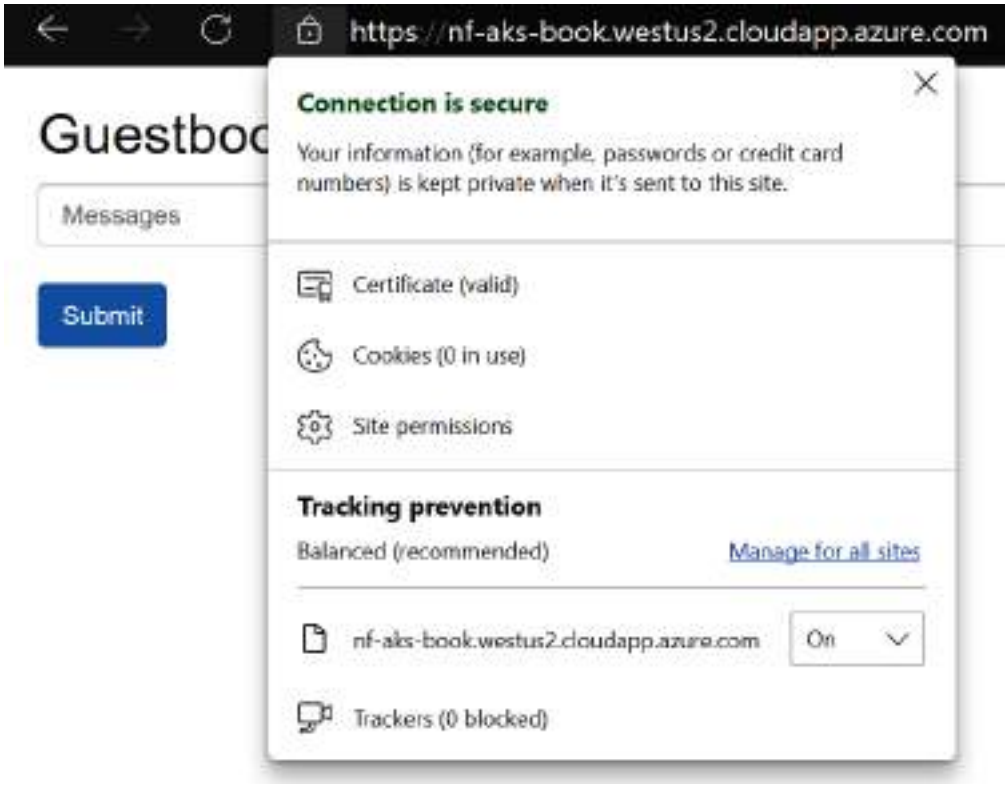


Figure 6.14: The web page displaying a valid certificate

In this section, you have learned how to add TLS support to an ingress. You did this by installing the `cert-manager` Kubernetes add-on. `cert-manager` got a free certificate from Let's Encrypt and added this to the existing ingress deployed on the application gateway. The process that was described here is not specific to Azure and Azure Application Gateway. This process of adding TLS to an ingress works with other ingress controllers as well.

Let's delete the resources you created during this chapter:

```
kubectl delete -f https://github.com/jetstack/cert-manager/releases/
download/v1.1.0/cert-manager.yaml
az aks disable-addons -n handsonaks \
  -g rg-handsonaks -a ingress-appgw
```

Summary

In this chapter, you added HTTPS security to the guestbook application without actually changing the source code. You started by setting up a new application gateway and configured AGIC on AKS. This gives you the ability to create Kubernetes ingresses that can be configured on the application gateway.

Then, you installed a certificate manager that interfaces with the Let's Encrypt API to request a certificate for the domain name we subsequently specified. You leveraged a certificate issuer to get the certificate from Let's Encrypt. You then reconfigured the ingress to request a certificate from this issuer in the cluster. Using these capabilities of both the certificate manager as well as the ingress, you are now able to secure your websites using TLS.

In the next chapter, you will learn how to monitor your deployments and set up alerts. You will also learn how to quickly identify root causes when errors do occur, and how to debug applications running on AKS. At the same time, you'll learn how to perform the correct fixes once you have identified the root causes.

7

Monitoring the AKS cluster and the application

Now that you know how to deploy applications on an AKS cluster, let's focus on how you can ensure that your cluster and applications remain available. In this chapter, you will learn how to monitor your cluster and the applications running on it. You'll explore how Kubernetes makes sure that your applications are running reliably using readiness and liveness probes.

You will also learn how **AKS Diagnostics** and **Azure Monitor** are used, and how they are integrated within the Azure portal. You will see how you can use AKS Diagnostics to monitor the status of the cluster itself, and how Azure Monitor helps monitor the pods on the cluster and allows you to get access to the logs of the pods at scale.

In brief, the following topics will be covered in this chapter:

- Monitoring and debugging applications using `kubectl`
- Reviewing metrics reported by Kubernetes
- Reviewing metrics from Azure Monitor

Let's start the chapter by reviewing some of the commands in `kubectl` that you can use to monitor your applications.

Commands for monitoring applications

Monitoring the health of applications deployed on Kubernetes as well as the Kubernetes infrastructure itself is essential for providing a reliable service to your customers. There are two primary use cases for monitoring:

- Ongoing monitoring to get alerts if something is not behaving as expected
- Troubleshooting and debugging application errors

When observing an application running on top of a Kubernetes cluster, you'll need to examine multiple things in parallel, including containers, pods, services, and the nodes in the cluster. For ongoing monitoring, you'll need a monitoring system such as Azure Monitor or Prometheus. Azure Monitor will be introduced later in this chapter. Prometheus (https://prometheus.io/) is a popular open-source solution within the Kubernetes ecosystem to monitor Kubernetes environments. For troubleshooting, you'll need to interact with the live cluster. The most common commands used for troubleshooting are as follows:

```
kubectl get <resource type> <resource name>
kubectl describe <resource type> <resource name>
kubectl logs <pod name>
```

Each of these commands will be described in detail later in this chapter.

To begin with the practical examples, recreate the guestbook example again using the following command:

```
kubectl create -f guestbook-all-in-one.yaml
```

While the `create` command is running, you will watch its progress in the following sections. Let's start by exploring the `get` command.

The kubectl get command

To see the overall picture of deployed applications, `kubectl` provides the `get` command. The `get` command lists the resources that you specify. Resources can be pods, ReplicaSets, ingresses, nodes, deployments, secrets, and so on. You have already run this command in the previous chapters to verify that an application was ready for use.

Perform the following steps:

1. Run the following `get` command, which will get us the resources and their statuses:

    ```
    kubectl get all
    ```

    This will show you all the deployments, ReplicaSets, pods, and services in your namespace:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get all
NAME                                    READY   STATUS        RESTARTS   AGE
pod/frontend-766d4f77cb-859v8           1/1     Running       0          17s
pod/frontend-766d4f77cb-grfcx           1/1     Running       0          17s
pod/frontend-766d4f77cb-vq5dd           1/1     Running       0          17s
pod/redis-master-f46ff57fd-fb5k2        1/1     Running       0          17s
pod/redis-replica-57c8c66cc4-8jx7t      1/1     Running       0          17s
pod/redis-replica-57c8c66cc4-crltb      1/1     Running       0          17s
pod/redis-replica-57c8c66cc4-fddvg      0/1     Terminating   0          76m
pod/redis-replica-57c8c66cc4-mwtp9      0/1     Terminating   0          76m

NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)        AGE
service/frontend        LoadBalancer   10.0.184.216   51.143.114.234   80:30977/TCP   17s
service/kubernetes      ClusterIP      10.0.0.1       <none>           443/TCP        34h
service/redis-master    ClusterIP      10.0.102.11    <none>           6379/TCP       17s
service/redis-replica   ClusterIP      10.0.48.214    <none>           6379/TCP       17s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/frontend           3/3     3            3           17s
deployment.apps/redis-master       1/1     1            1           17s
deployment.apps/redis-replica      2/2     2            2           17s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/frontend-766d4f77cb           3         3         3       17s
replicaset.apps/redis-master-f46ff57fd        1         1         1       17s
replicaset.apps/redis-replica-57c8c66cc4      2         2         2       17s
```

Figure 7.1: All the resources running in the default namespace

2. Focus your attention on the pods in your deployment. You can get the status of the pods with the following command:

```
kubectl get pods
```

You will see that only the pods are shown, as seen in *Figure 7.2*. Let's investigate this in detail:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-ds6gb         1/1     Running   0          103s
frontend-766d4f77cb-gvbjj         1/1     Running   0          103s
frontend-766d4f77cb-qw8kv         1/1     Running   0          103s
redis-master-f46ff57fd-qs6gk      1/1     Running   0          103s
redis-replica-57c8c66cc4-fddvg    1/1     Running   0          103s
redis-replica-57c8c66cc4-mwtp9    1/1     Running   0          103s
```

Figure 7.2: All the pods in your namespace

The first column indicates the pod name, for example, `frontend-766d4f77cb-ds6gb`. The second column indicates how many containers in the pod are ready against the total number of containers in the pod. Readiness is defined via a readiness probe in Kubernetes. There is a dedicated section called *Readiness and liveness probes* later in this chapter.

The third column indicates the status, for example, `Pending`, `ContainerCreating`, `Running`, and so on. The fourth column indicates the number of restarts, while the fifth column indicates the age when the pod was asked to be created.

3. If you need more information about your pod, you can add extra columns to the output of a `get` command by adding `-o wide` to the command like this:

```
kubectl get pods -o wide
```

This will show you additional information, as shown in *Figure 7.3*:

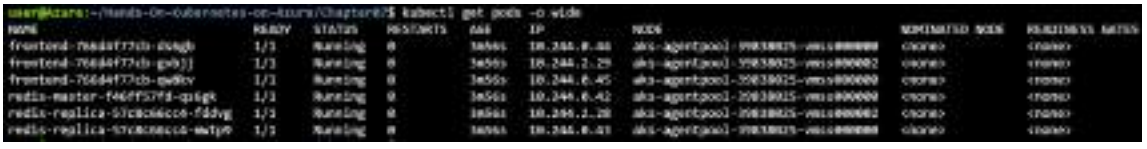


Figure 7.3: Adding -o wide shows more details on the pods

The extra columns include the IP address of the pod, the node it is running on, the nominated node, and readiness gates. A nominated node is only set when a higher-priority pod preempts a lower-priority pod. The nominated node field would then be set on the higher-priority pod. It signifies the node that the higher-priority pod will be scheduled once the lower-priority pod has terminated gracefully. A readiness gate is a way to introduce external system components as the readiness for a pod.

Executing a `get pods` command only shows the state of the current pod. As we will see next, things can fail at any of the states, and we need to use the `kubectl describe` command to dig deeper.

The kubectl describe command

The `kubectl describe` command gives you a detailed view of the object you are describing. It contains the details of the object itself, as well as any recent events related to that object. While the `kubectl get events` command lists all the events for the entire namespace, with the `kubectl describe` command, you would get only the events for that specific object. If you are interested in just pods, you can use the following command:

```
kubectl describe pods
```

The preceding command lists all the information pertaining to all pods. This is typically too much information to contain in a typical shell.

If you want information on a particular pod, you can type the following:

```
kubectl describe pod/<pod-name>
```

> **Note**
>
> You can either use a slash or a space in between pod and `<pod-name>`.
> The following two commands will have the same output:
> `kubectl describe pod/<pod-name>`
> `kubectl describe pod <pod-name>`

You will get an output similar to *Figure 7.4*, which will be explained in detail later:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl describe pod frontend-766d4f77cb-ds6gb
Name:         frontend-766d4f77cb-ds6gb
Namespace:    default
Priority:     0
Node:         aks-agentpool-39838025-vmss000000/10.240.0.4
Start Time:   Tue, 26 Jan 2021 02:10:33 +0000
Labels:       app=guestbook
              pod-template-hash=766d4f77cb
              tier=frontend
Annotations:  <none>
Status:       Running
IP:           10.244.0.44
IPs:
  IP:           10.244.0.44
Controlled By:  ReplicaSet/frontend-766d4f77cb
Containers:
  php-redis:
    Container ID:   containerd://f202c0fc671be873362ff3a097c30193b04182ffdd1f5065aeb9b5daac724762
    Image:          gcr.io/google-samples/gb-frontend:v4
    Image ID:       sha256:c8cb3a8f677bc4b7fb210d98368dae7b6268451897d43ebbc4add5265574b610
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Tue, 26 Jan 2021 02:10:34 +0000
    Ready:          True
    Restart Count:  0
    Requests:
      cpu:     10m
      memory:  10Mi
    Environment:
      GET_HOSTS_FROM:  dns
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-ktl66 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-ktl66:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-ktl66
    Optional:    false
QoS Class:       Burstable
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/memory-pressure:NoSchedule op=Exists
                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                 node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  50m   default-scheduler  Successfully assigned default/frontend-766d4f77cb-ds6gb to
 aks-agentpool-39838025-vmss000000
  Normal  Pulled     50m   kubelet            Container image "gcr.io/google-samples/gb-frontend:v4" alr
eady present on machine
  Normal  Created    49m   kubelet            Created container php-redis
  Normal  Started    49m   kubelet            Started container php-redis
```

Figure 7.4: Describing an object shows the detailed output of that object

From the description, you can get the node on which the pod is running, how long it has been running, its internal IP address, the Docker image name, the ports exposed, the env variables, and the events (from within the past hour).

In the preceding example, the pod name is `frontend-766d4f77cb-ds6gb`. As mentioned in *Chapter 1*, *Introduction to containers and Kubernetes*, it has the `<ReplicaSet name>-<random 5 chars>` format. The `replicaset` name itself is randomly generated from the deployment name front end: `<deployment name>-<random-string>`.

Figure 7.5 shows the relationship between a deployment, a ReplicaSet, and pods:

Figure 7.5: Relationship between a deployment, a ReplicaSet, and pods

The namespace under which this pod runs is `default`. So far, you have just been using the `default` namespace, appropriately named `default`.

Another section that is important from the preceding output is the node section:

```
Node:           aks-agentpool-39838025-vmss000000/10.240.0.4
```

The node section lets you know which physical node/VM the pod is running on. If the pod is repeatedly restarting or having issues running and everything else seems OK, there might be an issue with the node itself. Having this information is essential to perform advanced debugging.

The following is the time the pod was initially scheduled:

```
Start Time:   Tue, 26 Jan 2021 02:10:33 +0000
```

This doesn't mean that the pod has been running since that time, so the time can be misleading in that sense. If a health event occurs (for example, a container crashes), the pod will reset automatically.

You can add more information about a workload in Kubernetes using `Labels`, as shown here:

```
Labels:app=guestbook
pod-template-hash=57d8c9fb45
tier=frontend
```

Labels are a commonly used functionality in Kubernetes. For example, this is how links between objects, such as `service` to pod and `deployment` to `ReplicaSet` to pod (*Figure* 7.5), are made. If you see that traffic is not being routed to a pod from a service, this is the first thing you should check. Also, you'll notice that the `pod-template-hash` label also occurs in the pod name. This is how the link between the ReplicaSet and the pod is made. If the labels don't match, the resources won't attach.

The following shows the internal IP of the pod and its status:

```
Status:       Running
IP:           10.244.0.44
IPs:
  IP:           10.244.0.44
```

As mentioned in previous chapters, when building out your application, the pods can be moved to different nodes and get a different IP, so you should avoid using these IP addresses. However, when debugging application issues, having a direct IP for a pod can help with troubleshooting. Instead of connecting to your application through a service object, you can connect directly from one pod to another using the other pod's IP address to test connectivity.

The containers running in the pod and the ports that are exposed are listed in the following block:

```
Containers:
  php-redis:
    ...
    Image:          gcr.io/google-samples/gb-frontend:v4
    ...
    Port:           80/TCP
    ...
    Requests:
      cpu:     10m
```

```
      memory:  10Mi
    Environment:
      GET_HOSTS_FROM:  dns
    ...
```

In this case, you are getting the `gb-frontend` container with the `v4` tag from the `gcr.io` container registry, and the repository name is `google-samples`.

Port `80` is exposed to outside traffic. Since each pod has its own IP, the same port can be exposed for multiple instances of the same pod even when running on the same host. For instance, if you had two pods running a web server on the same node, both could use port `80`, since each pod has its own IP address. This is a huge management advantage as you don't have to worry about port collisions on the same node.

Any events that occurred in the previous hour show up here:

```
Events:
```

Using `kubectl describe` is very useful to get more context about the resources you are running. The final section contains events related to the object you were describing. You can get all events in your cluster using the `kubectl get events` command.

To see the events for all resources in the system, run the following command:

```
kubectl get events
```

> **Note**
>
> Kubernetes maintains events for only 1 hour by default.

If everything goes well, you should have an output similar to *Figure 7.6*:

Figure 7.6: Getting the events shows all events from the past hour

Figure 7.6 only shows the event for one pod, but as you can see in your output, the output for this command contains the events for all resources that were recently created, updated, or deleted.

In this section, you have learned about the commands you can use to inspect a Kubernetes application. In the next section, you'll focus on debugging application failures.

Debugging applications

Now that you have a basic understanding of how to inspect applications, you can start seeing how you can debug issues with deployments.

In this section, common errors will be introduced, and you'll determine how to debug and fix them.

If you haven't implemented the Guestbook application already, run the following command:

```
kubectl create -f guestbook-all-in-one.yaml
```

After a couple of seconds, the application should be up and running.

Image pull errors

In this section, you are going to introduce image pull errors by setting the image tag value to a non-existent one. An image pull error occurs when Kubernetes cannot download the image for the container it needs to run.

1. Run the following command on Azure Cloud Shell:

   ```
   kubectl edit deployment/frontend
   ```

   Next, change the image tag from `v4` to `v_non_existent` by executing the following steps.

2. Type `/gb-frontend` and hit the *Enter* key to have your cursor brought to the image definition.

Hit the *I* key to go into insert mode. Delete `v4` and type `v_non_existent` as shown in *Figure 7.7*:

```
spec:
  containers:
  - env:
    - name: GET_HOSTS_FROM
      value: dns
    image: gcr.io/google-samples/gb-frontend:v_non_existent
    imagePullPolicy: IfNotPresent
    name: php-redis
    ports:
```

Figure 7.7: Changing the image tag from v4 to v_non_existent

3. Now, close the editor by first hitting the *Esc* key, then type `:wq!` and hit *Enter*.
4. Run the following command to list all the pods in the current namespace:

```
kubectl get pods
```

The preceding command should indicate errors, as shown in *Figure 7.8*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods
NAME                              READY   STATUS             RESTARTS   AGE
frontend-69f4b6d547-2xq6n         0/1     ErrImagePull       0          32s
frontend-766d4f77cb-ds6gb         1/1     Running            0          64m
frontend-766d4f77cb-gvbjj         1/1     Running            0          64m
frontend-766d4f77cb-qw8kv         1/1     Running            0          64m
redis-master-f46ff57fd-qs6gk      1/1     Running            0          64m
redis-replica-57c8c66cc4-fddvg    1/1     Running            0          64m
redis-replica-57c8c66cc4-mwtp9    1/1     Running            0          64m
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods
NAME                              READY   STATUS             RESTARTS   AGE
frontend-69f4b6d547-2xq6n         0/1     ImagePullBackOff   0          66s
frontend-766d4f77cb-ds6gb         1/1     Running            0          64m
frontend-766d4f77cb-gvbjj         1/1     Running            0          64m
frontend-766d4f77cb-qw8kv         1/1     Running            0          64m
redis-master-f46ff57fd-qs6gk      1/1     Running            0          64m
redis-replica-57c8c66cc4-fddvg    1/1     Running            0          64m
redis-replica-57c8c66cc4-mwtp9    1/1     Running            0          64m
```

Figure 7.8: One of the pods has the status of either ErrImagePull or ImagePullBackOff

You might see either a status called `ErrImagePull` or `ImagePullBackOff`. Both errors refer to the fact that Kubernetes cannot pull the image from the registry. The `ErrImagePull` error describes just this; `ImagePullBackOff` describes that Kubernetes will back off (wait) before retrying to download the image. This back-off has an exponential delay, going from 10 to 20 to 40 seconds and beyond, up to 5 minutes.

5. Run the following command to get the full error details:

```
kubectl describe pods/<failed pod name>
```

A sample error output is shown in *Figure* 7.9. The key error message is highlighted in red:

```
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  5m3s                 default-scheduler  Successfully assigned default/frontend-69
f4b6d547-2xq6n to aks-agentpool-39838025-vmss000000
  Normal   Pulling    3m33s (x4 over 5m3s)  kubelet           Pulling image "gcr.io/google-samples/gb-f
rontend:v_non_existent"
  Warning  Failed     3m33s (x4 over 5m2s)  kubelet           Failed to pull image "gcr.io/google-sampl
es/gb-frontend:v_non_existent": rpc error: code = NotFound desc = failed to pull and unpack image "gcr.i
o/google-samples/gb-frontend:v_non_existent": failed to resolve reference "gcr.io/google-samples/gb-fron
tend:v_non_existent": gcr.io/google-samples/gb-frontend:v_non_existent: not found
  Warning  Failed     3m33s (x4 over 5m2s)  kubelet           Error: ErrImagePull
  Warning  Failed     3m4s (x7 over 5m2s)   kubelet           Error: ImagePullBackOff
  Normal   BackOff    1s (x20 over 5m2s)    kubelet           Back-off pulling image "gcr.io/google-sam
ples/gb-frontend:v_non_existent"
```

Figure 7.9: Using describe shows more details on the error

The events clearly show that the image does not exist. Errors such as passing invalid credentials to private Docker repositories will also show up here.

6. Let's fix the error by setting the image tag back to `v4`. First, type the following command in Cloud Shell to edit the deployment:

```
kubectl edit deployment/frontend
```

7. Type `/gb-frontend` and hit *Enter* to have your cursor brought to the image definition.
8. Hit the *I* key to go into insert mode. Delete `v_non_existent`, and type `v4`.
9. Now, close the editor by first hitting the *Esc* key, then type `:wq!` and hit *Enter*.
10. This should automatically fix the deployment. You can verify it by getting the events for the pods again.

> **Note**
>
> Because Kubernetes did a rolling update, the front end was continuously available with zero downtime. Kubernetes recognized a problem with the new specification and stopped rolling out additional changes automatically.

Image pull errors can occur when images aren't available or when you don't have access to the container registry. In the next section, you'll explore an error within the application itself.

Application errors

You will now see how to debug an application error. The errors in this section will be self- induced, similar to the last section. The method for debugging the issue is the same as the one we used to debug errors on running applications.

1. To start, get the public IP of the front-end service:

```
kubectl get service
```

2. Connect to the service by pasting its public IP in a browser. Create a couple of entries:

Figure 7.10: Make a couple of entries in the guestbook application

You now have an instance of the guestbook application running. To improve the experience with the example, it's best to scale down the front end so there is only a single replica running.

Scaling down the front end

In *Chapter 3*, *Application deployment on AKS*, you learned how the deployment of the front end has a configuration of `replicas=3`. This means that the requests the application receives can be handled by any of the pods. To introduce the application error and note the errors, you'll need to make changes in all three of them.

But to make this example easier, set `replicas` to 1, so that you have to make changes to only one pod:

```
kubectl scale --replicas=1 deployment/frontend
```

Having only one replica running will make introducing the error easier. Let's now introduce this error.

Introducing an app error

In this case, you are going to make the **Submit** button fail to work. You will need to modify the application code for this:

> **Note:**
>
> It is not advised to make production changes to your application by using `kubectl exec` to execute commands in your pods. If you need to make changes to your application, the preferred way is to create a new container image and update your deployment.

1. You will use the `kubectl exec` command. This command lets you run commands on the command line of that pod. With the `-it` option, it attaches an interactive terminal to the pod and gives you a shell that you can run commands on. The following command launches a Bash terminal on the pod:

   ```
   kubectl exec -it <frontend-pod-name> -- bash
   ```

This will enter a Bash shell environment as shown in *Figure 7.11*:

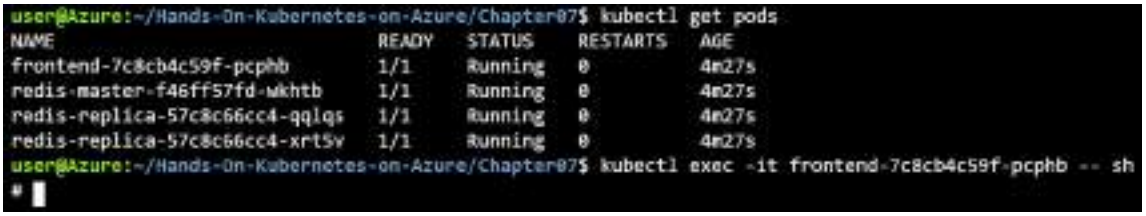


Figure 7.11: Getting a pod's name and getting access to a shell inside the pod

2. Once you are in the container shell, run the following command:

```
apt update
apt install -y vim
```

   The preceding code installs the `vim` editor so that we can edit the file to introduce an error.

3. Now, use `vim` to open the `guestbook.php` file:

```
vim guestbook.php
```

4. Add the following code at line 17, below the line `if ($_GET['cmd'] == 'set') {`. Remember, to edit a line in `vim`, you hit the *I* key. After you are done editing, you can exit by hitting *Esc*, and then type `:wq!` and press *Enter*:

```
$host = 'localhost';
if(!defined('STDOUT')) define('STDOUT', fopen('php://stdout', 'w'));
fwrite(STDOUT, "hostname at the beginning of 'set' command ");
fwrite(STDOUT, $host);
fwrite(STDOUT, "\n");
```

The file will look like *Figure 7.12*:

```
<?php

error_reporting(E_ALL);
ini_set('display_errors', 1);

require 'Predis/Autoloader.php';

Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
  $host = 'redis-master';
  if (getenv('GET_HOSTS_FROM') == 'env') {
    $host = getenv('REDIS_MASTER_SERVICE_HOST');
  }
  header('Content-Type: application/json');
  if ($_GET['cmd'] == 'set') {
    $host = 'localhost';
    if(!defined('STDOUT')) define('STDOUT', fopen('php://stdout', 'w'));
    fwrite(STDOUT, "hostname at the beginning of 'set' command ");
    fwrite(STDOUT, $host);
    fwrite(STDOUT, "\n");

    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => $host,
      'port'   => 6379,
    ]);
```

Figure 7.12: The updated code that introduced an error and additional logging

5. You have now introduced an error where reading messages will work, but not writing them. You have done this by asking the front end to connect to the Redis master at the non-existent localhost server. The writes should fail. At the same time, to make this demo more visual, we added some additional logging to this section of the code.

Open your guestbook application by browsing to its public IP, and you should see the entries from earlier:

Figure 7.13: The entries from earlier are still present

6. Now, create a new message by typing a message and hitting the **Submit** button:

Figure 7.14: A new message was created

Submitting a new message makes it appear in the application. If you did not know any better, you would have thought the entry was written successfully to the database. However, if you refresh your browser, you will see that the message is no longer there.

7. To verify that the message has not been written to the database, hit the **Refresh** button in your browser; you will see just the initial entries, and the new entry has disappeared:

Figure 7.15: The new message has disappeared

As an app developer or operator, you'll probably get a ticket like this: `After the new deployment, new entries are not persisted. Fix it`.

Using logs to identify the root cause

The first step toward resolution is to get the logs.

1. Exit out of the front-end pod for now and get the logs for this pod:

```
exit
kubectl logs <frontend-pod-name>
```

> **Note:**
>
> You can add the `-f` flag after `kubectl logs` to get a live log stream, as follows: `kubectl logs <pod-name> -f`. This is useful during live debugging sessions.

2. You will see entries such as those seen in *Figure 7.16*:

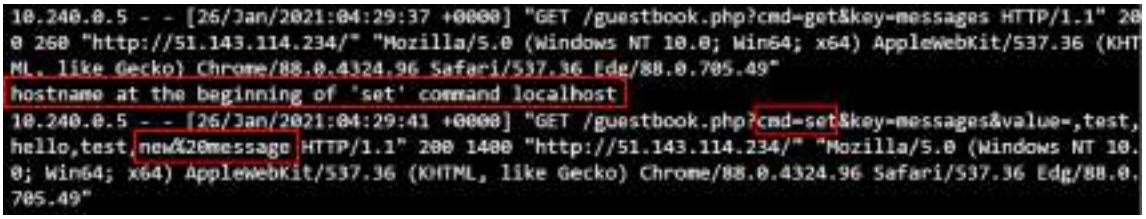


Figure 7.16: The new message shows up as part of the application logs

3. Hence, you know that the error is somewhere when writing to the database in the `set` section of the code. When you see the entry `hostname at the beginning of 'set' command localhost`, you know that the error is between this line and the start of the client, so the setting of `$host = 'localhost'` must be the offending error. This error is not as uncommon as you would think and, as you just saw, could have easily gone through QA unless there had been a specific instruction to refresh the browser. It could have worked perfectly well for the developer, as they could have a running Redis server on the local machine.

Now that you have used logs in Kubernetes to root cause the issue, let's get to resolving the error and getting our application back to a healthy state.

Solving the issue

There are two options to fix this bug you introduced: you can either navigate into the pod and make the code changes, or you can ask Kubernetes to give us a healthy new pod. It is not recommended to make manual changes to pods, so in the next step, you will use the second approach. Let's fix this bug by deleting the faulty pod:

```
kubectl delete pod <podname>
```

As there is a ReplicaSet that controls the pods, you should immediately get a new pod that has started from the correct image. Try to connect to the guestbook again and verify that messages persist across browser refreshes.

The following points summarize what was covered in this section on how to identify an error and how to fix it:

- Errors can come in many shapes and forms.
- Most of the errors encountered by the deployment team are configuration issues.
- Use logs to identify the root cause.
- Using `kubectl exec` on a container is a useful debugging strategy.
- Note that broadly allowing `kubectl exec` is a serious security risk, as it lets the Kubernetes operator execute commands directly in the pods they have access to. Make sure that only a subset of operators has the ability to use the `kubectl exec` command. You can use role-based access control to manage this access restriction, as you'll learn in *Chapter 8*, *Role-based access control in AKS*.
- Anything printed to `stdout` and `stderr` shows up in the logs (independent of the application/language/logging framework).

In this section, you introduced an application error to the guestbook application and leveraged Kubernetes logs to pinpoint the issue in the code. In the next section, you will learn about a powerful mechanism in Kubernetes called **readiness** and **liveness probes**.

Readiness and liveness probes

Readiness and liveness probes were briefly touched upon in the previous section. In this section, you'll explore them in more depth.

Kubernetes uses liveness and readiness probes to monitor the availability of your applications. Each probe serves a different purpose:

- A **liveness probe** monitors the availability of an application while it is running. If a liveness probe fails, Kubernetes will restart your pod. This could be useful to catch deadlocks, infinite loops, or just a "stuck" application.
- A **readiness probe** monitors when your application becomes available. If a readiness probe fails, Kubernetes will not send any traffic to the unready pods. This is useful if your application has to go through some configuration before it becomes available, or if your application has become overloaded but is recovering from the additional load. By having a readiness probe fail, your application will temporarily not get any more traffic, giving it the ability to recover from the increased load.

Liveness and readiness probes don't need to be served from the same endpoint in your application. If you have a smart application, that application could take itself out of rotation (meaning no more traffic is sent to the application) while still being healthy. To achieve this, it would have the readiness probe fail but have the liveness probe remain active.

Let's build this out in an example. You will create two nginx deployments, each with an index page and a health page. The index page will serve as the liveness probe.

Building two web containers

For this example, you'll use a couple of web pages that will be used to connect to a readiness and a liveness probe. The files are present in the code files for this chapter. Let's first create `index1.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Server 1</title>
  </head>
  <body>
    Server 1
  </body>
</html>
```

After that, create `index2.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Server 2</title>
  </head>
  <body>
    Server 2
  </body>
</html>
```

Let's also create a health page, `healthy.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>All is fine here</title>
  </head>
  <body>
    OK
  </body>
</html>
```

In the next step, you'll mount these files to your Kubernetes deployments. To do this, you'll turn each of these into a `configmap` that you will connect to your pods. You've already learned about configmaps in *Chapter 3*, *Application deployment on AKS*. Use the following commands to create the `configmap`:

```
kubectl create configmap server1 --from-file=index1.html
kubectl create configmap server2 --from-file=index2.html
kubectl create configmap healthy --from-file=healthy.html
```

With that out of the way, you can go ahead and create your two web deployments. Both will be very similar, with just the `configmap` changing. The first deployment file (`webdeploy1.yaml`) looks like this:

```
1   apiVersion: apps/v1
2   kind: Deployment
...
17    spec:
18      containers:
19        - name: nginx-1
20          image: nginx:1.19.6-alpine
21          ports:
22            - containerPort: 80
23          livenessProbe:
24            httpGet:
25              path: /healthy.html
26              port: 80
27            initialDelaySeconds: 3
28            periodSeconds: 3
29          readinessProbe:
30            httpGet:
31              path: /index.html
32              port: 80
33            initialDelaySeconds: 3
34            periodSeconds: 3
35          volumeMounts:
36            - name: html
37              mountPath: /usr/share/nginx/html
38            - name: index
39              mountPath: /tmp/index1.html
40              subPath: index1.html
41            - name: healthy
42              mountPath: /tmp/healthy.html
43              subPath: healthy.html
44          command: ["/bin/sh", "-c"]
45          args: ["cp /tmp/index1.html /usr/share/nginx/html/index.
html; cp /tmp/healthy.html /usr/share/nginx/html/healthy.html; nginx;
sleep inf"]
46      volumes:
47        - name: index
48          configMap:
49            name: server1
50        - name: healthy
51          configMap:
52            name: healthy
53        - name: html
54          emptyDir: {}
```

There are a few things to highlight in this deployment:

- **Lines 23-28**: This is the liveness probe. The liveness probe points to the health page. Remember, if the health page fails, the container will restart.
- **Lines 29-32**: This is the readiness probe. The readiness probe in our case points to the index page. If this page fails, the pod will temporarily not be sent any traffic but will remain running.
- **Lines 44-45**: These two lines contain a couple of commands that get executed when the container starts. Instead of simply running the nginx server, this copies the index and ready files in the right location, then starts nginx, and then uses a `sleep` command (so the container keeps running).

You can create this deployment using the following command. You can also deploy the second version for `server 2`, which is similar to `server 1`:

```
kubectl create -f webdeploy1.yaml
kubectl create -f webdeploy2.yaml
```

Finally, you can also create a service (`webservice.yaml`) that routes traffic to both deployments:

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: web
5   spec:
6     selector:
7       app: web-server
8     ports:
9     - protocol: TCP
10     port: 80
11     targetPort: 80
12   type: LoadBalancer
```

You can create that service using the following:

```
kubectl create -f webservice.yaml
```

You now have the application up and running. In the next section, you'll introduce some failures to verify the behavior of the liveness and readiness probes.

Experimenting with liveness and readiness probes

In the previous section, the functionality of the liveness and readiness probes was explained, and you created a sample application. In this section, you will introduce errors in this application and verify the behavior of the liveness and readiness probes. You will see how a failure of the readiness probe will cause the pod to remain running but no longer accept traffic. After that, you will see how a failure of the liveness probe will cause the pod to be restarted.

Let's start by failing the readiness probe.

Failing the readiness probe causes traffic to temporarily stop

Now that you have a simple application up and running, you can experiment with the behavior of the liveness and readiness probes. To start, let's get the service's external IP to connect to our web server using the browser:

```
kubectl get service
```

If you hit the external IP in the browser, you should see a single line that either says **Server 1** or **Server 2**:

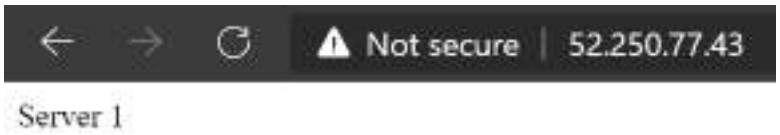


Figure 7.17: Our application is returning traffic from server 1

During the upcoming tests, you'll use a small script called `testWeb.sh` that has been provided in the code samples for this chapter to connect to your web page 50 times, so you can monitor a good distribution of results between servers 1 and 2. You'll first need to make that script executable, and then you can run that script while your deployment is fully healthy:

```
chmod +x testWeb.sh
./testWeb.sh <external-ip>
```

During healthy operations, we can see that server 1 and server 2 are hit almost equally, with 24 hits for server 1 and 26 for server 2:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ ./testWeb.sh 52.250.77.43
server 1:
     24      48     216
server 2:
     26      52     234
```

Figure 7.18: While the application is healthy, traffic is load-balanced between server 1 and server 2

Let's now move ahead and fail the readiness probe in server 1. To do this, you will use the `kubectl exec` command to move the index file to a different location:

```
kubectl get pods #note server1 pod name
kubectl exec <server1 pod name> -- \
  mv /usr/share/nginx/html/index.html \
  /usr/share/nginx/html/index1.html
```

Once this is executed, we can view the change in the pod status with the following command:

```
kubectl get pods -w
```

You should see the readiness state of the server 1 pod change to `0/1`, as shown in *Figure 7.19*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl exec server1-698686949-tffwp -- \
>   mv /usr/share/nginx/html/index.html \
>   /usr/share/nginx/html/index1.html
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods -w
NAME                       READY   STATUS    RESTARTS   AGE
server1-698686949-tffwp    1/1     Running   0          2m30s
server2-6c9f779df7-k7gm7   1/1     Running   0          2m30s
server1-698686949-tffwp    0/1     Running   0          2m33s
```

Figure 7.19: The failing readiness probes causes server 1 to not have any READY containers

This should direct no more traffic to the server 1 pod. Let's verify that:

```
./testWeb.sh <external-ip>
```

Traffic should be redirected to server 2:

Figure 7.20: All traffic is now served by server 2

You can now restore the state of server 1 by moving the file back to its rightful place:

```
kubectl exec <server1 pod name> -- mv \
  /usr/share/nginx/html/index1.html \
  /usr/share/nginx/html/index.html
```

This will return the pod to a **Ready** state and should again split traffic equally:

```
./testWeb.sh <external-ip>
```

This will show an output similar to *Figure 7.21*:

Figure 7.21: Restoring the readiness probe causes traffic to be load-balanced again

A failing readiness probe will cause Kubernetes to no longer send traffic to the failing pod. You have verified this by causing a readiness probe in your example application to fail. In the next section, you'll explore the impact of a failing liveness probe.

A failing liveness probe restarts the pod

You can repeat the previous process with the liveness probe as well. When the liveness probe fails, Kubernetes is expected to restart that pod. Let's try this by deleting the health file:

```
kubectl exec <server 2 pod name> -- \
  rm /usr/share/nginx/html/healthy.html
```

Let's see what this does to the pod:

```
kubectl get pods -w
```

You should see that the pod restarts within a couple of seconds:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl exec server2-6c9f779df7-k7gm7 -- \
>   rm /usr/share/nginx/html/healthy.html
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods -w
NAME                       READY   STATUS    RESTARTS   AGE
server1-698686949-tffwp    1/1     Running   0          3m56s
server2-6c9f779df7-k7gm7   1/1     Running   0          3m56s
server2-6c9f779df7-k7gm7   0/1     Running   1          4m30s
server2-6c9f779df7-k7gm7   1/1     Running   1          4m35s
```

Figure 7.22: A failing liveness probe will cause the pod to be restarted

As you can see in *Figure 7.22*, the pod was successfully restarted, with limited impact. You can inspect what was going on in the pod by running a `describe` command:

```
kubectl describe pod <server2 pod name>
```

The preceding command will give you an output similar to *Figure 7.23*:

```
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  5m8s                 default-scheduler  Successfully assigned default/server2-6c9f779df7-k7gm7 to aks-agentpool-39838025-vmss000000
  Warning  Unhealthy  69s (x3 over 75s)    kubelet            Liveness probe failed: HTTP probe failed with statuscode: 404
  Normal   Killing    69s                  kubelet            Container nginx-2 failed liveness probe, will be restarted
  Normal   Pulled     39s (x2 over 5m8s)   kubelet            Container image "nginx:1.19.4-alpine" already present on machine
  Normal   Created    39s (x2 over 5m8s)   kubelet            Created container nginx-2
  Normal   Started    39s (x2 over 5m8s)   kubelet            Started container nginx-2
```

Figure 7.23: More details on the pod showing how the liveness probe failed

In the `describe` command, you can clearly see that the pod failed the liveness probe. After three failures, the container was killed and restarted.

This concludes the experiment with liveness and readiness probes. Remember that both are useful for your application: a readiness probe can be used to temporarily stop traffic to your pod, so it has to deal with less load. A liveness probe is used to restart your pod if there is an actual failure in the pod.

Let's also make sure to clean up the deployments you just created:

```
kubectl delete deployment server1 server2
kubectl delete service web
```

Liveness and readiness probes are useful to ensure that only healthy pods will receive traffic in your cluster. In the next section, you will explore different metrics reported by Kubernetes that you can use to verify the state of your application.

Metrics reported by Kubernetes

Kubernetes reports multiple metrics. In this section, you'll first use a number of `kubectl` commands to get these metrics. Afterward, you'll look into Azure Monitor for containers to see how Azure helps with container monitoring.

Node status and consumption

The nodes in your Kubernetes are the servers running your application. Kubernetes will schedule pods to different nodes in the cluster. You need to monitor the status of your nodes to ensure that the nodes themselves are healthy and that the nodes have enough resources to run new applications.

Run the following command to get information about the nodes on the cluster:

```
kubectl get nodes
```

The preceding command lists their name, status, and age:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-agentpool-39838025-vmss000000   Ready    agent   3d      v1.19.6
aks-agentpool-39838025-vmss000002   Ready    agent   2d23h   v1.19.6
```

Figure 7.24: There are two nodes in this cluster

You can get more information by passing the `-o wide` option:

```
kubectl get -o wide nodes
```

The output lists the underlying `OS-IMAGE` and `INTERNAL-IP`, and other useful information, which can be viewed in *Figure 7.25*:

Figure 7.25: Using -o wide adds more details about the nodes

You can find out which nodes are consuming the most resources by using the following command:

```
kubectl top nodes
```

It shows the CPU and memory usage of the nodes:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl top nodes
NAME                                CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
aks-agentpool-39838025-vmss000000   41m          2%     648Mi           14%
aks-agentpool-39838025-vmss000002   154m         8%     966Mi           21%
```

Figure 7.26: CPU and memory utilization of the nodes

Note that this is the actual consumption at that point in time, not the number of requests a certain node has. To get the requests, you can execute the following:

```
kubectl describe node <node name>
```

This will show you the requests and limits per pod, as well as the cumulative amount for the whole node:

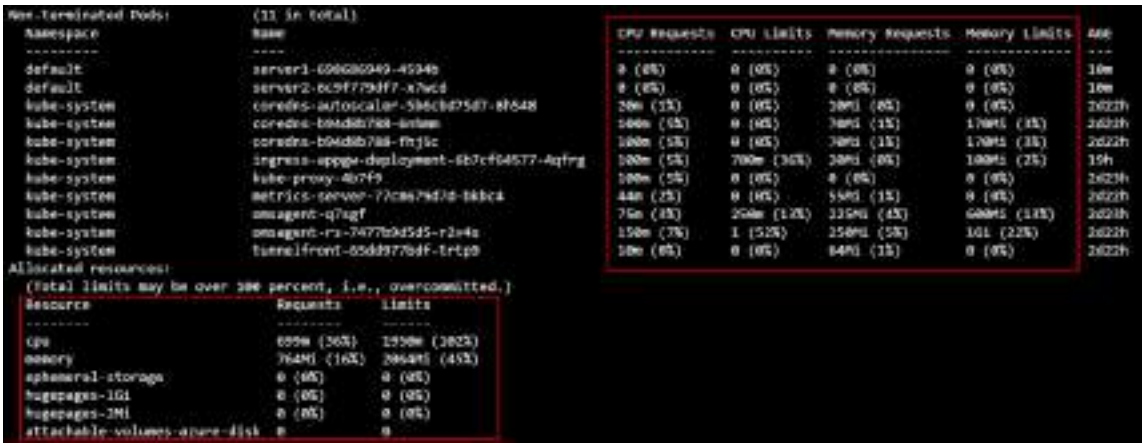


Figure 7.27: Describing the nodes shows details on requests and limits

As you can see in *Figure* 7.27, the `describe node` command outputs the requests and limits per pod, across namespaces. This is a good way for cluster operators to verify how much load is being put on the cluster, across all namespaces.

You now know where you can find information about the utilization of your nodes. In the next section, you will look into how you can get the same metrics for individual pods.

Pod consumption

Pods consume CPU and memory resources from an AKS cluster. Requests and limits are used to configure how much CPU and memory a pod can consume. Requests are used to reserve a minimum amount of CPU and memory, while limits are used to set a maximum amount of CPU and memory per pod.

In this section, you will learn how you can use `kubectl` to get information about the CPU and memory utilization of pods.

Let's start by exploring how you can see the requests and limits for a pod that you currently have running:

1. For this example, you will use the pods running in the `kube-system` namespace. Get all the pods in this namespace:

   ```
   kubectl get pods -n kube-system
   ```

   This should show something similar to *Figure 7.28*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods -n kube-system
NAME                                    READY   STATUS    RESTARTS   AGE
coredns-autoscaler-5b6cbd75d7-8h548     1/1     Running   1          2d22h
coredns-b94d8b788-6nhmm                 1/1     Running   1          2d22h
coredns-b94d8b788-fhj5c                 1/1     Running   1          2d22h
ingress-appgw-deployment-6b7cf64577-4qfrg   1/1     Running   0          19h
kube-proxy-4b7f9                        1/1     Running   1          2d23h
kube-proxy-x66h9                        1/1     Running   2          3d
metrics-server-77c8679d7d-bkbc4         1/1     Running   1          2d22h
omsagent-n796q                          1/1     Running   2          3d
omsagent-q7sgf                          1/1     Running   1          2d23h
omsagent-rs-7477b9d5d5-r2v4s            1/1     Running   1          2d22h
tunnelfront-65dd977bdf-trtp9            1/1     Running   1          2d22h
```

Figure 7.28: The pods running in the kube-system namespace

2. Let's get the requests and limits for one of the `coredns` pods. This can be done using the `describe` command:

   ```
   kubectl describe pod coredns-<pod id> -n kube-system
   ```

   In the `describe` command, there should be a section similar to *Figure 7.29*:

```
Limits:
  memory:  170Mi
Requests:
  cpu:      100m
  memory:   70Mi
```

Figure 7.29: Limits and requests for the CoreDNS pod

This shows you that this pod has a memory limit of `170Mi`, no CPU limit, and has a request for 100 m CPU (which means 0.1 CPU) and `70Mi` of memory. This means that if this pod were to consume more than 170 MiB of memory, Kubernetes would restart that pod. Kubernetes has also reserved 0.1 CPU core and 70 MiB of memory for this pod.

Requests and limits are used to perform capacity management in a cluster. You can also get the actual CPU and memory consumption of a pod. Run the following command and you'll get the actual pod consumption in all namespaces:

```
kubectl top pods --all-namespaces
```

This should show you anoutput similar to *Figure* 7.30:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl top pods --all-namespaces
NAMESPACE     NAME                                      CPU(cores)   MEMORY(bytes)
default       server1-698686949-4594b                   1m           3Mi
default       server2-6c9f779df7-x7wcd                  1m           3Mi
kube-system   coredns-autoscaler-5b6cbd75d7-8h548       1m           7Mi
kube-system   coredns-b94d8b788-6nhmm                   3m           10Mi
kube-system   coredns-b94d8b788-fhj5c                   3m           10Mi
kube-system   ingress-appgw-deployment-6b7cf64577-4qfrg  4m           12Mi
kube-system   kube-proxy-4b7f9                          1m           19Mi
kube-system   kube-proxy-x66h9                          1m           22Mi
kube-system   metrics-server-77c8679d7d-bkbc4           2m           13Mi
kube-system   omsagent-n796q                            8m           142Mi
kube-system   omsagent-q7sgf                            8m           149Mi
kube-system   omsagent-rs-7477b9d5d5-r2v4s              6m           156Mi
kube-system   tunnelfront-65dd977bdf-trtp9              83m          63Mi
```

Figure 7.30: Seeing the CPU and memory consumption of pods

Using the `kubectl top` command shows the CPU and memory consumption at the point in time when the command was run. In this case, you can see that the `coredns` pods are using `3m` CPU and `10Mi` of memory.

In this section, you have used the `kubectl` command to get an insight into the resource utilization of the nodes and pods in your cluster. This is useful information, but it is limited to that specific point in time. In the next section, you'll use the Azure portal to get more detailed information on the cluster and the applications on top of the cluster. You'll start by exploring the **AKS Diagnostics** pane.

Using AKS Diagnostics

When you are experiencing issues in AKS, a good place to start your exploration is the **AKS Diagnostics** pane. It provides you with tools that help investigate any issues related to underlying infrastructure or system cluster components.

> **Note:**
>
> AKS Diagnostics is in preview at the time of writing this book. This means functionality might be added or removed.

To access AKS Diagnostics, hit the **Diagnose and solve problems** option in the AKS menu. This will open up Diagnostics, as shown in *Figure 7.31*:

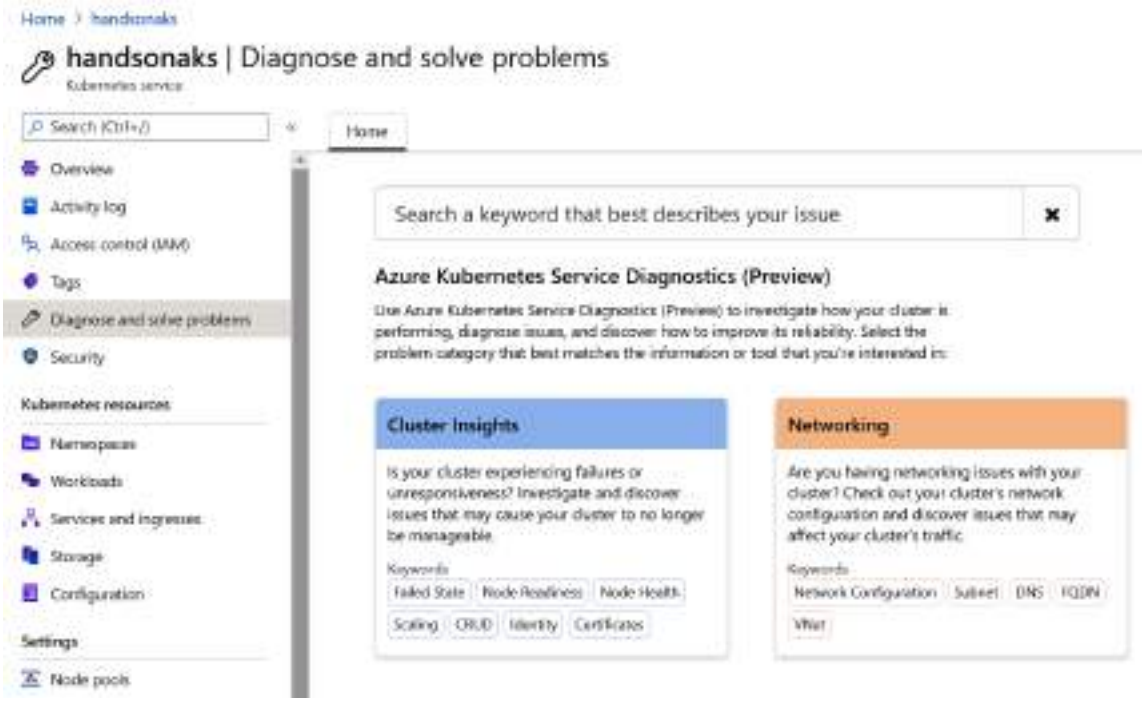


Figure 7.31: Accessing AKS Diagnostics

AKS Diagnostics gives you two tools to diagnose and explore issues. One is **Cluster Insights**, and the other is **Networking**. Cluster Insights uses cluster logs and configuration on your cluster to perform a health check and compare your cluster against best practices. It contains useful information and relevant health indicators in case anything is misconfigured in your cluster. An example output of Cluster Insights is shown in *Figure 7.32*:

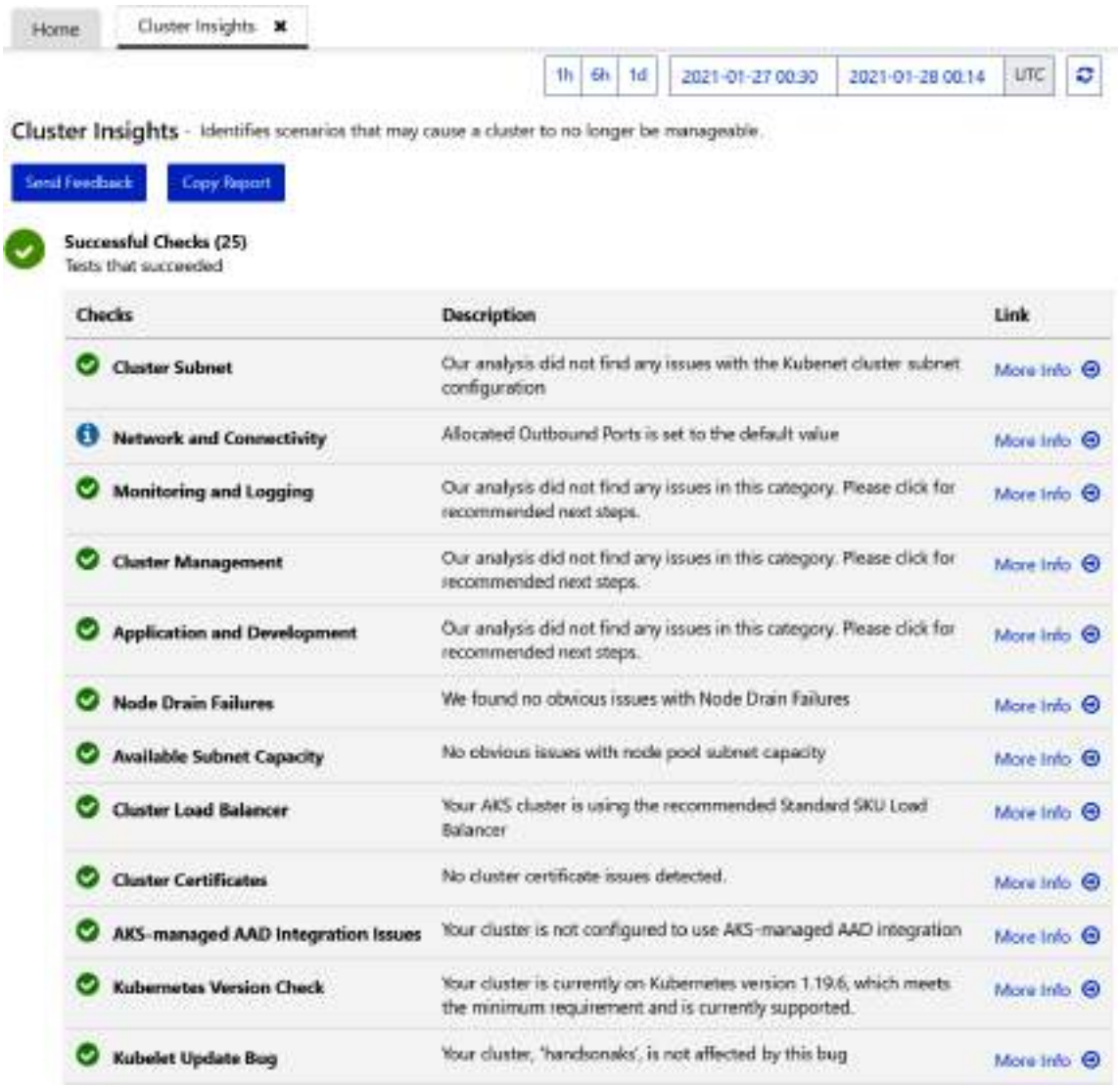


Figure 7.32: Example output from Cluster Insights

The **Networking** section of AKS Diagnostics allows you to interactively troubleshoot networking issues in your cluster. As you open the **Networking** view, you are presented with several questions that will then trigger network health checks and configuration reviews. Once you select one of those options, the interactive tool will give you the output from those checks, as shown in *Figure* 7.33:

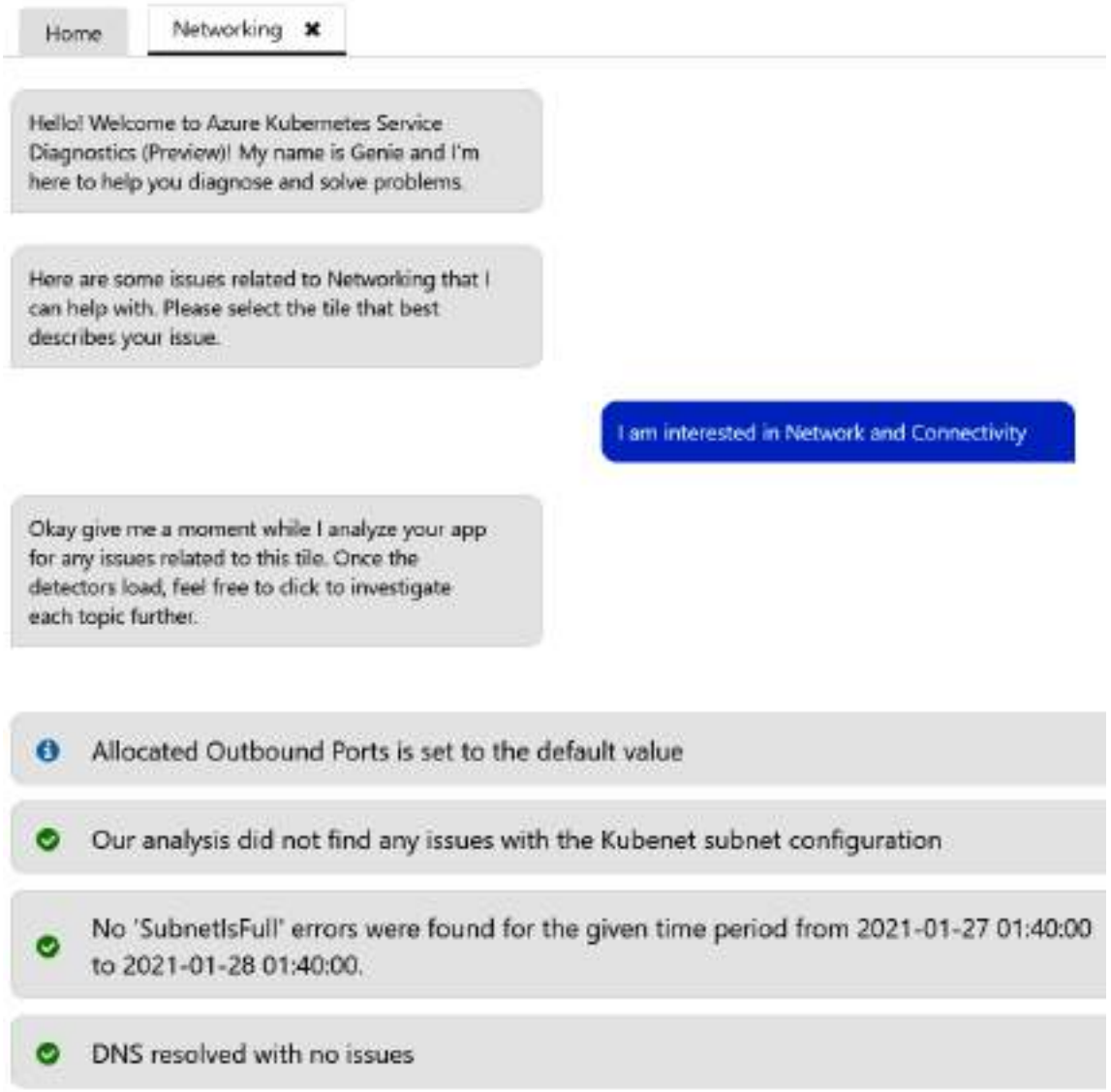


Figure 7.33: Diagnosing networking issues using AKS Diagnostics

Using AKS Diagnostics is very useful when you are facing infrastructure issues on your cluster. The tool does a scan of your environment and verifies whether everything is running and configured well. However, it does not scan your applications. That is where Azure Monitor comes in; it allows you to monitor your application and access your application logs.

Azure Monitor metrics and logs

Previously in this chapter, you explored the status and metrics of nodes and pods in your cluster using the `kubectl` command-line tool. In Azure, you can get more metrics from nodes and pods and explore the logs from pods in your cluster. Let's start by exploring AKS Insights in the Azure portal.

AKS Insights

The **Insights** section of the AKS pane provides most of the metrics you need to know about your cluster. It also has the ability to drill down to the container level. You can also see the logs of the container.

> **Note:**
>
> The Insights section of the AKS pane relies on Azure Monitor for containers. If you created the cluster using the portal defaults, this is enabled by default.

Kubernetes makes metrics available but doesn't store them. Azure Monitor can be used to store these metrics and make them available to query over time. To collect the relevant metrics and logs into Insights, Azure connects to the Kubernetes API to collect the metrics and logs to then store them in Azure Monitor.

> **Note:**
>
> Logs of a container could contain sensitive information. Therefore, the rights to review logs should be controlled and audited.

Let's explore the **Insights** tab of the AKS pane, starting with the cluster metrics.

Cluster metrics

Insights shows the cluster metrics. *Figure* 7.34 shows the CPU utilization and the memory utilization of all the nodes in the cluster. You can optionally add additional filters to filter to a particular namespace, node, or node pool. There also is a live option, which gives you more real-time information on your cluster status:

Figure 7.34: The Cluster tab shows CPU and memory utilization for the cluster

The cluster metrics also show the node count and the number of active pods. The node count is important, as you can track whether you have any nodes that are in a **Not Ready** state:

Figure 7.35: The Cluster tab shows the node count and the number of active pods

The **Cluster** tab can be used to monitor the status of the nodes in the cluster. Next, you'll explore the **Reports** tab.

Reports

The **Reports** tab in AKS Insights gives you access to a number of preconfigured monitoring workbooks. These workbooks combine text, log queries, metrics, and parameters together and give you rich interactive reports. You can drill down into each individual report to get more information and prebuilt log queries. The available reports are shown in *Figure* 7.36:

> **Note**
>
> The Reports functionality is in preview at the time of writing this book.

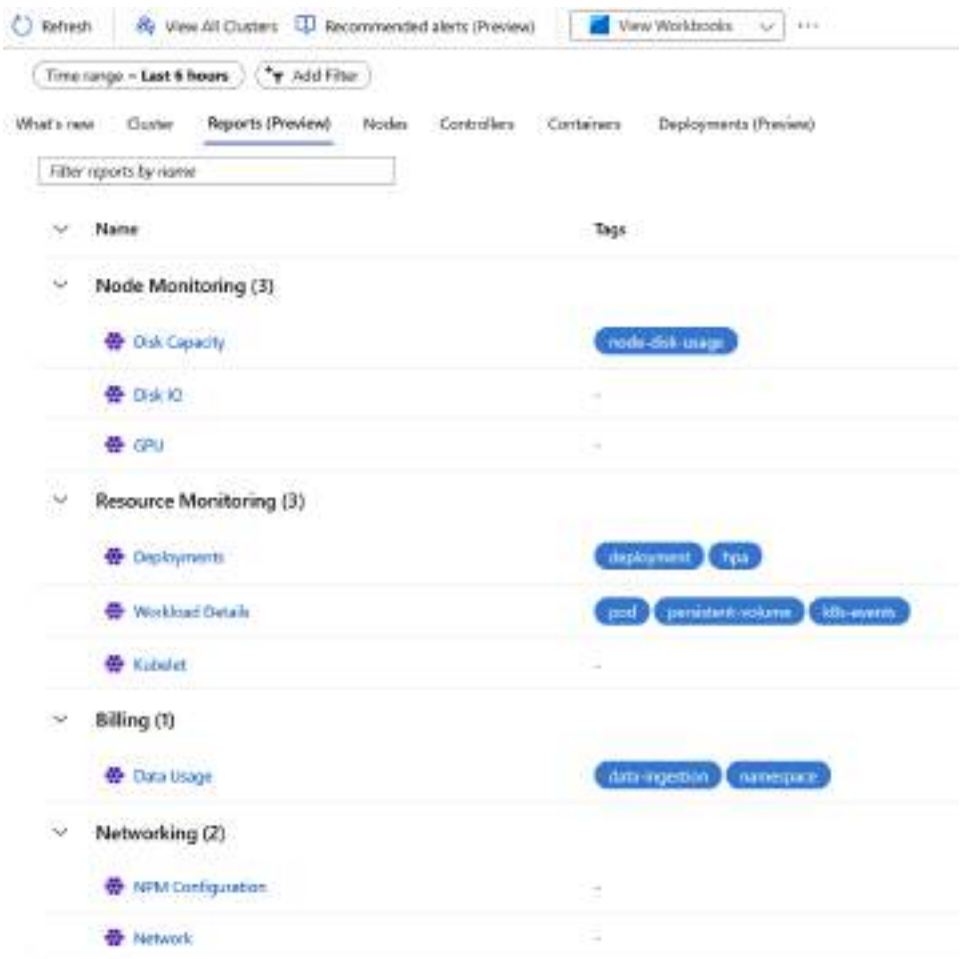


Figure 7.36: The Reports tab gives you access to preconfigured monitoring workbooks

As an example, you can explore the **Deployments** workbook. This is shown in *Figure* 7.37:

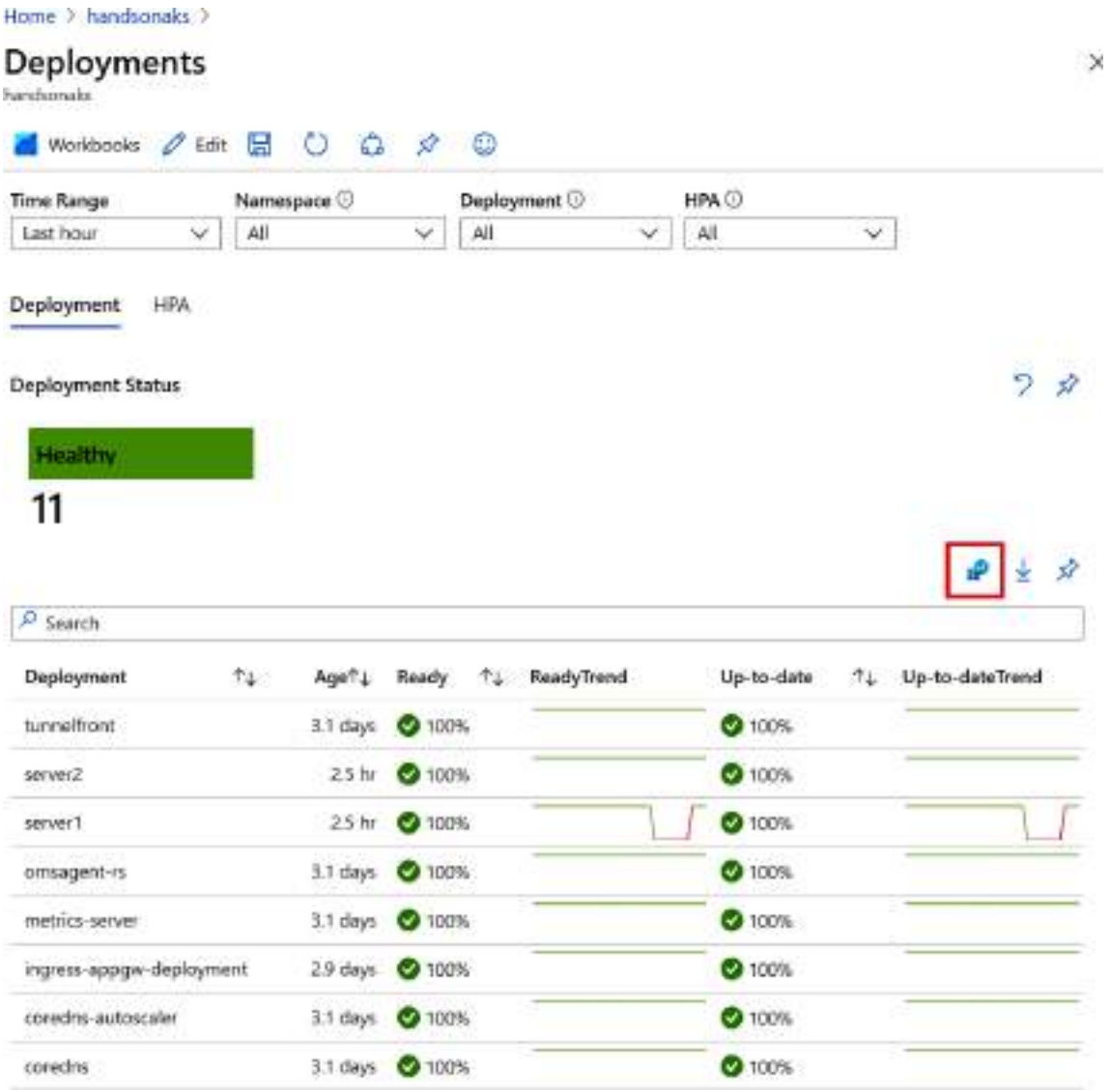


Figure 7.37: The Deployments workbook shows you the status of your deployments

This shows you all the deployments by default, their health, and up-to-date status. As you can see, it shows you that **server1** was temporarily unavailable when you were doing the exploration with liveness and readiness probes earlier in this chapter.

You can drill down further into the status of the individual deployments. If you click on the **Log** button highlighted in *Figure* 7.37, you get redirected to Log Analytics with a prebuilt query. You can then modify this query and get deeper insights into your workload, as shown in *Figure* 7.38.

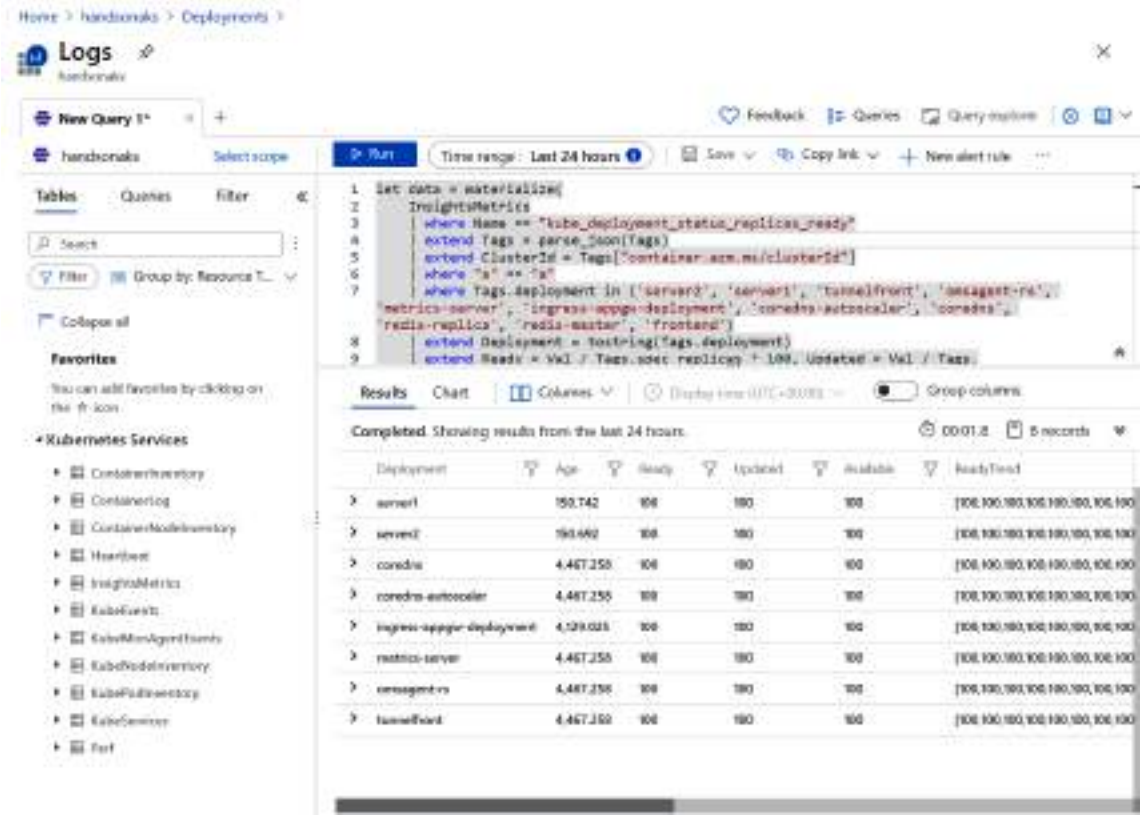


Figure 7.38: Drilling down in Log Analytics to get more details on your deployments

> **Note:**
>
> The queries used in Log Analytics make use of the **Kusto Query Language** (**KQL**). To learn more about KQL, please refer to the documentation: https://docs.microsoft.com/azure/data-explorer/kusto/concepts/

The **Reports** tab in AKS Insights gives you a number of prebuilt monitoring workbooks. The next tab is the **Nodes** tab.

Nodes

The **Nodes** view shows you detailed metrics for your nodes. It also shows you which pods are running on each node, as you can see in *Figure* 7.39:

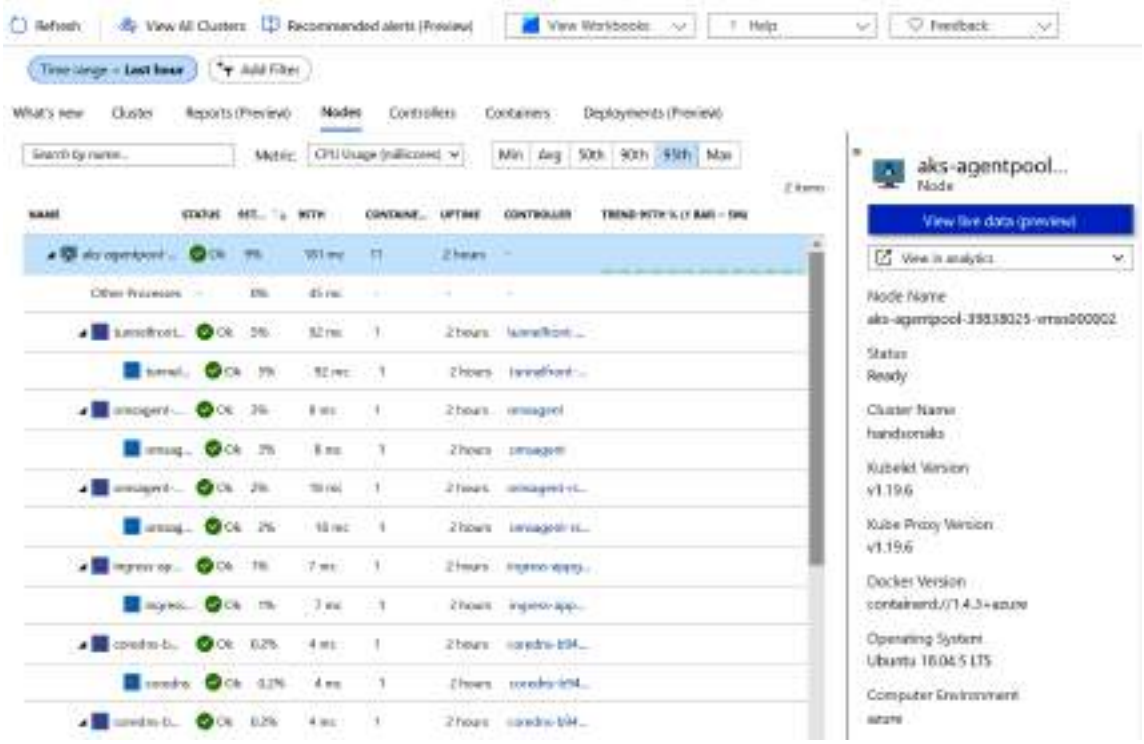


Figure 7.39: Detailed metrics of the nodes in the Nodes pane

Note that different metrics can be viewed from the dropdown menu right next to the search bar. If you need even more details, you can click through and get Kubernetes event logs from your nodes as well:

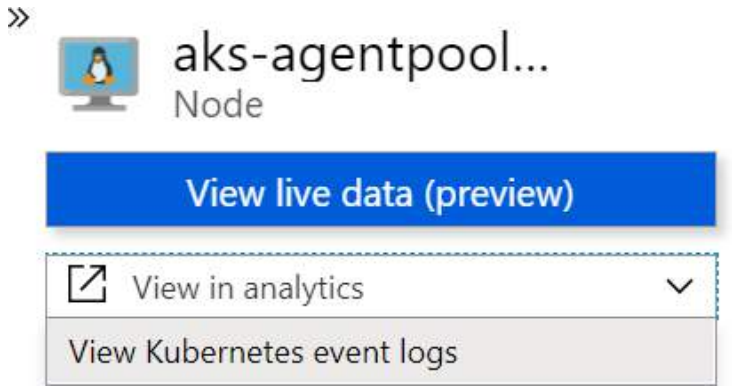


Figure 7.40: Click on View Kubernetes event logs to get the logs from a cluster

This will open Azure Log Analytics and will have pre-created a query for you that shows the logs for your node. In the example in *Figure* 7.41, you can see that the node was rebooted a couple of times and hit an `InvalidDiskCapacity` warning as well:

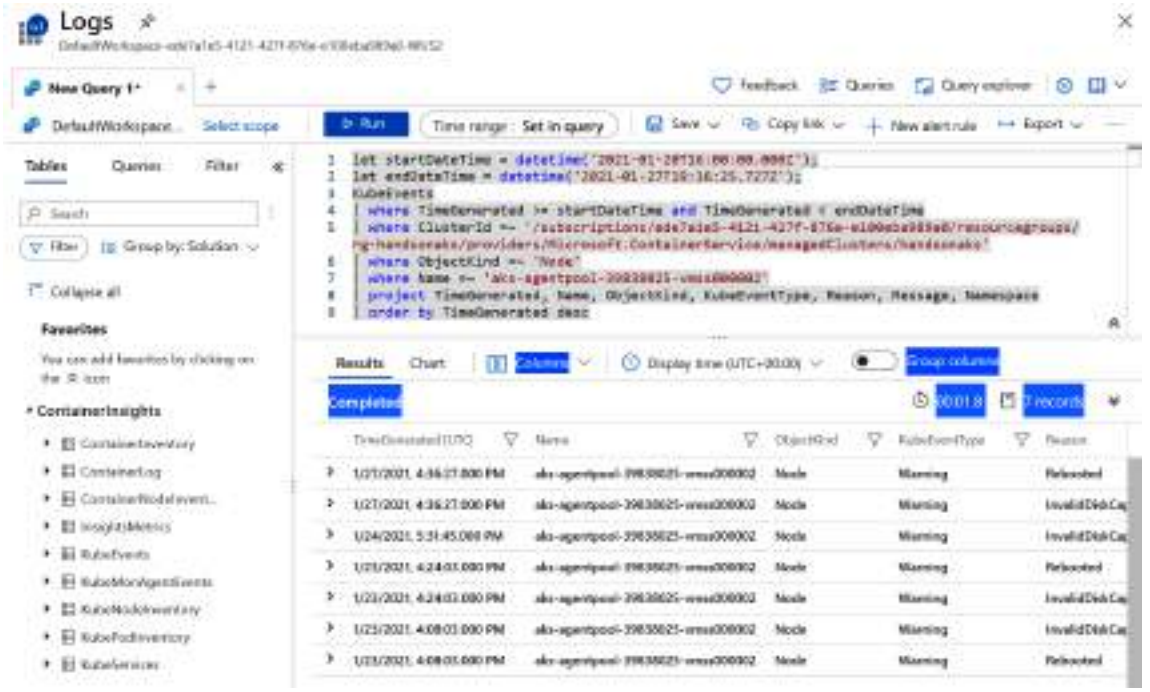


Figure 7.41: Log Analytics showing the logs for the nodes

This gives you information about the status of your nodes. Next, you'll explore the **Controllers** tab.

Controllers

The **Controllers** tab shows you details on all the controllers (that is, ReplicaSets, DaemonSets, and so on) on your cluster and the pods running in them. This shows you a controller-centric view of running pods. For instance, you can find the **server1** ReplicaSet and see all the pods and containers running in it, as shown in *Figure 7.42*:

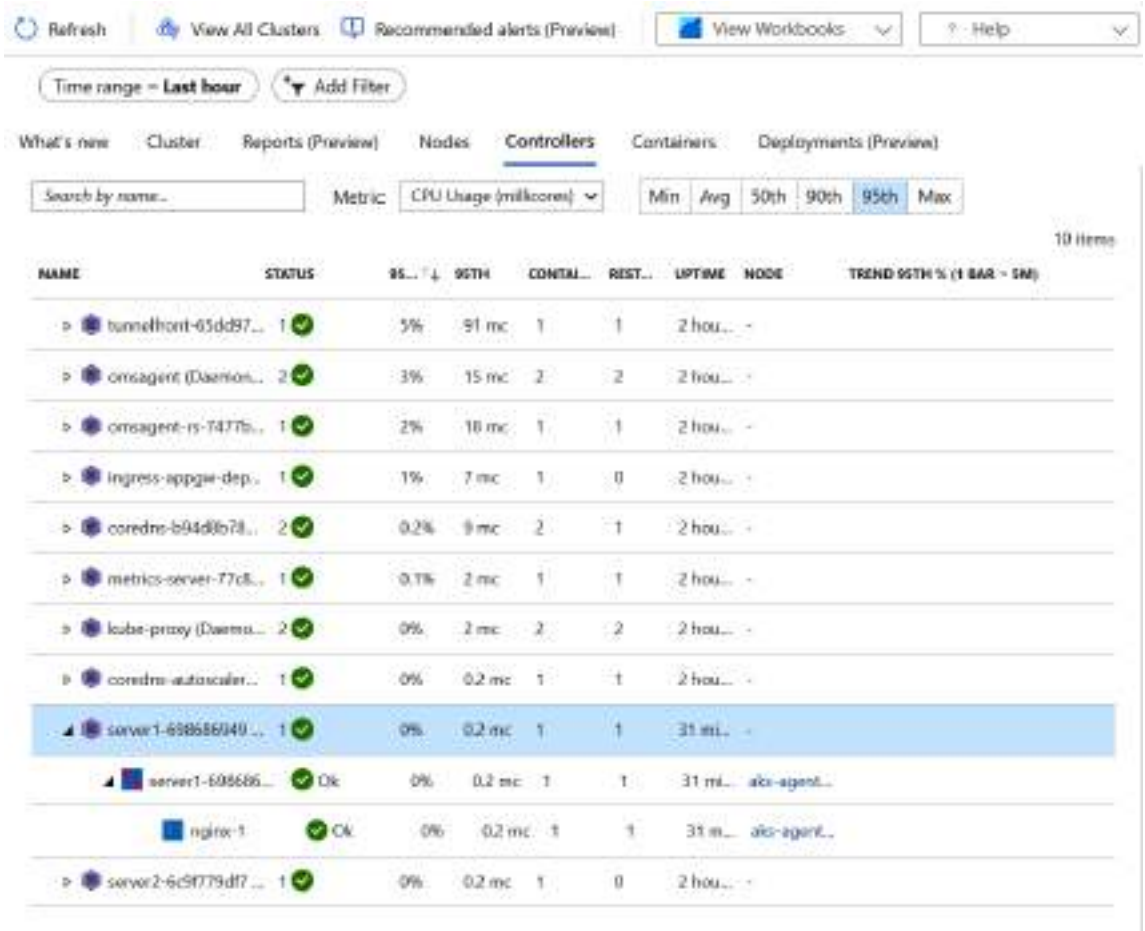


Figure 7.42: The Controllers tab shows you all the pods running in a ReplicaSet

The next tab is the **Containers** tab, which will show you the metrics, logs, and environment variables for a container.

Container metrics, logs, and environment variables

Clicking on the **Containers** tab lists the container metrics, environment variables, and access to its logs, as shown in *Figure* 7.43:

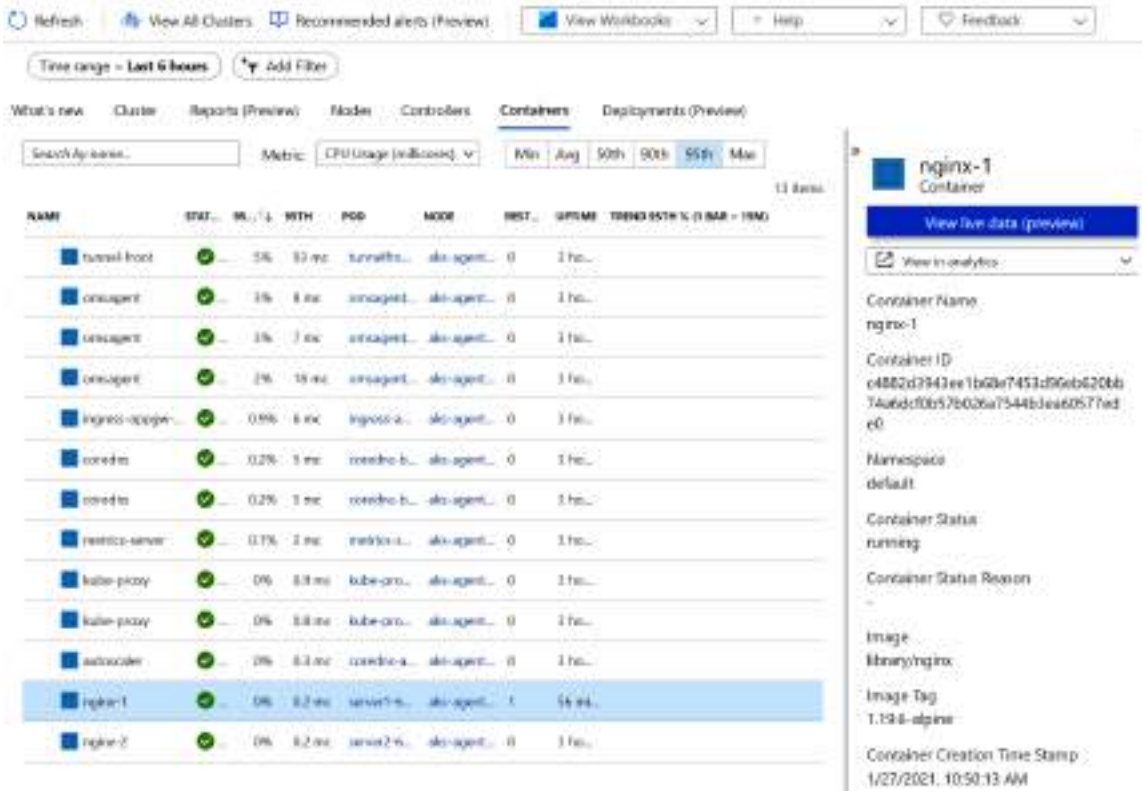


Figure 7.43: The Containers tab shows us all the individual containers

> **Note:**
>
> You might notice a couple of containers with an `Unknown` state. If a container in the **Insights** pane has an unknown status, that is because Azure Monitor has logs and information about that container, but the container is no longer running on the cluster.

You can get access to the container's logs from this view as well:

Figure 7.44: Access the container's logs

This will show you all the logs that Kubernetes logged from your application. Earlier in the chapter, you used `kubectl` to get access to container logs. Using this approach can be a lot more productive, as you can edit the log queries and correlate logs from different pods and applications in a single view:

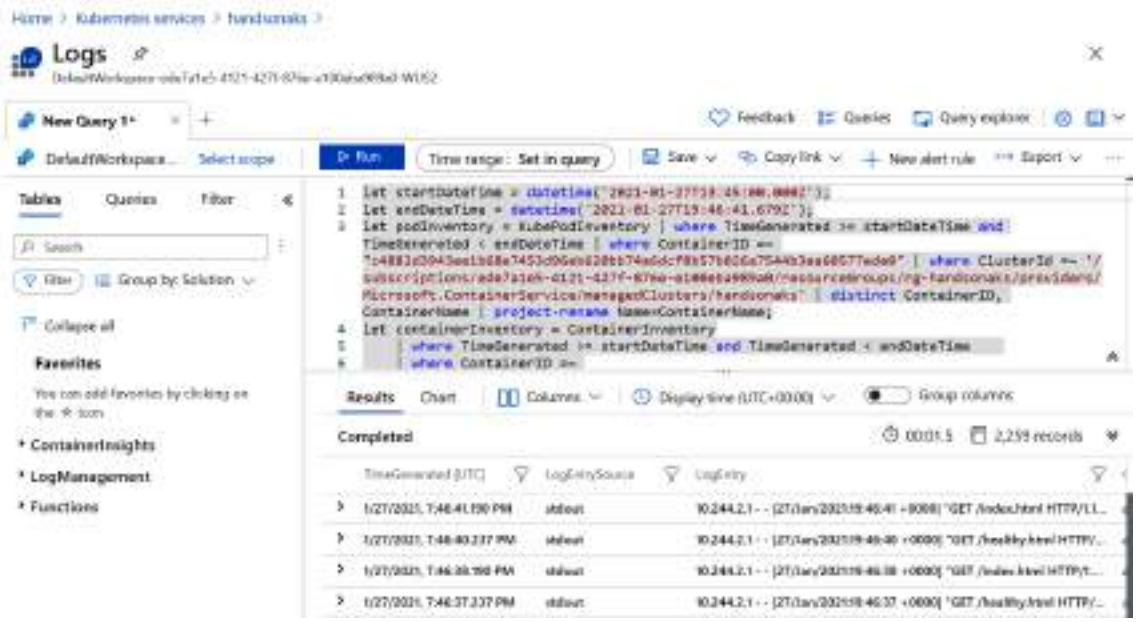


Figure 7.45: Logs are collected and can be queried

Apart from the logs, this view also shows the environment variables that are set for the container. To see the environment variables, scroll down in the right cell of the **Containers** view:

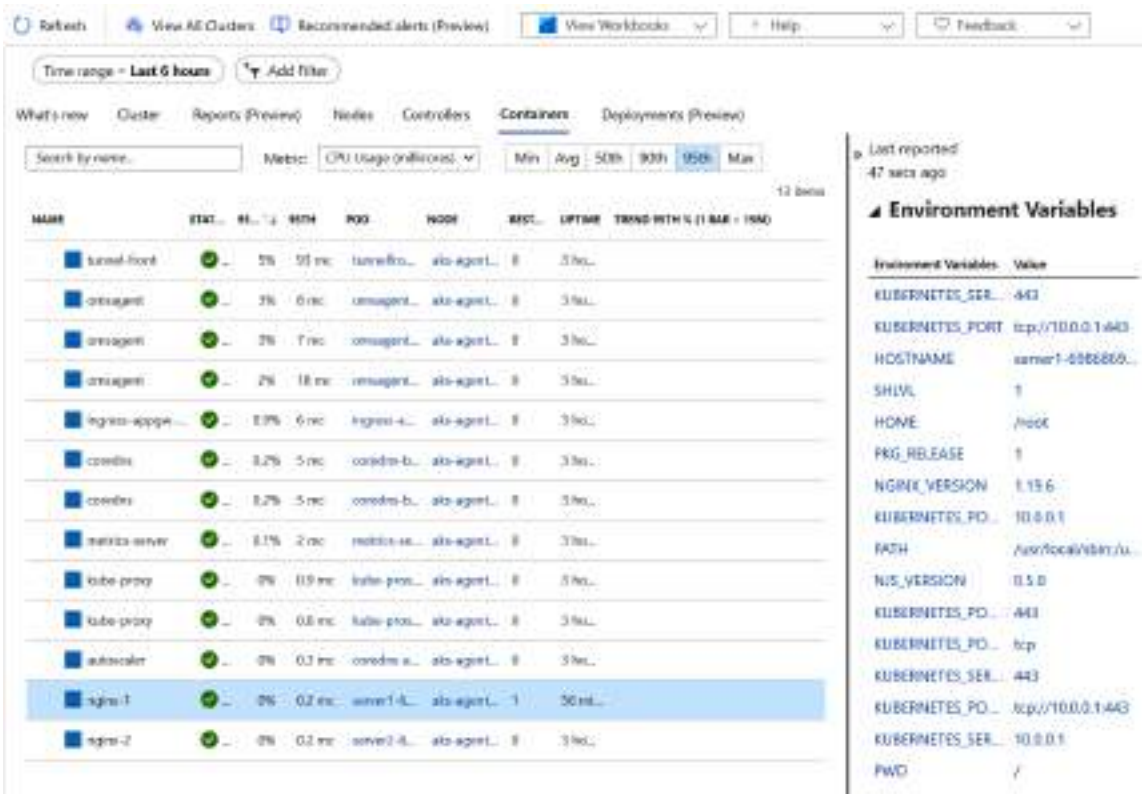


Figure 7.46: The environment variables set for the container

The final tab in AKS Insights is the **Deployments** tab, which you'll explore next.

Deployments

The final tab is the **Deployments** tab. This tab gives you an overview of all deployments in the cluster and allows you to get the definition of the deployment by selecting it. As you can see in *Figure* 7.47, you can get this view either in **Describe** (in text format) or in **RAW** (YAML format):

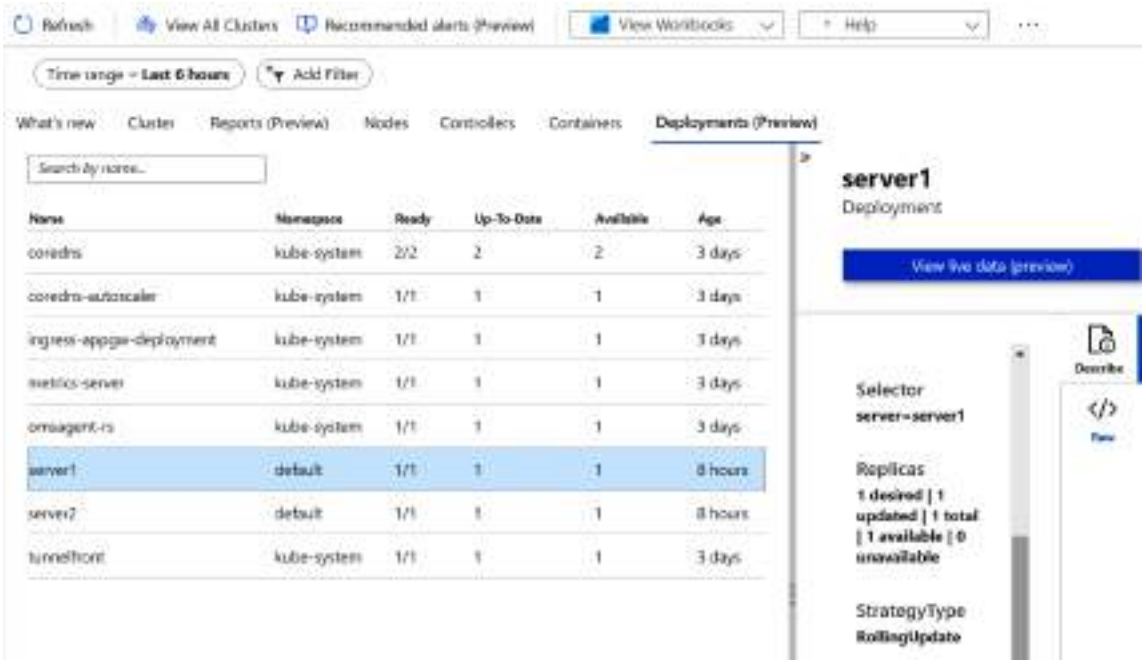


Figure 7.47: The Deployments tab in AKS Insights

By using the **Insights** pane in AKS, you can get detailed information about your cluster. You explored the different tabs in this section and learned how you can drill down and get access to customizable log queries to get even more information.

And that concludes this section. Let's make sure to clean up all the resources created in this chapter by using the following command:

```
kubectl delete -f
```

In this section, you explored monitoring applications running on top of Kubernetes. You used the AKS **Insights** tab in the Azure portal to get a detailed view of your cluster and the containers running on the cluster.

Summary

You started this chapter by learning how to use different `kubectl` commands to monitor an application. Then, you explored how logs created in Kubernetes can be used to debug that application. The logs contain all the information that is written to `stdout` and `stderr`.

After that, you switched to the Azure portal and started using AKS Diagnostics to explore infrastructure issues. Lastly, you explored the use of Azure Monitor and AKS Insights to show the AKS metrics and environment variables, as well as logs with log filtering.

In the next chapter, you will learn how to connect an AKS cluster to Azure PaaS services. You will specifically focus on how you can connect an AKS cluster to a MySQL database managed by Azure.

Section 3: Securing your AKS cluster and workloads

Loose lips sink ships is a phrase that describes how easy it can be to jeopardize the security of a Kubernetes-managed cluster (*Kubernetes*, by the way, is Greek for *helmsman*, as in the helmsman of a *ship*). If your cluster is left open with the wrong ports or services exposed, or plain text is used for secrets in application definitions, bad actors can take advantage of this negligent security and do pretty much whatever they want in your cluster.

There are multiple items to consider when securing an **Azure Kubernetes Service** (**AKS**) cluster and workloads running on top of it. In this section, you will learn about four ways to secure your cluster and applications. You will learn about role-based access control in Kubernetes and how this can be integrated with **Azure Active Directory** (**Azure AD**). After that, you'll learn how to allow your pods to get access to Azure resources such as Blob Storage or Key Vault using an Azure AD pod identity. Subsequently, you'll learn about Kubernetes secrets and how to safely integrate them with Key Vault. Finally, you'll learn about network security and how to isolate your Kubernetes cluster.

In this chapter, you will be routinely deleting clusters and creating new clusters with new functionalities enabled. The reason you will delete existing clusters is to save costs and optimize the free trial, if you are using it.

This section contains the following chapters:

- *Chapter 8*, *Role-based access control in AKS*
- *Chapter 9*, *Azure Active Directory pod-managed identities in AKS*
- *Chapter 10*, *Storing secrets in AKS*
- *Chapter 11*, *Network security in AKS*

You will start this section with *Chapter 8*, *Role-based access control in AKS*, in which you will configure role-based access control in Kubernetes and integrate this with Azure AD.

8

Role-based access control in AKS

Up to this point, you've been using a form of access to **Azure Kubernetes Service** (**AKS**) that gave you permissions to create, read, update, and delete all objects in your cluster. This has worked great for testing and development but is not recommended on production clusters. On production clusters, the recommendation is to leverage **role-based access control** (**RBAC**) in Kubernetes to only grant a limited set of permissions to users.

In this chapter, you will explore Kubernetes RBAC in more depth. You will be introduced to the concept of RBAC in Kubernetes. You will then configure RBAC in Kubernetes and integrate it with **Azure Active Directory** (**Azure AD**).

The following topics will be covered in this chapter:

- RBAC in Kubernetes
- Enabling Azure AD integration in your AKS cluster
- Creating a user and a group in Azure AD
- Configuring RBAC in AKS
- Verifying RBAC for a user

> **Note**
>
> To complete the example on RBAC, you need access to an Azure AD instance, with global administrator permissions.

Let's start this chapter by explaining RBAC.

RBAC in Kubernetes explained

In production systems, you need to allow different users different levels of access to certain resources; this is known as **RBAC**. The benefit of establishing RBAC is that it not only acts as a guardrail against the accidental deletion of critical resources but also is an important security feature that limits full access to the cluster to roles that really need it. On an RBAC-enabled cluster, users can only access and modify those resources for which they have permission.

Up until now, using Cloud Shell, you have been acting as *root*, which allowed you to do anything and everything in the cluster. For production use cases, root access is dangerous and should be restricted as much as possible. It is a generally accepted best practice to use the **principle of least privilege** (**PoLP**) to sign in to any computer system. This prevents both access to secure data and unintentional downtime through the deletion of key resources. Anywhere between 22% and 29% of data loss is attributed to human error. You don't want to be a part of that statistic.

Kubernetes developers realized this was a problem and added RBAC to Kubernetes along with the concept of service roles to control access to clusters. Kubernetes RBAC has three important concepts:

- **Role**: A role contains a set of permissions. A role defaults to `no permissions`, and every permission needs to be specifically called out. Examples of permissions include `get`, `watch`, and `list`. The role also contains which resources these permissions are given to. Resources can be either all pods, deployments, and so on, or can be a specific object (such as `pod/mypod`).

- **Subject**: The subject is either a person or a service account that is assigned a role. In AKS clusters integrated with Azure AD, these subjects can be Azure AD users or groups.
- **RoleBinding**: A RoleBinding links a subject to a role in a certain namespace or, in the case of a ClusterRoleBinding, the whole cluster.

An important concept to understand is that when interfacing with AKS, there are two layers of RBAC: Azure RBAC and Kubernetes RBAC, as shown in *Figure 8.1*. Azure RBAC deals with the roles given to people to make changes in Azure, such as creating, modifying, and deleting clusters. Kubernetes RBAC deals with the access rights to resources in a cluster. Both are independent control planes but can use the same users and groups originating in Azure AD.

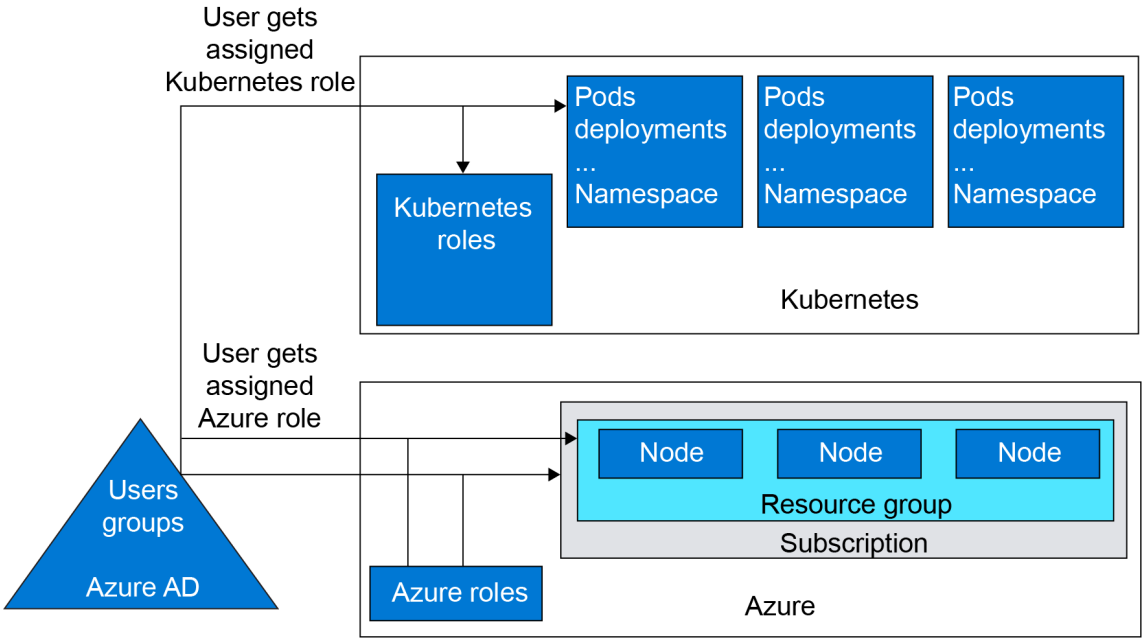


Figure 8.1: Two different RBAC planes, Azure and Kubernetes

RBAC in Kubernetes is an optional feature. The default in AKS is to create clusters that have RBAC enabled. However, by default, the cluster is not integrated with Azure AD. This means that by default you cannot grant Kubernetes permissions to Azure AD users. In the coming section, you will enable Azure AD integration in your cluster.

Enabling Azure AD integration in your AKS cluster

In this section, you will update your existing cluster to include Azure AD integration. You will do this using the Azure portal:

> **Note**
>
> Once a cluster has been integrated with Azure AD, this functionality cannot be disabled.

1. To start, you will need an Azure AD group. You will later give admin privileges for your AKS cluster to this group. To create this group, search for `azure active directory` in the Azure search bar:

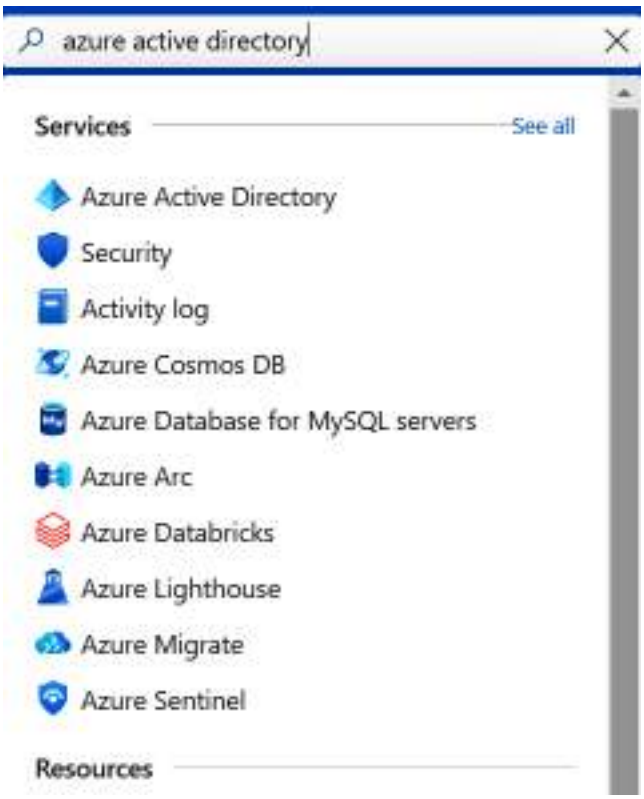


Figure 8.2: Searching for azure active directory in the Azure search bar

2. In the left pane, select **Groups**, which will bring you to the **All groups** screen. Click **+ New Group**, as shown in *Figure 8.3*:

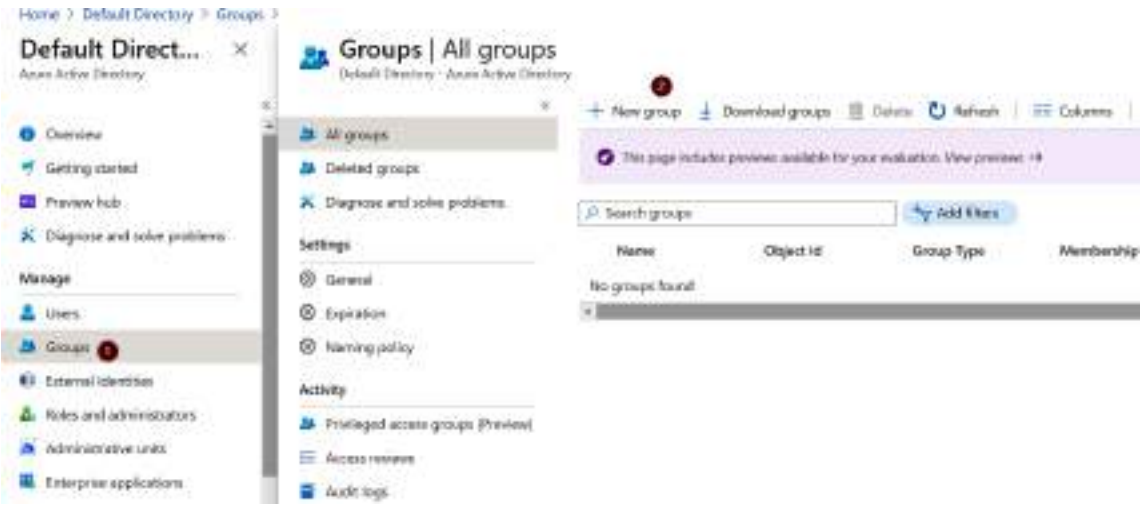


Figure 8.3: Creating a new Azure AD group

3. On the resulting page, create a security group and give it a name and description. Select your user as the owner and a member of this group. Click the **Create** button on the screen:

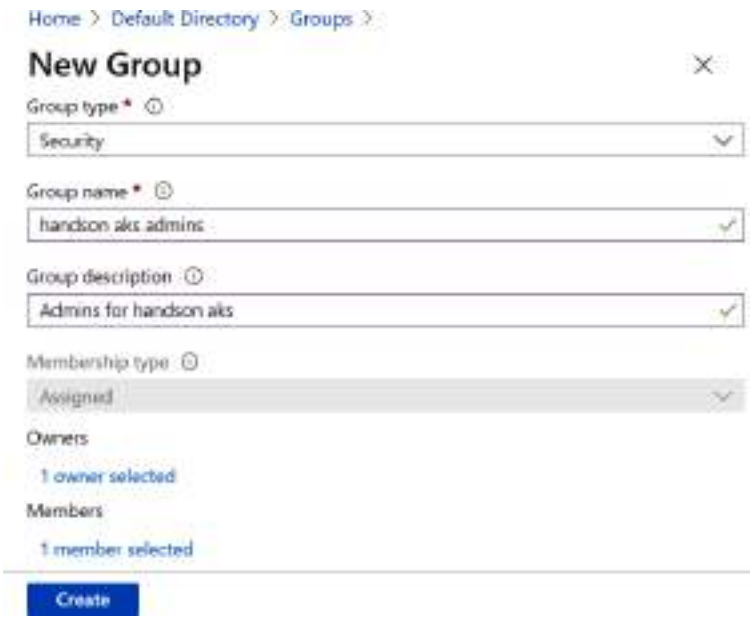


Figure 8.4: Providing details for creating the Azure AD group

4. Now that this group is created, search for your Azure cluster in the Azure search bar to open the AKS pane:

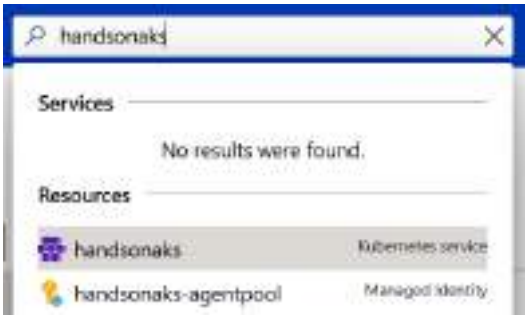


Figure 8.5: Searching for your cluster in the Azure search bar

5. In the AKS pane, select **Cluster configuration** under **Settings**. In this pane, you will be able to turn on **AKS-managed Azure Active Directory**. Enable the functionality and select the Azure AD group you created earlier to set as the admin Azure AD group. Finally, hit the **Save** button in the command bar, as shown in *Figure 8.6*:

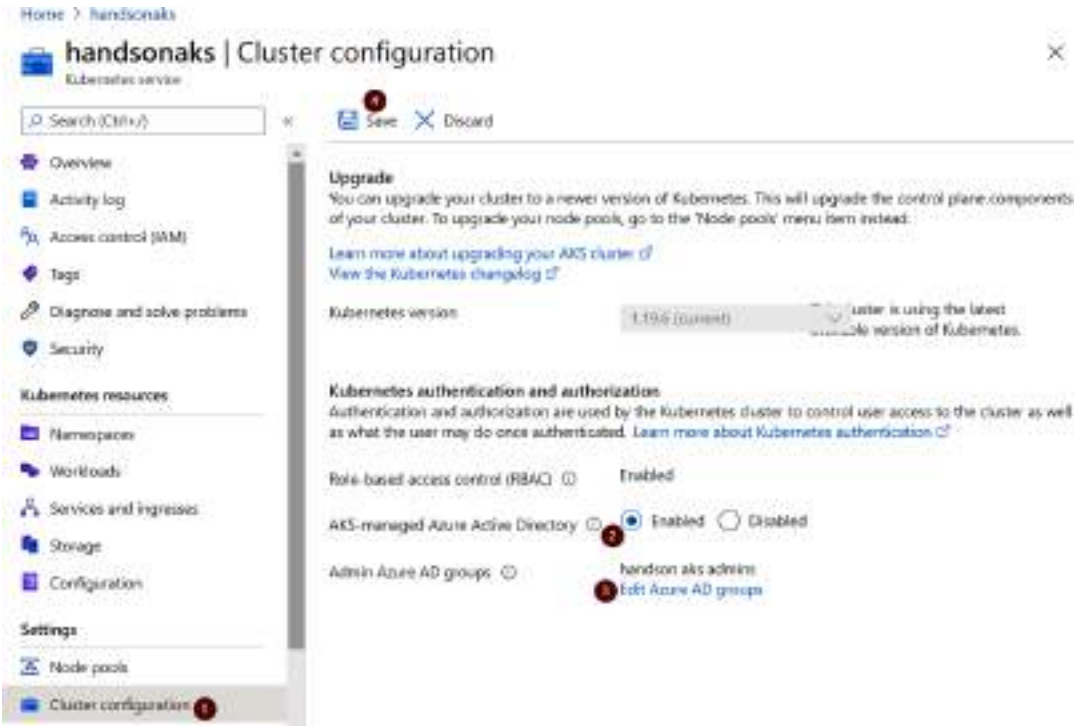


Figure 8.6: Enabling AKS-managed Azure Active Directory and clicking the Save button

This enables Azure AD–integrated RBAC on your AKS cluster. In the next section, you will create a new user and a new group that will be used in the section afterward to set up and test RBAC in Kubernetes.

Creating a user and group in Azure AD

In this section, you will create a new user and a new group in Azure AD. You will use them later on in the chapter to assign them permissions to your AKS cluster:

> **Note**
>
> You need the *User Administrator role* in Azure AD to be able to create users and groups.

1. To start with, search for `azure active directory` in the Azure search bar:

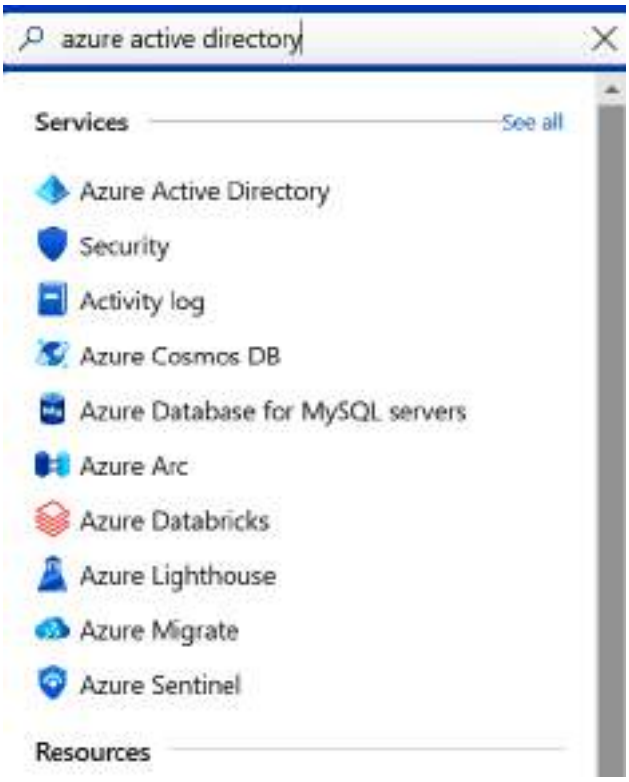


Figure 8.7: Searching for azure active directory in the search bar

2. Click on **All users** in the left pane. Then select **+ New user** to create a new user:

Figure 8.8: Clicking on + New user to create a new user

3. Provide the information about the user, including the username. Make sure to note down the password, as this will be required to sign in:

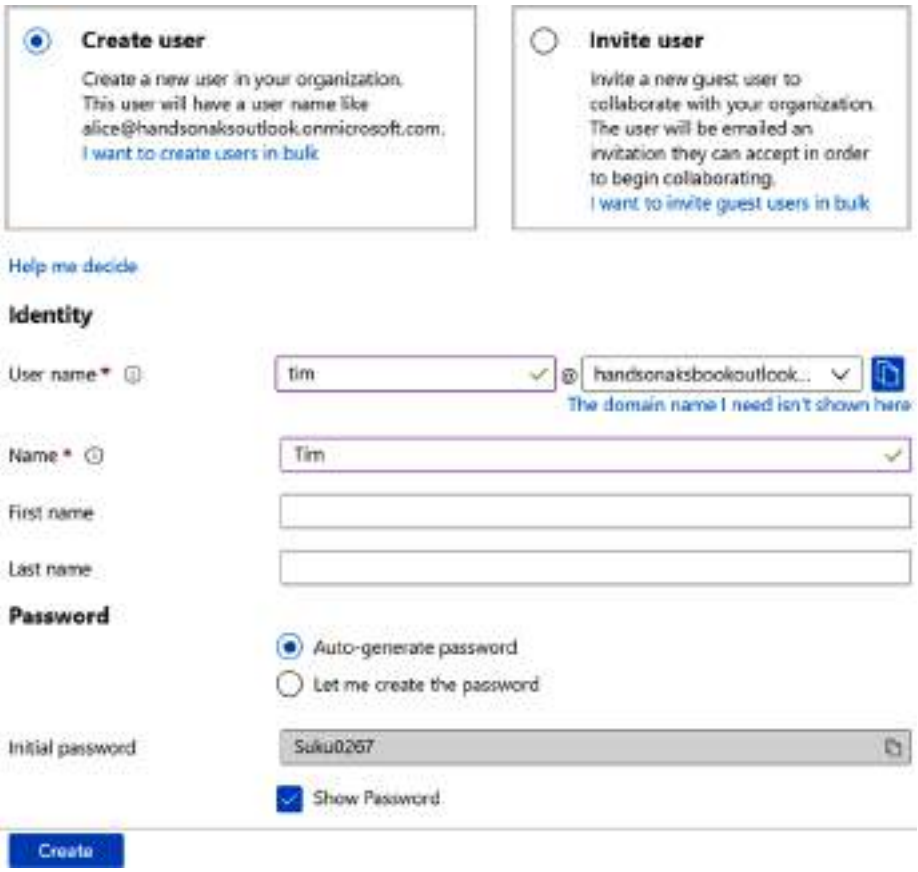


Figure 8.9: Providing the user details

4. Once the user is created, go back to the Azure AD pane and select **Groups**. Then click the **+ New group** button to create a new group:

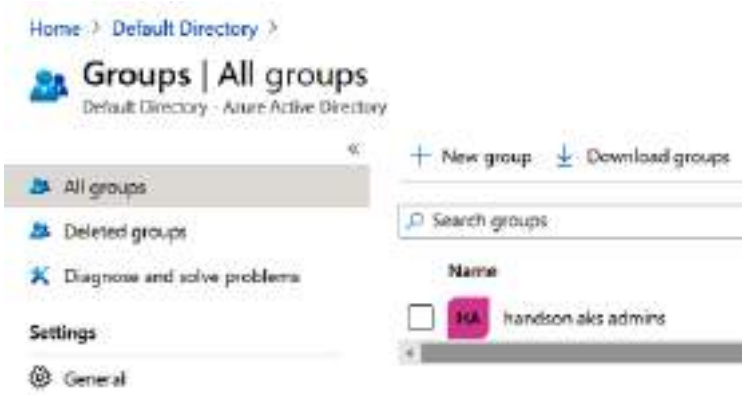


Figure 8.10: Clicking on + New group to create a new group

5. Create a new security group. Call the group `handson aks users` and add `Tim` as a member of the group. Then hit the **Create** button at the bottom:

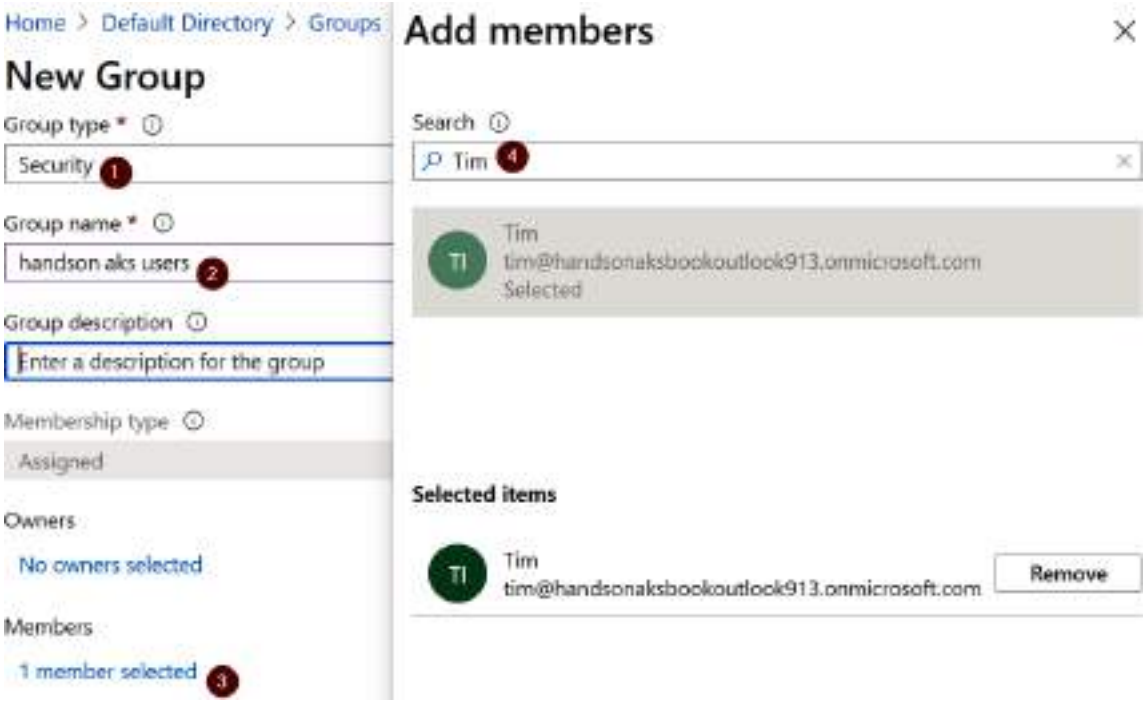


Figure 8.11: Providing the group type, group name, and group description

6. You have now created a new user and a new group. Next, you'll make that user a cluster user in AKS RBAC. This enables them to use the Azure CLI to get access to the cluster. To do that, search for your cluster in the Azure search bar:

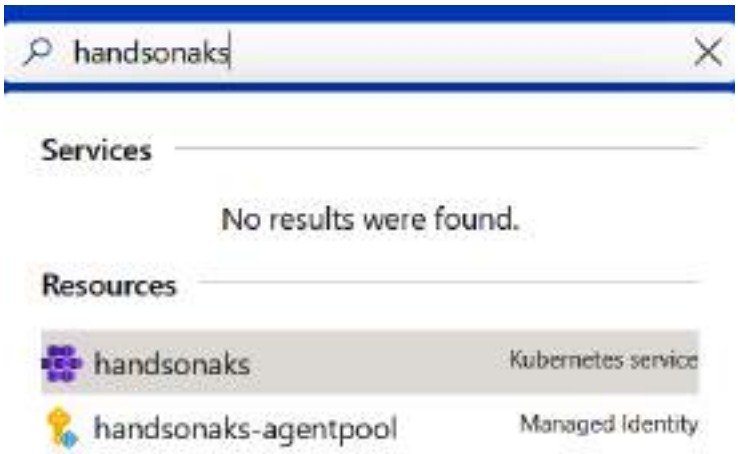


Figure 8.12: Searching for your cluster in the Azure search bar

7. In the cluster pane, click on **Access control (IAM)** and then click on the **+ Add** button to add a new role assignment. Select **Azure Kubernetes Service Cluster User Role** and assign that to the new user you just created:

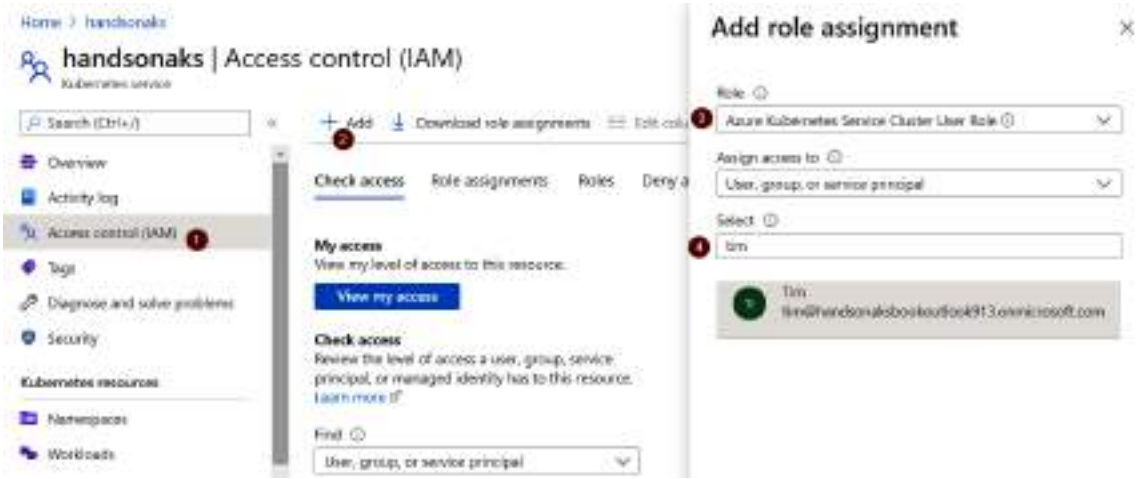


Figure 8.13: Assigning the cluster user role to the new user you created

8. As you will also be using Cloud Shell with the new user, you will need to give them contributor access to the Cloud Shell storage account. First, search for `storage` in the Azure search bar:

Figure 8.14: Searching for storage in the Azure search bar

9. There should be a storage account under **Resource group** with a name that starts with **cloud-shell-storage**. Click on the resource group:

Figure 8.15: Selecting the resource group

10. Go to **Access control (IAM)** and click on the **+ Add** button. Give the **Storage Account Contributor** role to your newly created user:

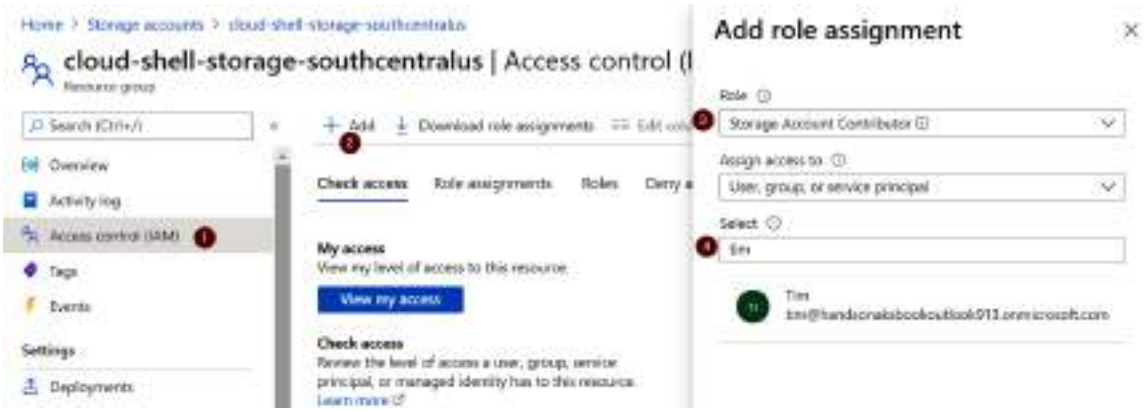


Figure 8.16: Assigning Storage Account Contributor role to the new user

This has concluded the creation of a new user and a group and giving that user access to AKS. In the next section, you will configure RBAC for that user and group in your AKS cluster.

Configuring RBAC in AKS

To demonstrate RBAC in AKS, you will create two namespaces and deploy the Azure voting application in each namespace. You will give the group cluster-wide read-only access to pods, and you will give the user the ability to delete pods in only one namespace. Practically, you will need to create the following objects in Kubernetes:

- `ClusterRole` to give read-only access
- `ClusterRoleBinding` to grant the group access to this role
- `Role` to give delete permissions in the `delete-access` namespace
- `RoleBinding` to grant the user access to this role

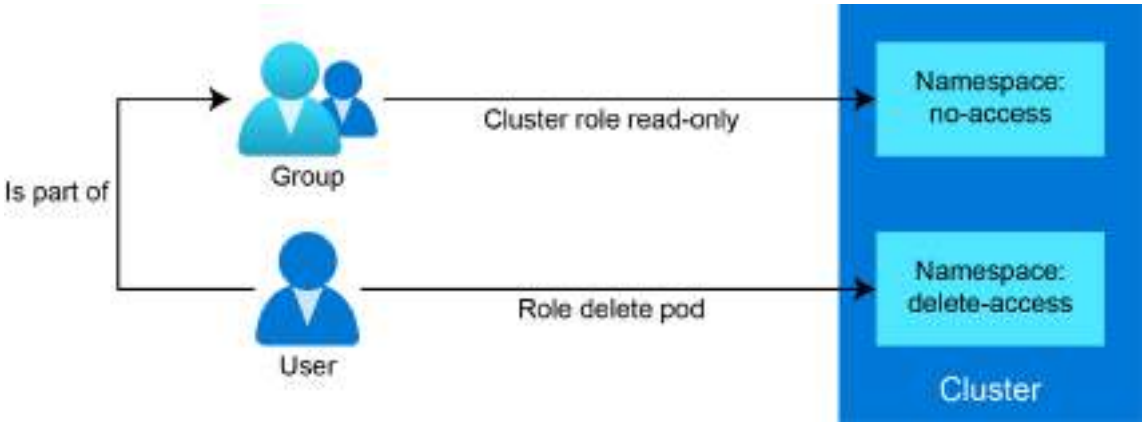


Figure 8.17: The group getting read-only access to the whole cluster, and the user getting delete permissions to the delete-access namespace

Let's set up the different roles on your cluster:

1. To start our example, you will need to retrieve the ID of the group. The following commands will retrieve the group ID:

```
az ad group show -g 'handson aks users' \
  --query objectId -o tsv
```

   This will show your group ID. Note this down because you'll need it in the next steps:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter08$ az ad group show -g 'handson aks users' \
>   --query objectId -o tsv
2f5de9a5-dc1b-4ee6-b4a2-4898a6ee3fb6
```

Figure 8.18: Getting the group ID

2. In Kubernetes, you will create two namespaces for this example:

```
kubectl create ns no-access
kubectl create ns delete-access
```

3. You will also deploy the `azure-vote` application in both namespaces:

```
kubectl create -f azure-vote.yaml -n no-access
kubectl create -f azure-vote.yaml -n delete-access
```

4. Next, you will create the `ClusterRole` object. This is provided in the `clusterRole.yaml` file:

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: ClusterRole
3   metadata:
4     name: readOnly
5   rules:
6   - apiGroups: [""]
7     resources: ["pods"]
8     verbs: ["get", "watch", "list"]
```

   Let's have a closer look at this file:

   - **Line 2**: Defines the creation of a `ClusterRole` instance
   - **Line 4**: Gives a name to our `ClusterRole` instance
   - **Line 6**: Gives access to all API groups
   - **Line 7**: Gives access to all pods
   - **Line 8**: Gives access to the actions `get`, `watch`, and `list`

   We will create `ClusterRole` using the following command:

```
kubectl create -f clusterRole.yaml
```

5. The next step is to create a cluster role binding. The binding links the role to a user or a group. This is provided in the `clusterRoleBinding.yaml` file:

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: ClusterRoleBinding
3   metadata:
4     name: readOnlyBinding
5   roleRef:
6     kind: ClusterRole
7     name: readOnly
8     apiGroup: rbac.authorization.k8s.io
9   subjects:
10  - kind: Group
11    apiGroup: rbac.authorization.k8s.io
12    name: "<group-id>"
```

Let's have a closer look at this file:

- **Line 2**: Defines that we are creating a `ClusterRoleBinding` instance.
- **Line 4**: Gives a name to `ClusterRoleBinding`.
- **Lines 5–8**: Refer to the `ClusterRole` object we created in the previous step
- **Lines 9–12**: Refer to your group in Azure AD. Make sure to replace `<group-id>` on *line 12* with the group ID you got earlier.

We can create `ClusterRoleBinding` using the following command:

```
kubectl create -f clusterRoleBinding.yaml
```

6. Next, you'll create a role that is limited to the `delete-access` namespace. This is provided in the `role.yaml` file:

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: Role
3   metadata:
4     name: deleteRole
5     namespace: delete-access
6   rules:
7   - apiGroups: [""]
8     resources: ["pods"]
9     verbs: ["delete"]
```

This file is similar to the `ClusterRole` object from earlier. There are two meaningful differences:

- **Line 2**: Defines that you are creating a `Role` instance and not a `ClusterRole` instance
- **Line 5**: Defines the namespace this role is created in

You can create `Role` using the following command:

```
kubectl create -f role.yaml
```

7. Finally, you will create a `RoleBinding` instance that links our user to the namespace role. This is provided in the `roleBinding.yaml` file:

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: RoleBinding
3   metadata:
4     name: deleteBinding
5     namespace: delete-access
6   roleRef:
7     kind: Role
8     name: deleteRole
9     apiGroup: rbac.authorization.k8s.io
10  subjects:
11  - kind: User
12    apiGroup: rbac.authorization.k8s.io
13    name: "<user e-mail address>"
```

   This file is similar to the `ClusterRoleBinding` object from earlier. There are a couple of meaningful differences:

   - **Line 2**: Defines the creation of a `RoleBinding` instance and not a `ClusterRoleBinding` instance
   - **Line 5**: Defines the namespace this `RoleBinding` instance is created in
   - **Line 7**: Refers to a regular role and not a `ClusterRole` instance
   - **Lines 11–13**: Defines a user instead of a group

   You can create `RoleBinding` using the following command:

```
kubectl create -f roleBinding.yaml
```

This has concluded the requirements for RBAC. You have created two roles—`ClusterRole` and one namespace-bound role, and set up two `RoleBindings` objects—`ClusterRoleBinding` and the namespace-bound `RoleBinding`. In the next section, you will explore the impact of RBAC by signing in to the cluster as the new user.

Verifying RBAC for a user

To verify that RBAC works as expected, you will sign in to the Azure portal using the newly created user. Go to https://portal.azure.com in a new browser, or an InPrivate window, and sign in with the newly created user. You will be prompted immediately to change your password. This is a security feature in Azure AD to ensure that only that user knows their password:

Figure 8.19: You will be asked to change your password

Once you have changed your password, you can start testing the different RBAC roles:

1. You will start this experiment by setting up Cloud Shell for the new user. Launch Cloud Shell and select **Bash**:

Figure 8.20: Selecting Bash in Cloud Shell

2. In the next dialog box, select **Show advanced settings**:

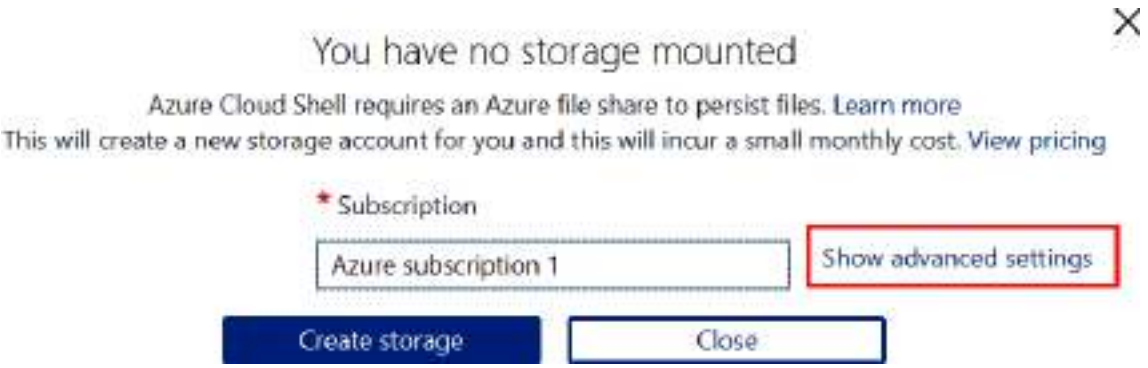


Figure 8.21: Selecting Show advanced settings

3. Then, point Cloud Shell to the existing storage account and create a new file share:

Figure 8.22: Pointing to the existing storage account and creating a new file share

4. Once Cloud Shell is available, get the credentials to connect to the AKS cluster:

```
az aks get-credentials -n handsonaks -g rg-handsonaks
```

Then, try a command in `kubectl`. Let's try to get the nodes in the cluster:

```
kubectl get nodes
```

Since this is the first command executed against an RBAC-enabled cluster, you are asked to sign in again. Browse to https://microsoft.com/devicelogin and provide the code Cloud Shell showed you (this code is highlighted in *Figure 8.24*). Make sure you sign in here with your new user credentials:

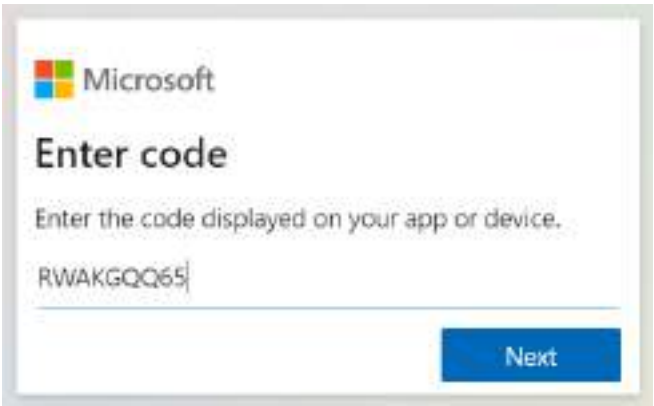


Figure 8.23: Copying and pasting the code Cloud Shell showed you in the prompt

After you have signed in, you should get a `Forbidden` error message from `kubectl`, informing you that you don't have permission to view the nodes in the cluster. This was expected since the user is configured only to have access to pods:

```
tim@Azure:~$ kubectl get nodes
To sign in, use a web browser to open the page https://microsoft.com/devicelogi
n and enter the code RWAKGQQ65 to authenticate.
Error from server (Forbidden): nodes is forbidden: User "tim@handsonaksbookoutl
ook913.onmicrosoft.com" cannot list resource "nodes" in API group "" at the clu
ster scope
```

Figure 8.24: The prompt asking you to sign in and the Forbidden message

5. Now you can verify that your user has access to view pods in all namespaces and that the user has permission to delete pods in the `delete-access` namespace:

```
kubectl get pods -n no-access
kubectl get pods -n delete-access
```

This should succeed for both namespaces. This is due to the `ClusterRole` object configured for the user's group:

```
tim@Azure:~$ kubectl get pods -n no-access
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-6bbcb89698-tjpv7    1/1     Running   0          27m
azure-vote-front-64b6fd89d6-vm9c8   1/1     Running   0          27m
tim@Azure:~$ kubectl get pods -n delete-access
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-6bbcb89698-qrbl4    1/1     Running   0          27m
azure-vote-front-64b6fd89d6-wkp9l   1/1     Running   0          27m
```

Figure 8.25: The user has access to view pods in both namespaces

6. Let's also verify the `delete` permissions:

```
kubectl delete pod --all -n no-access
kubectl delete pod --all -n delete-access
```

   As expected, this is denied in the `no-access` namespace and allowed in the `delete-access` namespace, as seen in *Figure 8.26*:

```
tim@Azure:~$ kubectl delete pod --all -n no-access
Error from server (Forbidden): pods "azure-vote-back-6bbcb89698-tjpv7" is forbi
dden: User "tim@handsonaksbookoutlook913.onmicrosoft.com" cannot delete resourc
e "pods" in API group "" in the namespace "no-access"
Error from server (Forbidden): pods "azure-vote-front-64b6fd89d6-vm9c8" is forb
idden: User "tim@handsonaksbookoutlook913.onmicrosoft.com" cannot delete resour
ce "pods" in API group "" in the namespace "no-access"
tim@Azure:~$ kubectl delete pod --all -n delete-access
pod "azure-vote-back-6bbcb89698-qrb14" deleted
pod "azure-vote-front-64b6fd89d6-wkp91" deleted
```

Figure 8.26: Deletes are denied in the no-access namespace and allowed in the delete-access namespace

In this section, you have verified the functionality of RBAC on your Kubernetes cluster. Since this is the last section of this chapter, let's make sure to clean up the deployments and namespaces in the cluster. Make sure to execute these steps from Cloud Shell with your main user, not the new user:

```
kubectl delete -f azure-vote.yaml -n no-access
kubectl delete -f azure-vote.yaml -n delete-access
kubectl delete -f .
kubectl delete ns no-access
kubectl delete ns delete-access
```

This concludes the overview of RBAC on AKS.

Summary

In this chapter, you learned about RBAC on AKS. You enabled Azure AD–integrated RBAC in your cluster. After that, you created a new user and group and set up different RBAC roles on your cluster. Finally, you signed in using that user and were able to verify that the RBAC roles that were configured gave you limited access to the cluster you were expecting.

This deals with how users can get access to your Kubernetes cluster. The pods running on your cluster might also need an identity in Azure AD that they can use to access resources in Azure services such as Blob Storage or Key Vault. You will learn more about this use case and how to set this up using an Azure AD pod identity in AKS in the next chapter.

9

Azure Active Directory pod-managed identities in AKS

In the previous chapter, *Chapter 8*, *Role-based access control in AKS*, you integrated your AKS cluster with **Azure Active Directory** (**Azure AD**). You then assigned Kubernetes roles to users and groups in Azure AD. In this chapter, you will explore how you can integrate your applications running on AKS with Azure AD, and you will learn how you can give your pods an identity in Azure so they can interact with other Azure resources.

In Azure, application identities use a functionality called service principals. A service principal is the equivalent of a service account in the cloud. An application can use a service principal to authenticate to Azure AD and get access to resources. Those resources could be either Azure resources such as Azure Blob Storage or Azure Key Vault, or they could be applications that you developed that are integrated with Azure AD.

There are two ways to authenticate a service principal: you can either use a password or a combination of a certificate and a private key. Although these are secure ways to authenticate your applications, managing passwords or certificates and the rotation associated with them can be cumbersome.

Managed identities in Azure are a functionality that makes authenticating to a service principal easier. It works by assigning an identity to a compute resource in Azure, such as a virtual machine or an Azure function. Those compute resources can authenticate using that managed identity by calling an endpoint that only that machine can reach. This is a secure type of authentication that does not require you to manage passwords or certificates.

Azure AD pod-managed identities allow you to assign managed identities to pods in Kubernetes. Since pods in Kubernetes run on virtual machines, by default, each pod would be able to access the managed identity endpoint and authenticate using that identity. Using Azure AD pod-managed identities, pods can no longer reach the internal endpoint for the virtual machine, and rather only get access to identities assigned to that specific pod.

In this chapter, you'll configure an Azure AD pod-managed identity on an AKS cluster and use it to get access to Azure Blob Storage. In the next chapter, you will then use these Azure AD pod-managed identities to get access to Azure Key Vault and manage Kubernetes secrets.

The following topics will be covered briefly in this chapter:

- An overview of Azure AD pod-managed identities
- Setting up a new cluster with Azure AD pod-managed identities
- Linking an identity to your cluster
- Using a pod with managed identity

Let's start with an overview of Azure AD pod-managed identities.

An overview of Azure AD pod-managed identities

The goal of this section is to describe Azure managed identities and Azure AD pod-managed identities.

As explained in the introduction, managed identities in Azure are a way to securely authenticate applications running inside Azure. There are two types of managed identities in Azure. The difference between them is how they are linked to resources:

- **System assigned**: This type of managed identity is linked 1:1 to the resource (such as a virtual machine) itself. This managed identity also shares the lifecycle of the resource, meaning that once the resource is deleted, the managed identity is also deleted.
- **User assigned**: User-assigned managed identities are standalone Azure resources. A user-assigned managed identity can be linked to multiple resources. When a resource is deleted, the managed identity is not deleted.

Both types of managed identities work the same way once they are created and linked to a resource. This is how managed identities work from an application perspective:

1. Your application running in Azure requests a token to the **Instance Metadata Service** (**IMDS**). The IMDS is only available to that resource itself, at a non-routable IP address (`169.254.169.254`).
2. The IMDS will request a token from Azure AD. It uses a certificate that is configured for your managed identity and is only known by the IMDS.
3. Azure AD will return a token to the IMDS, which will, in turn, return that token to your application.
4. Your application can use this token to authenticate to other resources, for instance, Azure Blob Storage.

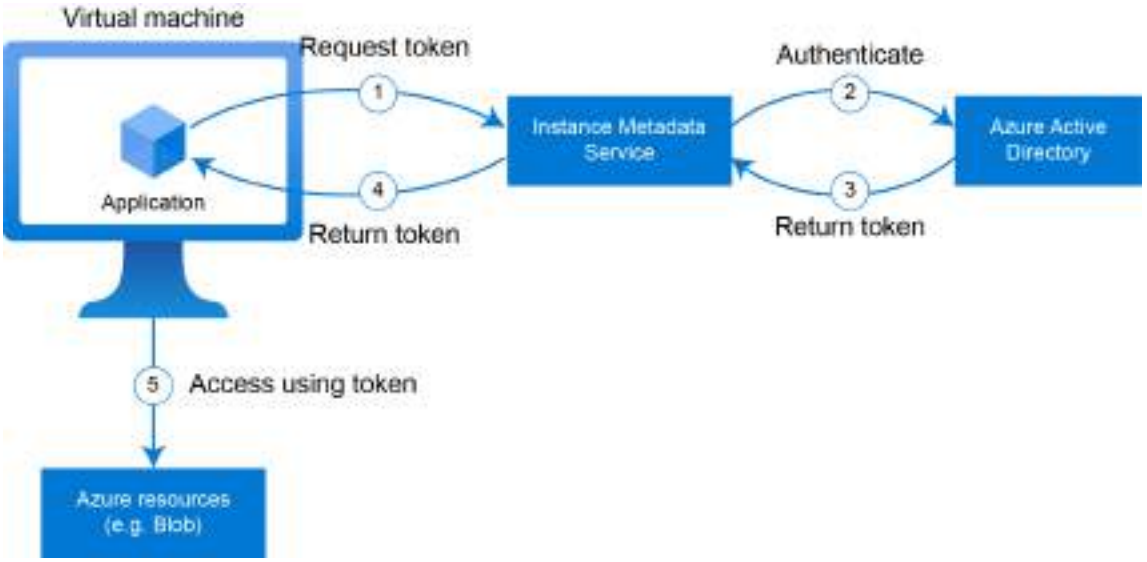


Figure 9.1: Managed identity in an Azure virtual machine

When running multiple pods on a single virtual machine in a Kubernetes cluster, by default each pod can reach the IMDS endpoint. This means that each pod could get access to the identities configured for that virtual machine.

The Azure AD pod-managed identities add-on for AKS configures your cluster in such a way that pods can no longer access the IMDS endpoint directly to request an access token. It configures your cluster in such a way that pods trying to access to IMDS endpoint (1) will connect to a DaemonSet running on the cluster. This DaemonSet is called the **node managed identity** (**NMI**). The NMI will verify which identities that pod should have access to. If the pod is configured to have access to the requested identity, then the DaemonSet will connect to the IMDS (2 to 5) to get the token, and then deliver the token to the pod (6). The pods can then use this token to access Azure resources (7).

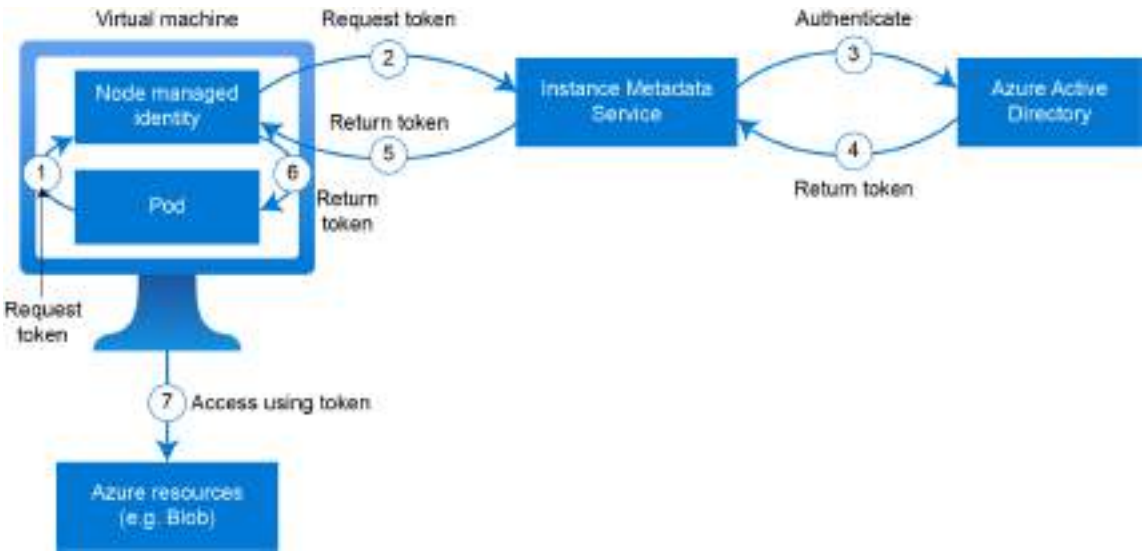


Figure 9.2: Azure AD pod-managed identity

This way, you can control which pods on your cluster have access to certain identities.

Azure AD pod-managed identities were initially developed as an open-source project by Microsoft on GitHub. More recently, Microsoft has released Azure AD pod-managed identities as an AKS add-on. The benefit of using Azure AD pod-managed identities as an AKS add-on is that the functionality is supported by Microsoft and the software will be updated automatically as part of regular cluster operations.

> **Note**
>
> At the time of writing, the Azure AD pod-managed identities add-on is in preview. Currently, it is also not supported for Windows containers. Using preview functionality for product use cases is not recommended.

Now that you know how Azure AD pod-managed identities work, let's set it up on an AKS cluster in the next section.

Setting up a new cluster with Azure AD pod-managed identities

As mentioned in the previous section, there are two ways to set up Azure AD pod-managed identities in AKS. It can either be done using the open-source project on GitHub, or by setting it up as an AKS add-on. By using the add-on, you'll get a supported configuration, which is why you'll set up a cluster using the add-on in this section.

At the time of writing, it is not yet possible to enable the Azure AD pod-managed identities add-on on an existing cluster, which is why in the following instructions you'll delete your existing cluster and create a new one with the add-on installed. By the time you are reading this, it might be possible to enable this add-on on an existing cluster without recreating your cluster.

Also, because the functionality is in preview at the time of this writing, you'll have to register for the preview. That'll be the first step in this section:

1. Start by opening Cloud Shell and registering for the preview of Azure AD pod-managed identities:

```
az feature register --name EnablePodIdentityPreview \
  --namespace Microsoft.ContainerService
```

2. You'll also need a preview extension of the Azure CLI, which you can install using the following command:

```
az extension add --name aks-preview
```

3. Now you can go ahead and delete your existing cluster. This is required to ensure you have enough core quota available in Azure. You can do this using the following command:

```
az aks delete -n handsonaks -g rg-handsonaks --yes
```

4. Once your previous cluster is deleted, you'll have to wait until the pod identity preview is registered on your subscription. You can use the following command to verify this status:

```
az feature show --name EnablePodIdentityPreview \
  --namespace Microsoft.ContainerService -o table
```

   Wait until the status shows as registered, as shown in *Figure 9.3*:

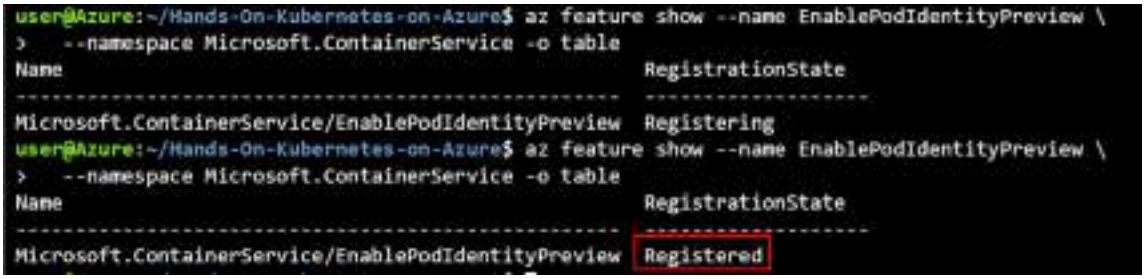


Figure 9.3: Waiting for the feature to be registered

5. If the feature is registered and your old cluster is deleted, you need to refresh the registration of the namespace before creating a new cluster. Let's first refresh the registration of the namespace:

```
az provider register --namespace Microsoft.ContainerService
```

6. And now you can create a new cluster using the Azure AD pod-managed identities add-on. You can use the following command to create a new cluster with the add-on enabled:

```
az aks create -g rg-handsonaks -n handsonaks \
  --enable-managed-identity --enable-pod-identity \
  --network-plugin azure --node-vm-size Standard_DS2_v2 \
  --node-count 2 --generate-ssh-keys
```

7. This will take a couple of minutes to finish. Once the command finishes, obtain the credentials to access your cluster and verify you can access your cluster using the following commands:

```
az aks get-credentials -g rg-handsonaks \
  -n handsonaks --overwrite-existing
kubectl get nodes
```

This should return an output similar to *Figure 9.4*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09$ az aks get-credentials -g rg-handsonaks \
>   -n handsonaks --overwrite-existing
The behavior of this command has been altered by the following extension: aks-preview
Merged "handsonaks" as current context in /home/kelly/.kube/config
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09$ kubectl get nodes
NAME                                STATUS   ROLES   AGE    VERSION
aks-nodepool1-36910094-vmss000000   Ready    agent   107s   v1.18.14
aks-nodepool1-36910094-vmss000001   Ready    agent   105s   v1.18.14
```

Figure 9.4: Getting cluster credentials and verifying access

Now you have a new AKS cluster with Azure AD pod-managed identities enabled. In the next section, you will create a managed identity and link it to your cluster.

Linking an identity to your cluster

In the previous section, you created a new cluster with Azure AD pod-managed identities enabled. Now you are ready to create a managed identity and link it to your cluster. Let's get started:

1. To start, you will create a new managed identity using the Azure portal. In the Azure portal, look for `managed identity` in the search bar, as shown in *Figure 9.5*:

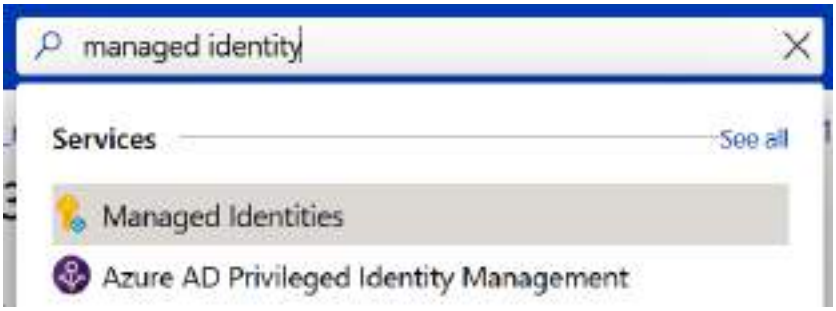


Figure 9.5: Navigating to Managed Identities in the Azure portal

2. In the resulting pane, click the **+ New** button at the top. To organize the resources for this chapter together, it's recommended to create a new resource group. In the resulting pane, click the **Create new** button to create a new resource group. Call it `aad-pod-id`, as shown in *Figure 9.6*:

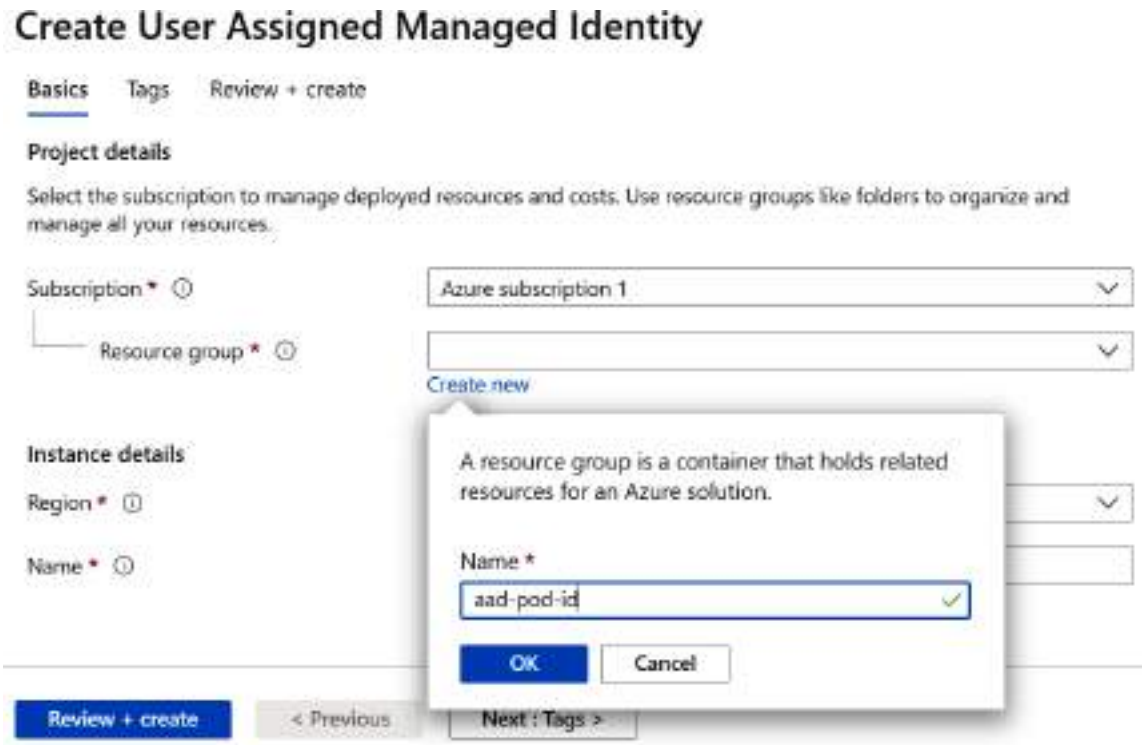


Figure 9.6: Creating a new resource group

3. Now, select the region you created your cluster in as the region for your managed identity and give it a name (`aad-pod-id` in this example), as shown in *Figure 9.7*. To finish, click the **Review + create** button and in the final window click the **Create** button to create your managed identity:

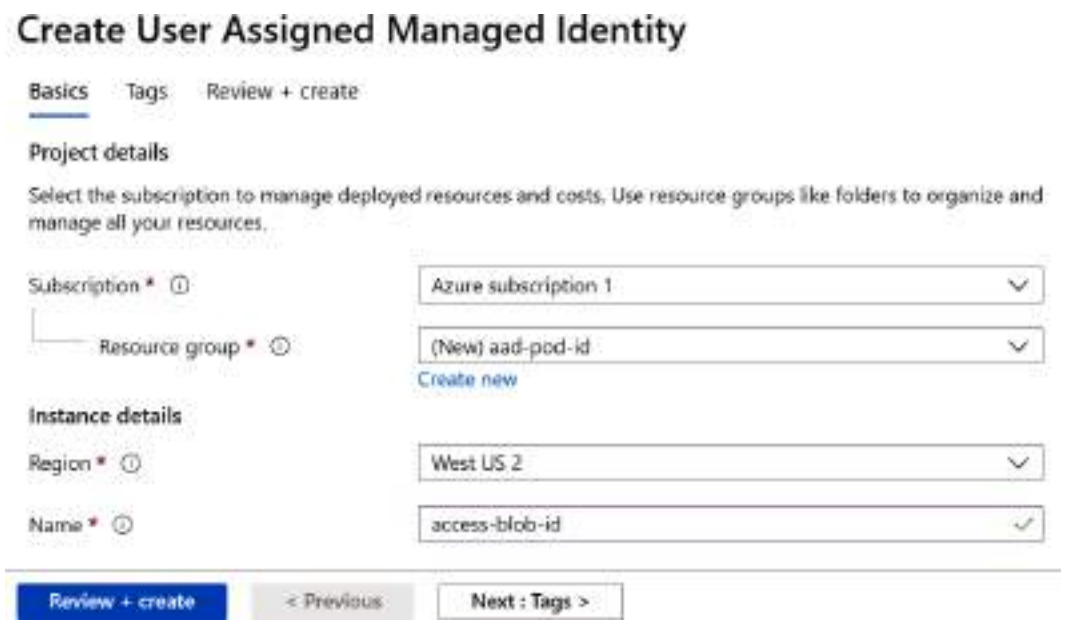


Figure 9.7: Providing Instance details for the managed identity

4. Once the managed identity has been created, hit the **Go to resource** button to go to the resource. Here, you will need to copy the client ID and the resource ID. They will be used later in this chapter. Copy and paste the values somewhere that you can access later. First, you will need the client ID of the managed identity. You can find that in the **Overview** pane of the managed identity, as shown in *Figure* 9.8:

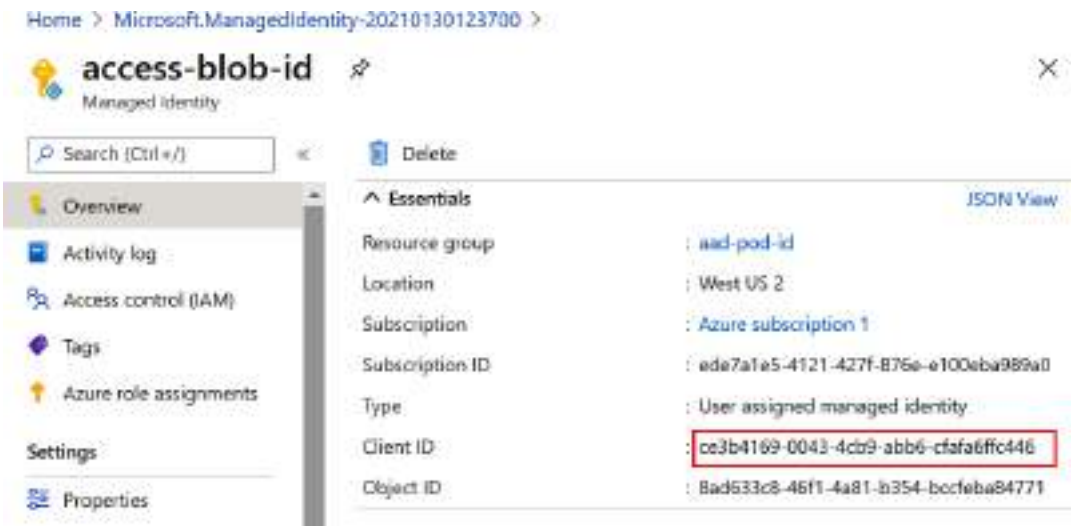


Figure 9.8: Getting the client ID of the managed identity

5. Finally, you will also need the resource ID of the managed identity. You can find that in the Properties pane of the managed identity, as shown in *Figure* 9.9:

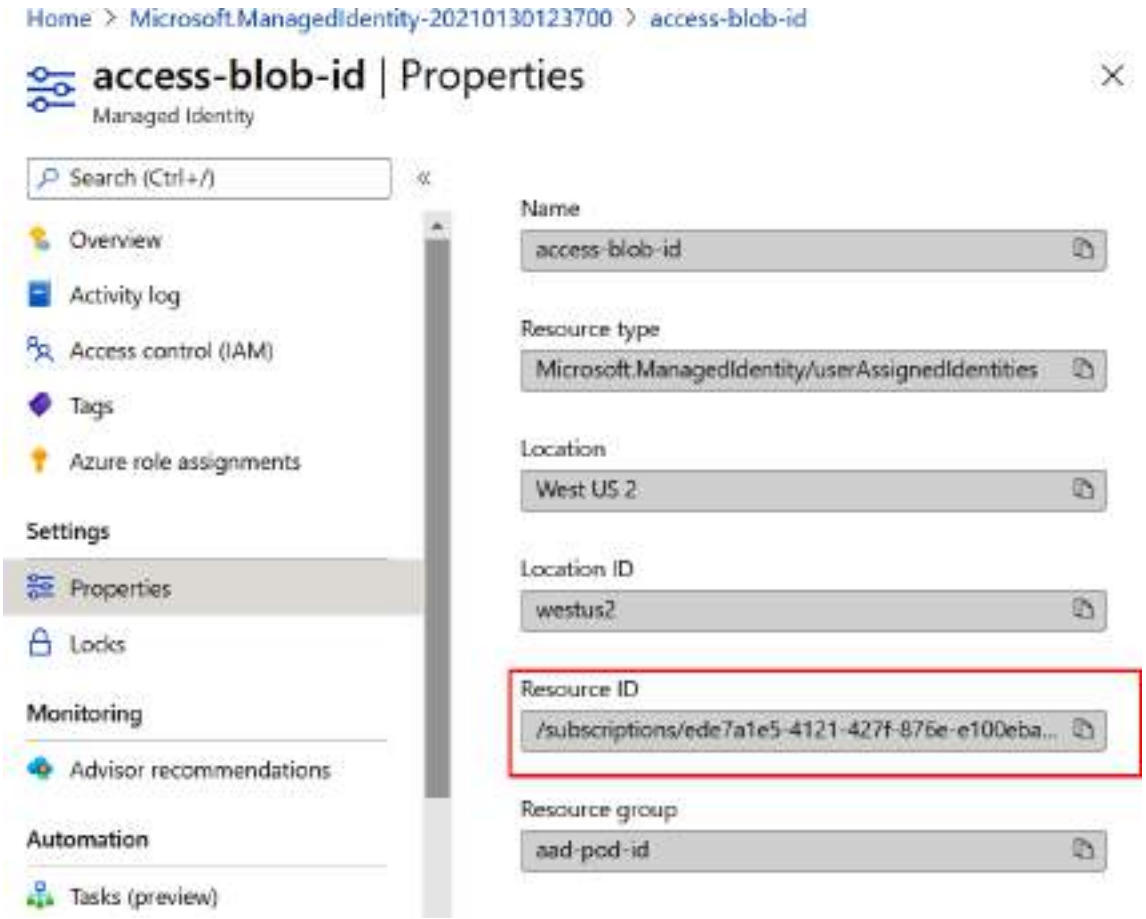


Figure 9.9: Getting the resource ID of the managed identity

6. Now you are ready to link the managed identity to your AKS cluster. To do this, you will run a command in Cloud Shell, and afterward you will be able to verify that the identity is available in your cluster. Let's start with linking the identity. Make sure to replace `<Managed identity resource ID>` with the resource you copied earlier:

```
az aks pod-identity add --resource-group rg-handsonaks \
  --cluster-name handsonaks --namespace default \
  --name access-blob-id \
  --identity-resource-id <Managed identity resource ID>
```

7. You can verify that your identity was successfully linked to your cluster by running the following command:

```
kubectl get azureidentity
```

This should give you an output similar to *Figure 9.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09$ kubectl get azureidentity
NAME              AGE
access-blob-id    59s
```

Figure 9.10: Verifying the availability of the identity in the cluster

This means that the identity is now available for you to use in your cluster. How you do this will be explained in the next section.

Using a pod with managed identity

In the previous section, you created a managed identity and linked it to your cluster. In this section, you will create a new blob storage account and give the managed identity you created permission over this storage account. Then, you will create a new pod in your cluster that can use that managed identity to interact with that storage account. Let's get started by creating a new storage account:

1. To create a new storage account, look for `storage accounts` in the Azure search bar, as shown in *Figure 9.11*:

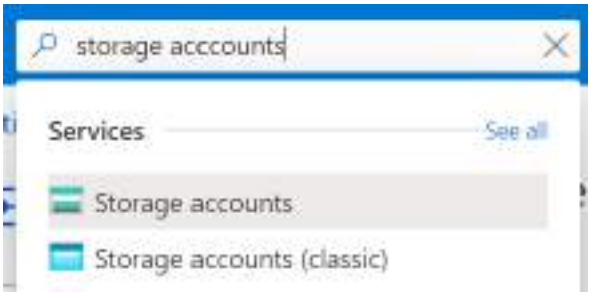


Figure 9.11: Looking for storage accounts in the Azure search bar

In the resulting pane, click the **+ New** button at the top of the screen as shown in *Figure 9.12*:

Figure 9.12: Creating a new storage account

Select the `aad-pod-id` resource group you created earlier, give the account a unique name, and select the same region as your cluster. To optimize costs, it is recommended that you select the **Standard** performance, **StorageV2** as the Account kind, and **Locally-redundant storage (LRS)** for Replication, as shown in *Figure 9.13*:

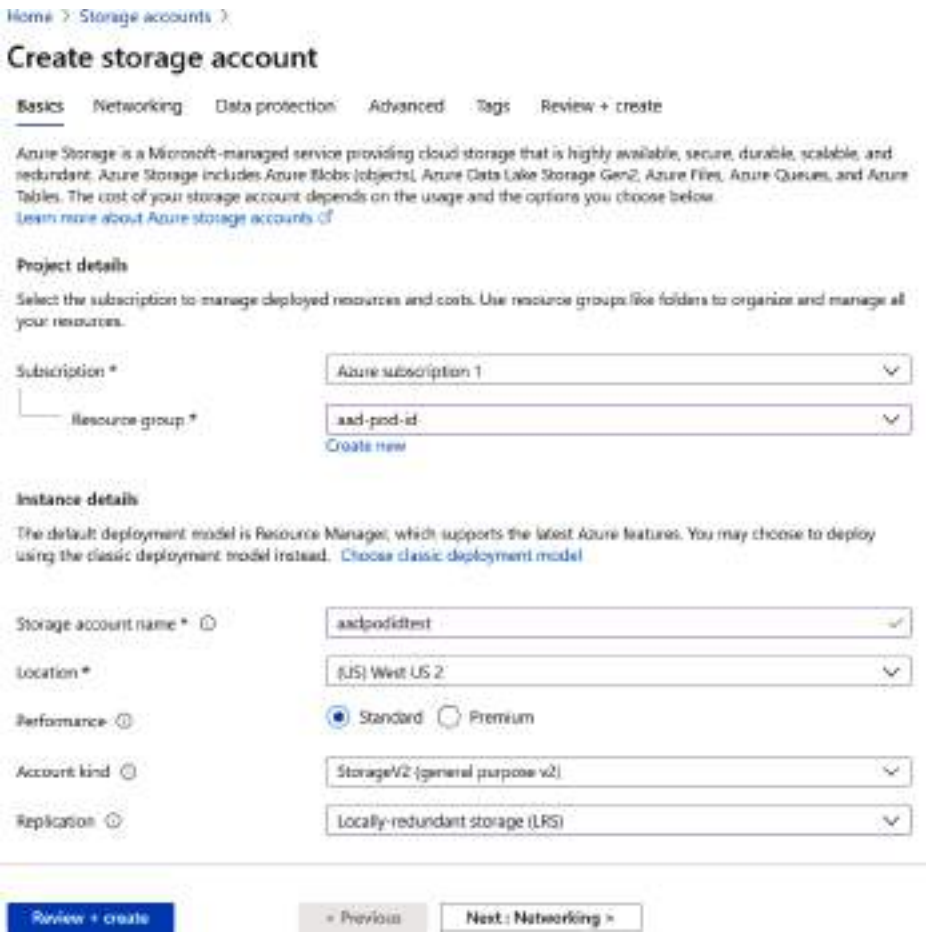


Figure 9.13: Configuring your new storage account

2. After you have provided all the values, click **Review + create** and then the **Create** button on the resulting screen. This will take about a minute to create. Once the storage account is created, click the **Go to resource** button to move on to the next step.
3. First, you will give the managed identity access to the storage account. To do this, click **Access Control (IAM)** in the left-hand navigation bar, click **+ Add** and **Add role assignment**. Then select the **Storage Blob Data Contributor** role, select **User assigned managed identity** in the **Assign access to** dropdown, and select the **access-blob-id** managed identity you created, as shown in *Figure 9.14*. Finally, hit the **Save** button at the bottom of the screen:

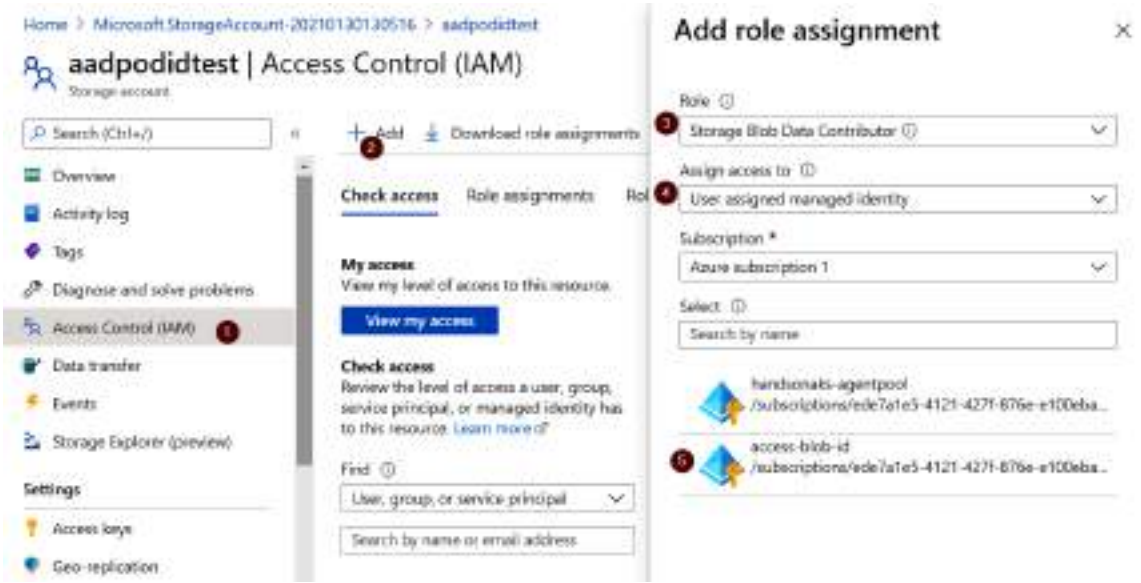


Figure 9.14: Providing access to the storage account for the managed identity

4. Next, you will upload a random file to this storage account. Later, you will try to access this file from within a Kubernetes pod to verify you have access to the storage account. To do this, go back to the **Overview** pane of the storage account. There, click on **Containers**, as shown in *Figure 9.15*:

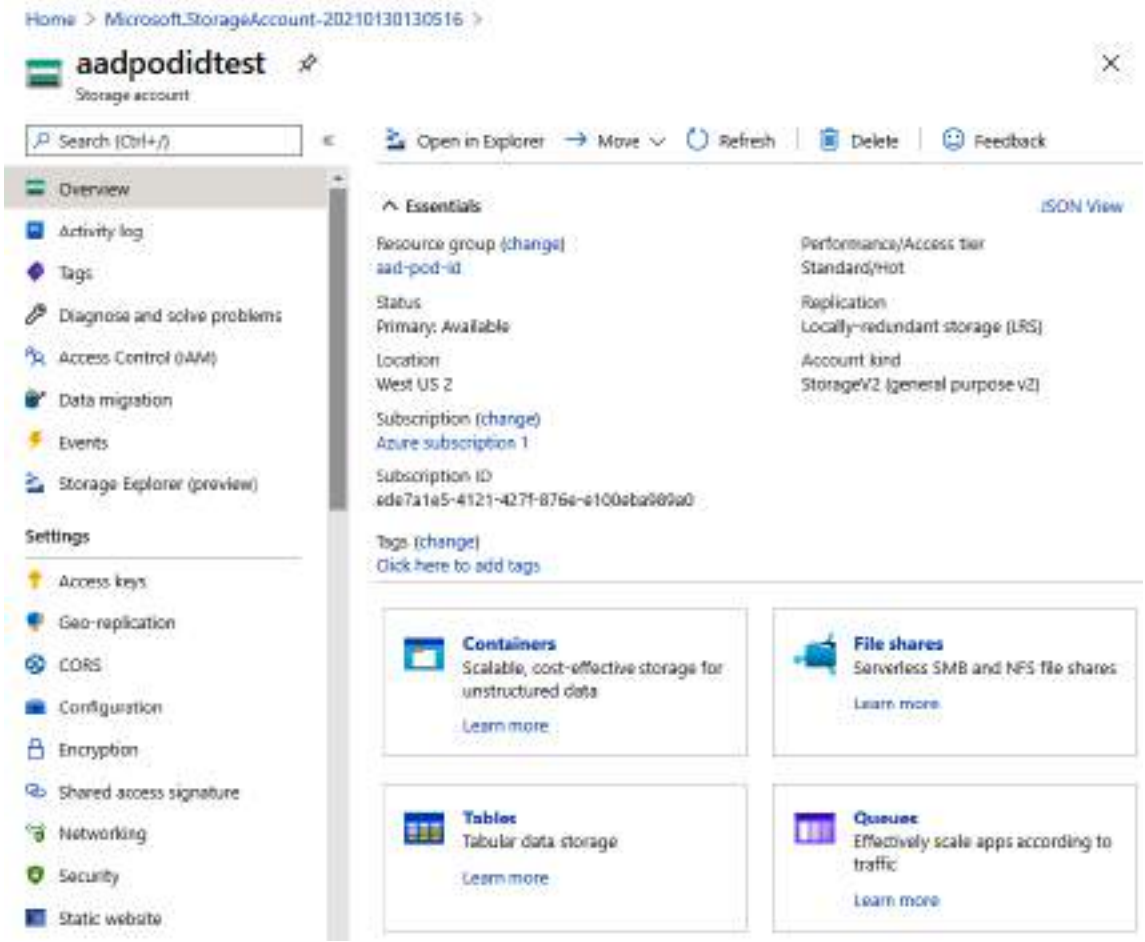


Figure 9.15: Clicking on Containers in the overview pane

5. Then hit the **+ Container** button at the top of the screen. Give the container a name, such as `uploadedfiles`. Make sure to set Public access level to **Private (no anonymous access)**, and then click the **Create** button at the bottom of the screen, as shown in *Figure 9.16*:

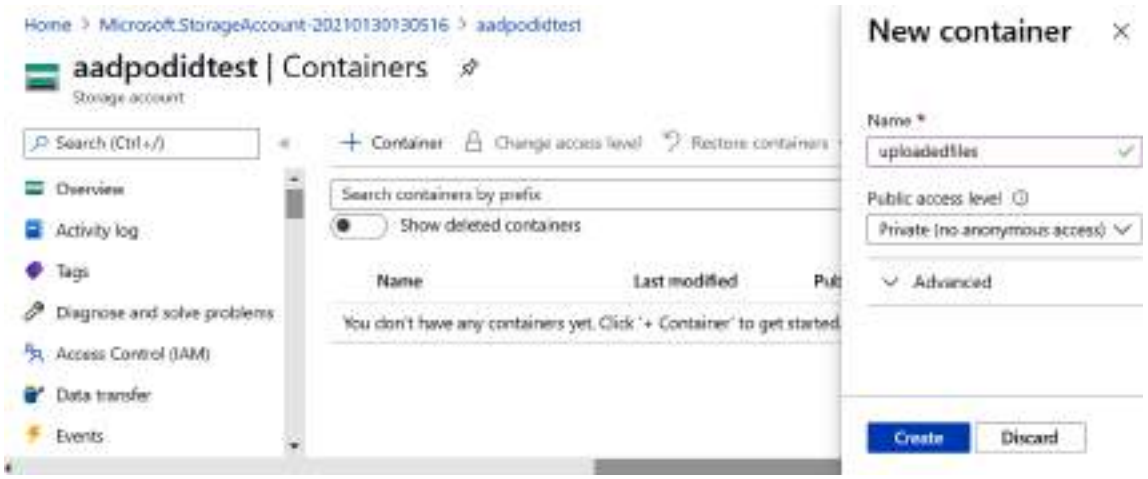


Figure 9.16: Creating a new blob storage container

6. Finally, upload a random file into this storage container. To do this, click on the container name, and then click the **Upload** button at the top of the screen. Select a random file from your computer and click **Upload** as shown in *Figure 9.17*:

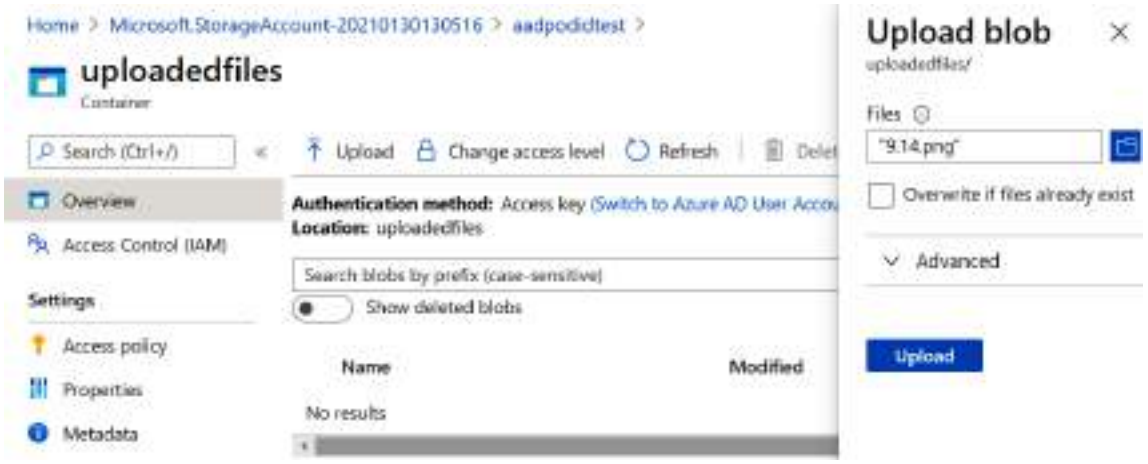


Figure 9.17: Uploading a new file to blob storage

7. Now that you have a file in blob storage, and your managed identity has access to this storage account, you can go ahead and try connecting to it from Kubernetes. To do this, you will create a new deployment using the Azure CLI container image. This deployment will contain a link to the managed identity that was created earlier. The deployment file is provided in the code files for this chapter as `deployment-with-identity.yaml`:

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: access-blob
5   spec:
6     selector:
7       matchLabels:
8         app: access-blob
9     template:
10      metadata:
11        labels:
12          app: access-blob
13          aadpodidbinding: access-blob-id
14      spec:
15        containers:
16        - name: azure-cli
17          image: mcr.microsoft.com/azure-cli
18          command: [ "/bin/bash", "-c", "sleep inf" ]
```

   There are a few things to draw attention to in the definition of this deployment:

   - **Line 13**: This is where you link the pod (created by the deployment) with the managed identity. Any pod with that label will be able to access the managed identity.
   - **Line 16-18**: Here, you define which container will be created in this pod. As you can see, the image (`mcr.microsoft.com/azure-cli`) is referring to the Azure CLI, and you're running a `sleep` command in this container to make sure the container doesn't continuously restart.

8. You can create this deployment using the following command:

```
kubectl create -f deployment-with-identity.yaml
```

9. Watch the pods until the `access-blob` pod is in the **Running** state. Then copy and paste the name of the `access-blob` pod and `exec` into it using the following command:

```
kubectl exec -it <access-blob pod name> -- sh
```

10. Once you are connected to the pod, you can authenticate to the Azure API using the following command. Replace `<client ID of managed identity>` with the client ID you copied earlier:

```
az login --identity -u <client ID of managed identity> \
  --allow-no-subscription -o table
```

This should return you an output similar to *Figure 9.18*:

Figure 9.18: Logging in to the Azure CLI using the Azure AD pod-managed identity

11. Now, you can try accessing the blob storage account and download the file. You can do this by executing the following command:

```
az storage blob download --account-name <storage account name> \
 --container-name <container name> --auth-mode login \
 --file <filename> --name <filename> -o table
```

This should return you an output similar to *Figure 9.19*:

```
/ # az storage blob download --account-name aadpodidtest \
>  --container-name uploadedfiles --auth-mode login \
>  --file 9.14.png --name 9.14.png -o table
Finished[#############################################################]  100.0000%
Name      Blob Type    Blob Tier    Length    Content Type    Last Modified              Snapshot
--------  -----------  -----------  --------  --------------  -------------------------  ----------
9.14.png  BlockBlob                76133     image/png       2021-01-30T21:20:48+00:00
```

Figure 9.19: Downloading a blob file using the managed identity

12. You can now exit the container using the `exit` command.

13. If you would like to verify that pods that don't have a managed identity configured and cannot download the file, you can use the file called `deployment-without-identity.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: no-access-blob
spec:
  selector:
    matchLabels:
      app: no-access-blob
  template:
    metadata:
      labels:
        app: no-access-blob
    spec:
      containers:
      - name: azure-cli
        image: mcr.microsoft.com/azure-cli
        command: [ "/bin/bash", "-c", "sleep inf" ]
```

    As you can see, this deployment isn't similar to the deployment you created earlier in the chapter. The difference here is that the pod definition doesn't contain the label with the Azure AD pod-managed identity. This means that this pod won't be able to log in to Azure using any managed identity. You can create this deployment using the following:

```
kubectl create -f deployment-without-identity.yaml
```

14. Watch the pods until the `no-access-blob` pod is in the **Running** state. Then copy and paste the name of the `access-blob` pod and `exec` into it using the following command:

```
kubectl exec -it <no-access-blob pod name> -- sh
```

15. Once you are connected to the pod, you can try to authenticate to the Azure API using the following command, which should fail:

```
az login --identity -u <client ID of managed identity> \
  --allow-no-subscription -o table
```

    This should return an output similar to *Figure 9.20*:

Figure 9.20: The new pod cannot authenticate using the managed identity

16. Finally, you can exit the container using the `exit` command.

This has successfully shown you how to use Azure AD pod-managed identities to connect to blob storage from within your Kubernetes cluster. A deployment with an identity label could log in to the Azure CLI and then access blob storage. A deployment without this identity label didn't get permission to log in to the Azure CLI, and hence was also not able to access blob storage.

This has concluded this chapter. Let's make sure to delete the resources you created for this chapter:

```
az aks pod-identity delete --resource-group rg-handsonaks \
  --cluster-name handsonaks --namespace default \
  --name access-blob-id
az group delete -n aad-pod-id --yes
kubectl delete -f
```

You can keep the cluster you created in this chapter since in the next chapter you will use Azure AD pod-managed identities to access Key Vault secrets.

Summary

In this chapter, you've continued your exploration of security in AKS. Whereas *Chapter 8*, *Role-based access control in* AKS, focused on identities for users, this chapter focused on identities for pods and applications running in pods. You learned about managed identities in Azure and how you can use Azure AD pod-managed identities in Azure to assign those managed identities to pods.

You created a new cluster with the Azure AD pod-managed identities add-on enabled. You then created a new managed identity and linked that to your cluster. In the final section, you gave this identity permissions over a blob storage account and finally verified that pods with the managed identity were able to log in to Azure and download files, but pods without the managed identity couldn't log in to Azure.

In the next chapter, you'll learn more about Kubernetes secrets. You'll learn about the built-in secrets and then also learn how you can securely connect Kubernetes to Azure Key Vault, and even use Azure AD pod-managed identities to do this.

10

Storing secrets in AKS

All production applications require some sensitive information to function, such as passwords or connection strings. Kubernetes has a pluggable back end to manage these secrets. Kubernetes also provides multiple ways of using the secrets in your deployment. The ability to manage secrets and use them properly will make your applications more secure.

You have already used secrets previously in this book. You used them when connecting to the WordPress site to create blog posts in *Chapter 3*, *Application deployment on AKS*, and *Chapter 4*, *Building scalable applications*. You also used secrets in *Chapter 6*, *Securing your application with HTTPS*, when you were configuring the Application Gateway Ingress Controller with TLS.

Kubernetes has a built-in secret system that stores secrets in a semi-encrypted fashion in the default Kubernetes database. This system works well but isn't the most secure way to deal with secrets in Kubernetes. In AKS, you can make use of a project called **Azure Key Vault provider for Secrets Store CSI driver** (**CSI driver**), which is a more secure way of working with Secrets in Kubernetes. This project allows you to store and retrieve secrets in/from Azure Key Vault.

In this chapter, you will learn about the various built-in secret types in Kubernetes and the different ways in which you can create these Secrets. After that, you will install the CSI driver on your cluster, and use it to retrieve Secrets.

Specifically, you will cover the following topics in this chapter:

- Different types of secret in Kubernetes
- Creating and using secrets in Kubernetes
- Installing the Azure Key Vault provider for secrets Store CSI driver
- Using the Azure Key Vault provider for secrets Store CSI driver

Let's start with exploring the different secret types in Kubernetes.

Different secret types in Kubernetes

As mentioned in the introduction to this chapter, Kubernetes comes with a default secrets implementation. This default implementation will store secrets in the etcd database that Kubernetes uses to store all object metadata. When Kubernetes stores secrets in etcd, it will store them in base64-encoded format. Base64 is a way to encode data in an obfuscated manner but is not a secure way of doing encryption. Anybody with access to base64-encoded data can easily decode it. AKS adds a layer of security on top of this by encrypting all data at rest within the Azure platform.

The default secret implementation in Kubernetes allows you to store multiple types of Secrets:

- **Opaque secrets**: These can contain any arbitrary user-defined secret or data.
- **Service account tokens**: These are used by Kubernetes pods for built-in cluster RBAC.
- **Docker config secrets**: These are used to store Docker registry credentials for Docker command-line configuration.
- **Basic authentication secrets**: These are used for storing authentication information in the form of a username and password.
- **SSH authentication secrets**: These are used to store SSH private keys.

- **TLS certificates**: These are used to store TLS/SSL certificates.
- **Bootstrap token Secrets**: These are used to store bearer tokens that are used when creating new clusters or joining new nodes to an existing cluster.

As a user of Kubernetes, you most typically will work with opaque secrets and TLS certificates. You've already worked with TLS secrets in *Chapter 6*, *Securing your application with HTTPS*. In this chapter, you will focus on opaque secrets.

Kubernetes provides three ways of creating secrets, as follows:

- Creating secrets from files
- Creating secrets from YAML or JSON definitions
- Creating secrets from the command line

Using any of the preceding methods, you can create any type of secret.

Kubernetes gives you two ways of consuming secrets:

- Using secrets as an environment variable
- Mounting secrets as a file in a pod

In the next section, you will create secrets using the three ways mentioned here, and you will later consume them using both the methods listed here.

Creating secrets in Kubernetes

In Kubernetes, there are three different ways to create secrets: from files, from YAML or JSON definitions, or directly from the command line. Let's start the exploration of how to create secrets by creating them from files.

Creating Secrets from files

The first way to create secrets in Kubernetes is to create them from a file. In this way, the contents of the file will become the value of the secret, and the filename will be the identifier of each value within the secret.

Let's say that you need to store a URL and a secure token for accessing an API. To achieve this, follow these steps:

1. Store the URL in `secreturl.txt`, as follows:

```
echo https://my-url-location.topsecret.com \
  > secreturl.txt
```

2. Store the token in another file, as follows:

```
echo 'superSecretToken' > secrettoken.txt
```

3. Let Kubernetes create the secret from the files, as follows:

```
kubectl create secret generic myapi-url-token \
  --from-file=./secreturl.txt --from-file=./secrettoken.txt
```

   Please note that you are creating a single secret object in Kubernetes, referring to both text files. In this command, you are creating an opaque secret by using the `generic` keyword.

   The command should return an output similar to *Figure 10.1*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo https://my-url-location.topsecret.com \
>   > secreturl.txt
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo 'superSecretToken' > secrettoken.txt
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl create secret generic myapi-url-token \
>   --from-file=./secreturl.txt --from-file=./secrettoken.txt
secret/myapi-url-token created
```

Figure 10.1: Creating an opaque secret

4. You can check whether the secrets were created in the same way as any other Kubernetes resource by using the `get` command:

```
kubectl get secrets
```

   This command will return an output similar to *Figure 10.2*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-xphvc   kubernetes.io/service-account-token   3      7h5m
myapi-url-token       Opaque                                2      8s
```

Figure 10.2: List of the created secrets

Here, you will see the secret you just created, and any other secrets that are present in the `default` namespace. The secret is of the `Opaque` type, which means that, from Kubernetes' perspective, the schema of the contents is unknown. It is an arbitrary key-value pair with no constraints, as opposed to, for example, SSH auth or TLS secrets, which have a schema that will be verified as having the required details.

5. For more details about the secret, you can also run the `describe` command:

```
kubectl describe secrets myapi-url-token
```

You will get an output similar to *Figure 10.3*:

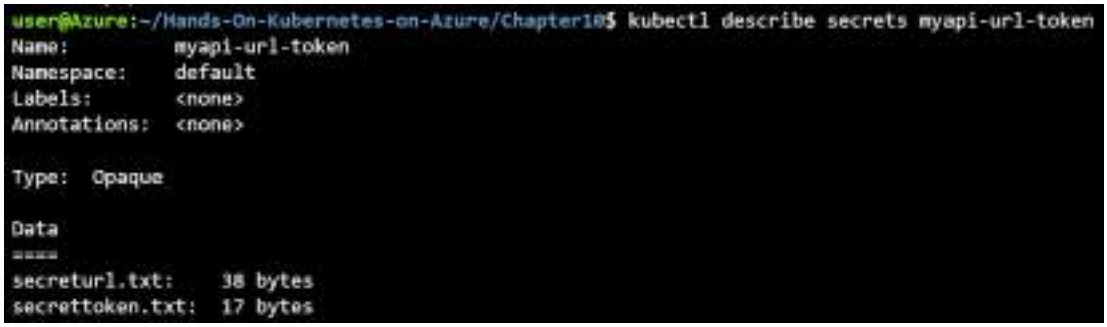


Figure 10.3: Description of the created secret

As you can see, neither of the preceding commands displayed the actual secret values.

6. To see the secret's value, you can run the following command:

```
kubectl get -o yaml secrets/myapi-url-token
```

You will get an output similar to *Figure 10.4*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get -o yaml secrets/myapi-url-token
apiVersion: v1
data:
  secrettoken.txt: c3VwZXJTZWNyZXRUb2tlbgo=
  secreturl.txt: aHR0cHM6Ly9teS1zcmwtbG9jYXRpb24udG9wc2VjcmV0LmNvbQo=
kind: Secret
metadata:
  creationTimestamp: "2021-01-31T03:32:11Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:secrettoken.txt: {}
        f:secreturl.txt: {}
      f:type: {}
    manager: kubectl-create
    operation: Update
    time: "2021-01-31T03:32:11Z"
  name: myapi-url-token
  namespace: default
  resourceVersion: "59163"
  selfLink: /api/v1/namespaces/default/secrets/myapi-url-token
  uid: 82027703-dda6-449f-a999-7e00f7365662
type: Opaque
```

Figure 10.4: Using the -o yaml switch in kubectl get secret fetches the encoded value of the secret

The data is stored as key-value pairs, with the filename as the key and the base64-encoded contents of the file as the value.

7. The preceding values are base64-encoded. Base64 encoding isn't secure. It obfuscates the secret so it isn't easily readable by an operator, but any bad actor can easily decode a base64-encoded secret. To get the actual values, you can run the following command:

```
echo 'c3VwZXJTZWNyZXRUb2tlbgo=' | base64 -d
echo 'aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW9uLnRvcHNlY3JldC5jb20K'|
base64 -d
```

You will get the values of the secrets that were originally created:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo 'c3VwZXJTZWNyZXRUb2tlbgo=' | base64 -d
superSecretToken
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo 'aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW
9uLnRvcHNlY3JldC5jb20K'| base64 -d
https://my-secret-url-location.topsecret.com
```

Figure 10.5: Base64-encoded secrets can easily be decoded

This shows you that the secrets are not securely encrypted in the default Kubernetes secret store.

In this section, you were able to create a secret containing an example URL with a secure token using files as the source. You were also able to get the actual secret values back by decoding the base64-encoded secrets.

Let's move on and explore the second method of creating Kubernetes secrets, creating secrets from YAML definitions.

Creating secrets manually using YAML files

In the previous section, you created a secret from a text file. In this section, you will create the same secret using YAML files by following these steps:

1. First, you need to encode the secret to `base64`, as follows:

   ```
   echo 'superSecretToken' | base64
   ```

   You will get the following value:

   ```
   c3VwZXJTZWNyZXRUb2tlbgo=
   ```

   You might notice that this is the same value that was present when you got the `yaml` definition of the secret in the previous section.

2. Similarly, for the `url` value, you can get the base64-encoded value, as shown in the following code block:

   ```
   echo 'https://my-secret-url-location.topsecret.com' | base64
   ```

   This will give you the base64-encoded URL:

   ```
   aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW9uLnRvcHNlY3JldC5jb20K
   ```

3. You can now create the secret definition manually; then, save the file. This file has been provided in the code bundle as `myfirstsecret.yaml`:

```
1   apiVersion: v1
2   kind: Secret
3   metadata:
4     name: myapiurltoken-yaml
5   type: Opaque
6   data:
7     url:
aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW9uLnRvcHNlY3JldC5jb20K
8     token: c3VwZXJTZWNyZXRUb2tlbgo=
```

   Let's investigate this file:

   - **Line 2**: This specifies that you are creating a secret.
   - **Line 5**: This specifies that you are creating an `Opaque` secret, meaning that from Kubernetes' perspective, values are unconstrained key-value pairs.
   - **Lines 7-8**: These are the base64-encoded values of the secret.

   You might notice that this YAML is very similar to the return you got in the previous section. This is because the object you use to create the secret in Kubernetes is stored with a bit more metadata on the Kubernetes API.

4. Now you can create the secret in the same way as any other Kubernetes resource by using the `create` command:

```
kubectl create -f myfirstsecret.yaml
```

   This will return an output similar to *Figure 10.6*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl create -f myfirstsecret.yaml
secret/myapiurltoken-yaml created
```

Figure 10.6: The secret was successfully created from a YAML file

5. You can verify whether the secret was successfully created using this:

```
kubectl get secrets
```

This will show you an output similar to *Figure 10.7*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-xphvc   kubernetes.io/service-account-token   3      7h19m
myapi-url-token       Opaque                                2      14m
myapiurltoken-yaml    Opaque                                2      9s
```

Figure 10.7: List of the created secrets

6. You can double-check that the secrets are the same by using `kubectl get -o yaml secrets myapiurltoken-yaml` in the same way that was described in the previous section.

This described a second way of creating secrets in Kubernetes. In the next section, you will learn the final way to create secrets, using literals in kubectl.

Creating generic secrets using literals in kubectl

The third method of creating secrets is by using the `literal` method, which means you pass the value in kubectl on the command line. As you have seen in the previous examples, a single secret in Kubernetes can contain multiple values. In the command to create a secret using the `literal` method, you use the syntax `--from-literal=<key>=<value>` to identify the different values in a secret:

1. To create a secret using the `literal` method, run the following command:

```
kubectl create secret generic myapiurltoken-literal \
  --from-literal=token='superSecretToken' \
  --from-literal=url=https://my-secret-url-location.topsecret.com
```

This will return an output similar to *Figure 10.8*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl create secret generic myapiurltoken-literal \
>   --from-literal=token='superSecretToken' \
>   --from-literal=url=https://my-secret-url-location.topsecret.com
secret/myapiurltoken-literal created
```

Figure 10.8: The secret was successfully created using a literal value in kubectl

2. You can verify that the secret was created by running the following command:

```
kubectl get secrets
```

This will give us a similar output to *Figure 10.9*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get secret
NAME                  TYPE                                  DATA   AGE
default-token-xphvc   kubernetes.io/service-account-token   3      7h26m
myapi-url-token       Opaque                                2      20m
myapiurltoken-literal Opaque                                2      72s
myapiurltoken-yaml    Opaque                                2      6m37s
```

Figure 10.9: Verifying the secret created using the literal method

Thus, you have created secrets using literal values in addition to the preceding two methods.

In this section, you've created Kubernetes secrets using three methods. In the next section, you'll explore two methods of using those secrets in your pods and applications.

Using your secrets

Once secrets have been created, they need to be linked to the application. This means that Kubernetes needs to pass the value of the secret to the running pods in some way. Kubernetes offers two ways to link your secrets to your application:

- Using secrets as environment variables
- Mounting secrets as files

Mounting secrets as files is the best way to consume secrets in your application. In this section, we will explain both methods, and also show why it's best to use the second method. Let's start by accessing secrets as environment variables.

Secrets as environment variables

You can use a secret in Kubernetes by referencing it as an environment variable. secrets can then be referenced in the pod definition under the `containers` and `env` sections. You will use the secrets that you previously created in a pod and learn how to use them in an application:

1. You can configure a pod with environment variable secrets like the definition provided in `pod-with-env-secrets.yaml`:

```
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: secret-using-env
5   spec:
6     containers:
7     - name: nginx
8       image: nginx
9       env:
10        - name: SECRET_URL
11          valueFrom:
12            secretKeyRef:
13              name: myapi-url-token
14              key: secreturl.txt
15        - name: SECRET_TOKEN
16          valueFrom:
17            secretKeyRef:
18              name: myapi-url-token
19              key: secrettoken.txt
20    restartPolicy: Never
```

   Let's inspect this file:

   - **Line 9**: Here, you are setting the environment variables.
   - **Lines 11-14**: Here, you refer to the `secreturl.txt` file in the `myapi-url-token` secret.
   - **Lines 16-19**: Here, you refer to the `secrettoken.txt` file in the `myapi-url-token` secret.

When Kubernetes creates a pod on a node that needs to use a secret, it will store that secret on that host in `tmpfs`, a temporary file system that is not written to disk. When the last pod referencing that secret is no longer running on that node, the secret is deleted from the node's `tmpfs`. If a node is shut down or rebooted, `tmpfs` is always erased.

2. Let's now create the pod and see whether you can access the secrets:

```
kubectl create -f pod-with-env-secrets.yaml
```

3. Check whether the environment variables are set correctly:

```
kubectl exec -it secret-using-env -- sh
echo $SECRET_URL
echo $SECRET_TOKEN
```

This should show you a result similar to *Figure 10.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl exec -it secret-using-env -- sh
# echo $SECRET_URL
https://my-url-location.topsecret.com
# echo $SECRET_TOKEN
superSecretToken
#
```

Figure 10.10: You can get the secrets inside the pod

4. You can now exit out of the shell to the container using the `exit` command.

There are a couple of things to note in this example. First, note that when you access the environment variables, you get the actual value of the secret back, not the base64-encoded value. This is as expected, since the base64 encoding is only applied at the Kubernetes API level, not at the application level.

The second thing to note is that you were able to access the secret by opening a shell into that running container and echoing the secret. It is important to apply the right level of RBAC to pods in Kubernetes, so that not every cluster user is able to run the `exec` command and open a shell.

Also note that both the application, in the form of the container image, and the pod definition had no hardcoded secrets. The secrets were provided by the dynamic configuration in Kubernetes.

The final thing to note is that any application can use the secret values by referencing the appropriate env variables. There is no way to limit which processes in a container can access which environment variables.

An important thing to know about secrets that are used as environment variables is that the value of the environment variable will not be updated when the secret itself is updated. This might cause you to end up in a state where pods that are created after a secret is updated have a different environment variable value compared to the pods created before the secret was updated.

In this section, you explored how to access secrets from within a running pod using environment variables. In the next section, you will explore how to achieve this using files.

Secrets as files

Let's take a look at how to mount the same secrets as files rather than environment variables:

1. You will use the following pod definition to demonstrate how this can be done. It is provided in the `pod-with-vol-secrets.yaml` file:

```
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: secret-using-volume
5   spec:
6     containers:
7     - name: nginx
8       image: nginx
9       volumeMounts:
10      - name: secretvolume
11        mountPath: "/etc/secrets"
12        readOnly: true
13    volumes:
14    - name: secretvolume
15      secret:
16        secretName: myapi-url-token
```

Let's have a closer look at this file:

- **Lines 9-12**: Here, you provide the mount details. You mount the secret in the `/etc/secrets` directory as read-only.
- **Lines 13-16**: Here, you refer to the secret. Please note that both values in the secret will be mounted in the container. You can optionally – although not shown here – specify which parts of a secret should be mounted in a volume.

Note that this is more succinct than the `env` definition, as you don't have to define a name for each secret. However, applications need to have special code to read the contents of the file in order to load it properly.

2. Let's see whether the secrets made it through. Create the pod using the following command:

```
kubectl create -f pod-with-vol-secret.yaml
```

3. Echo the contents of the files in the mounted volume:

```
kubectl exec -it secret-using-volume -- sh
cd /etc/secrets/
cat secreturl.txt
cat secrettoken.txt
```

As you can see in *Figure 10.11*, the secrets are present in the pod:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl exec -it secret-using-volume -- sh
# cd /etc/secrets/
# cat secreturl.txt
https://my-url-location.topsecret.com
# cat secrettoken.txt
superSecretToken
```

Figure 10.11: The secrets are available as files in our pod

4. You can now exit out of the shell to the container using the `exit` command.

There are a couple of things to note here as well. First, note that the secrets again are available in plain text and not in base64.

Second, since the secrets are mounted as a file, file system permissions apply to these secrets. This means that you can limit which processes can get access to the contents of these files.

Finally, Secrets mounted as files will be dynamically updated as the secrets are updated.

You have now learned two ways in which secrets can be passed to a running container. In the next section, it will be explained why it's best practice to use the file method.

Why secrets as files is the best method

Although it is a common practice to use secrets as environment variables, it is more secure to mount secrets as files. Kubernetes treats secrets as environment variables securely, but the container runtime doesn't treat them securely. To verify this, you can run the following commands to see the secret in plain text in the Docker runtime:

1. Start by getting the node that the pod using environment variables from the earlier example is running on with the following command:

   ```
   kubectl describe pod secret-using-env | grep Node
   ```

   This should show you the instance ID, as seen in *Figure 10.12*:

Figure 10.12: Getting the instance ID

2. Next, get the Docker ID of the running pod:

   ```
   kubectl describe pod secret-using-env | grep 'Container ID'
   ```

   This should give you the container ID:

Figure 10.13: Getting the Docker ID

3. Finally, you will execute a command on the node running your container to show the secret that was passed as an environment variable. First, let's set a couple of variables you'll use later:

```
INSTANCE=<provide instance number>
DOCKERID=<provide Docker ID>
VMSS=$(az vmss list --query '[].name' -o tsv)
RGNAME=$(az vmss list --query '[].resourceGroup' -o tsv)
```

> **Note**
>
> The previous command assumes you have a single AKS cluster with one node pool in your subscription. If this is not the case, please change the values of VMSS and RGNAME to the name of the value of the scale set and resource group running your cluster.

4. Depending on your node version, you will run either of the following commands. For clusters running on Kubernetes version 1.18.x or earlier, run the following command:

```
az vmss run-command invoke -g $RGNAME -n $VMSS --command-id \
RunShellScript --instance-id $INSTANCE --scripts \
"docker inspect -f '{{ .Config.Env }}' $DOCKERID" \
-o yaml | grep SECRET
```

This should return an output similar to *Figure 10.14*:

Figure 10.14: The Secrets are decoded in the Docker runtime

For clusters running version 1.19 or later, run the following command:

```
az vmss run-command invoke -g $RGNAME -n $VMSS --command-id \
RunShellScript --instance-id $INSTANCE --scripts \
"crictl inspect --output yaml $DOCKERID" \
-o yaml | grep SECRET
```

This will show you an output similar to *Figure 10.15*:

Figure 10.15: The secrets are decoded in the containerd runtime

This shows you both secrets in plain text in the container runtime, whether Docker (AKS version before 1.19) or containerd (AKS versions 1.19 and above).

As you can see, the secrets are decoded in the container runtime command. This means that most logging systems will log these sensitive secrets. Hence, it's advised to use secrets as files, since they are not passed in plain text except to the pod and the application.

Let's make sure to clean up the resources we created in this example:

```
kubectl delete pod --all
kubectl delete secret myapi-url-token \
myapiurltoken-literal myapiurltoken-yaml
```

Now that you have explored secrets in Kubernetes using the default secrets mechanism, let's go ahead and use a more secure option, namely Azure Key Vault.

Installing the Azure Key Vault provider for Secrets Store CSI driver

In the previous section, you explored secrets that were stored natively in Kubernetes. This means they were base64-encoded on the Kubernetes API server. You saw in the previous section that base64-encoded secrets are not secure at all. For highly secure environments, you will want to use a better secret store.

Azure offers an industry-compliant key and secret storage solution called Azure Key Vault. It is a managed service that makes creating, storing, and retrieving keys and secrets easy, and offers auditing of access to your keys and secrets.

The Kubernetes community maintains a project called the Kubernetes Secrets Store CSI driver (https://github.com/kubernetes-sigs/secrets-store-csi-driver). This project allows you to integrate external secret stores with volumes in Kubernetes through the CSI driver. The Container Storage Interface is a standardized way in Kubernetes to interface with storage systems. There are multiple implementations of the Secret Store CSI driver. At the time of writing, the current implementations are Hashicorp Vault, Google Cloud Platform, and Azure Key Vault.

Microsoft maintains the Key Vault implementation of the Secret Store CSI driver, named Azure Key Vault provider for Secrets Store CSI driver. This implementation allows you as a user to access Key Vault secrets from within Kubernetes. It is also integrated with pod identities to restrict access to secrets. Optionally, this implementation can also sync Key Vault secrets with Kubernetes secrets so you can use them as an environment variable if needed.

> **Note**
>
> For brevity, we'll refer to Azure Key Vault provider for Secrets Store CSI driver as the CSI driver for Key Vault.

At the time of writing, the CSI driver for Key Vault is only available as an open-source project that you can install on your cluster. It is worth noting that this solution might be introduced as a managed add-on to AKS in the future. For more up-to-date details, please refer to this issue on GitHub at https://github.com/Azure/AKS/issues/1876.

To work with the CSI driver for Key Vault, there are two things you need to do. First, you need to set up the driver itself on your cluster. That is the goal of this section. Secondly, you'll need to create an object in Kubernetes called a `SecretProviderClass` for each secret from Key Vault you need to access. You will learn more about this in the next section.

In this section, you will set up the CSI driver for Key Vault. First, you will create a new user-assigned managed identity. After that, you'll create a new key vault and give the user-assigned managed identity permissions to that key vault. Finally, you'll set up the CSI driver for Key Vault on your cluster.

Let's start by creating a new managed identity.

Creating a managed identity

The CSI driver for Key Vault supports different ways of getting data out of Key Vault. It is recommended that you use a managed identity to link your Kubernetes cluster to Key Vault. For this, you can use the AAD pod-managed identity add-on that you set up in the previous chapter. In this section, you'll create a new managed identity in Azure to later use with Key Vault:

1. Let's create a new managed identity. You will use the Azure portal to do this. To start, look for `managed identity` in the Azure search bar, as shown in *Figure 10.16*:

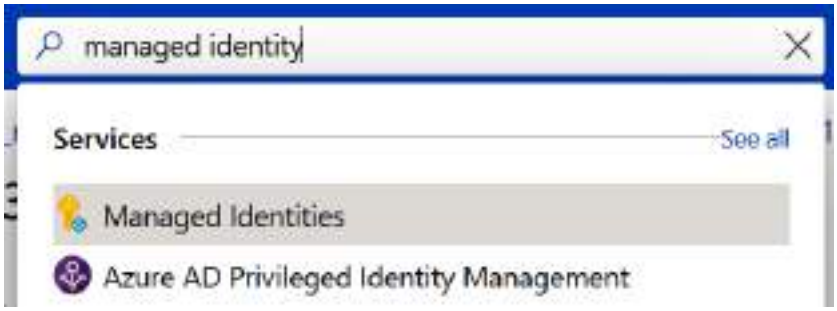


Figure 10.16: Looking for managed identity in the Azure search bar

2. In the resulting pane, click the **+ New** button at the top. To organize the resources for this chapter together, it's recommended to create a new resource group. In the resulting pane, click the **Create new** button to create a new resource group. Call it `csi-key-vault`, as shown in *Figure 10.17*:

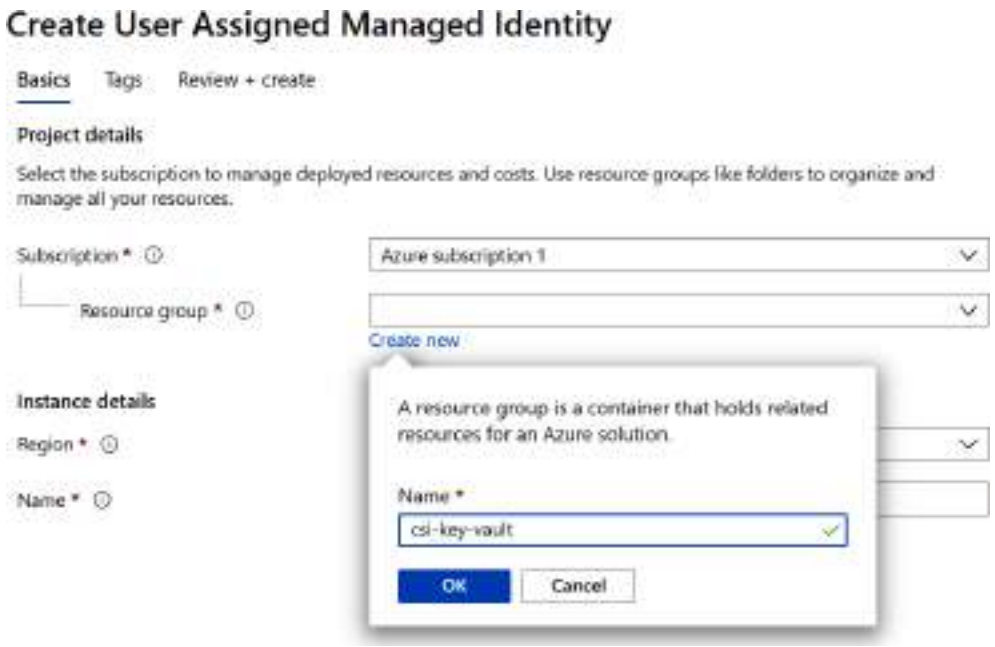


Figure 10.17: Creating a new resource group

3. Now, select the region you created your cluster in as the region for your managed identity and give it a name, `csi-key-vault` if you follow the example, as shown in *Figure* 10.18. To finish, click the **Review + create** button and in the final window, click the **Create** button to create your managed identity:

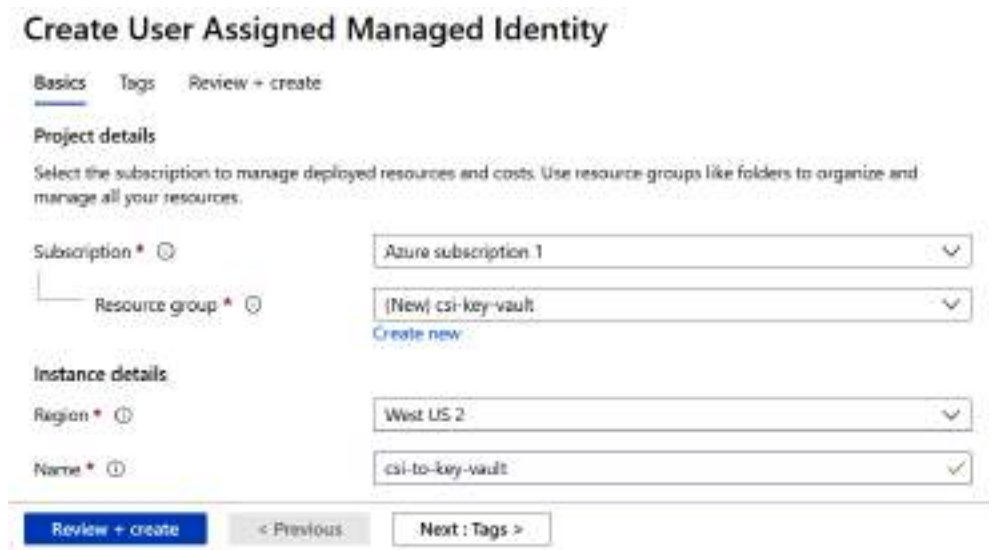


Figure 10.18: Providing Instance details

4. Once the managed identity has been created, hit the **Go to resource** button to go to the resource. Here, you will need to copy the resource ID that will be used later in the next step. You can find that in the **Properties** pane of the managed identity, as shown in *Figure 10.19*:

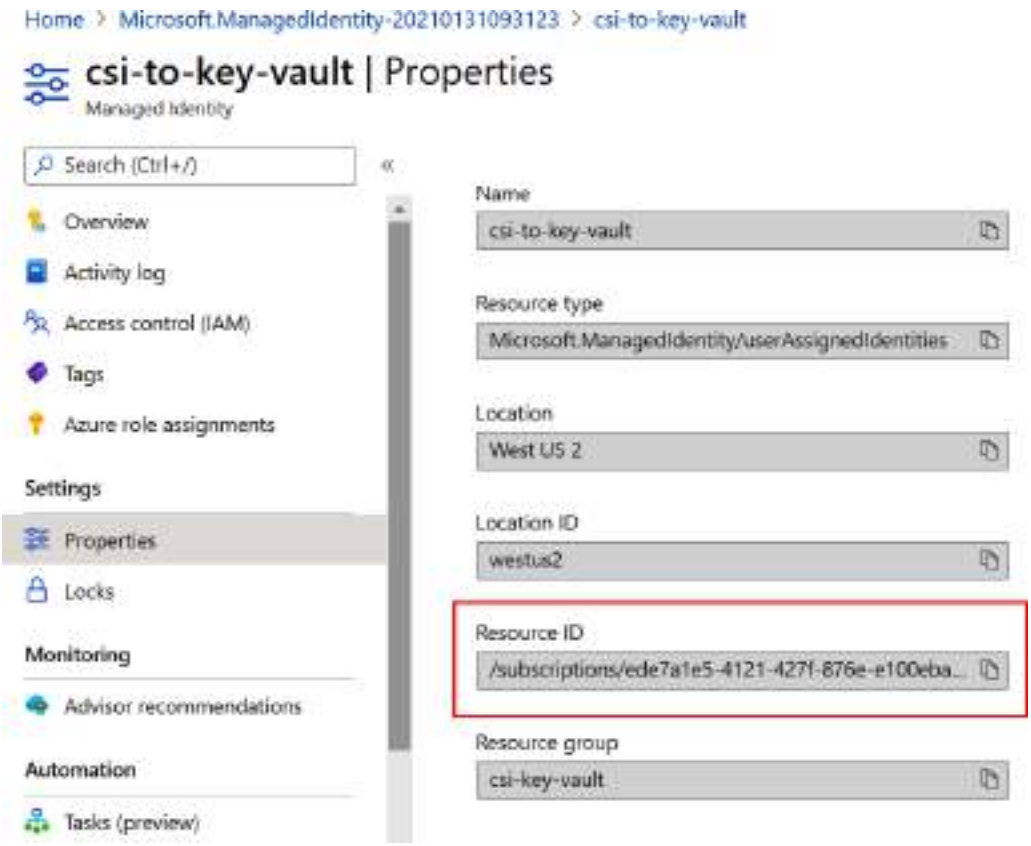


Figure 10.19: Getting the Resource ID of the managed identity

5. Now you are ready to link the managed identity to your AKS cluster. To do this, you will run a command in cloud shell as you did in the previous chapter. Afterward, you will verify that the identity is available in your cluster. Let's start with linking the identity:

```
az aks pod-identity add --resource-group rg-handsonaks \
  --cluster-name handsonaks --namespace default \
  --name csi-to-key-vault \
  --identity-resource-id <Managed identity resource ID>
```

6. You can verify that your identity was successfully linked to your cluster by running the following command:

```
kubectl get azureidentity
```

This should produce an output similar to *Figure 10.20*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get azureidentity
NAME                AGE
csi-to-key-vault    44s
```

Figure 10.20: Verifying the availability of the identity in the cluster

In this section, you created a new managed identity and linked that to your Kubernetes cluster using the AAD Pod-managed identity add-on. In the next section, you'll create a key vault and give the new identity you created access to the secrets. Finally, you'll create a secret in Key Vault that you try to access later from your cluster.

Creating a key vault

In the previous section, you set up the managed identity that the CSI driver for Key Vault will use. In this section, you'll create the key vault that will be used:

1. To start the creation process, look for `Key vaults` in the Azure search bar:

Figure 10.21: Navigating to Key vaults through the Azure portal

2. Click the **+ New** button to start the creation process:

Figure 10.22: Click the Add button to start creating a key vault

3. Provide the details to create the key vault. Create the key vault in the resource group you created in the previous step. The key vault's name has to be globally unique, so consider adding your initials to the name. It is recommended that you create the key vault in the same region as your cluster:

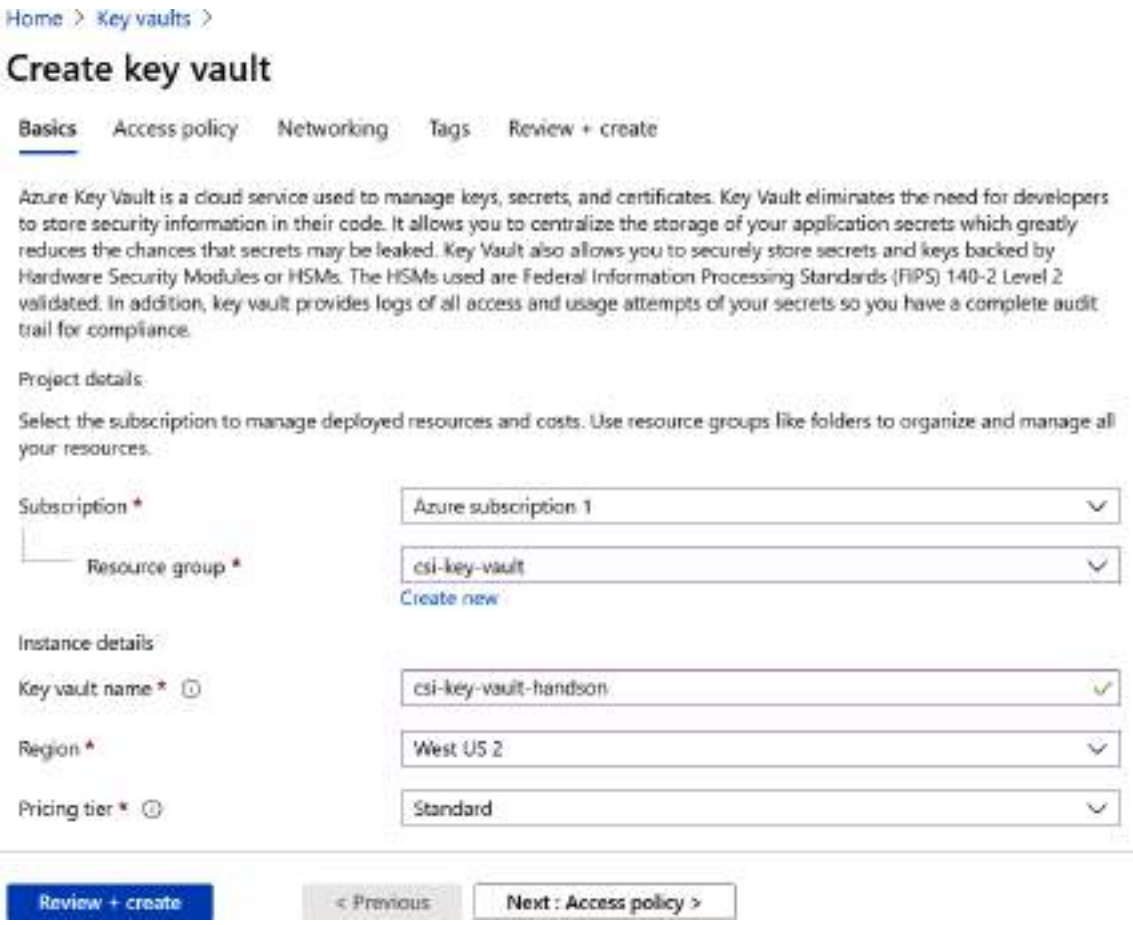


Figure 10.23: Providing the details to create the key vault

4. After you have provided the details for your key vault, click the **Next: Access policy >** button to give the managed identity access to secrets. Click on the **+ Add Access Policy** to give permission to your managed identity, as shown in *Figure 10.24*:

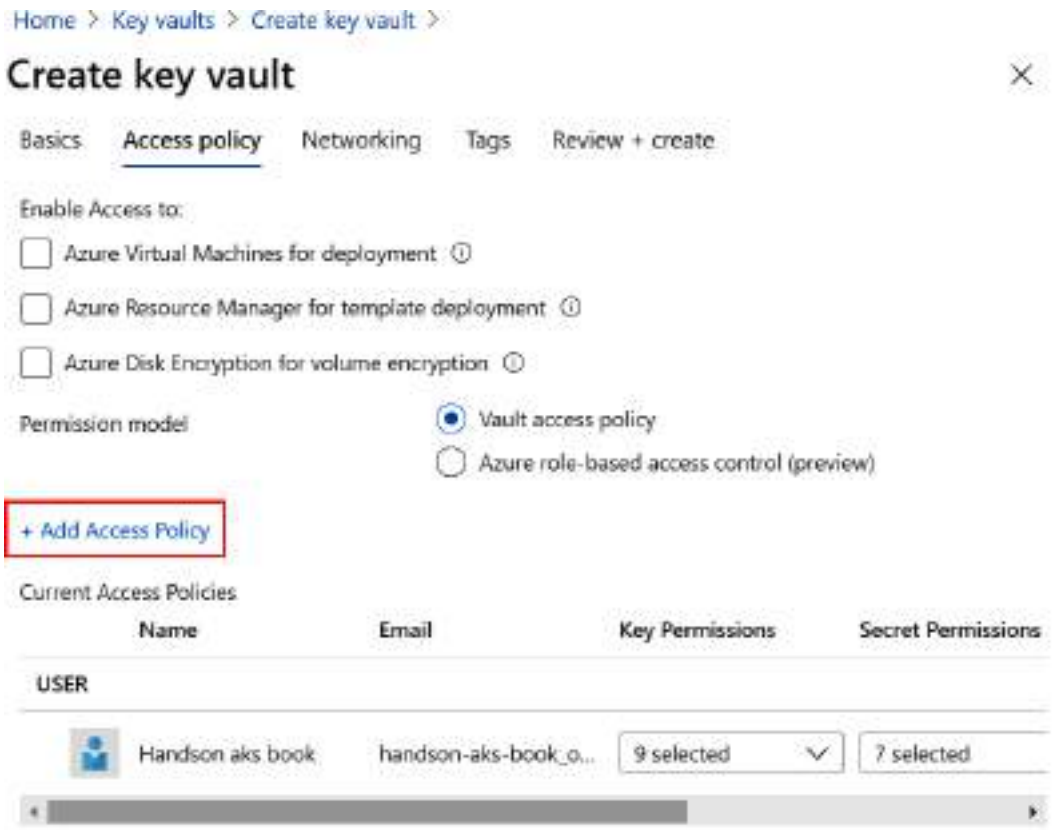


Figure 10.24: Adding an Access policy

In the resulting pane, select the **Secret Management** template, click on the **None Selected** button underneath **Select principal**, and in the resulting pane look for the `csi-to-key-vault` you created earlier. Finally, click on **Select** at the bottom of the screen and then on **Add**, as shown in *Figure 10.25*:

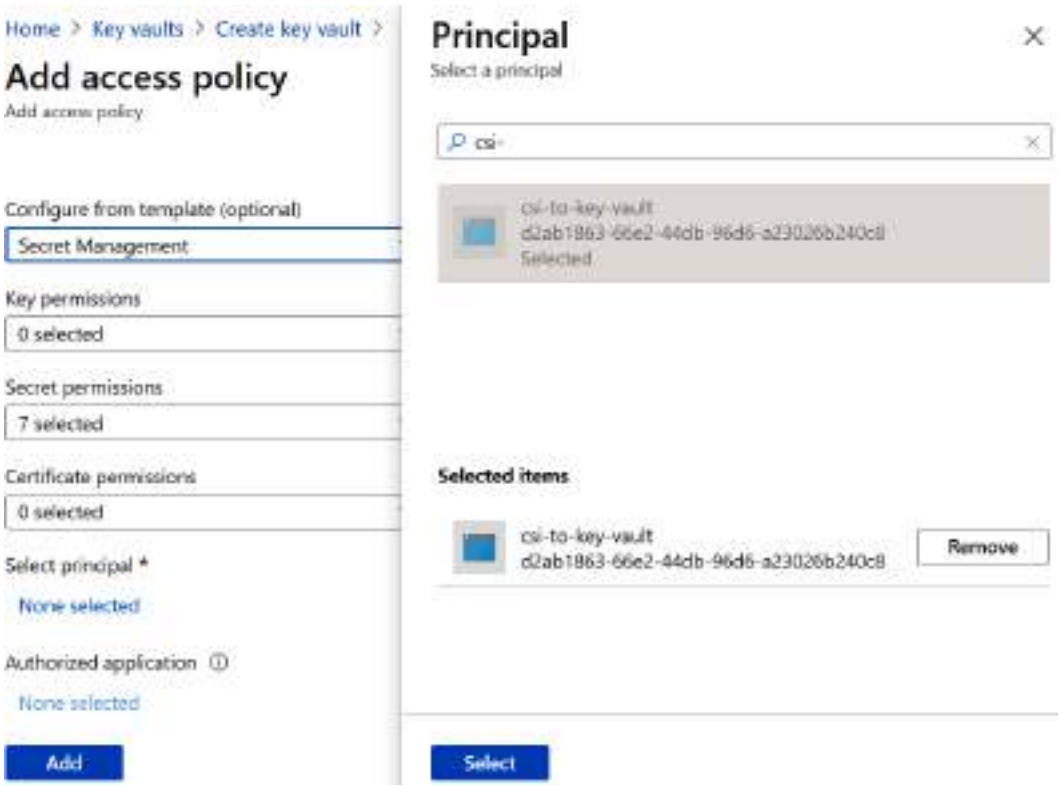


Figure 10.25: Assigning the Secret Management template to your managed identity

5. Once you have provided permissions to this managed identity, hit the **Review + create** button to review and create your key vault. Hit the **Create** button to finish the creation process.

6. It will take a couple of seconds to create your key vault. Once the vault is created, click on the **Go to resource** button, go to **Secrets**, and hit the **Generate/Import** button to create a new secret as shown in *Figure 10.26*:

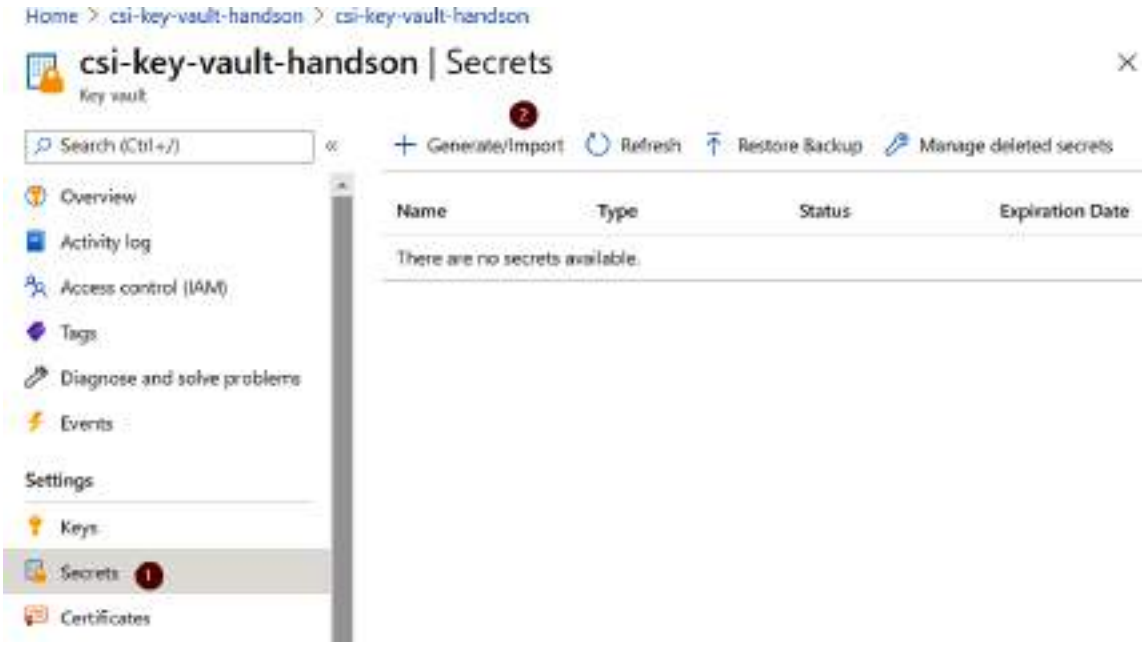


Figure 10.26: Creating a new secret

7. In the secret creation wizard, provide the details about your secret. To make this demonstration easier to follow, use the name `k8s-secret-demo`. Give the secret a memorable value, such as `secret-coming-from-key-vault`. Click the **Create** button at the bottom of the screen to create the secret:

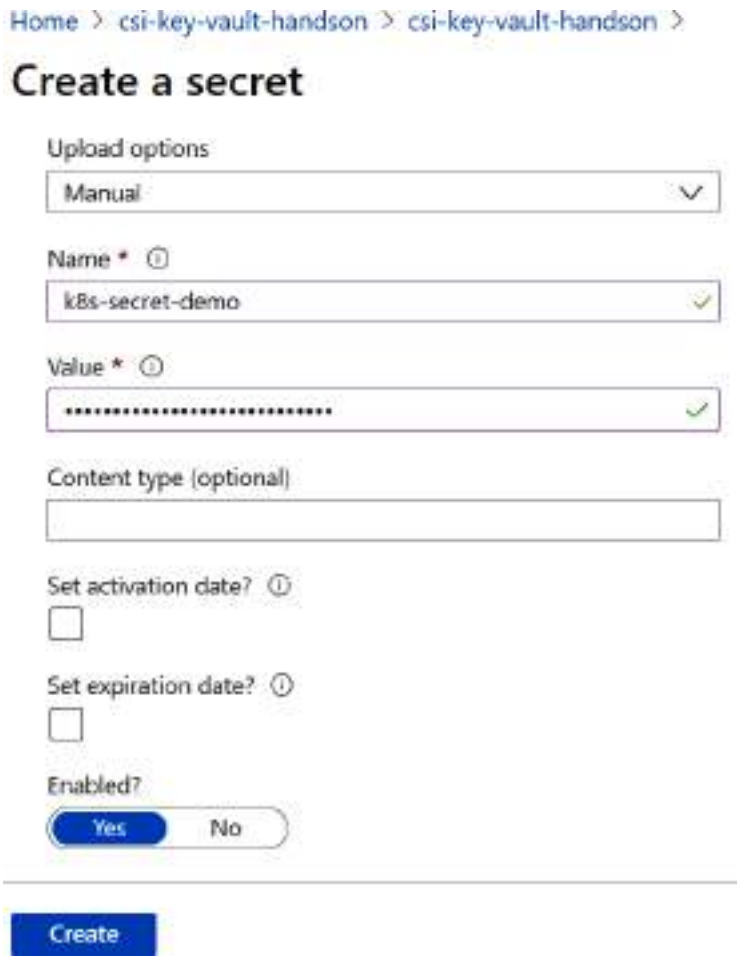


Figure 10.27: Providing the details for your new secret

Now that you have a secret in Key Vault, you can move ahead and install the actual CSI driver for Key Vault in your cluster.

Installing the CSI driver for Key Vault

In this section, you will set up the CSI driver for Key Vault in your cluster. This will allow you, in the next section, to retrieve secrets from Key Vault. The installation is a short process, as you will see here:

1. The easiest way to install the CSI driver for Key Vault is to use Helm, as you've done before. Note that this feature may be available as an add-on after the release of this book. To do this, add the repo for the CSI driver for Key Vault:

   ```
   helm repo add csi-secrets-store-provider-azure \
     https://raw.githubusercontent.com/Azure/secrets-store-csi-driver-
   provider-azure/master/charts
   ```

2. Once the repo has been added, you can install the actual CSI driver for Key Vault using the following command:

   ```
   helm install csi-secrets \
     csi-secrets-store-provider-azure/csi-secrets-store-provider-azure
   ```

3. To verify that the installation succeeded, you can verify that the SecretProviderClass CRD has been added to your cluster via the following command:

   ```
   kubectl get crd
   ```

   This should show you an output that contains the SecretProviderClass CRD as shown in *Figure 10.28*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get crd
NAME                                                  CREATED AT
azureidentities.aadpodidentity.k8s.io                 2021-01-30T20:27:18Z
azureidentitybindings.aadpodidentity.k8s.io           2021-01-30T20:27:18Z
azurepodidentityexceptions.aadpodidentity.k8s.io      2021-01-30T20:27:18Z
secretproviderclasses.secrets-store.csi.x-k8s.io      2021-01-31T18:05:20Z
secretproviderclasspodstatuses.secrets-store.csi.x-k8s.io   2021-01-31T18:05:20Z
```

Figure 10.28: The SecretProviderClass CRD has been added to the cluster

This concludes the setup of the CSI driver for Key Vault. In this section, you first created a managed identity, then created a key vault with a secret in it, and then finally set up the CSI driver for Key Vault on your cluster.

You are now ready to use the CSI driver for Key Vault, which you'll do in the next section.

Using the Azure Key Vault provider for Secrets Store CSI driver

Now that the CSI driver for Key Vault has been set up on your cluster, you are ready to start using it. In this section, you'll run through two examples of using the CSI driver for Key Vault. First, you will use it to mount a secret as a file in Kubernetes. Afterward, you will also use it to sync Key Vault secrets to Kubernetes secrets and use them as an environment variable.

Let's get started with the first example, how to mount Key Vault secrets as a file.

Mounting a Key Vault secret as a file

In this first example, you will create a new SecretProviderClass in your cluster. This object will allow you to link a secret in Key Vault to a pod in Kubernetes. After that, you'll create a pod that uses that SecretProviderClass and mounts the secrets in that pod. Let's get started:

1. The SecretProviderClass requires you to know your Azure Active Directory tenant ID. To get this, run the following command:

   ```
   az account show --query tenantId
   ```

   This will show you an output similar to *Figure 10.29*. Copy-paste this value and store it in a file you can refer to later:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ az account show --query tenantId
"1cf4b872-ae04-44c8-8318-2ba43e95f591"
```

Figure 10.29: Getting your tenant ID

Next, you'll create the SecretProviderClass. An example has been provided in the code files for this chapter, in the `secretproviderclass-file.yaml` file:

```
1   apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
2   kind: SecretProviderClass
3   metadata:
4     name: key-vault-secret-file
5   spec:
6     provider: azure
7     parameters:
8       usePodIdentity: "true"
9       keyvaultName: "<key vault name>"
10      objects:  |
11        array:
12          - |
13            objectName: k8s-secret-demo
14            objectType: secret
15      tenantId: "<your tenant ID>"
```

Let's investigate this file:

- **Line 2**: Here, you define you are creating a `SecretProviderClass`.
- **Line 6**: Here, you create an Azure secret. As mentioned in the introduction, the secret-store project supports multiple implementations.
- **Line 8**: You configure this secret to use pod identities for authentication. You will link a pod identity to your pod later on.
- **Line 9**: The name of the key vault.
- **Line 10-14**: Here, you refer to the secrets that need to be accessed. In this example, you're only accessing a single secret, but you could access multiple secrets in a single `SecretProviderClass`.
- **Line 15**: The AAD tenant ID of your AAD tenant.

Make sure to edit this with the values for your environment.

2. You can create this `SecretProviderClass` using the following command:

```
kubectl create -f secretproviderclass-file.yaml
```

3. Once the SecretProviderClass has been created, you can go ahead and create a pod that references that SecretProviderClass. An example has been provided in the `pod-keyvault-file.yaml` file:

```
1   kind: Pod
2   apiVersion: v1
3   metadata:
4     name: csi-demo-file
5     labels:
6       aadpodidbinding: "csi-to-key-vault"
7   spec:
8     containers:
9       - name: nginx
10        image: nginx
11        volumeMounts:
12        - name: keyvault
13          mountPath: "/mnt/secrets-store"
14          readOnly: true
15    volumes:
16      - name: keyvault
17        csi:
18          driver: secrets-store.csi.k8s.io
19          readOnly: true
20          volumeAttributes:
21            secretProviderClass: "key-vault-secret-file"
```

Let's have a look at the key parts of this file:

- **Line 5-6**: This is where you link this pod to the managed identity you created earlier.
- **Line 11-14**: Here, you define where you want to mount the secrets.
- **Line 15-21**: Here, you define the actual Volume and the link to Key Vault. On line 21, you refer to the SecretProviderClass you created earlier.

4. You can create this pod using the following command:

```
kubectl create -f pod-keyvault-file.yaml
```

5. Monitor the Pod's creation using the following command:

```
kubectl get pods -w
```

This should return an output similar to *Figure 10.30*:

Figure 10.30: Status of the csi-demo-file pod changes to Running

6. Once the pod is created and running, you can open a shell in the pod using the `kubectl exec` command and verify that the secret is present:

```
kubectl exec -it csi-demo-file -- sh
cd /mnt/secrets-store
cat k8s-secret-demo
```

This should output the secret you created in Key Vault, as seen in *Figure 10.31*:

Figure 10.31: The secret you configured in Key Vault is mounted in the pod as a file

And as expected, you are able to get the secret you configured in Key Vault to show up in Kubernetes.

7. You can now exit out of the shell to the container using the `exit` command.

As you can see, you successfully used the CSI driver for Key Vault to get a secret from Key Vault to show up as a file in a pod.

It is also possible to sync secrets in Key Vault to secrets in Kubernetes and then use them as an environment variable in running pods. That's what you'll explore in the next and final section of this chapter.

Using a Key Vault secret as an environment variable

In the previous section, you saw how to access Key Vault secrets as a file in a pod. As you learned earlier in this chapter, it is recommended that you use Kubernetes secrets as a file.

However, there are situations where you cannot modify an application to use secrets as a file and you need to use them as environment variables. This can be done using the CSI driver for Key Vault, and you will configure the driver that way in this section. Please note that in order for the CSI driver to sync secrets in Key Vault to Secrets in Kubernetes, you need to mount the secret as a Volume in Kubernetes; you cannot only rely on the secret syncing.

Let's configure all of this:

1. First, you'll create the SecretProviderClass. An example has been provided in the code files for this chapter, in the `secretproviderclass-env.yaml` file:

```
1   apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
2   kind: SecretProviderClass
3   metadata:
4     name: key-vault-secret-env
5   spec:
6     provider: azure
7     parameters:
8       usePodIdentity: "true"
9       keyvaultName: "<key vault name>"
10      objects:  |
11        array:
12          - |
13            objectName: k8s-secret-demo
14            objectType: secret
15      tenantId: "<tenant ID>"
16    secretObjects:
17    - secretName: key-vault-secret
18      type: Opaque
19      data:
20      - objectName: k8s-secret-demo
21        key: secret-content
```

Let's investigate what's different in this file versus the previous one you created:

- **Line 16-21**: This is where you link the Key Vault secret to a Kubernetes secret. The names used here are important since they provide critical information about the different objects:
- **Line 17** `secretName`: This refers to the name of the secret in Kubernetes that will be created.
- **Line 20** `objectName`: This refers to the `objectName` on line 13, which is the name of the secret in Key Vault.
- **Line 21** `key`: This is the name of the key in the secret in Kubernetes. As was explained earlier in this chapter, a single secret in Kubernetes can contain multiple keys.

The remaining sections of this file are similar to the earlier SecretProviderClass you created.

2. You can create this SecretProviderClass using the following command:

```
kubectl create -f secretproviderclass-env.yaml
```

3. Once the SecretProviderClass has been created, you can go ahead and create a pod that references that SecretProviderClass. You cannot rely solely on the syncing of the secrets, the SecretProviderClass has to be mounted in order for the CSI driver to sync the secrets. An example has been provided in the `pod-keyvault-env.yaml` file:

```
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: csi-demo-env
5     labels:
6       aadpodidbinding: "csi-to-key-vault"
7   spec:
8     containers:
9     - name: nginx
10      image: nginx
```

```
11      env:
12        - name: KEYVAULT_SECRET
13          valueFrom:
14            secretKeyRef:
15              name: key-vault-secret
16              key: secret-content
17      volumeMounts:
18        - name: keyvault
19          mountPath: "/mnt/secrets-store"
20          readOnly: true
21    volumes:
22      - name: keyvault
23        csi:
24          driver: secrets-store.csi.k8s.io
25          readOnly: true
26          volumeAttributes:
27            secretProviderClass: "key-vault-secret-env"
```

The difference between this pod and the previous one you created is on *lines 11 to 16*. This should seem familiar, as this is the typical way to use a secret as an environment variable.

4. You can create this pod using the following command:

```
kubectl create -f pod-keyvault-env.yaml
```

5. Monitor the Pod's creation using the following command:

```
kubectl get pods -w
```

This should return an output similar to *Figure 10.32*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get pods -w
NAME                                                READY   STATUS              RESTARTS   AGE
csi-demo-env                                        0/1     ContainerCreating   0          1s
csi-demo-file                                       1/1     Running             0          38m
csi-secrets-csi-secrets-store-provider-azure-lnslh  1/1     Running             0          120m
csi-secrets-csi-secrets-store-provider-azure-ndzfg  1/1     Running             0          120m
csi-secrets-secrets-store-csi-driver-kvc82          3/3     Running             0          120m
csi-secrets-secrets-store-csi-driver-xw56w          3/3     Running             0          120m
csi-demo-env                                        1/1     Running             0          4s
```

Figure 10.32: Waiting for the csi-demo-env pod to run

6. Once the pod is created and running, you can open a shell in the pod using the `kubectl exec` command and verify that the secret is present:

```
kubectl exec -it csi-demo-env -- sh
echo $KEYVAULT_SECRET
```

This should output the secret you created in Key Vault, as seen in *Figure 10.33*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl exec -it csi-demo-env -- sh
# echo $KEYVAULT_SECRET
secret-coming-from-key-vault
```

Figure 10.33: The secret you configured in Key Vault is used as an environment variable

7. You can now exit out of the shell to the container using the `exit` command.
8. Finally, you can also verify that the secret was created in Kubernetes by running the following command:

```
kubectl get secret
```

This should show you an output similar to *Figure 10.34*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get secret
NAME                                          TYPE                                  DATA   AGE
csi-secrets-store-provider-azure-token-8cm9k  kubernetes.io/service-account-token   3      124m
default-token-xphvc                           kubernetes.io/service-account-token   3      23h
key-vault-secret                              Opaque                                1      4m17s
secrets-store-csi-driver-token-c44td          kubernetes.io/service-account-token   3      124m
sh.helm.release.v1.csi-secrets.v1             helm.sh/release.v1                    1      124m
```

Figure 10.34: The key-vault-secret secret in Kubernetes is synced with the Key Vault secret

9. This secret will disappear once no more pods that mount the secret are present. You can verify this with the following commands:

```
kubectl delete -f pod-keyvault-env.yaml
kubectl get secret
```

This should show you an output similar to *Figure 10.35*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl delete -f pod-keyvault-env.yaml
pod "csi-demo-env" deleted
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get secret
NAME                                          TYPE                                  DATA   AGE
csi-secrets-store-provider-azure-token-8cm9k  kubernetes.io/service-account-token   3      126m
default-token-xphvc                           kubernetes.io/service-account-token   3      23h
secrets-store-csi-driver-token-c44td          kubernetes.io/service-account-token   3      126m
sh.helm.release.v1.csi-secrets.v1             helm.sh/release.v1                    1      126m
```

Figure 10.35: Deleting the pod automatically deletes the secret as well

This shows you that although you have a SecretProviderClass that tries to sync a Key Vault secret to a Kubernetes secret, that syncing only happens once a pod references that SecretProviderClass and mounts the secret.

In this section, you've been able to sync a secret in Key Vault to a secret in Kubernetes. You were able to access that secret's value in a pod using environment variables.

This also concludes this chapter on secrets in Kubernetes. Let's make sure to clean up all the objects that we created:

```
kubectl delete -f .
helm delete csi-secrets
az aks pod-identity delete --resource-group rg-handsonaks \
  --cluster-name handsonaks --namespace default \
  --name csi-to-key-vault
az group delete -n csi-key-vault --yes
```

Once the resources are deleted, you are ready to move on to the next chapter about network security in AKS.

Summary

In this chapter, you learned about secrets in Kubernetes. You worked with both the default secret mechanism in Kubernetes as well as with the Azure Key Vault provider for Secrets Store CSI driver.

This chapter started by explaining different secret types in Kubernetes. After that, you used different mechanisms in Kubernetes to create secrets. You then used two methods of accessing those secrets, using them as files or as environment variables.

After that, you created a managed identity and a key vault to experiment with the CSI driver for Key Vault. You installed that on your cluster and used two mechanisms to access secrets in Key Vault: either using files or using environment variables.

In the next chapter, you'll learn more about network security in AKS.

11

Network security in AKS

Securing a network is a critical activity in the protection of an application. The goal of a secure network is, on the one hand, to allow your users to connect to your applications and use all the functionalities you offer. On the other hand, you also need to protect your network from attackers. This means making sure that they cannot get access to critical parts of your network, and that even if they were to gain access, this would be limited.

When it comes to network security in AKS, there are two different layers to secure the network. The first is the control plane. The control plane refers to the managed Kubernetes master servers that host the Kubernetes API. By default, the control plane is exposed to the internet. You can secure the control plane either by limiting which public IP addresses can access it using a feature called **Authorized IP ranges**, or by deploying a private cluster, which means only the machines connected to your virtual network can access the control plane.

The second network layer to secure is the workload running on your cluster. There are multiple ways to secure the workload. The first way is by using Azure networking functionalities, such as the Azure Firewall or **Network Security Groups** (**NSGs**). The second way to protect the workload is by using a Kubernetes functionality called network policies.

In this chapter, you will explore the different ways to secure the network of an AKS cluster. Specifically, this chapter contains the following sections:

- Networking and network security in AKS
- Control plane network security
- Workload network security

Since most networking configurations of an AKS cluster are only configurable during cluster creation, you will create and destroy multiple clusters throughout this chapter.

Let's start this chapter by exploring the concepts of networking and network security in AKS.

Networking and network security in AKS

This section serves as an introduction to the concepts of networking and security in AKS. You'll first cover the control plane, then workload networking, and then network security.

Control plane networking

The control plane of a Kubernetes cluster is the infrastructure that hosts the Kubernetes API server for your cluster, manages the scheduler, and stores the cluster state. When you interact with a Kubernetes cluster, for instance, by using `kubectl`, you are sending commands to the Kubernetes API server. In AKS, this control plane is managed by Microsoft and provided to you as a service.

By default, the control plane is exposed over the internet and is accessible to everybody that is connected to the internet. This doesn't mean that the control plane is not secure though. Even if an attacker had network access to your control plane, they would still need to have cluster credentials to execute commands against the control plane.

Frequently, though, organizations still want to limit network access to the control plane of their AKS clusters. There are two functionalities in AKS that enable you to limit network access to the control plane of a cluster.

The first functionality is called **authorized IP address ranges**. By configuring authorized IP address ranges on your AKS, you configure which IP addresses are allowed to access your API server. This means that IP addresses that are not allowed to access your API server cannot interact with your API server. This is explained in *Figure 11.1*:

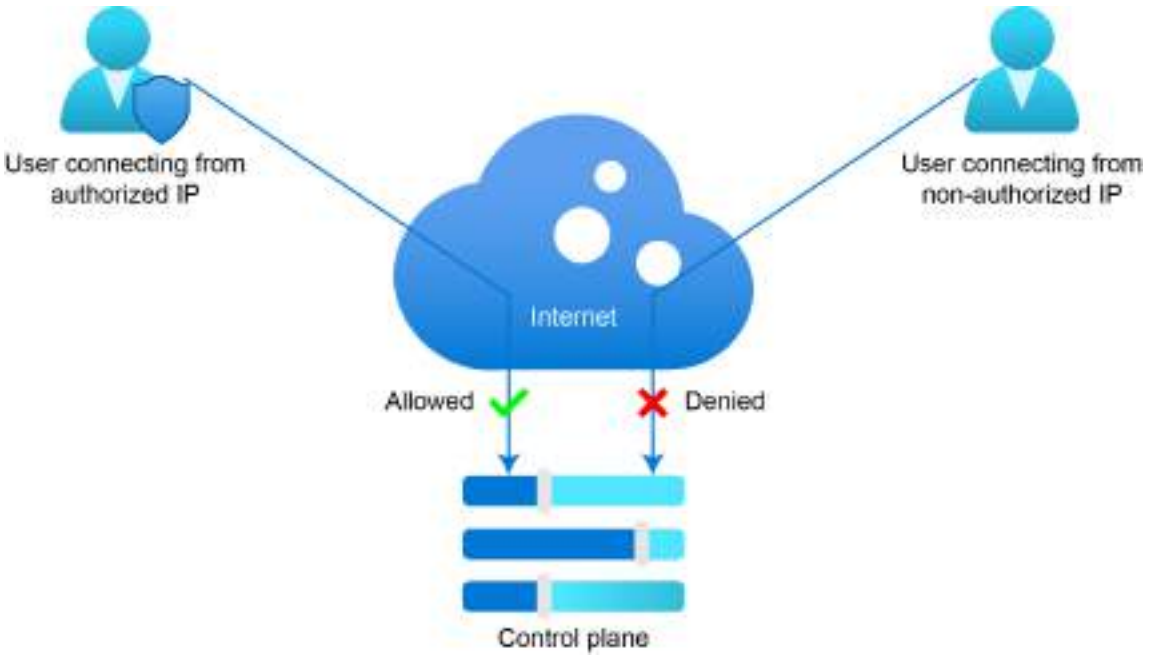


Figure 11.1: Authorized IP ranges explained

Another way to limit network access to your control plane is by using a feature called **private clusters**. By configuring private clusters, you do not give your control plane a publicly reachable address. The cluster is only reachable from a private network. To connect to the control plane, you would need to use a machine that is connected to an **Azure Virtual Network** (**VNet**). This machine would communicate to the control plane using an Azure functionality called Azure Private Link.

Private Link is an Azure feature that allows you to connect to managed services using a private IP address in your VNet. When using Private Link, a Private Link endpoint is created in your VNet. To connect to this Private Link endpoint, you would have to connect from either a VM hosted in the same VNet, in a peered VNet, or through a VPN or Azure ExpressRoute that is connected to that VNet. In *Figure 11.2*, you see an example of how this works using a VM hosted in the same VNet. You can see that the node pools (1) that host your workloads as well as VMs (2) connected to the same VNet can connect to the control plane, but a user connecting over the internet (3) cannot:

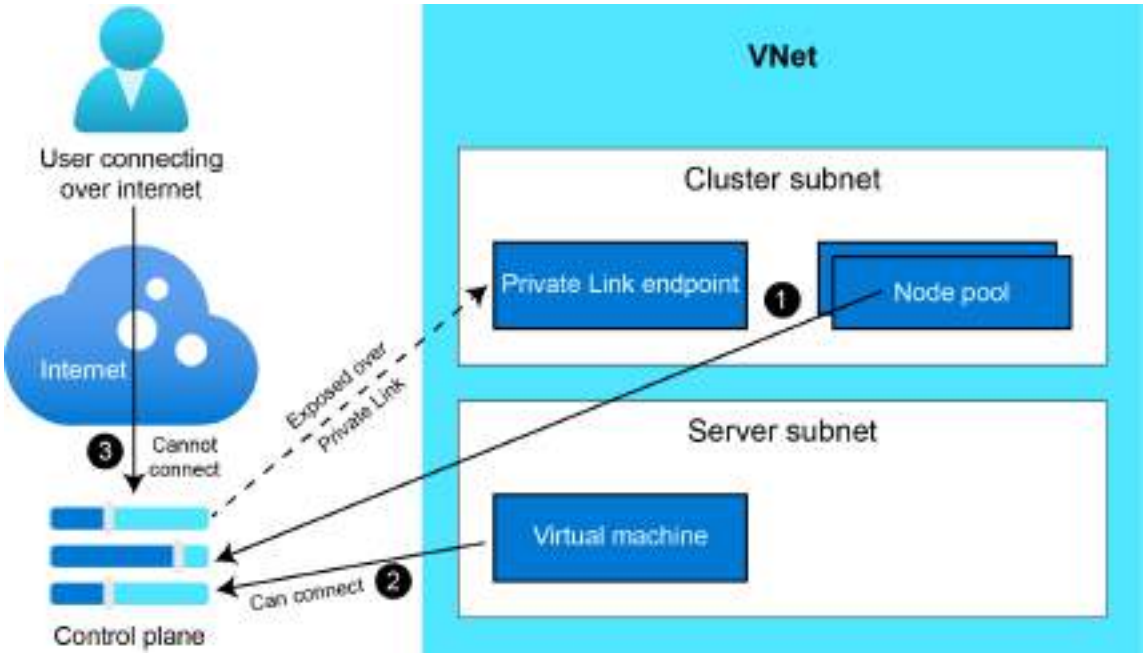


Figure 11.2: Private clusters explained

It is important to understand that both authorized IP address ranges and private clusters only provide network security to the Kubernetes control plane; they do not influence the workload network. Workload networking will be covered in the next section.

Workload networking

Your workloads in AKS are deployed on a cluster that is deployed in a VNet. There are many ways to configure and secure networking in a VNet. In this section, we will introduce several important configuration options for network security of the workload deployed in a VNet. This is, however, only an introduction to these concepts. Before deploying a production cluster, please refer to the AKS documentation for a more in-depth review of the different configuration options: https://docs.microsoft.com/azure/aks/concepts-network.

You'll first need to choose the networking model with which you'll deploy your cluster. This configuration has a limited impact on security, but it is important to understand from a networking perspective. There are two options:

- **Kubenet networking (default)**: By using kubenet networking, cluster nodes get an IP address from a subnet in a VNet. The pods running on those nodes get an IP address from an overlay network, which uses a different address space from the nodes. Pod-to-pod networking is enabled by **Network Address Translation** (**NAT**). The benefit of kubenet is that only nodes consume an IP address from the cluster subnet.
- **Azure Container Network Interface (CNI) networking (advanced)**: With Azure CNI, the pods and the nodes all get an IP address from the subnet that the cluster is created in. This has the benefit that pods can be accessed directly by resources outside the cluster. It has the disadvantage that you need to execute careful IP address planning, since each pod requires an IP address in the cluster subnet.

In both networking models, you can create the cluster in an existing VNet or have AKS create a new VNet on your behalf.

The second network security configuration to consider is the routing of inbound and outbound traffic through an external firewall. This could either be an Azure Firewall or a third-party **network virtual appliance** (**NVA**). By routing traffic through an external firewall, you can apply centralized security rules, do traffic inspection, and log traffic access patterns. To configure this, you would configure a **user-defined route** (**UDR**) on the cluster subnet, to route traffic from your cluster through the external firewall. If you wish to explore this further, please refer to the documentation: https://docs.microsoft.com/azure/aks/limit-egress-traffic.

Another network security option is the use of NSGs in Azure to limit inbound and outbound traffic. By default, when you create a service of the `LoadBalancer` type in AKS, AKS will also configure an NSG to allow traffic from everywhere to that service. You can tune the configuration of this NSG in AKS, to limit which IPs can access those services.

Finally, you can limit traffic in your cluster by using a Kubernetes feature called **network policies**. A network policy is a Kubernetes object that allows you to configure which traffic is allowed on certain pods. With network policies, you can secure pod-to-pod traffic, external to pod traffic, and pod to external traffic. It is recommended that you use network policies mainly for pod-to-pod traffic (also called east-west traffic), and to use an external firewall or NSGs for external-to-pod or pod-to-external traffic (also called north-south traffic).

AKS supports two options in terms of configuring network policies on your cluster. You can either use Azure network policies or Calico network policies. Azure network policies are developed, maintained, and supported by Microsoft, whereas Calico network policies are developed as an open-source project, with optional commercial support by a company called Tigera (http://tigera.io/).

In the section on workload network security, you will configure network security groups and network policies on your cluster. Configuring an external firewall is beyond the scope of this book; please refer to the documentation mentioned earlier to learn more about this setup.

Control plane network security

In this section, you will explore two ways in which to protect the control plane of your AKS cluster: Authorized IP ranges and private clusters. You'll start by updating your existing cluster to use authorized IP ranges.

Securing the control plane using authorized IP ranges

Configuring authorized IP ranges on AKS will limit which public IP addresses can access the control plane of your AKS cluster. In this section, you will configure authorized IP ranges on your existing cluster. You will limit traffic to a random public IP address to verify that traffic blocking works. You will then configure the IP address from the Azure Cloud Shell to be authorized and will see how that then allows traffic.

1. To start, browse to the Azure portal and open the pane for your AKS cluster. Select **Networking** in the left-hand navigation. Then, select the checkbox next to **Set authorized IP ranges**, and fill in the IP address, `10.0.0.0`, in the box below, as shown in *Figure 11.3*. You are not using this IP; this configuration is only to verify that you will no longer be able to connect to your Kubernetes control plane if your IP address is not authorized. Finally, hit the **Save** button at the top of the screen.

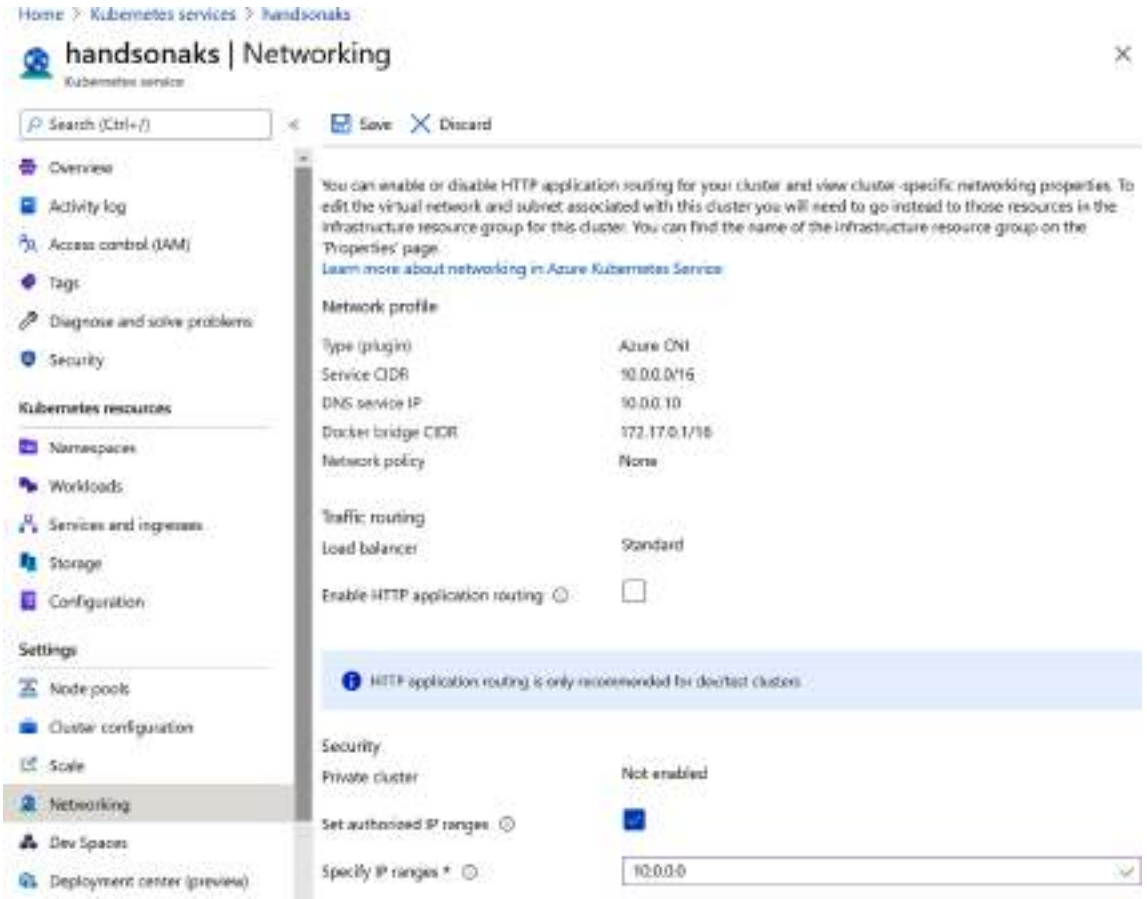


Figure 11.3: Configuring an authorized IP

2. Now, open the Azure Cloud Shell. In the Cloud Shell, execute the following command:

```
Watch kubectl get nodes
```

Initially, this might still return the list of nodes as shown in *Figure 11.4*. This is because it takes a couple of minutes for the authorized IP ranges to become configured on AKS.

```
Every 2.0s: kubectl get nodes       cc-9c4498d9-69cbc6fb69-n2kdc: Sat Feb  6 23:56:13 2021

NAME                                STATUS   ROLES   AGE    VERSION
aks-nodepool1-36910094-vmss000000   Ready    agent   7d3h   v1.18.14
aks-nodepool1-36910094-vmss000001   Ready    agent   7d3h   v1.18.14
```

Figure 11.4: The command might initially still show the list of nodes

After a couple of minutes, however, the output of this command should return an error, as shown in *Figure 11.5*. This is as expected, since you limited the access to the control plane.

```
Every 2.0s: kubectl get nodes       cc-9c4498d9-69cbc6fb69-n2kdc: Sat Feb  6 23:57:16 2021

Unable to connect to the server: dial tcp 20.59.49.243:443: i/o timeout
```

Figure 11.5: An error showing that you can no longer connect to the control plane

3. You can stop the `watch` command by pressing *Ctrl* + C. You will now get the IP address used by your current Cloud Shell session, and will then configure this as an authorized IP. To get the IP address used by your current Cloud Shell session, you can connect to icanhazip.com, which is a simple website that will return your public IP. To do this, execute the following command:

   ```
   curl icanhazip.com
   ```

   This will return an output similar to *Figure 11.6*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ curl icanhazip.com
40.78.15.47
```

Figure 11.6: Getting the IP address used by Cloud Shell

4. You can now configure this IP address as an authorized IP address in AKS. You can do this in the **Networking** section of the AKS pane as you did in *step 1*. This is shown in *Figure 11.7*. Make sure to click the **Save** button at the top of the screen.

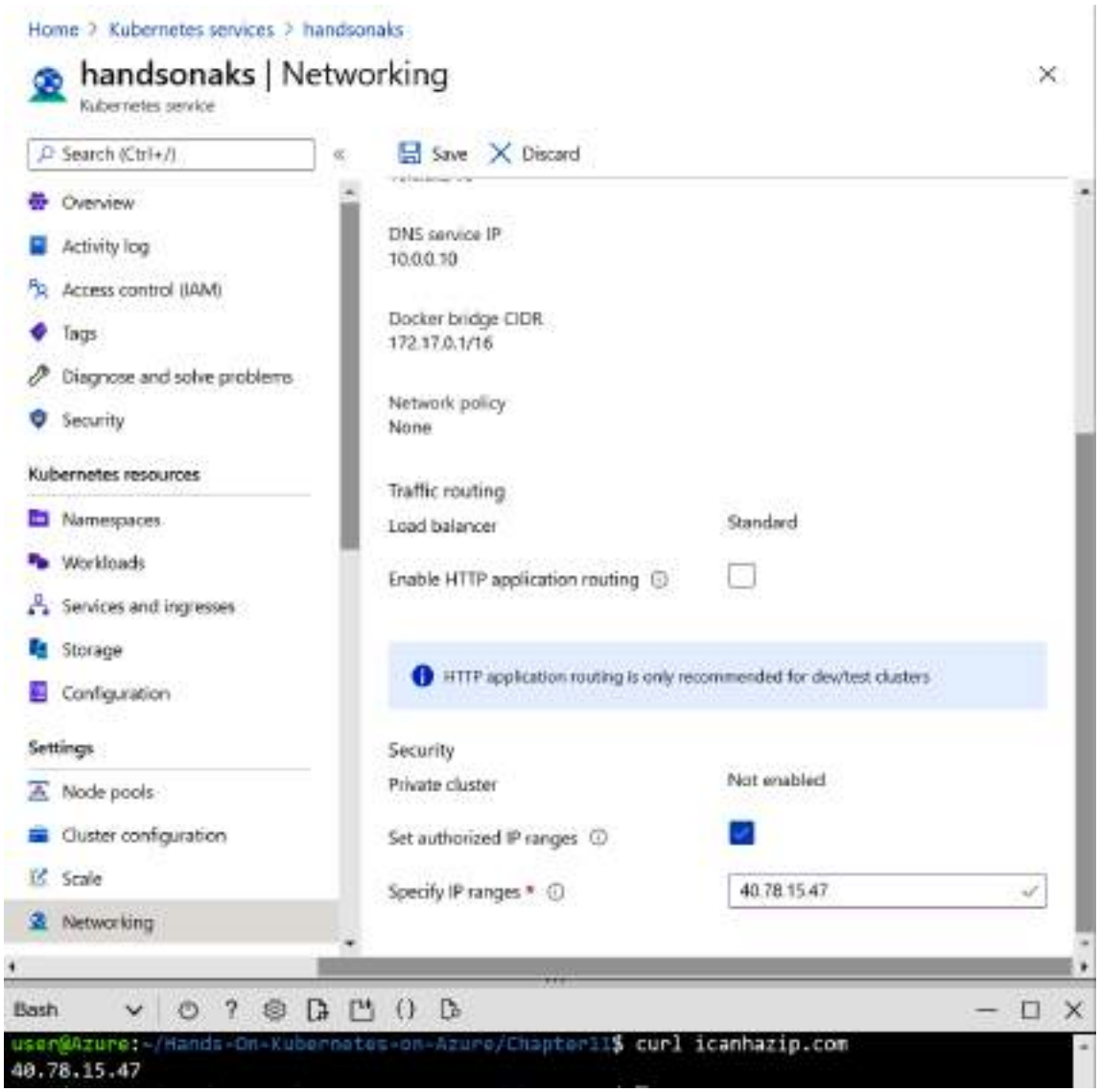


Figure 11.7: Configuring the IP address of Cloud Shell as an authorized IP in AKS

5. Now, execute the same command as before to get the list of nodes in your AKS cluster.

   ```
   watch kubectl get nodes
   ```

   Initially, this might still return the error you saw earlier, as shown in *Figure 11.8*. This is because it takes a couple of minutes for the authorized IP ranges to become configured on AKS.

```
Every 2.0s: kubectl get nodes       cc-9c4498d9-69cbc6fb69-n2kdc: Sun Feb  7 00:08:17 2021

Unable to connect to the server: dial tcp 20.59.49.243:443: i/o timeout
```

Figure 11.8: The command initially still gives an error

After a couple of minutes, however, the output of this command should return a list of nodes, as shown in *Figure 11.9*. This shows you that you successfully configured authorized IP ranges.

```
Every 2.0s: kubectl get nodes       cc-9c4498d9-69cbc6fb69-n2kdc: Sun Feb  7 00:13:04 2021

NAME                                STATUS   ROLES   AGE    VERSION
aks-nodepool1-36910094-vmss000000   Ready    agent   7d3h   v1.18.14
aks-nodepool1-36910094-vmss000001   Ready    agent   7d3h   v1.18.14
```

Figure 11.9: By configuring an authorized IP, you can now connect to the API server

By configuring authorized IP ranges, you were able to confirm that when the IP address of Cloud Shell was not allowed access to the Kubernetes control plane, the connection is timed out. By configuring the IP address of Cloud Shell as an authorized IP, you were able to connect to the control plane.

In a typical production scenario, you wouldn't configure IP addresses from Cloud Shell as the authorized IP on an AKS cluster, but you would rather configure well-known IP addresses or ranges of either of your Kubernetes administrators, your datacenter, or known IPs of tools you use. The Cloud Shell was used here just as an example to show functionality.

There is a second way to secure the control plane, that is, by deploying a private cluster. You will do this in the next section.

Securing the control plane using a private cluster

By configuring authorized IP ranges in AKS, you were able to limit which public IP addresses can access your cluster. You can also completely limit any public traffic to your cluster by deploying a private cluster. A private cluster is only reachable through a private connection, established using Azure Private Link.

Let's start by configuring a private cluster and trying to access it:

1. The private cluster feature can only be enabled at cluster creation time. This means that you will have to create a new cluster. To do this on the free trial subscription, you will have to delete the existing cluster. You can do this using the following command on Cloud Shell:

   ```
   az aks delete -n handsonaks -g rg-handsonaks -y
   ```

   This command will take a couple of minutes to complete. Please wait for it to successfully delete your previous cluster.

2. You are now ready to create a new cluster. Because you will, in later steps, also create a new VM to access the cluster (as shown in *Figure 11.2*), you will create a new VNet instead of letting AKS create the VNet for you. To create the VNet, use the following command:

   ```
   az network vnet create -o table \
       --resource-group rg-handsonaks \
       --name vnet-handsonaks \
       --address-prefixes 192.168.0.0/16 \
       --subnet-name akssubnet \
       --subnet-prefix 192.168.0.0/24
   ```

3. You will require the ID of the subnet that was created in the VNet. To get that ID, use the following command:

   ```
   VNET_SUBNET_ID='az network vnet subnet show \
     --resource-group rg-handsonaks \
     --vnet-name vnet-handsonaks \
     --name akssubnet --query id -o tsv'
   ```

4. You will also need a managed identity that has permission to create resources in the subnet you just created. To create the managed identity and give it access to your subnet, use the following commands:

   ```
   az identity create --name handsonaks-mi \
     --resource-group rg-handsonaks
   IDENTITY_CLIENTID='az identity show --name handsonaks-mi \
     --resource-group rg-handsonaks \
     --query clientId -o tsv'
   ```

```
az role assignment create --assignee $IDENTITY_CLIENTID \
  --scope $VNET_SUBNET_ID --role Contributor
IDENTITY_ID='az identity show --name handsonaks-mi \
  --resource-group rg-handsonaks \
  --query id -o tsv'
```

The preceding code will first create the managed identity. Afterward, it gets the client ID of the managed identity and grants that access to the subnet. In the final command, it is getting the resource ID of the managed identity.

5. Finally, you can go ahead and create the private AKS cluster using the following command. As you might notice, you are creating a smaller cluster using only one node. This is to conserve the core quota under the free trial subscription:

```
az aks create \
  --resource-group rg-handsonaks \
  --name handsonaks \
  --vnet-subnet-id $VNET_SUBNET_ID \
  --enable-managed-identity \
  --assign-identity $IDENTITY_ID \
  --enable-private-cluster \
  --node-count 1 \
  --node-vm-size Standard_DS2_v2 \
  --generate-ssh-keys
```

The command creates a new AKS cluster with a number of special configurations that haven't been covered previously in the book. The first new configuration is `--vnet-subnet-id`. This allows you to create an AKS cluster in an existing subnet in an existing VNet. The `--enable-managed-identity` parameter enables the cluster to use a managed identity, and the `--assign-identity` parameter configures which managed identity to use. The final new configuration option you see here is `--enable-private-cluster`, which will create a private cluster with a private endpoint.

6. The preceding command will take a couple of minutes to complete. Once it's complete, you can try to access your cluster using the Azure Cloud Shell. This will fail, however, because the Azure Cloud Shell isn't deployed in your VNet. Let's explore this. First, get the cluster credentials:

```
az aks get-credentials -n handsonaks -g rg-handsonaks
```

This will ask you whether it may overwrite the existing `kubeconfig` files twice. Confirm this by pressing the *y* key, as shown in *Figure 11.10*:

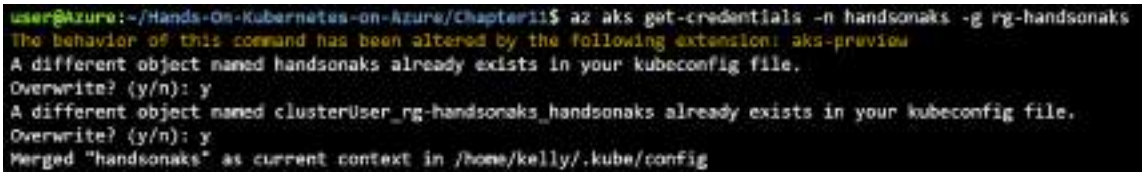


Figure 11.10: Getting cluster credentials

Now, try to get the nodes in the cluster with the following command:

```
kubectl get nodes
```

This will return an error, as shown in *Figure 11.11*. This error is as expected, since you have no private connection from Cloud Shell to the Private Link endpoint.

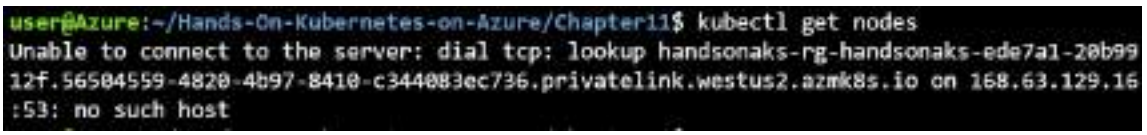


Figure 11.11: Error showing that you can no longer access the control plane from the Cloud Shell

> **Note**
>
> In the previous command, you noticed that your Cloud Shell couldn't reach the Kubernetes API server. It is possible to connect Azure Cloud Shell to a VNet in Azure and connect to your Kubernetes API server that way. You will not do so in the next steps of this example, but if you are interested in this approach, please refer to the documentation: https://docs.microsoft.com/azure/cloud-shell/private-vnet.

7. As mentioned in the introduction, when you create a private AKS cluster, AKS will use a service called Private Link to connect the control plane to your VNet. You can actually see this endpoint in your subscription in the Azure portal. To see the private endpoint, look for **Private Link** in the Azure search bar, as shown in *Figure 11.12*:

Figure 11.12: Searching for Private Link in the Azure search bar

In the resulting pane, click on **Private endpoints** to see your current Private Link endpoints. You should see a private endpoint by the name of **kube-apiserver** here, as shown in *Figure 11.13*. Here you see the link to the cluster and to the subnet where the private endpoint is created.

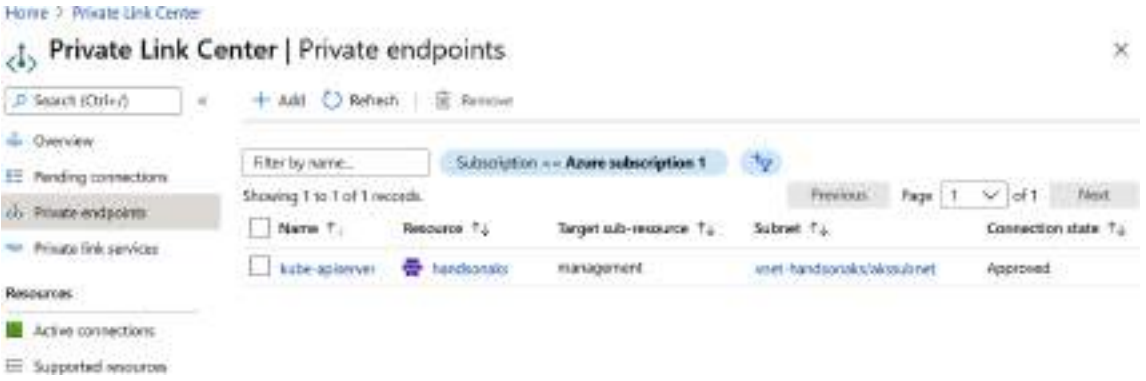


Figure 11.13: The private endpoints in your subscription

Private Link makes use of an Azure DNS private zone to link the DNS name of the cluster to the private IP of the private endpoint. To see the Azure DNS private zone, look for **Private DNS zones** via the Azure search bar, as shown in *Figure 11.14*:

Figure 11.14: Navigating to Private DNS zones through the Azure portal

In the resulting pane, you should see one private DNS zone. If you click on that zone, you will see more details from the DNS zone, as shown in *Figure 11.15*. You see here that a DNS record got created for your cluster DNS name, pointing to a private IP address in your VNet.

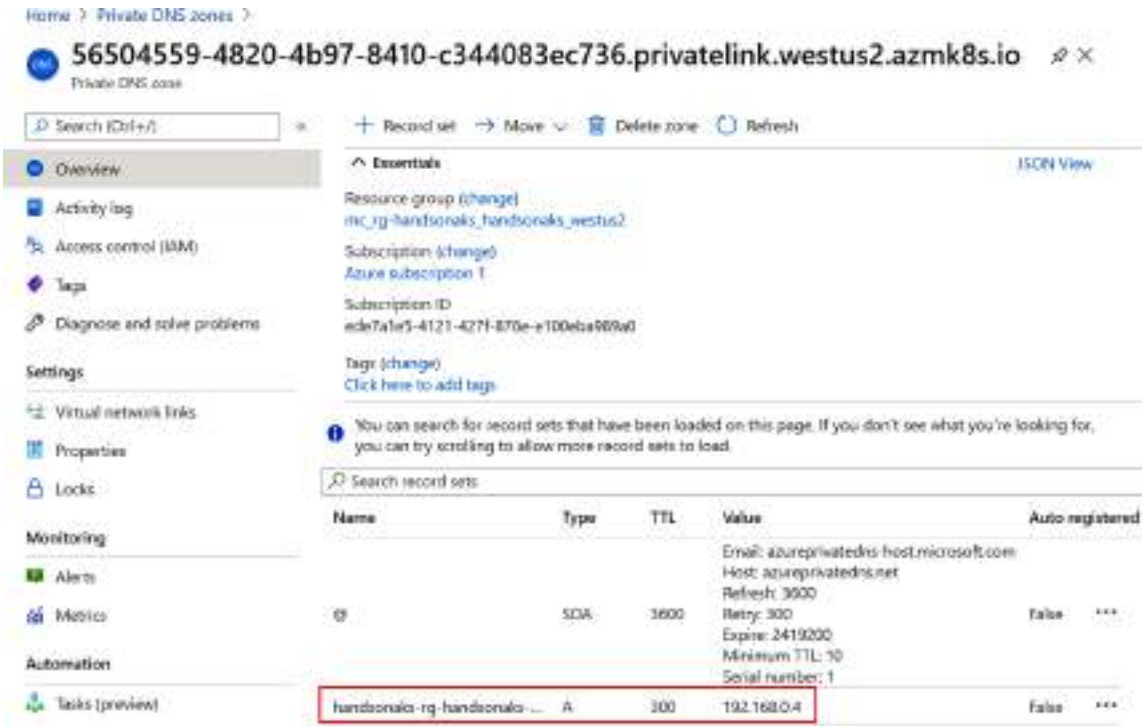


Figure 11.15: The DNS record in the Azure DNS private zone that got created by AKS

8. To establish a private connection to the control plane, you will now create a new VM and use it to connect to the control plane. For organization purposes, you'll create this VM in a new resource group. This will make it easier to delete the VM later. Use the following commands to create a new subnet in your VNet and to create a VM in that subnet:

```
az network vnet subnet create \
  --resource-group rg-handsonaks \
  --vnet-name vnet-handsonaks \
  --name vmsubnet \
  --address-prefix 192.168.1.0/24
VM_SUBNET_ID='az network vnet subnet show \
  --resource-group rg-handsonaks \
```

```
    --vnet-name vnet-handsonaks \
    --name vmsubnet --query id -o tsv'
az group create -l <your Azure location> \
    --name rg-handsonaks-vm
az vm create --name vm-handsonaks \
    --resource-group rg-handsonaks-vm \
    --image UbuntuLTS \
    --admin-username azureuser \
    --ssh-key-values ~/.ssh/id_rsa.pub \
    --subnet $VM_SUBNET_ID \
    --size Standard_D2_v2
```

It will take about a minute for the VM to be created. Once it is created, you should get an output similar to *Figure 11.16*. Copy the public IP address in your output:

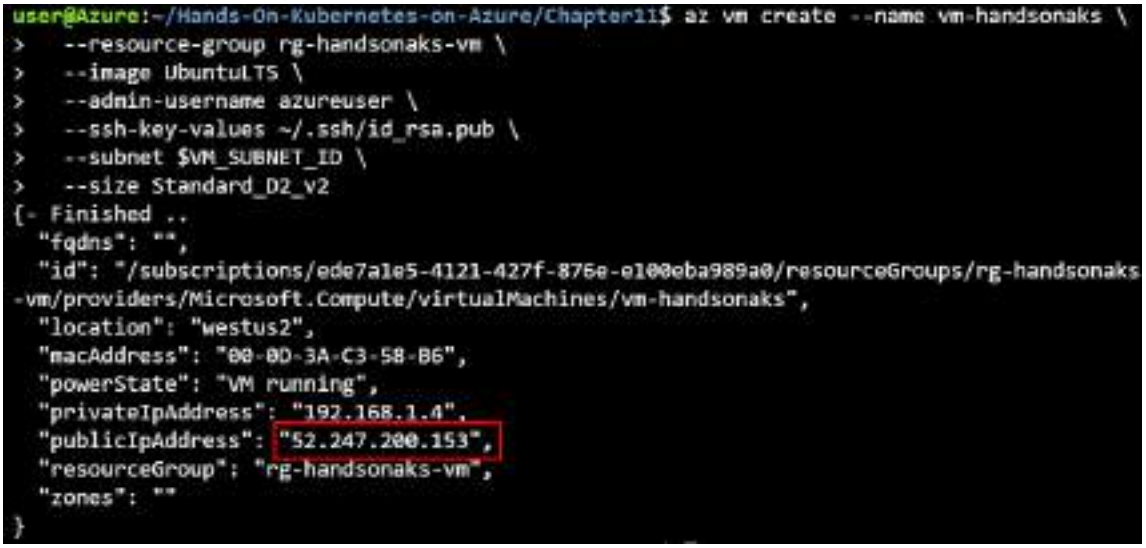


Figure 11.16: Creating a new VM and getting its public IP address

9. Now that the VM is created, you will move your Kubernetes config file containing the cluster credentials to that VM. This avoids you having to install the Azure CLI on the target machine to get the Kubernetes credentials. Make sure to replace <public IP> with the outcome from the previous step.

```
scp ~/.kube/config azureuser@<public IP>:~
```

You will be prompted if you trust this host. Confirm this by typing `yes`. This will create an output similar to *Figure 11.17*:

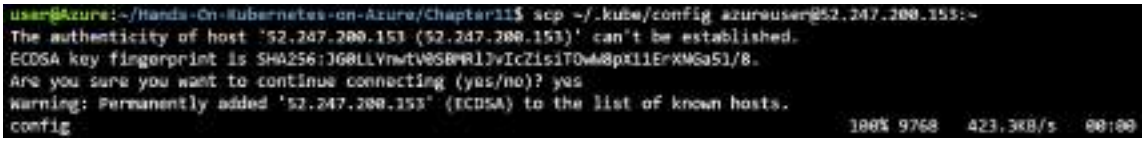


Figure 11.17: Copying the Kubernetes credentials to the target machine

10. You can now access the remote machine using the following command:

    ```
    ssh azureuser@<public IP>
    ```

11. Now that you're connected to the remote machine, you'll need to use `kubectl`. Download it, make it executable, and move it into the binaries folder using the following command:

    ```
    curl -LO https://dl.k8s.io/release/v1.20.0/bin/linux/amd64/kubectl
    chmod +x kubectl
    sudo mv ./kubectl /usr/local/bin/kubectl
    ```

12. To have `kubectl` recognize the config file you uploaded, you have to move it into the kube directory. You can do so using the following command:

    ```
    mkdir .kube
    mv config .kube/config
    ```

13. Now that you have this VM configured to connect to your cluster, you can verify that you can connect to this cluster by applying the following command:

    ```
    kubectl get nodes
    ```

    This should generate an output similar to *Figure 11.18*:

Figure 11.18: Accessing the private AKS cluster from a VM in the same VNet

14. You can also verify the DNS record that your VM is using to connect to the cluster. To do this, first get the **fully qualified domain name** (**FQDN**) cluster (refer to the highlighted section in *Figure 11.19* to see which output is the FQDN) and then use the `nslookup` command to get the DNS record. You can use the following commands to do this:

    ```
    kubectl cluster-info
    nslookup <cluster FQDN>
    ```

    This should produce an output similar to *Figure 11.19*:

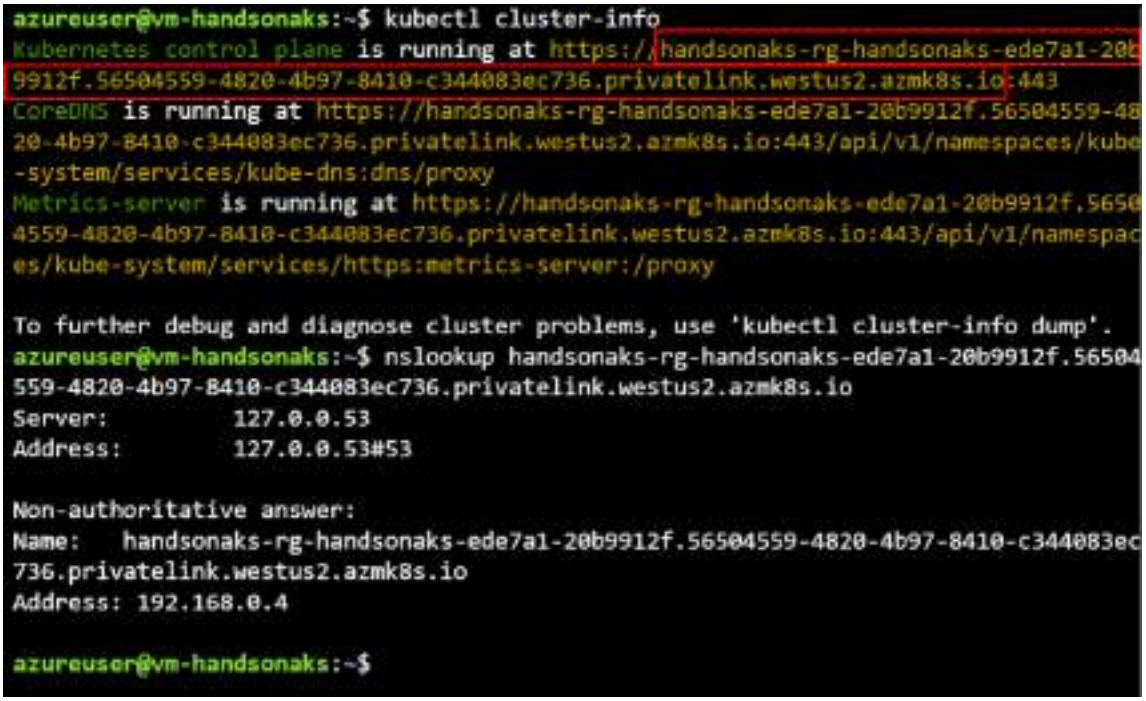


Figure 11.19: Getting the cluster's FQDN and looking up its IP address using nslookup

As you can see in *Figure 11.19*, the address that you are getting back from the `nslookup` command is a private IP address. This means that only machines connected to that VNet will be able to connect to the Kubernetes control plane.

You have now successfully created an AKS private cluster and verified that only machines connected to the AKS VNet can connect to the control plane. You couldn't connect to the control plane from within the Azure Cloud Shell, but you could connect to it from a VM in the same VNet. Since you now have a private cluster deployed, don't delete the VM you are using just yet. You will use it in the next example. You will delete this private cluster and the VM in the final example in this chapter.

This also concludes this section on control plane security. You have learned about authorized IP ranges and private clusters. In the next section, you'll learn more about how you can secure your workload.

Workload network security

You have now learned about how to protect the network of your control plane of your AKS cluster. This, however, hasn't influenced the network security of your workloads. In this section, you will explore three ways in which you can protect your workloads. First, you will create a Kubernetes service using an Azure internal load balancer. Then, you'll secure traffic to a service in Kubernetes using NSGs. Finally, you will use network policies to secure pod-to-pod traffic.

Securing the workload network using an internal load balancer

Kubernetes has multiple types of services, as you learned in *Chapter 3*, *Application Deployment on AKS*. You have used the service type load balancer multiple times before to have AKS create an Azure load balancer. These have always been public load balancers. You can also configure AKS in such a way that it will create an internal load balancer. This is useful in cases where you are creating a service that only needs to be accessible from within a VNet or networks connected to that VNet.

You will create such a service in this section:

1. If you are no longer connected to the VM you created in the previous example, reconnect to it. You can get the VM's public IP address using the following command:

   ```
   az vm show -n vm-handsonaks \
     -g rg-handsonaks-vm -d --query publicIps
   ```

   And you can connect to the VM using the following command:

   ```
   ssh azureuser@<public IP>
   ```

2. Once connected to this VM, you will need to retrieve the git repository linked with this book. You can get this using the following command:

   ```
   git clone https://github.com/PacktPublishing/Hands-on-Kubernetes-on-Azure-Third-Edition
   ```

   Once the repository is cloned, navigate into the samples for this chapter using the following command:

   ```
   cd Hands-On-Kubernetes-on-Azure-Third-Edition/Chapter11
   ```

3. As the example application in this section, you will use the guestbook application you've already used in the first half of this book. However, the all-in-one YAML file you used before has been broken up into two files: `guestbook-without-service.yaml` and `front-end-service-internal.yaml`. The reason for this is to make it easier for you to explore the service-specific configuration.

The `front-end-service-internal.yaml` file contains the configuration to create a Kubernetes service using an Azure internal load balancer. The following code is part of that example:

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: frontend
5     annotations:
6       service.beta.kubernetes.io/azure-load-balancer-internal:
"true"
7     labels:
8       app: guestbook
9       tier: frontend
10  spec:
11    type: LoadBalancer
12    ports:
13    - port: 80
14    selector:
15      app: guestbook
16      tier: frontend
```

You are using annotations in the YAML code to instruct AKS to create an Azure internal load balancer. You can see on lines 5-6 of the preceding code example that you are setting the `service.beta.kubernetes.io/azure-load-balancer-internal` annotation to `true`.

You can create the guestbook application and the service using the internal load balancer by applying the following commands:

```
kubectl create -f guestbook-without-service.yaml
kubectl create -f front-end-service-internal.yaml
```

You can then get the service and wait for it to get an external IP using the following command:

```
kubectl get service -w
```

This will return an output similar to *Figure 11.20*:

```
azureuser@vm-handsonaks:~/Hands-on-Kubernetes-on-Azure/Chapter11$ kubectl get service -w
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
frontend     LoadBalancer   10.0.131.115   <pending>     80:31416/TCP   3s
kubernetes   ClusterIP      10.0.0.1       <none>        443/TCP        56s
frontend     LoadBalancer   10.0.131.115   10.240.0.35   80:31416/TCP   80s
```

Figure 11.20: Getting the service's external IP

4. As you can see, the service has a private IP as an external IP. You can only access this IP from the VNet that the cluster is deployed into, or from other networks connected to that VNet.

> **Note**
>
> You may ask yourself the question: "Each service gets a cluster IP as well, which is a private IP. Why can't that be used instead of the internal load balancer?"
> The answer to that question is that a cluster IP is only reachable from within the cluster, not from outside the cluster. You can, however, create services of the NodePort type to make a service exposed to calls from outside the cluster. This would expose the service on the IP of the node, on a certain port. The downside of NodePort services is that they expose the service on the same port on each node, so you can't expose two services on the same port in your cluster. The internal private load balancer does have the ability to expose the same port on multiple services for the same cluster.

You can try accessing the service using the following command:

```
curl <external IP>
```

This will return a result similar to *Figure 11.21*:

```
azureuser@vm-handsonaks:~/Hands-on-Kubernetes-on-Azure/Chapter11$ curl 10.240.0.35
<html ng-app="redis">
  <head>
    <title>Guestbook</title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css">
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.12/angular.min.js"></script>
    <script src="controllers.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/0.11.0/ui-bootstrap-tpls.js"></script>
  </head>
  <body ng-controller="RedisCtrl">
    <div style="width: 50%; margin-left: 20px">
      <h2>Guestbook</h2>
    <form>
    <fieldset>
    <input ng-model="msg" placeholder="Messages" class="form-control" type="text" name="input"><br>
    <button type="button" class="btn btn-primary" ng-click="controller.onRedis()">Submit</button>
    </fieldset>
    </form>
    <div>
      <div ng-repeat="msg in messages track by $index">
        {{msg}}
      </div>
    </div>
    </div>
  </body>
</html>
```

Figure 11.21: Accessing the service exposed through the internal load balancer

5. AKS created an internal load balancer to expose this service. You can see this load balancer in the Azure portal as well. To see this internal load balancer, start by searching for **load balancer** in the Azure search bar, as shown in *Figure 11.22*:

Figure 11.22: Navigating to Load balancers through the Azure portal

6. In the resulting pane, you should see two load balancers, as shown in *Figure 11.23*:

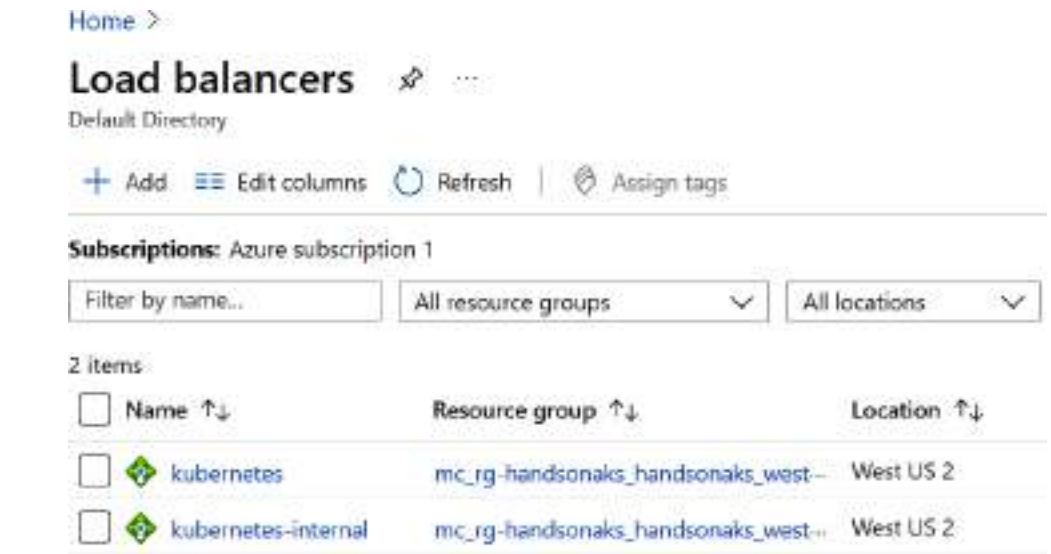


Figure 11.23: List of load balancers in the default directory

7. Click on the **kubernetes-internal** load balancer. This will take you to a pane similar to *Figure 11.24*:

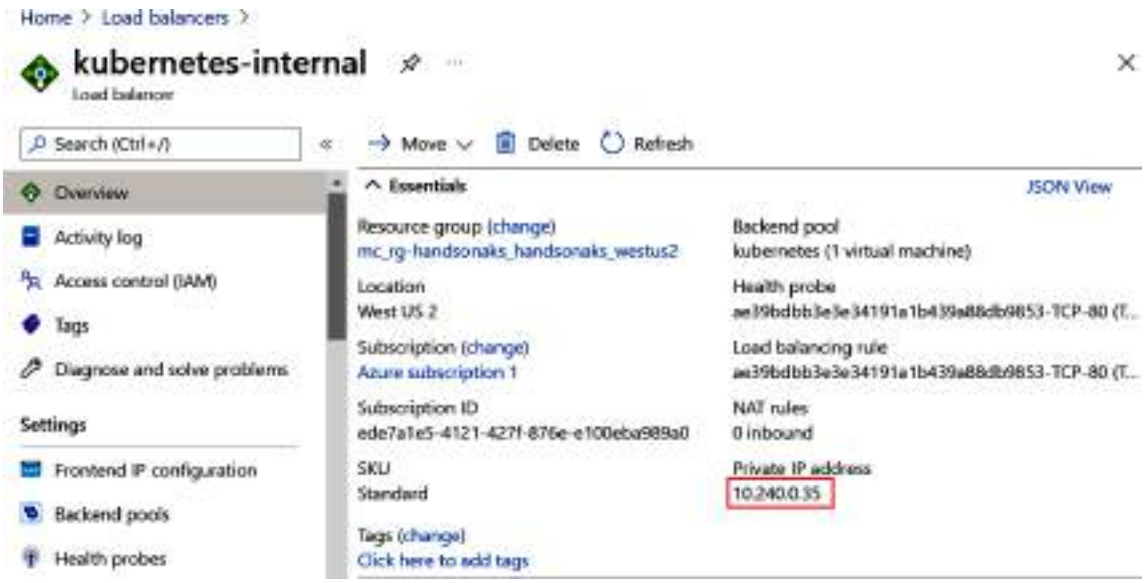


Figure 11.24: Details of the internal load balancer

Here, you can see the details of this internal load balancer. As you can see in the highlight in the screenshot, the same IP that you saw as the output of the `kubectl` command is configured on the load balancer.

8. This concludes the example of using an internal load balancer. You can now delete the service using the internal load balancer by applying the following command:

```
kubectl delete -f front-end-service-internal.yaml
kubectl delete -f guestbook-without-service.yaml
```

This will delete the guestbook application and the service. When deleting the service, both the service in Kubernetes, as well as the internal load balancer in Azure, will be deleted. This is because once there are no more services in your cluster requiring an internal load balancer, AKS will delete that internal load balancer.

In this section, you deployed a Kubernetes service using an internal load balancer. This gives you the ability to create services that are not exposed to the internet. There are, however, cases where you need to expose a service to the internet, but need to ensure that only trusted parties can connect to it. In the next section, you'll learn how you can create a service in AKS that uses a network security group to limit inbound traffic.

Securing the workload network using network security groups

Up to this point in the book, you have exposed multiple services in Kubernetes. You've exposed them both using the service object in Kubernetes, as well as using an ingress. However, you never restricted access to your application, except in the previous section, by deploying an internal load balancer. This means that the application was always publicly accessible. In the following example, you will create a service on your Kubernetes cluster that will have a public IP, but you will restrict access to it using an NSG that is configured by AKS.

1. As the example application in this section, you will again use the guestbook application. As in the previous section, the front-end service configuration has been moved to a separate file. For this example, you'll start by using the `front-end-service.yaml` file to create the service, and later update that using the `front-end-service-secured.yaml` file.

Let's start by deploying the application as-is, without any NSG configuration, by applying the following command:

```
kubectl apply -f guestbook-without-service.yaml
kubectl apply -f front-end-service.yaml
```

Then, get the front-end service's IP address using the following command:

```
kubectl get service -w
```

This will create an output similar to *Figure 11.25*. Once you get the public IP, you can exit out of the command by pressing *Ctrl + C*:

```
azureuser@vm-handsonaks:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl get service -w
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
frontend       LoadBalancer   10.0.102.40    <pending>     80:32508/TCP   4s
kubernetes     ClusterIP      10.0.0.1       <none>        443/TCP        157m
redis-master   ClusterIP      10.0.104.61    <none>        6379/TCP       76s
redis-replica  ClusterIP      10.0.197.130   <none>        6379/TCP       76s
frontend       LoadBalancer   10.0.102.40    40.64.65.202  80:32508/TCP   13s
```

Figure 11.25: Getting the front-end service's IP address

You are now able to connect to this service using both your browser as well as using the VM itself. If you connect using your browser, you should expect an output similar to *Figure 11.26*:

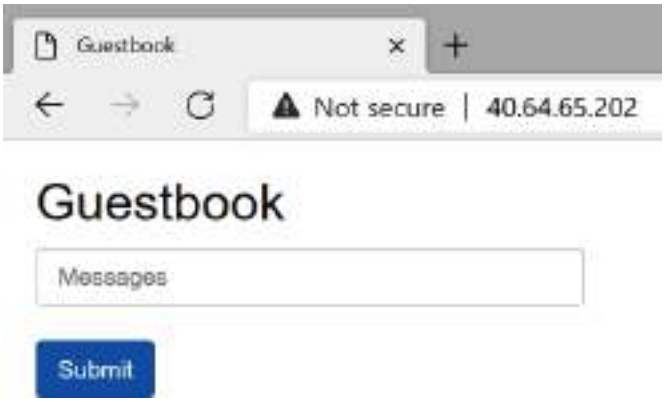


Figure 11.26: Accessing the guestbook application through a web browser

2. You can also connect to this application using the command line. To do this, use the following command:

```
curl <public IP>
```

   This should return an output similar to *Figure 11.27*:

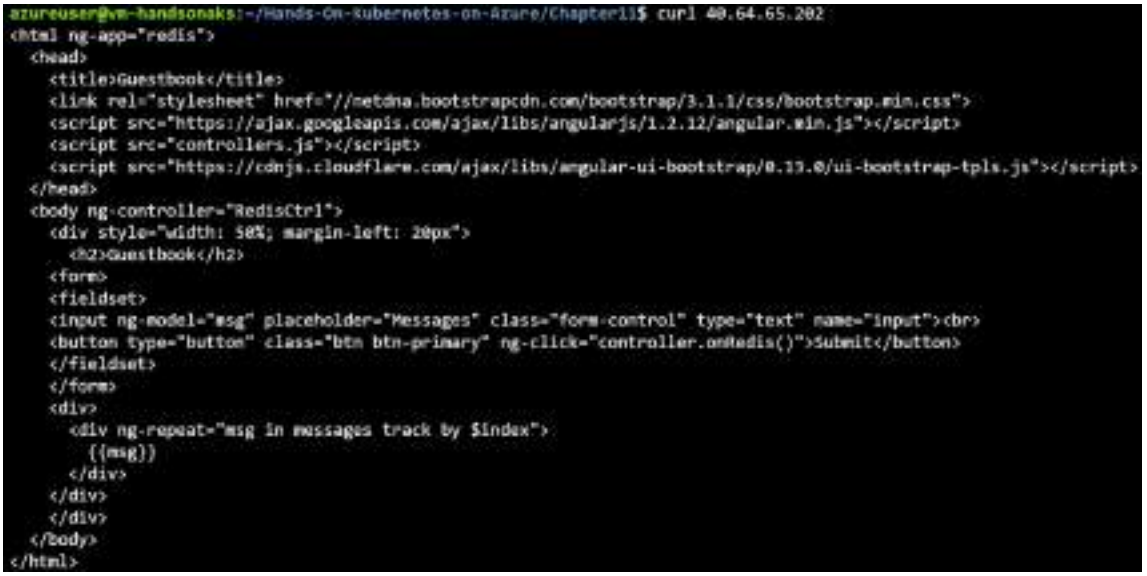


Figure 11.27: Connecting to the guestbook application using the command line

3. Let's now configure additional security on the front-end service by only allowing your browser to connect to the application. For this, you will require the public IP address you are using right now. If you don't know this, you can browse to https://www.whatismyip.com/ to get your IP address, as shown in *Figure 11.28*:

Figure 11.28: Getting your own public IP address

To secure the front-end service, you will edit the `front-end-service-secured.yaml` file. This is the code in that particular file:

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: frontend
5     labels:
6       app: guestbook
7       tier: frontend
8   spec:
9     type: LoadBalancer
10    ports:
11    - port: 80
12    selector:
13      app: guestbook
14      tier: frontend
15    loadBalancerSourceRanges:
16    - <your public IP address>
```

This file is very similar to the services you created earlier in this book. However, on lines 15 and 16, you now see `loadBalancerSourceRanges` and the option to add your own public IP address. You can provide multiple public IP addresses or ranges here; each address or range would be prepended with a dash. If you wish to enter an individual IP address rather than a range, append `/32` to that IP address. You need to do this since Kubernetes expects IP ranges, and a range of `/32` equals a single IP address.

To edit or add your own IP address in this file, use the following command:

```
vi front-end-service-secured.yaml
```

In the resulting application, use the arrow keys to navigate to the bottom line, hit the *i* button to enter `insert` mode, remove the placeholder, add in your IP address, and then append that with `/32`. To close and save the file, hit the *Esc* key, type `:wq!` to write and close the file, and finally hit *Enter*. An example is shown in *Figure 11.29*:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: guestbook
    tier: frontend
  loadBalancerSourceRanges:
  - 108.85.111.151/32
```

Figure 11.29: An example of the front-end-service-secured.yaml file with an IP address

4. You can update the exiting service that was deployed before using the following command:

```
kubectl apply -f front-end-service-secured.yaml
```

   This will cause AKS to update the NSG linked to this cluster to only allow traffic from your public IP address. You can confirm this by browsing to the IP address of the service again, and you should see the guestbook application. However, if you retry the command from earlier from the VM, you should see it eventually time out:

```
curl <public IP>
```

   This will time out after 2 minutes, with an output similar to *Figure 11.30*:

```
azureuser@vm-handsonaks:~/Hands-On-Kubernetes-on-Azure/Chapter11$ curl 40.64.65.202
curl: (7) Failed to connect to 40.64.65.202 port 80: Connection timed out
```

Figure 11.30: The connection from within the VM times out

5. You can verify the NSG configuration in Azure itself as well. To verify this, look for **Network security groups** via the Azure search bar, as shown in *Figure 11.31*:

Figure 11.31: Navigating to Network security groups through the Azure portal

6. In the resulting pane, you should see two NSGs. Select the one whose name starts with **aks-agentpool**, as shown in *Figure 11.32*:

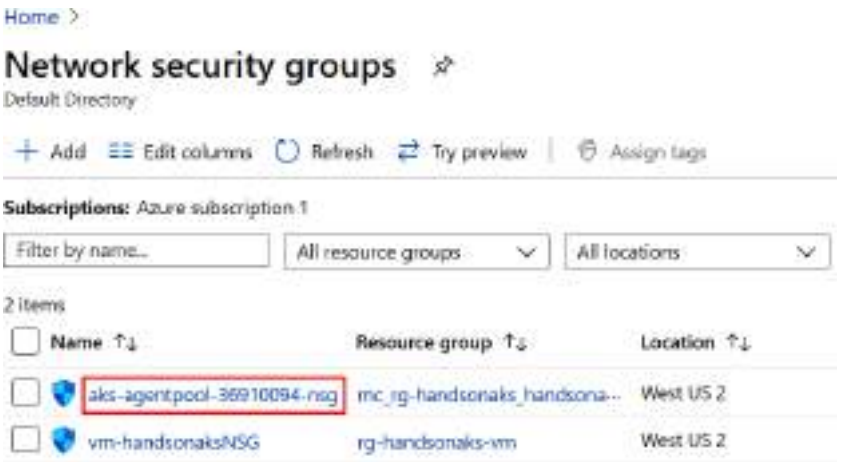


Figure 11.32: Selecting the aks-agentpool NSG

7. In the resulting detailed view of that NSG, you should see a rule that allows traffic from your IP address to the service's public IP address, as you can see in *Figure 11.33*:

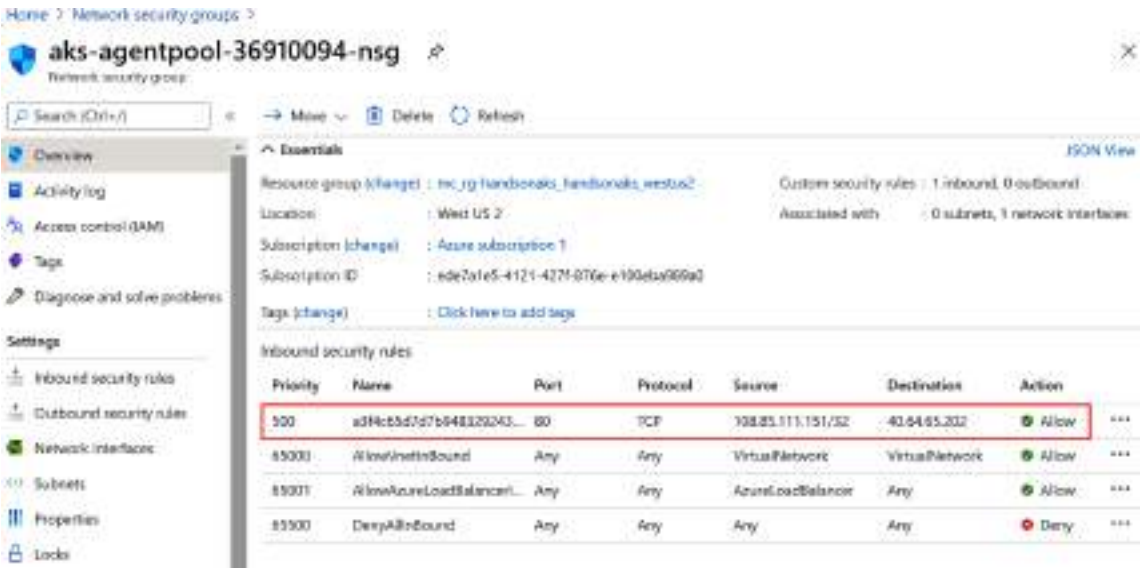


Figure 11.33: The NSG contains a rule that allows traffic only from the public IP defined earlier

Notice how this rule was created and is managed by AKS; you didn't have to create this yourself.

8. Here, we've concluded this example. Let's clean up the deployment, the VM, and the private cluster. From within the VM, delete the application using the following command:

```
kubectl delete -f guestbook-without-service.yaml
kubectl delete -f front-end-service-secured.yaml
```

   Then, exit out of the VM using the `exit` command. This will return you to Cloud Shell. Here, you can go ahead and delete the private cluster and the VM you used to connect to it:

```
az group delete -n rg-handsonaks-vm -y
az aks delete -g rg-handsonaks -n handsonaks -y
```

By adding additional configuration to a Kubernetes service, you were able to limit who was able to connect to your service. You were able to confirm that only the public IP that was allowed to connect to the service was able to connect to it. A connection not coming from this public IP address timed out.

This is an example of protecting what is called north-south traffic, meaning traffic coming from the outside to your cluster. You can also add additional protections to east-west traffic, meaning traffic inside your cluster. To do this, you will use a feature called network policies in Kubernetes. You will do that in the next section.

Securing the workload network using network policies

In the previous section, you let Kubernetes configure an NSG in Azure to protect north-south traffic. This is a good practice for limiting the network traffic coming to your public services. In most scenarios, you will also need to protect the east-west traffic, meaning the traffic between your pods. That way, you can ensure that if a potential attacker were to get access to part of your application, they'd have limited ability to connect to other parts of the application or different applications. This is also known as protecting from lateral movement.

To protect the traffic between pods, Kubernetes has a functionality called network policies. Network policies can be used to protect traffic from the outside to your pods, and from your pods to the outside, as well as traffic between pods. Since you have already learned about one way to protect traffic from the outside to your pods, in this section, you will learn how to use network policies to protect pod-to-pod traffic.

In AKS, network policies are something you need to configure on your cluster at cluster creation time (it is this way at the time of this writing). If your cluster has network policies enabled, you can create new network policy objects on your cluster. When there is no network policy selecting a certain pod, all traffic to and from that pod is allowed. When you apply a network policy to a pod, depending on the configuration, all traffic to and/or from that pod is blocked, except for the traffic allowed by that network policy.

Let's try this out:

1. Start by creating a new cluster with network policies enabled. In this example, you'll create a cluster with Azure network policies enabled. You can create this new cluster using the following command:

```
az aks create \
  --resource-group rg-handsonaks \
  --name handsonaks \
  --enable-managed-identity \
  --node-count 2 \
  --node-vm-size Standard_DS2_v2 \
  --generate-ssh-keys \
  --network-plugin azure \
  --network-policy azure
```

2. Once the cluster is created, make sure to refresh the credentials to get access to the cluster. You can do this using the following command:

```
az aks get-credentials -g rg-handsonaks -n handsonaks
```

   This will prompt you to overwrite the existing credentials. You can confirm this by typing *y* in the two prompts, as shown in *Figure 11.34*:

Figure 11.34: Getting credentials for the new cluster

3. For this example, you will test connections between two web servers in a pod running `nginx`. The code for these has been provided in the `web-server-a.yaml` and `web-server-b.yaml` files. This is the code for `web-server-a.yaml`:

```
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: web-server-a
5     labels:
6       app: web-server
7       env: A
8   spec:
9     containers:
10    - name: webserver
11      image: nginx:1.19.6-alpine
```

   This is the code for `web-server-b.yaml`:

```
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: web-server-b
5     labels:
6       app: web-server
7       env: B
8   spec:
9     containers:
10    - name: webserver
11      image: nginx:1.19.6-alpine
```

   As you can see in the code for each of the pods, each pod has a label app, `web-server`, and another label called `env`, with the value of each server (A for `web-server-a` and B for `web-server-b`). You will use these labels later in this example to selectively allow traffic between these pods.

   To create both pods, use the following command:

```
kubectl create -f web-server-a.yaml
kubectl create -f web-server-b.yaml
```

Verify that the pods are running before moving forward by running the following command:

```
kubectl get pods -w
```

This should return an output similar to *Figure 11.35*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl get pods -w
NAME           READY   STATUS              RESTARTS   AGE
web-server-a   0/1     Pending             0          0s
web-server-a   0/1     Pending             0          0s
web-server-a   0/1     ContainerCreating   0          0s
web-server-b   0/1     Pending             0          0s
web-server-b   0/1     Pending             0          0s
web-server-b   0/1     ContainerCreating   0          0s
web-server-b   1/1     Running             0          1s
web-server-a   1/1     Running             0          2s
```

Figure 11.35: Both pods are running

4. For this example, we'll use the pod's IP addresses to test the connection. Get the IP address for `web-server-b` using the following command:

   ```
   kubectl get pods -o wide
   ```

   This should return an output similar to *Figure 11.36*, in which you'll see the IP address highlighted:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE    IP            NODE
web-server-a   1/1     Running   0          111s   10.240.0.17   aks-nodepool1-36910094-vmss000000
web-server-b   1/1     Running   0          110s   10.240.0.42   aks-nodepool1-36910094-vmss000001
```

Figure 11.36: Getting the IP address of web-server-b

5. Now, try to connect from `web-server-a` to `web-server-b`. You can test this connection using the following command:

   ```
   kubectl exec -it web-server-a -- \
     wget -qO- -T 2 http://<web-server-b IP>
   ```

This should return an output similar to *Figure 11.37*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl exec -it web-server-a -- \
>   wget -qO- -T 2 http://10.240.0.42
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Figure 11.37: Verifying that web-server-a can connect to web-server-b

6. Let's now create a new `NetworkPolicy` object that will limit all traffic to and from the pods with the label app `web-server`. This policy has been provided in the `deny-all.yaml` file:

```
1   kind: NetworkPolicy
2   apiVersion: networking.k8s.io/v1
3   metadata:
4     name: deny-all
5   spec:
6     podSelector:
7       matchLabels:
8         app: web-server
9     ingress: []
10    egress: []
```

Let's explore what's contained in this code:

- **Line 1**: Here, you define that you are creating a `NetworkPolicy` object.
- **Lines 6-8**: Here, you define which pods this network policy will apply to. In this case, you are applying this network policy to all pods that have the label `app: web-server`.
- **Lines 9-10**: Here, you define the allow rules. As you can see, you are not defining any allow rules, which will mean that all traffic will be blocked.

Later in this example, you will add more specific ingress and egress rules to selectively allow traffic to flow.

7. Let's now create this network policy. You can do this using the following command:

```
kubectl create -f deny-all.yaml
```

This will return an output similar to *Figure 11.38*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl create -f deny-all.yaml
networkpolicy.networking.k8s.io/deny-all created
```

Figure 11.38: Creating the network policy

8. Let's now test the connection between `web-server-a` and `web-server-b` again. You can test this using the following command.

```
kubectl exec -it web-server-a -- \
  wget -qO- -T 2 http://<web-server-b IP>
```

This should return an output similar to *Figure 11.39*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl exec -it web-server-a -- \
>   wget -qO- -T 2 http://10.240.0.42
wget: download timed out
command terminated with exit code 1
```

Figure 11.39: Traffic is no longer flowing between web-server-a and web-server-b

9. You will now create another network policy that will selectively allow traffic from `web-server-a` to `web-server-b`. This policy is included in the `allow-a-to-b.yaml` file:

```
1   kind: NetworkPolicy
2   apiVersion: networking.k8s.io/v1
3   metadata:
4     name: allow-a-to-b
5   spec:
6     podSelector:
7       matchLabels:
8         app: web-server
9     ingress:
10    - from:
11      - podSelector:
12          matchLabels:
13            env: A
14    egress:
15    - to:
16      - podSelector:
17          matchLabels:
18            env: B
```

Let's explore the difference in this file versus the earlier network policy in more depth:

- **Lines 9-13**: Here, you are defining which ingress traffic is allowed. Specifically, you are allowing traffic from pods with the label `env: A`.
- **Lines 14-18**: Here, you are defining which egress traffic is allowed. In this case, you are allowing egress traffic to pods with the label `env: B`.

Also, note that you are creating this network policy with a new name. This means you will have two network policies active on your cluster selecting the pods with the label `app: web-server`. Both the `deny-all` and `allow-a-to-b` network policies will be present on your cluster, and both apply to pods with the label `app: web-server`. Network policies, by design, are additive, meaning that if any one of the policies allows the traffic, the traffic will be allowed.

10. Let's create this policy using the following command:

```
kubectl create -f allow-a-to-b.yaml
```

This will return an output similar to *Figure 11.40*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl create -f allow-a-to-b.yaml
networkpolicy.networking.k8s.io/allow-a-to-b created
```

Figure 11.40: Creating a new network policy to allow traffic from web-server-a to web-server-b

11. Let's test the connection between `web-server-a` and `web-server-b` again. You can test this by applying the following command:

```
kubectl exec -it web-server-a -- \
  wget -qO- -T 2 http://<web-server-b IP>
```

This should return an output similar to *Figure 11.41*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl exec -it web-server-a -- \
>   wget -qO- -T 2 http://10.240.0.42
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Figure 11.41: Traffic is again allowed from web-server-a to web-server-b

12. You have now allowed traffic from `web-server-a` to `web-server-b`. You have, however, not allowed the traffic to pass the other way, meaning traffic from `web-server-b` to `web-server-a` is blocked. Let's test this as well. To test this, get the IP address of `web-server-a` using the following command:

    ```
    kubectl get pods -o wide
    ```

    This will return an output similar to *Figure 11.42*, where the IP address of `web-server-a` has been highlighted:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP            NODE
web-server-a   1/1     Running   0          39m   10.240.0.17   aks-nodepool1-36910094-vmss000000
web-server-b   1/1     Running   0          39m   10.240.0.42   aks-nodepool1-36910094-vmss000001
```

Figure 11.42: Getting the IP address of web-server-a

    You can now test the traffic path from `web-server-b` to `web-server-a`:

    ```
    kubectl exec -it web-server-b -- \
      wget -qO- -T 2 http://<web-server-a IP>
    ```

    This should return an output similar to *Figure 11.43*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter11$ kubectl exec -it web-server-b -- \
>   wget -qO- -T 2 http://10.240.0.17
wget: download timed out
command terminated with exit code 1
```

Figure 11.43: Traffic from web-server-b to web-server-a is not allowed, as expected

    As you can see in *Figure 11.43*, the traffic from `web-server-b` to `web-server-a` times out, showing you that the traffic is blocked.

13. This concludes the example regarding network policies in Azure. In the next chapter, you will create a new cluster again, so to conclude this chapter, it is safe to delete this cluster with network policies enabled, using the following command:

    ```
    az aks delete -n handsonaks -g rg-handsonaks -y
    ```

You have now used network policies to protect traffic between pods. You saw how a default policy will deny all traffic, and how you can add new policies to selectively allow traffic. You also saw that if you allow traffic from one pod to another, that the inverse is not automatically allowed.

Summary

This chapter introduced you to multiple network security options in AKS. You explored both securing the control plane and the workload in the cluster.

To secure the control plane, you first used authorized IP ranges to verify that only allowed public IP addresses can access the control plane of your cluster. After that, you created a new private cluster, which was only reachable using a private connection. You connected to that private cluster using Azure Private Link.

After that, you also explored workload network security. Initially, you deployed a public service, which was available for all users. You then had AKS configure Azure NSGs to secure that service only to an allowed connection. You verified that you could connect to the service from your machine, but not from a VM in Azure, as expected. Finally, you also configured Kubernetes network policies in a new cluster. You used those to protect pod-to-pod traffic and were able to secure traffic between different pods on your cluster.

In the next chapter, you will learn how you can use AKS to create Azure resources, such as an Azure Database for MySQL, using the Azure Service Operator.

Section 4: Integrating with Azure managed services

So far in the book, you have run multiple applications on top of **Azure Kubernetes Service** (**AKS**). The applications were always self-contained, meaning the full application was able to run in its entirety on top of AKS. There are certain advantages of running a full application on top of AKS. You gain application portability since you can move that application to any other Kubernetes cluster with little friction. You also have full control over the end-to-end application.

With great control comes great responsibility. There are certain advantages to offloading parts of your application to one of the PaaS services that Azure offers. For example, by offloading your database to a managed PaaS service, you no longer need to take care of updating the database service, backups are automatically performed for you, and a lot of logging and monitoring is done out of the box.

In the coming chapters, you will learn more about multiple advanced integrations and the advantages that come with them. Having read this section, you should be able to securely access other Azure services, such as Azure SQL Database and Azure Functions, and perform **continuous integration and continuous delivery** (**CI/CD**) using GitHub Actions.

This section contains the following chapters:

- *Chapter 12*, *Connecting an application to an Azure database*
- *Chapter 13*, *Azure Security Center for Kubernetes*
- *Chapter 14*, *Serverless functions*
- *Chapter 15*, *Continuous integration and continuous deployment for AKS*

You will start this section with *Chapter 12*, *Connecting an application to an Azure database*, in which you will connect an application to Azure Database for MySQL.

12

Connecting an application to an Azure database

In previous chapters, you stored the state of your application in your cluster, either on a Redis cluster or on MariaDB. You might remember that both had some issues when it came to high availability. This chapter will take you through the process of connecting to a MySQL database managed by Azure.

We will discuss the benefits of using a hosted database rather than running **StatefulSets** on Kubernetes. To create this hosted and managed database, you will make use of **Azure Service Operator** (**ASO**). ASO is a way to create Azure resources, such as a managed MySQL database, from within a Kubernetes cluster. In this chapter, you will learn more details about the ASO project, and you will set up and configure ASO on your cluster.

You will then make use of ASO to create a MySQL database in Azure. You will use this managed database as part of a WordPress application. This will show you how you can connect an application to a managed database. This chapter is broken down into the following topics:

- Azure Service Operator
- Installing ASO on your cluster
- Creating a MySQL database using ASO
- Creating an application using the MySQL database

Let's start by exploring ASO.

Azure Service Operator

In this section, you will learn more about ASO. We will start by exploring the benefits of using a hosted database versus running StatefulSets on Kubernetes itself, and then learn more details about ASO.

All the examples that you have gone through so far have been self-contained; that is, everything ran inside the Kubernetes cluster. Almost any production application has a state, which is generally stored in a database. While there is a great advantage to being mostly cloud-agnostic, this has a huge disadvantage when it comes to managing a stateful workload such as a database.

When you are running your own database on top of a Kubernetes cluster, you need to take care of scalability, security, high availability, DR, and backup. Managed database services offered by cloud providers can offload you or your team from having to execute these tasks. For example, Azure Database for MySQL comes with enterprise-grade security and compliance, built-in high availability, and automated backups. The service scales within seconds. Finally, you also have the option to configure DR to a secondary region.

It is a lot simpler to consume a production-grade database from Azure than it is to set up and manage your own on Kubernetes. In the next section, you will explore a way that Kubernetes can be used to create these databases on Azure.

What is ASO?

As with most applications these days, much of the hard work has already been done for us by the open-source community (including those who work for Microsoft). Microsoft has realized that many users would like to use their managed services from Kubernetes and that they require an easier way of using the same methodologies that are used for Kubernetes deployment. The ASO project was created to solve this problem.

ASO is a new project started in 2020 that succeeds the **Open Service Broker for Azure** (**OSBA**) project. OSBA was Microsoft's original implementation that allowed you to create Azure resources from within Kubernetes, but this project is no longer maintained and has been deprecated. ASO serves the same purpose and is actively maintained and developed.

There are two parts to ASO: a set of **CustomResourceDefinitions** (**CRDs**) and a controller that manages those CRDs. The CRDs are a set of API extensions for Kubernetes that allow you to specify which Azure resources you want to create. There are CRDs for resource groups, virtual machines, MySQL databases, and more.

Most APIs in ASO are still in either the alpha or beta stage, meaning they might change in the future. Please refer to the documentation at https://github.com/Azure/azure-service-operator for an up-to-date resource definition, as the definitions used in this chapter might have changed.

The controller is a pod that runs on your cluster and monitors the Kubernetes API for any objects that are created using these CRDs. It's this controller that will interface with the Azure API and create the resource you create using ASO.

ASO depends on two other projects that you have already learned about in this book, namely **Azure Active Directory** (**Azure AD**) pod-managed identities and cert-manager. ASO uses Azure AD pod-managed identities to link a managed identity to the ASO pod. This also means that this managed identity needs to have permissions to create those resources. ASO uses cert-manager to get access to a certificate for the ASO pod to use.

By default, ASO will store secrets such as connection strings in Kubernetes secrets. As you have learned in the preceding chapters, it's better to store secrets in Key Vault rather than in Kubernetes. ASO has the option to store secrets in Key Vault as well, and during the setup, you will configure ASO to store secrets in Key Vault.

For a user perspective using ASO, *Figure 12.1* describes what happens when you create a resource:

1. As a user, you submit a YAML definition for an Azure resource to the Kubernetes API. The Azure resources are defined in a CRD.
2. The ASO pod is monitoring the Kubernetes API for changes to the Azure CRD objects.
3. When changes are detected, ASO will create the resources in Azure.
4. If a connection string was created as part of the resource creation, this connection string will be stored either as a Kubernetes secret (default) or in Key Vault (if configured).

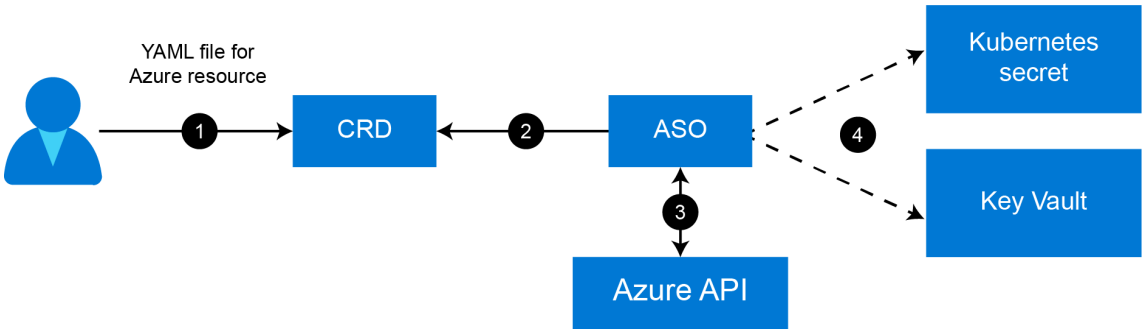


Figure 12.1: High-level process diagram of resource creation using ASO

In this section, you've learned the basics of the ASO project. In the next section, you will go ahead and install ASO on your cluster.

Installing ASO on your cluster

To install ASO on your cluster, you will need a cluster. At the end of the previous chapter, you deleted your cluster, so you will create a new one here. After that, you will need to create a managed identity and Key Vault. Both are best practices when setting up ASO, which is why this chapter will explain how to set up ASO this way. After the creation of these resources, you need to ensure that cert-manager is set up in your cluster. Once that is confirmed, you can install ASO using a Helm chart.

Let's start with the first step, creating a new AKS cluster.

Creating a new AKS cluster

Since you deleted your cluster at the end of the previous chapter, let's start by creating a new cluster. You can do all these steps using Cloud Shell. Let's get started:

1. First, you will create a new cluster. Since you will be making use of pod identities for the authorization of ASO, you will also enable the pod identity add-on on this new cluster. At the time of this writing, the pod identity add-on is in preview.

   If you haven't registered for your subscription for this preview as explained in *Chapter 9*, *Azure Active Directory pod-managed identities in AKS*, please do so now using the following commands:

   ```
   az feature register --name EnablePodIdentityPreview \
     --namespace Microsoft.ContainerService
   ```

   You will also need a preview extension of the Azure CLI, which you can install using the following command:

   ```
   az extension add --name aks-preview
   ```

   You will have to wait until the pod identity preview is registered on your subscription. You can use the following command to verify this status:

   ```
   az feature show --name EnablePodIdentityPreview \
     --namespace Microsoft.ContainerService -o table
   ```

Wait until the status shows as registered, as shown in *Figure 12.2*:

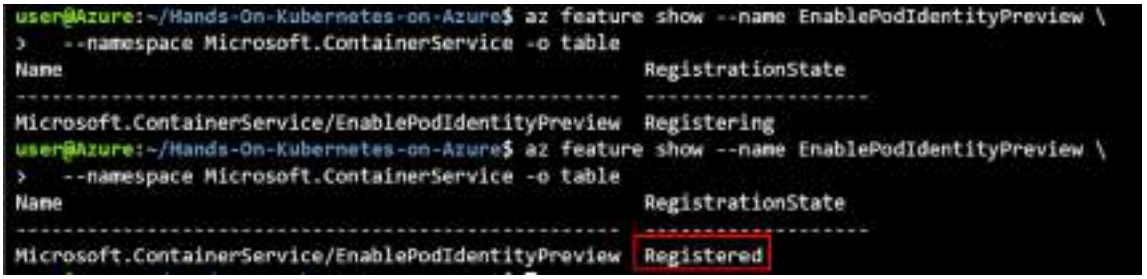


Figure 12.2: Waiting for the feature to be registered

Once the feature is registered, you need to refresh the registration of the namespace before creating a new cluster. Let's first refresh the registration of the namespace:

```
az provider register --namespace Microsoft.ContainerService
```

2. Once you registered the preview provider, or if you had already done so as part of *Chapter 9*, *Azure Active Directory pod-managed identities in AKS*, you can create a new cluster using the following command:

```
az aks create -g rg-handsonaks -n handsonaks \
  --enable-managed-identity --enable-pod-identity \
  --network-plugin azure --node-vm-size Standard_DS2_v2 \
  --node-count 2 --generate-ssh-keys
```

3. Once the command is finished, get the credentials to get access to your cluster using the following command:

```
az aks get-credentials -g rg-handsonaks \
  -n handsonaks --overwrite-existing
```

You now have a new Kubernetes cluster with pod identities enabled. To continue the setup of ASO, let's now create a managed identity.

Creating a managed identity

In this section, you will use the Azure portal to create a managed identity. You will then give permission to your AKS cluster to manage this managed identity and give the managed identity access to your subscription to create the resources. Let's start:

1. In the Azure search bar, look for *Managed Identities*, as shown in *Figure 12.3*:

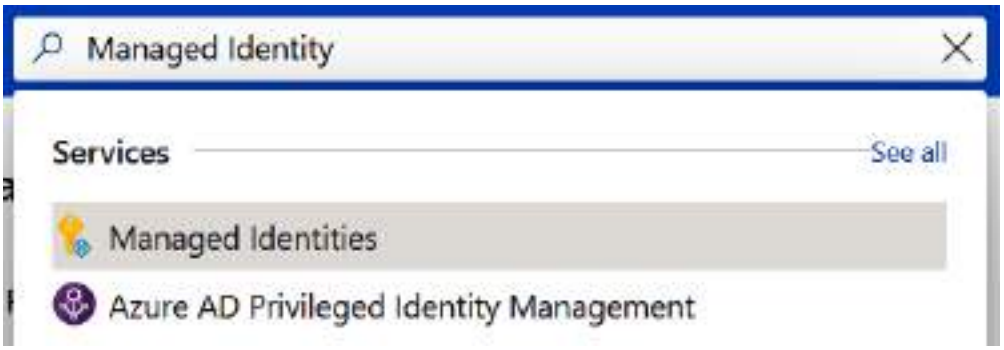


Figure 12.3: Searching for Managed Identities

2. In the resulting screen, click on **+ New** to create a new managed identity, as shown in *Figure 12.4*:

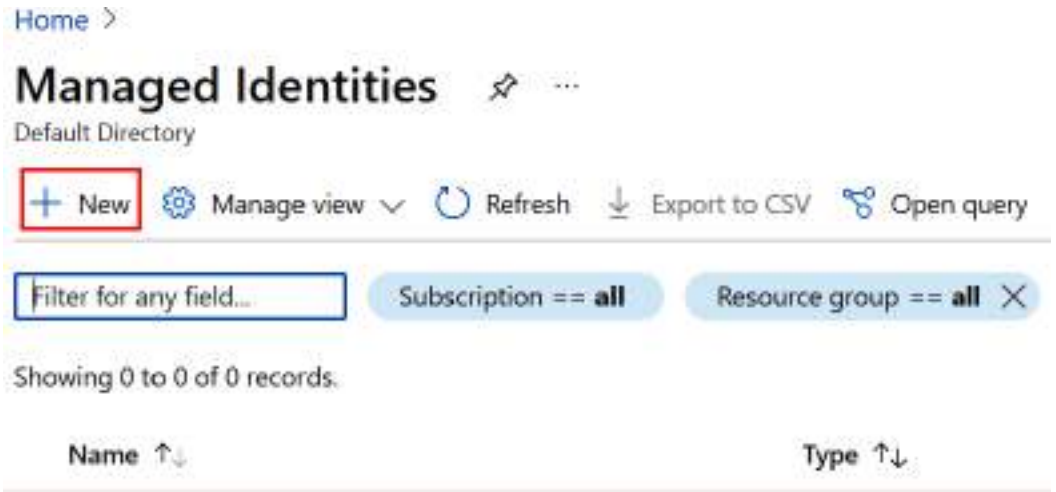


Figure 12.4: Creating a new managed identity

3. To organize the resources for this chapter together, create a new resource group called ASO, as shown in *Figure 12.5*:

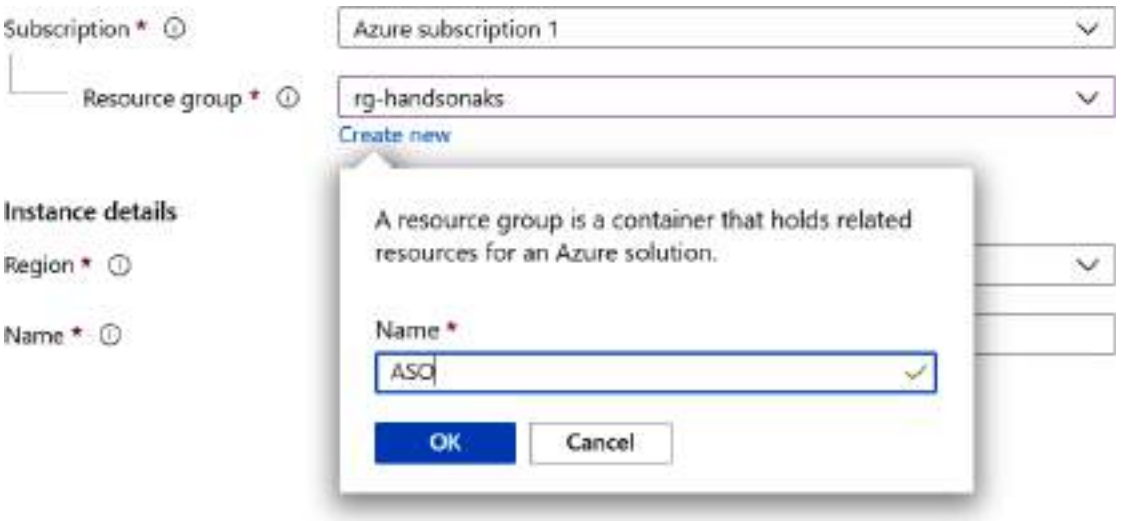


Figure 12.5: Creating a new resource group

4. Provide the location and a name for your managed identity; use the name `aso-mi` as shown in *Figure 12.6* if you wish to follow the example here. Make sure to select the same region as the region of your cluster:

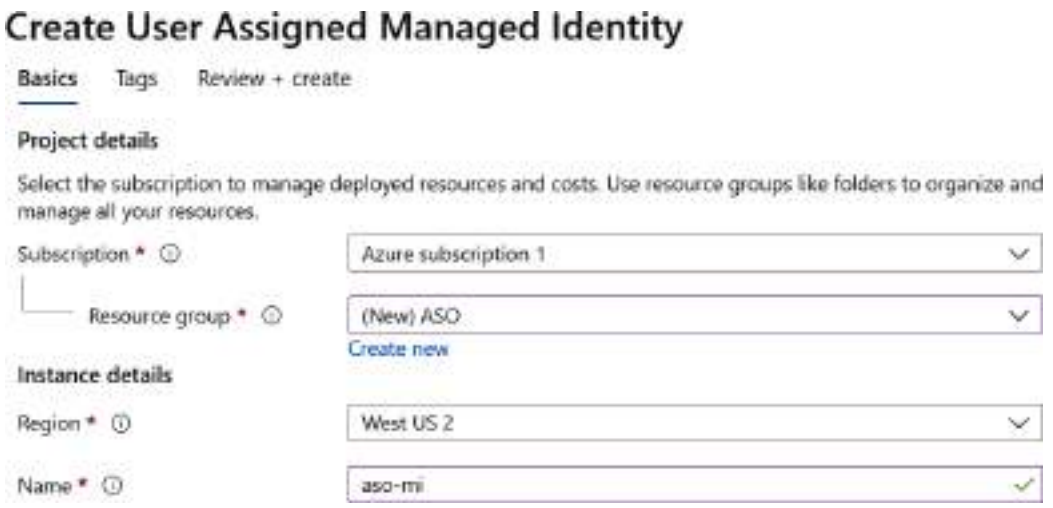


Figure 12.6: Providing Project and Instance details for creating the managed identity

5. Click **Review + create** at the bottom of the screen and create the managed identity.
6. Once the managed identity is created, you need to capture the client ID and resource ID for later use. Copy and paste this information in a location where you can access it later. You can get the client ID in the **Overview** pane, as shown in *Figure 12.7*:

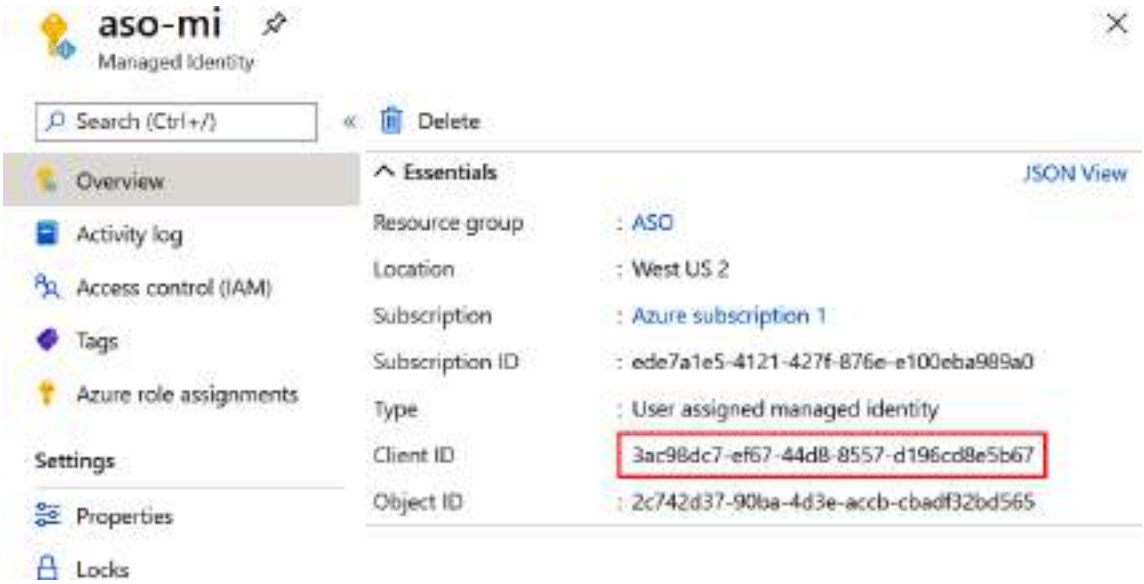


Figure 12.7: Getting the client ID from the managed identity

You can get the resource ID in the Properties pane, as shown in *Figure* 12.8:

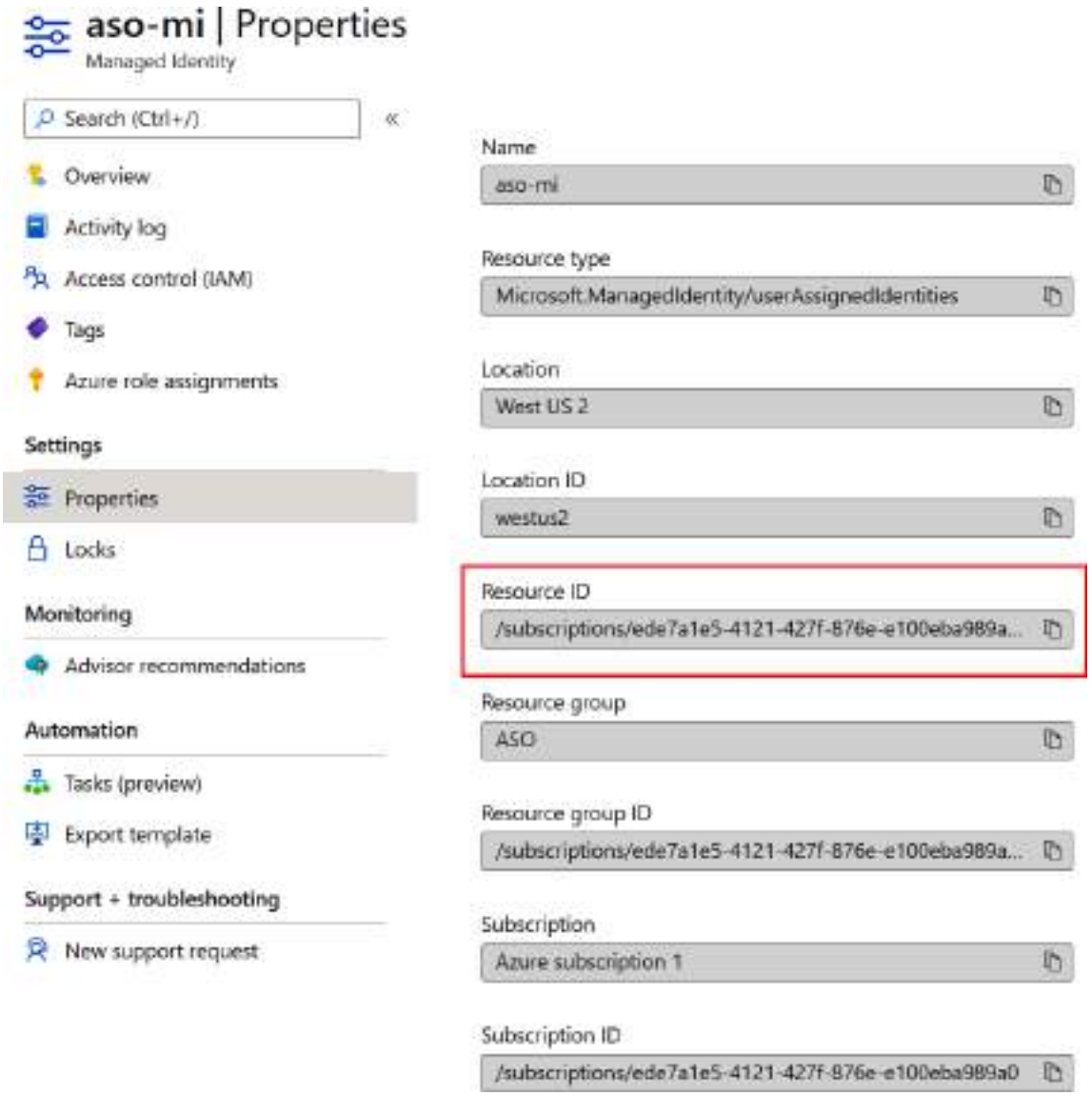


Figure 12.8: Getting the resource ID of the managed identity

7. The next thing to do on the managed identity is to give our AKS cluster permissions to it. To do this, click on **Access control (IAM)** in the left pane, click on the **+ Add** button at the top of the screen, click **Add role assignment** from the dropdown menu, select the **Managed Identity Operator** role, select **User assigned managed identity** from the **Assign access to** dropdown menu, and select the **handsonaks-agentpool** identity and save. This process is shown in *Figure 12.9*:

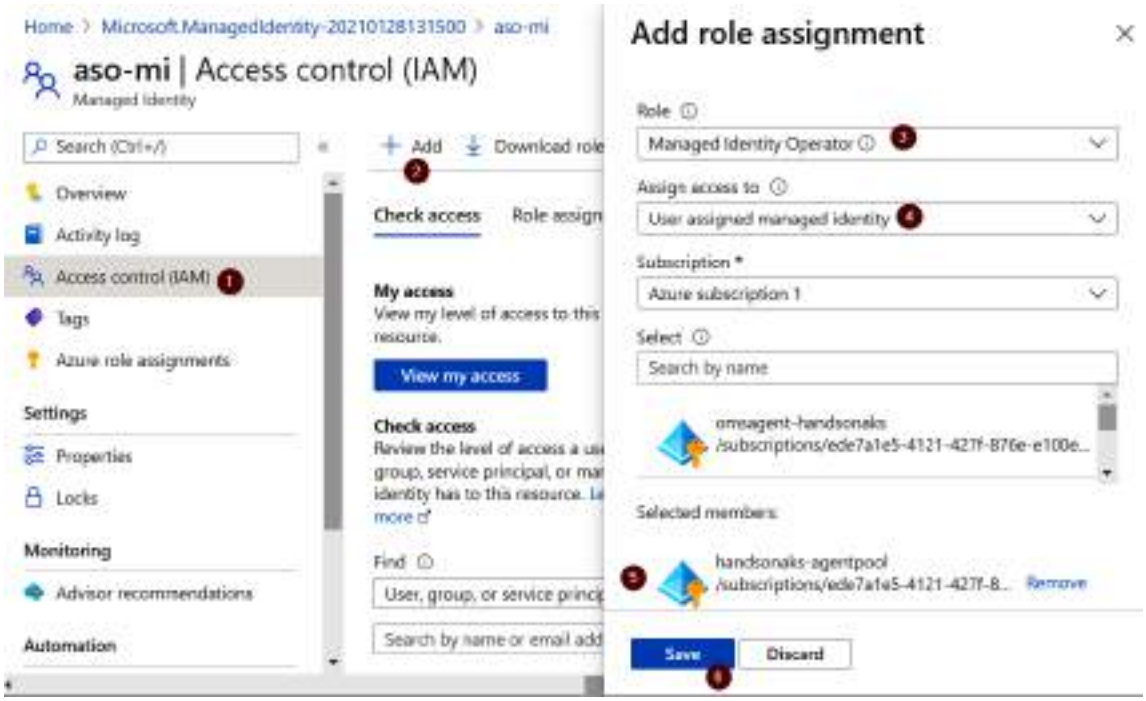


Figure 12.9: Giving AKS access to the managed identity

8. You will now give Managed Identities permission to create resources on your subscription. To do this, look for **Subscriptions** in the Azure search bar, as shown in *Figure 12.10*, and then select your subscription:

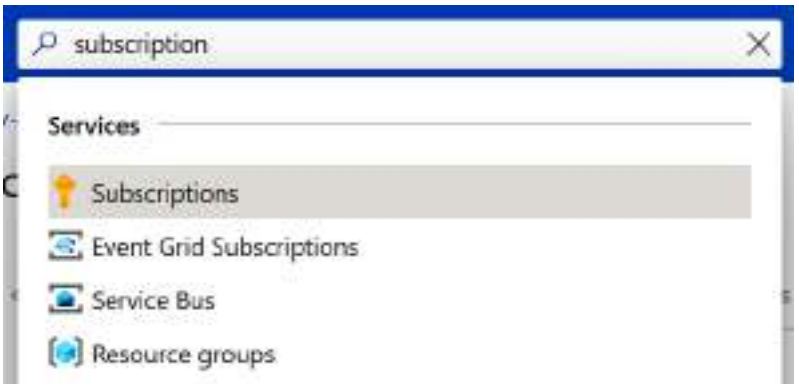


Figure 12.10: Looking for subscriptions in the Azure search bar

9. In the **Subscription** pane, click on **Access control (IAM)**, click on the **+ Add** button at the top of the screen, click **Add role assignment**, select the **Contributor** role, select **User assigned managed identity** from the **Assign access to** dropdown menu, and select the **aso-mi** identity and save. This process is shown in *Figure 12.11*:

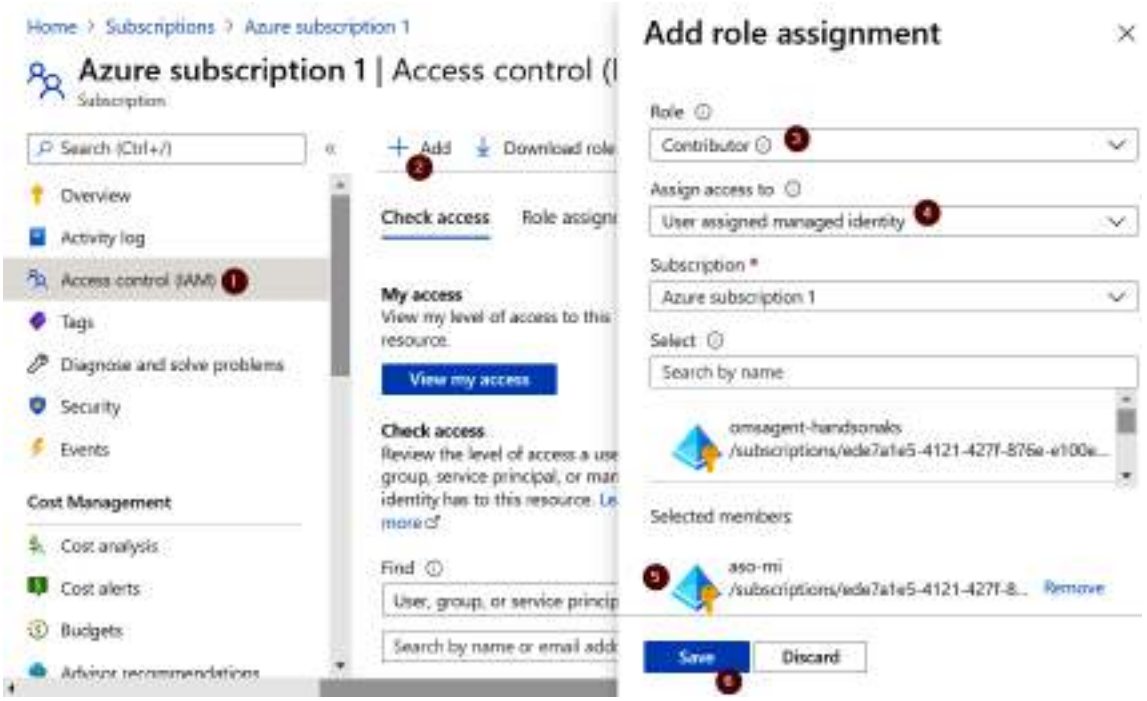


Figure 12.11: Giving the aso-mi permissions to your subscription

This completes the setup of the managed identity. In the next section, you will create a key vault and allow the managed identity you just created to create and read secrets.

Creating a key vault

In this section, you will create the key vault that ASO will use to store connection strings and secrets. This is optional in the ASO setup process but recommended.

1. To start, look for key vaults in the Azure search bar, as shown in *Figure 12.12*:

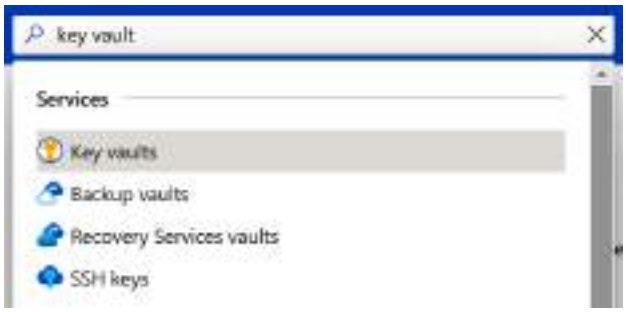


Figure 12.12: Looking for key vaults in the Azure search bar

2. Click the **+ New** button at the top of the screen to create a new key vault. Select the ASO resource group you created earlier and give your key vault a name. Please note that your key vault name has to be unique, so consider adding extra characters to the name if it is not unique. Also, make sure to create the key vault in the same region as your AKS cluster. The resulting configuration is shown in *Figure 12.13*:

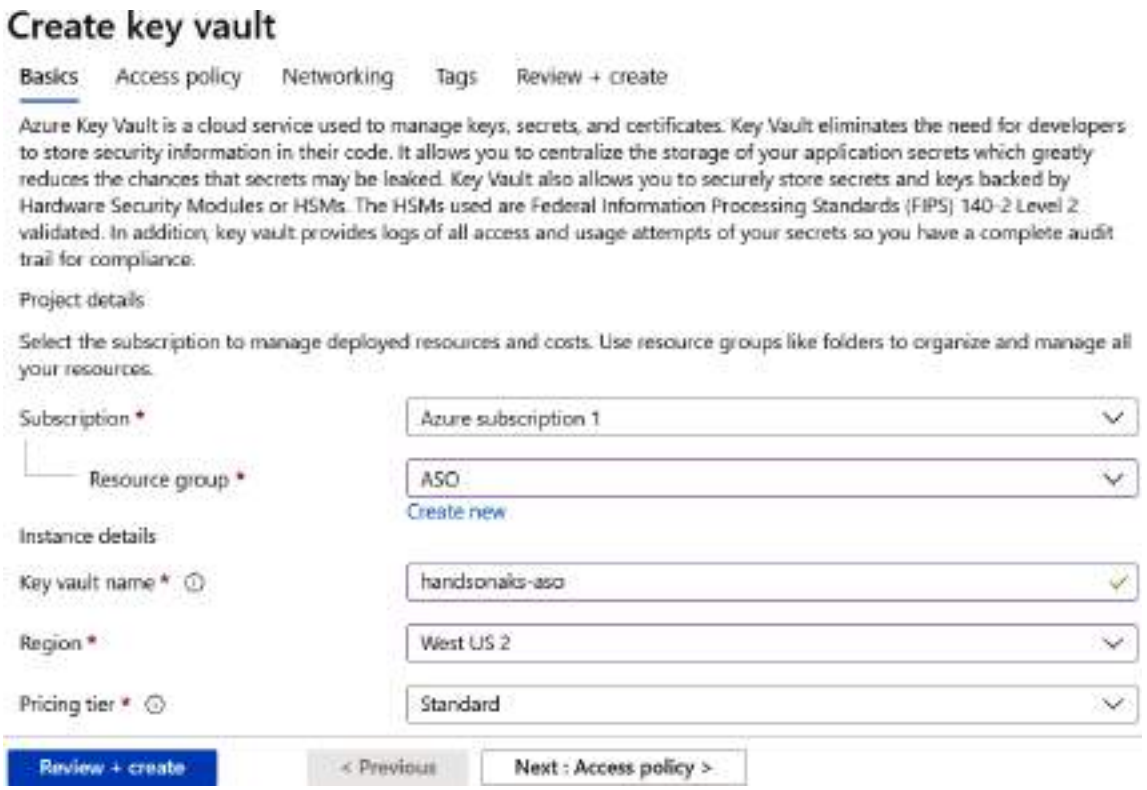


Figure 12.13: Key vault configuration

3. Now select **Next: Access policy >** to configure a new access policy. Here you will give the **aso-mi** managed identity you created in the previous section permission to do secret management in this key vault. To do this, start by clicking the **+ Add Access Policy** button, as shown in *Figure 12.14*:

Figure 12.14: Clicking the + Add Access Policy button

4. In the resulting popup, select the **Secret Management** template, then click on **None selected** to select your managed identity. In the resulting popup, look for the **aso-mi** managed identity, select it, and then click **Select** followed by clicking on **Add**, as shown in *Figure 12.15*:

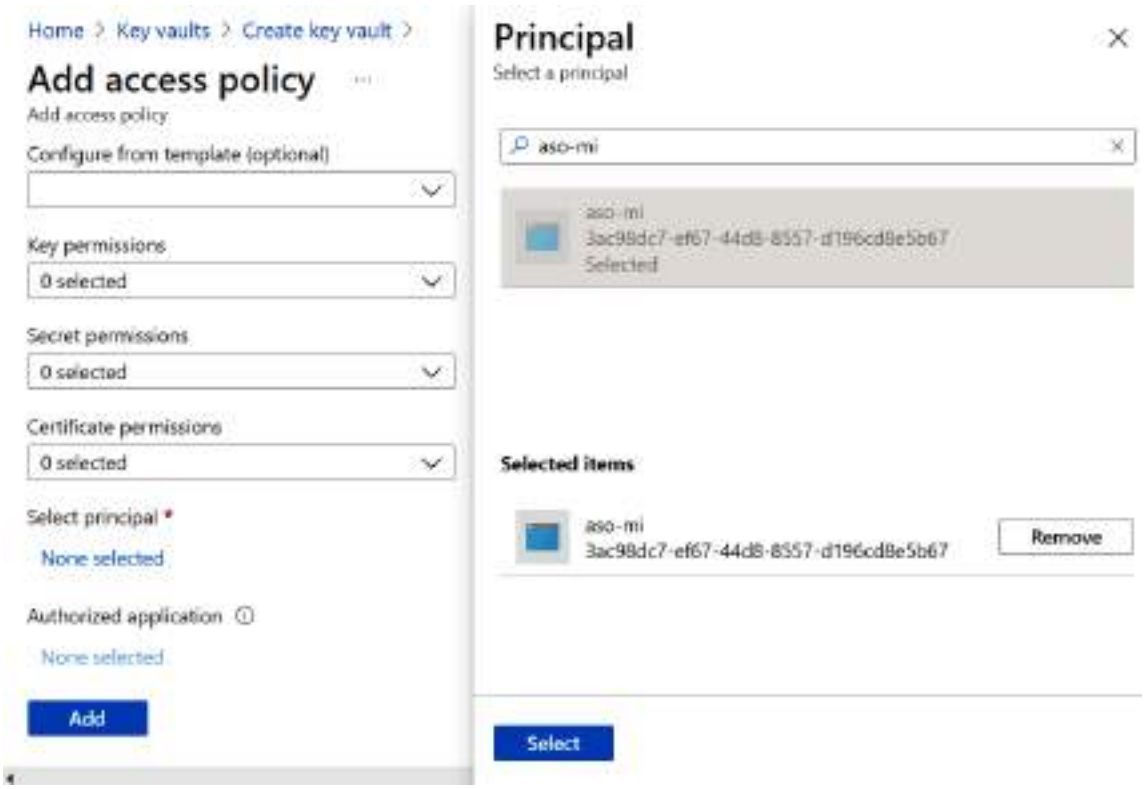


Figure 12.15: Adding the secret management permissions to the managed identity

5. This has configured the access policy in Key Vault. Now click the **Review + create** button, and in the last window hit **Create** to create the key vault. This should take a couple of minutes to complete.

Once your key vault has been deployed, you are ready to start installing ASO, which will be explained in the next section.

Setting up ASO on your cluster

Now that you have the required managed identity and Key Vault, you are ready to start deploying ASO on your cluster. You can do all these steps using Cloud Shell. Let's get started:

1. You created a new cluster in the *Creating a new AKS cluster* section. You will need to link the managed identity you created earlier to the cluster. The ASO components will be created in their own namespace, so you will also create a new namespace for this:

```
kubectl create namespace azureoperator-system
az aks pod-identity add --resource-group rg-handsonaks \
  --cluster-name handsonaks --namespace azureoperator-system \
  --name aso-identity-binding \
  --identity-resource-id <resource ID of managed identity>
```

2. Now you can install cert-manager on your cluster. You've done this once before in *Chapter 6*, *Securing your application with HTTPS*, but at the end of the chapter you were asked to remove this component. You can install it again using the following command:

```
kubectl apply -f https://github.com/jetstack/cert-manager/releases/
download/v1.1.0/cert-manager.yaml
```

3. Track the deployment status of cert-manager using the following command:

```
kubectl rollout status \
  -n cert-manager deploy cert-manager-webhook
```

   Wait until the rollout shows that it's successfully rolled out, as shown in *Figure 12.16*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter12$ kubectl rollout status \
>   -n cert-manager deploy cert-manager-webhook
Waiting for deployment "cert-manager-webhook" rollout to finish: 0 of 1 updated replicas are available...
deployment "cert-manager-webhook" successfully rolled out
```

Figure 12.16: Checking the rollout status of cert-manager

4. Once cert-manager has fully rolled out, you can start the ASO installation. Start by adding the Helm repo for ASO using the following command:

```
helm repo add azureserviceoperator \
https://raw.githubusercontent.com/Azure/azure-service-operator/
master/charts
```

5. Next, you need to provide configuration values for your ASO installation. Open the `values.yaml` file that is part of the code sample that comes with this chapter using the following command:

```
code values.yaml
```

Fill in all the required values in that file, as shown here:

```
1   azureTenantID: "<tenant ID>"
2   azureSubscriptionID: "<subscription ID>"
3   azureOperatorKeyvault: "<key vault name>"
4   azureClientID: "<client ID>"
5   cloudEnvironment: AzurePublicCloud
6   azureUseMI: true
7   image:
8     repository: mcr.microsoft.com/k8s/azureserviceoperator:0.1.16800
9   installAadPodIdentity: true
10  aad-pod-identity:
11    azureIdentityBinding:
12      name: aso-identity-binding
13      selector: aso_manager_binding
14    azureIdentity:
15      enabled: True
16      name: aso-identity
17      type: 0
18      resourceID: "<resource ID>"
19      clientID: "<client ID>"
```

As shown in the previous code sample, you will need to provide your tenant ID, subscription ID, key vault name, client ID of the managed identity (twice), and resource ID of the managed identity. You can find the tenant ID and subscription ID with the following command:

```
az account show
```

This will return an output similar to *Figure 12.17*, in which the tenant ID and subscription ID have been highlighted:

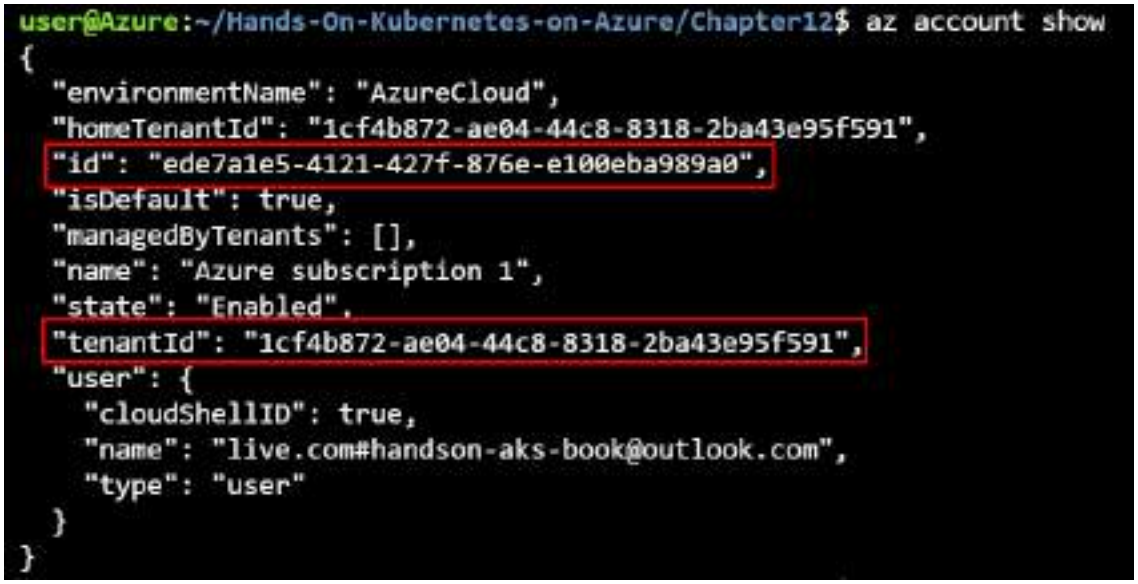


Figure 12.17: Getting the subscription ID and tenant ID

6. Once you have the values filled in, you can install ASO using the following command:

```
helm upgrade --install aso \
  azureserviceoperator/azure-service-operator \
  -n azureoperator-system --create-namespace \
  -f values.yaml
```

7. The installation process takes a couple of minutes. Wait until the following command returns a successful rollout:

```
kubectl rollout status deploy \
  -n azureoperator-system azureoperator-controller-manager
```

The output should look similar to *Figure 12.18*:

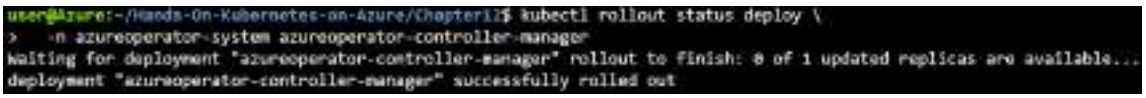


Figure 12.18: Checking the status of the deployments for ASO

8. At the time of writing, there was an issue with the `aadpodidbinding` label on the deployment of `azureoperator-controller-manager`. This can, however, be fixed by applying a patch, to apply a new label to that deployment. The patch has been provided in the files for the chapter, specifically in the `patch.yaml` file:

```
spec:
  template:
    metadata:
      labels:
        aadpodidbinding: aso-identity-binding
```

   As you can see, the patch itself applies a new label to the pods in the deployment. You can apply the patch using the following command:

```
kubectl patch deployment \
  azureoperator-controller-manager \
  -n azureoperator-system \
  --patch "$(cat patch.yaml)"
```

   This will ensure that you can use ASO in the next section.

Now that ASO has been deployed on your cluster, you are ready to start deploying Azure resources using Kubernetes and ASO. You will do that in the next section.

Deploying Azure Database for MySQL using ASO

In the previous section, you deployed ASO on your Kubernetes cluster. This means that now you can use the Kubernetes API to deploy Azure resources. In this section, you will create a MySQL database running on the Azure Database for MySQL service using YAML files that you will submit to Kubernetes using `kubectl`. Let's get started:

1. First, you need to create a resource group. The code for the resource group definition is also available in the code samples with this chapter. Create this file and save it as `rg.yaml`:

```
apiVersion: azure.microsoft.com/v1alpha1
kind: ResourceGroup
metadata:
  name: aso-resources
spec:
  location: <cluster location>
```

   As you can see in the code for the resource, `apiVersion` refers to `azure.microsoft.com` and `kind` is `ResourceGroup`. Furthermore, you provide the details for the resource group, being its name and its location. Make sure to change `location` to the location of your cluster.

   You can create this resource group using the following command:

```
kubectl create -f rg.yaml
```

   To monitor the process of the resource group creation, you can use the following command:

```
kubectl get resourcegroup -w
```

   This returns an output similar to *Figure 12.19*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter12$ kubectl get rg -w
NAME            PROVISIONED   MESSAGE
aso-resources
aso-resources
aso-resources   true          successfully provisioned
```

Figure 12.19: Monitoring the creation of a new resource group

2. Let's also verify that the resource group was created in Azure. To do so, look for the resource group name (`aso-resources`, in this example) in the Azure search bar, as shown in *Figure 12.20*:

Figure 12.20: Searching for the resource group in the Azure portal

As you can see, the resource group is returned in the search results, meaning the resource group was successfully created.

3. Now you can create the MySQL server. You won't create a virtual machine to run MySQL, but rather create a managed MySQL server on Azure. To create this, you can use the `mysql-server.yaml` file that is provided for you:

```
apiVersion: azure.microsoft.com/v1alpha1
kind: MySQLServer
metadata:
  name: <mysql-server-name>
spec:
  location: <cluster location>
  resourceGroup: aso-resources
  serverVersion: "8.0"
  sslEnforcement: Disabled
  createMode: Default
  sku:
    name: B_Gen5_1
    tier: Basic
    family: Gen5
    size: "5120"
    capacity: 1
```

This file contains specific configurations for the MySQL server. A number of elements are worth pointing out:

- **Line 2**: Here you define that you will create a `MySQLServer` instance.
- **Line 4**: Here you give the server a name. This name has to be globally unique, so consider appending your initials to the server name.
- **Line 6**: The location of the MySQL server you will create. Make sure to change `location` to the location of your cluster.
- **Line 9**: `sslEnforcement` is disabled for this demo. This has been done to make the demo easier to follow. If you create a production cluster, it is highly recommended to enable `sslEnforcement`.
- **Line 11-16**: Here you define the size of the MySQL server. In this case, you are creating a basic server with 5 GB of capacity. If you plan to use this for production use cases, you will likely need a larger server.

You can create the MySQL server using the following command:

```
kubectl create -f mysql-server.yaml
```

This will take a couple of minutes to complete. You can follow the progress using the following command:

```
kubectl get mysqlserver -w
```

This will return an output similar to *Figure 12.21*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter12$ kubectl get mysqlserver -w
NAME           PROVISIONED   MESSAGE
wp-helm-mysql
wp-helm-mysql
wp-helm-mysql
wp-helm-mysql                request submitted to Azure
wp-helm-mysql   true         successfully provisioned
```

Figure 12.21: Monitoring the creation of the MySQL server

If you were to run into errors when creating the MySQL server, please refer to the ASO documentation at https://github.com/Azure/azure-service-operator/blob/master/docs/troubleshooting.md.

Once you get the message that the server has successfully been provisioned, you can exit out of this command by pressing *Ctrl + C*.

4. After the MySQL server, you can create the MySQL database. The definition of the MySQL database has been provided in the `mysql-database.yaml` file:

```
1   apiVersion: azure.microsoft.com/v1alpha1
2   kind: MySQLDatabase
3   metadata:
4     name: wordpress-db
5   spec:
6     resourceGroup: aso-resources
7     server: <mysql-server-name>
```

The definition of the database is providing a name and referring to the server you created earlier. To create the database, you can use the following command:

```
kubectl create -f mysql-database.yaml
```

This will take a couple of seconds to complete. You can follow the progress using the following command:

```
kubectl get mysqldatabase -w
```

This will return an output similar to *Figure 12.22*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter12$ kubectl get mysqldatabase -w
NAME           PROVISIONED   MESSAGE
wordpress-db
wordpress-db
wordpress-db
wordpress-db                 mysql.DatabasesCreateOrUpdateFuture: asynchronous operation has not completed
wordpress-db   true          successfully provisioned
```

Figure 12.22: Monitoring the creation of the MySQL database

Once you get the message that the database has successfully been provisioned, you can exit out of this command by pressing *Ctrl + C*.

5. You can create a firewall rule that will allow traffic to your database. In this example, you will create a rule that will allow traffic from all sources. In a production environment, this is not recommended. For the recommended networking configurations for Azure Database for MySQL, please refer to the documentation: https://docs.microsoft.com/azure/mysql/flexible-server/concepts-networking.
The configuration for the firewall rule has been provided in the `mysql-firewall.yaml` file:

```
apiVersion: azure.microsoft.com/v1alpha1
kind: MySQLFirewallRule
metadata:
  name: allow-all-mysql
spec:
  resourceGroup: aso-resources
  server: <mysql-server-name>
  startIpAddress: 0.0.0.0
  endIpAddress: 255.255.255.255
```

As you can see, we refer to the MySQL server that was created earlier and allow traffic from all IP addresses (meaning from `0.0.0.0` to `255.255.255.255`).

To create the firewall rule, you can use the following command:

```
kubectl create -f mysql-firewall.yaml
```

This will take a couple of seconds to complete. You can follow the progress using the following command:

```
kubectl get mysqlfirewallrule -w
```

This will return an output similar to *Figure 12.23*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter12$ kubectl get mysqlfirewallrule -w
NAME             PROVISIONED   MESSAGE
allow-all-mysql
allow-all-mysql
allow-all-mysql
allow-all-mysql                mysql.FirewallRulesCreateOrUpdateFuture: asynchronous operation has not completed
allow-all-mysql   true         successfully provisioned
```

Figure 12.23: Monitoring the creation of the MySQL firewall rule

Once you get the message that the firewall rule has successfully been provisioned, you can exit out of this command by pressing *Ctrl + C*.

6. Let's verify that all of this was successfully created in the Azure portal. To do so, start by searching for the MySQL server name (`wp-helm-mysql` in this example) in the Azure search bar as shown in *Figure 12.24*. Click on the server to go to the details:

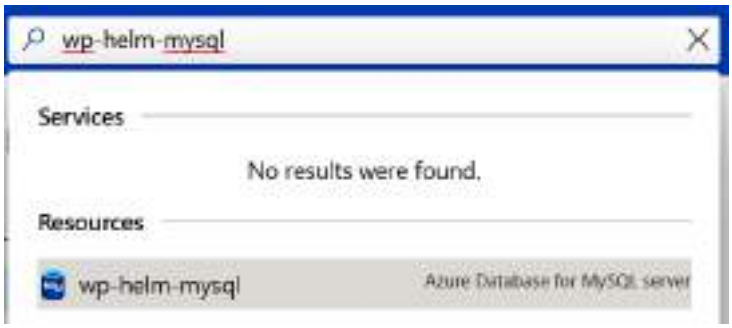


Figure 12.24: Searching for the MySQL server in the Azure portal

7. This will take you to the **Overview** pane of the MySQL server. Scroll down in this pane and expand the **Available resources** section. Here you should see that **wordpress-db** was created, as shown in *Figure 12.25*:

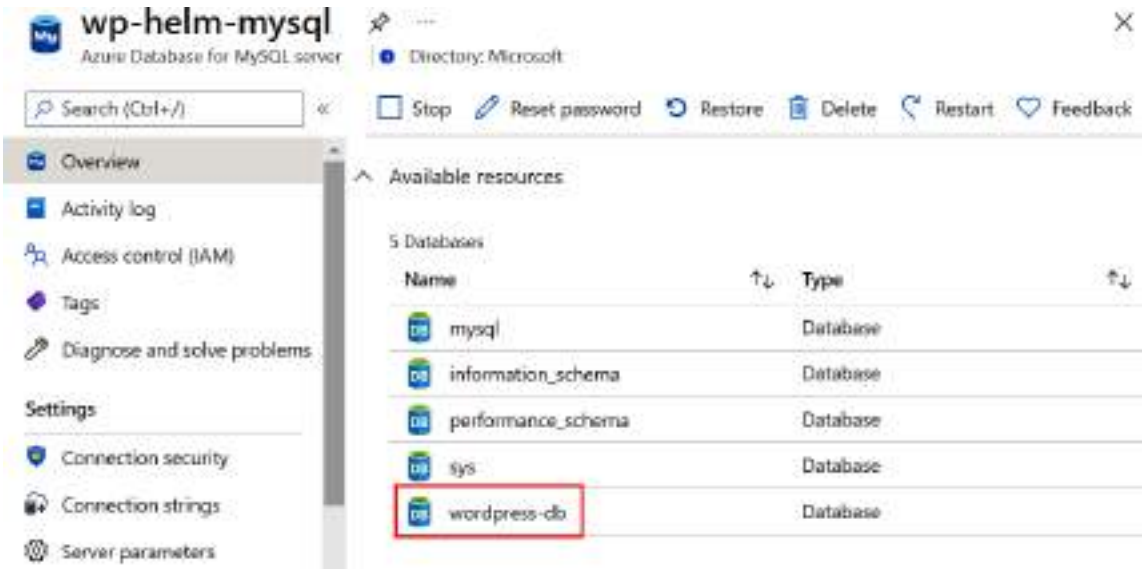


Figure 12.25: The database created through ASO is shown in the Azure portal

8. From the MySQL server pane, click on **Connection security** in the left-hand navigation to verify the firewall rule. You should see the firewall rule you created through ASO on this pane, as shown in *Figure 12.26*:

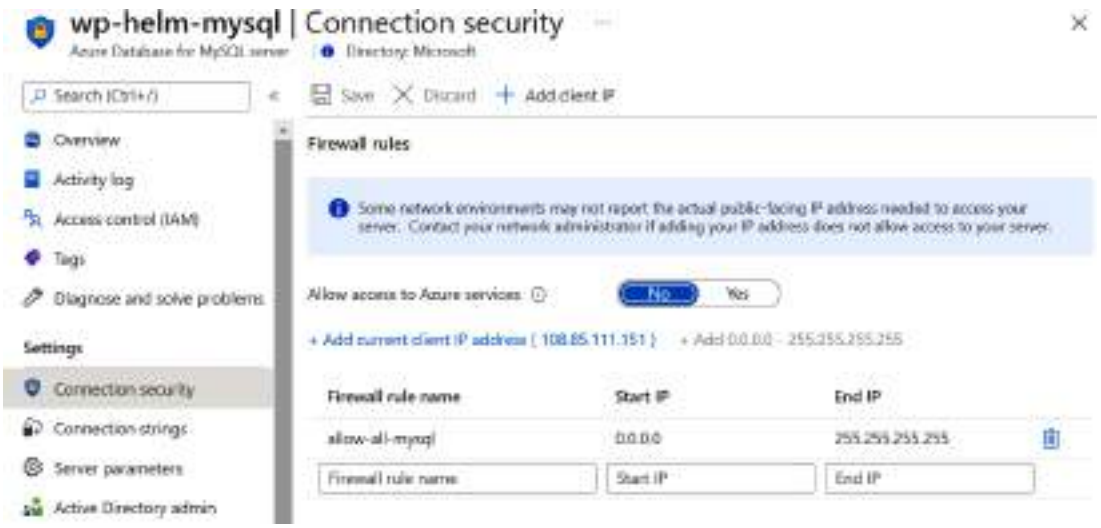


Figure 12.26: The firewall rule created through ASO is set on the MySQL server

This verifies that you were able to create a MySQL server with a database in Azure and configure its firewall settings.

In this section, you've used ASO to create a MySQL server, as well as a database on that server, and then finally configured its firewall. You were able to do all of this using Kubernetes YAML files. ASO translated those YAML files to Azure and created the resources for you. Finally, you were able to confirm everything was created and configured in the Azure portal.

In the next and final section, you will use this database to support the WordPress application.

Creating an application using the MySQL database

You now have a MySQL database. To showcase that you can use this database to configure an application, you will use the WordPress application. You can install this using Helm and provide the connection information to your database in the Helm configuration:

1. To start, you will need the connection information to your database server. When you installed ASO on your cluster, you configured it to use Key Vault as a secret store rather than Kubernetes secrets. You will need this connection information to connect WordPress to your Azure MySQL database.

   Search for `Key Vaults` in the Azure search bar, as shown in *Figure 12.27*, click on **Key vaults**, and then select the key vault you created earlier in the chapter:

Figure 12.27: Searching for key vaults in the Azure portal

2. In the resulting pane, click on **Secrets** in the left-hand navigation and then click on the secret, as shown in *Figure 12.28*. The name of this secret follows the naming convention `<object type>-<Kubernetes namesapce>-<object name>`.

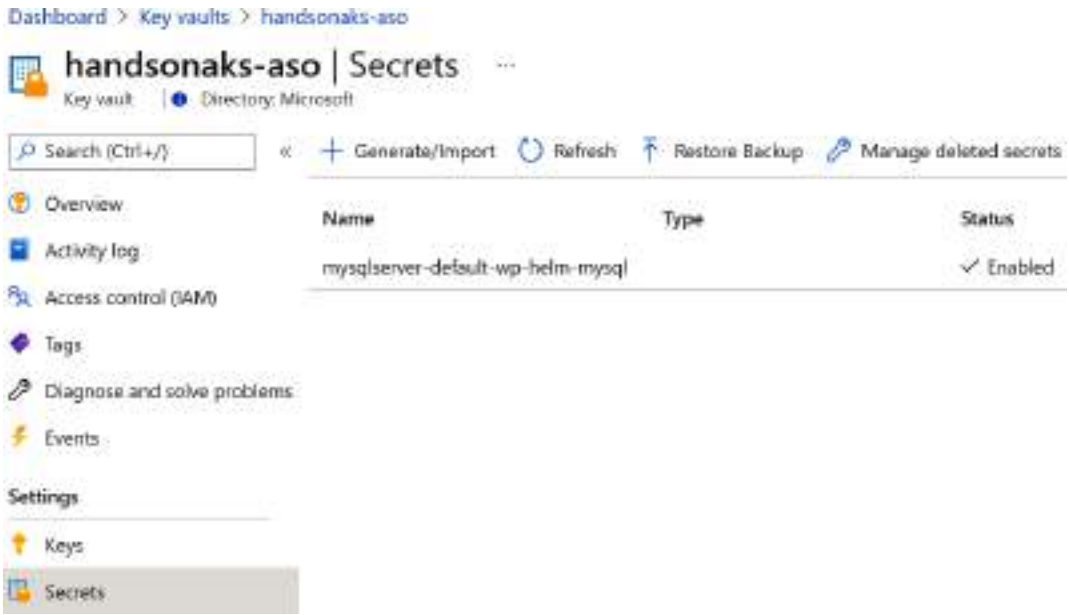


Figure 12.28: The MySQL secret in the Azure portal

3. You will then get a view with multiple versions of your secret; click the current version as shown in *Figure 12.29*:

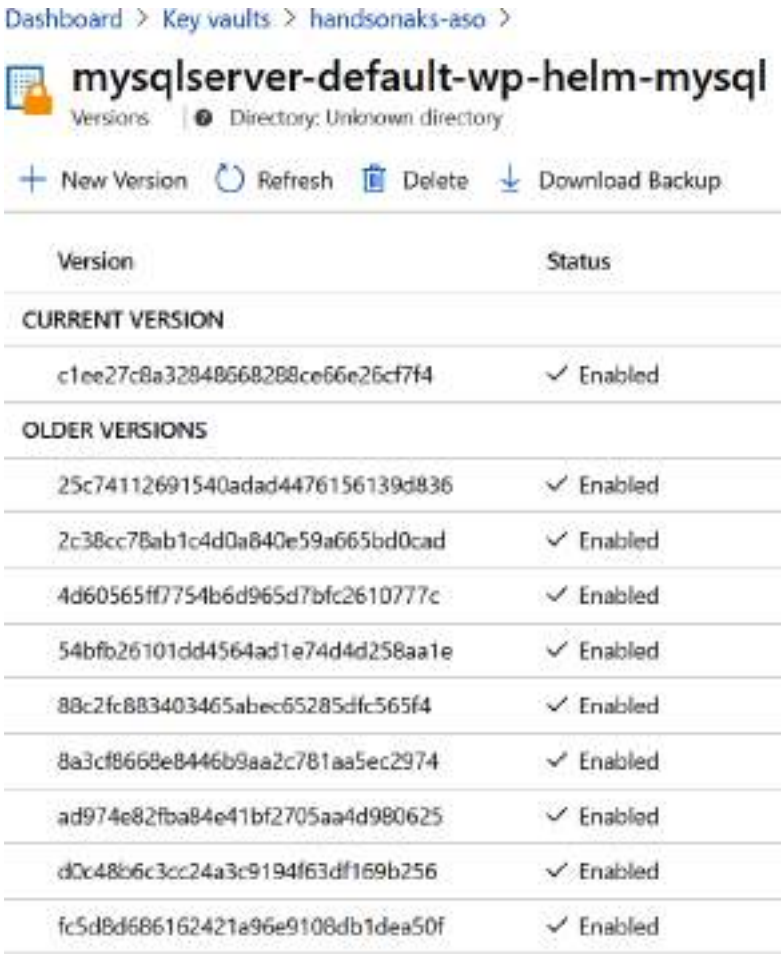


Figure 12.29: Different secret versions in your key vault

Now, copy the value of the secret, as shown in *Figure 12.30*:

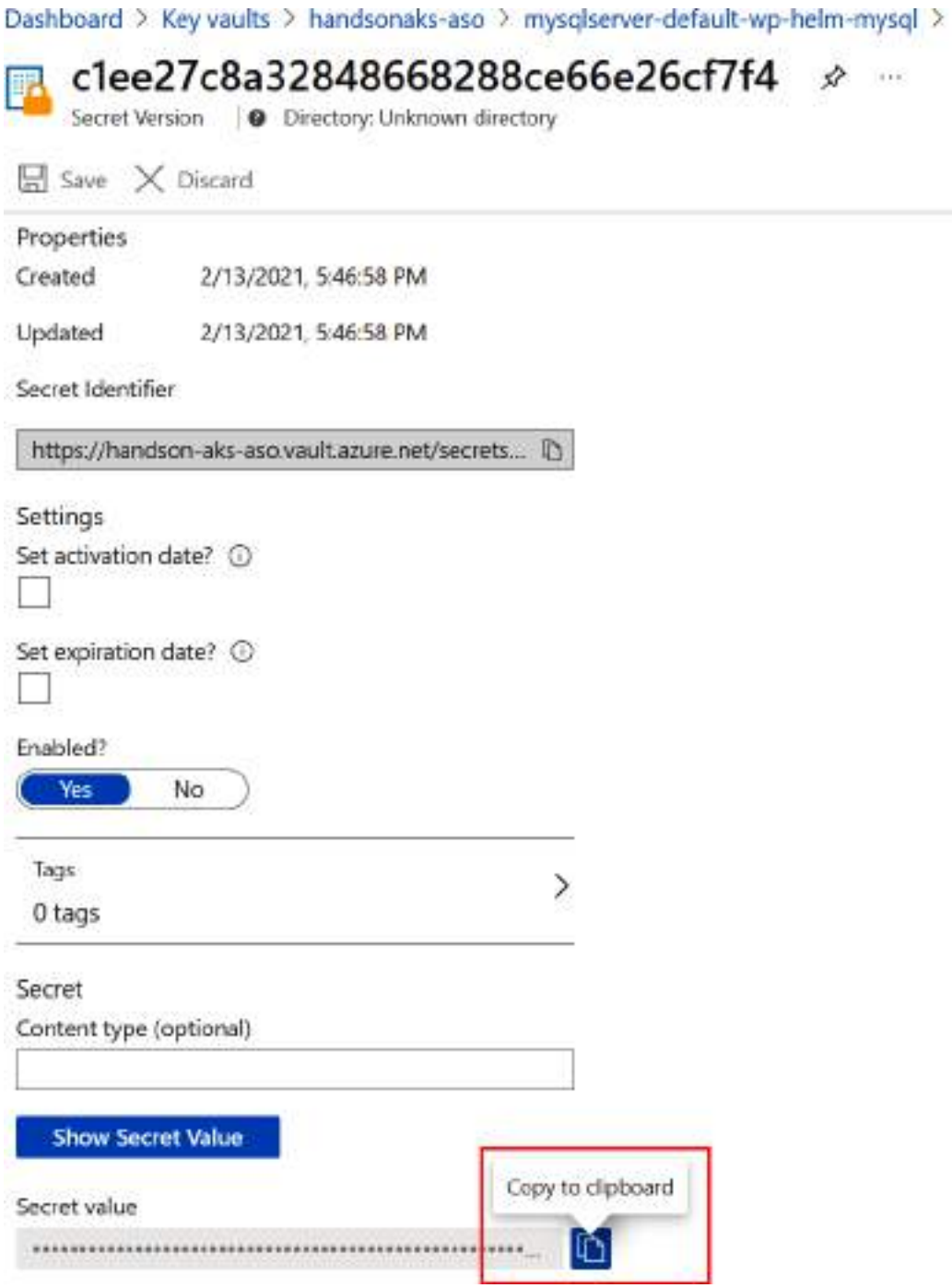


Figure 12.30: Copying the value of the secret to clipboard

4. The secret contains several pieces of information related to your database connection that you will need for the Helm installation. It contains the fully qualified server name, the username, and the password. The values in the secret are Base64 encoded. To make working with this secret easier, a shell script has been provided that will give you the required decoded values. To run this script, use the following command:

```
sh decode-secret.sh <secret value>
```

   An example is shown in *Figure 12.31*:

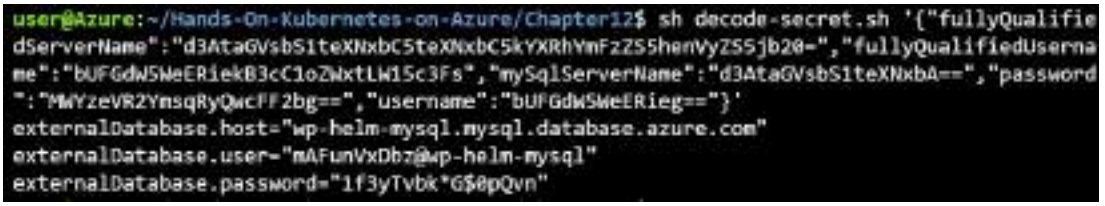


Figure 12.31: Decoding the secret

5. You can use the values outputted by the previous step to configure Helm to use your Azure MySQL database. The following Helm command will set up WordPress on your cluster, but use an external database:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install wp bitnami/wordpress \
  --set mariadb.enabled=false \
  --set externalDatabase.host='<decoded host value>' \
  --set externalDatabase.user='<decoded user value>' \
  --set externalDatabase.password='<decoded password value>' \
  --set externalDatabase.database='wordpress-db' \
  --set externalDatabase.port='3306'
```

   As you can see, with this command, you disabled the MariaDB installation by setting the `mariadb.enabled` value to `false` and then provided the connection information to the external database.

To monitor the setup of WordPress, you can use the following command:

```
kubectl get pods –w
```

This will take a couple of minutes to fully set up, and finally, you should see the WordPress pod in a running state and ready, as shown in *Figure 12.32*:

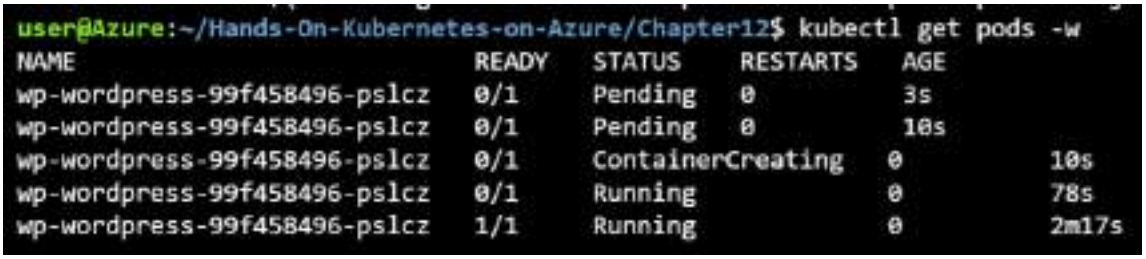


Figure 12.32: WordPress pod in a running state

Once the pod is running and ready, you can stop this command by pressing *Ctrl* + *C*. If you remember the WordPress deployment in *Chapter 3*, *Application deployment on AKS*, there was a second pod present in the WordPress installation hosting a MariaDB database. This pod is no longer there since we replaced it with an Azure MySQL database.

6. Let's now finally connect to this WordPress application. You can get the public IP address of the WordPress website using the following command:

```
kubectl get service
```

This will show you the public IP, as shown in *Figure 12.33*:

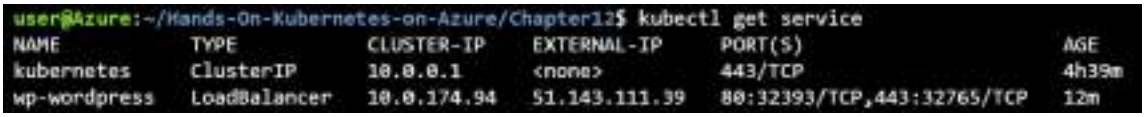


Figure 12.33: Getting the public IP of the WordPress website

Enter this IP address in your web browser's address bar and hit *Enter*. You should be able to see the WordPress landing page with the default demo post, as shown in *Figure 12.34*:

Figure 12.34: Browsing to the WordPress website

You now have a fully functional WordPress website hosted on Kubernetes, with the database being backed by Azure Database for MySQL.

7. This concluded the examples from this chapter. You created a number of resources and installed a number of cluster components. Let's also clean them up from the cluster using the following commands:

```
helm uninstall wp
kubectl delete -f mysql-firewall.yaml
kubectl delete -f mysql-database.yaml
kubectl delete -f mysql-server.yaml
kubectl delete -f rg.yaml
helm uninstall aso -n azureoperator-system
az aks pod-identity delete --resource-group rg-handsonaks \
  --cluster-name handsonaks --namespace azureoperator-system \
  --name aso-identity-binding
kubectl delete namespace azureoperator-system
kubectl delete -f https://github.com/jetstack/cert-manager/releases/
download/v1.1.0/cert-manager.yaml
az group delete -n aso --yes
```

You've been able to connect an application on Kubernetes to an Azure-managed MySQL database. You used the WordPress Helm chart and provided custom values to configure this Helm chart to make it connect to the managed database.

Summary

This chapter introduced **Azure Service Operator** (**ASO**). ASO is an open-source project that makes it possible to create Azure services using Kubernetes. This allows you as the user to not have to switch between the Azure portal or CLI and Kubernetes resource definitions.

In this chapter, you created a new AKS cluster and then installed ASO on this cluster. You then created a MySQL database on Azure using ASO. You verified that this database was available in Azure using the Azure portal.

Finally, you created a WordPress application on your Kubernetes cluster that connected to the external database. You verified that the application was running and available as you've seen in previous chapters.

In the next chapter, you will learn about other Azure integrations with AKS, namely Azure Security Center and Azure Defender for Kubernetes, which are used to monitor the security configuration of your cluster and mitigate threats.

13

Azure Security Center for Kubernetes

Kubernetes is a very powerful platform with a lot of configuration options. Configuring your workload the right way and making sure you follow best practices can be difficult. There are industry benchmarks that you can follow to get guidelines for how to deploy your workloads securely, such as the **Center for Internet Security** (**CIS**) Benchmarks for Kubernetes: https://www.cisecurity.org/benchmark/kubernetes/.

Azure Security Center is a unified infrastructure security management platform. It provides continuous security monitoring and alerting for resources in Azure as well as for hybrid workloads. It offers protection for many Azure resources, including Kubernetes clusters. This will allow you to ensure your workloads are configured securely and protected.

Azure Security Center offers two types of protection. First, it monitors your resource configuration and compares it to security best practices, and then gives you actionable recommendations to improve your security posture. Second, it also does threat protection by assessing your workloads and raising alerts when a potential threat is identified. This threat detection capability is part of a feature called Azure Defender within Azure Security Center.

When it comes to monitoring Kubernetes workloads, Azure Security Center can monitor both your cluster configuration as well as the configuration of the workloads running in your cluster. To monitor the configuration of the workloads in your cluster, Azure Security Center uses Microsoft Azure Policy for Kubernetes. This free add-on for **Azure Kubernetes Service** (**AKS**) will enable Azure Security Center to compare the configuration of your workloads against known best practices.

Azure Defender also has specific threat detection capabilities for Kubernetes. It monitors a combination of Kubernetes audit logs, node logs, as well as cluster and workload configuration to identify potential threats. Examples of threats that can be discovered are crypto-miners, the creation of high-privileged roles, or exposing the Kubernetes dashboard.

In this chapter, you'll enable Azure Security Center, Azure Policy for Kubernetes, and Azure Defender for Kubernetes and monitor several sample applications against threats.

In this chapter, we will cover the following topics:

- Azure Security Center for Kubernetes
- Azure Defender for Kubernetes
- Deploying offending workloads
- Analyzing configuration using Azure Secure Score
- Neutralizing threats using Azure Defender

Let's start by setting up Azure Security Center for Kubernetes.

Setting up Azure Security Center for Kubernetes

We'll start this chapter by setting up Azure Security Center for Kubernetes. To enable Azure Security Center for Kubernetes, you need to enable Azure Policy for AKS on your cluster. This will enable Azure Security to monitor your workload configuration. To benefit from Azure Defender's threat protection, you will also need to enable Azure Defender for Kubernetes on your subscription.

Let's get started.

1. Search for your AKS cluster in the Azure search bar, as shown in *Figure 13.1*:

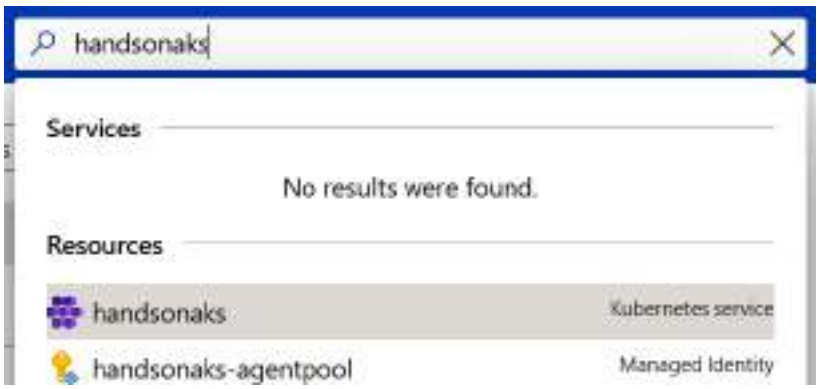


Figure 13.1: Looking for your cluster in the Azure search bar

2. You will now enable Azure Policy for AKS. To enable this, click the **Policies** button on the left-hand side and on the resulting pane, click on **Enable add-on**, as shown in *Figure 13.2*:

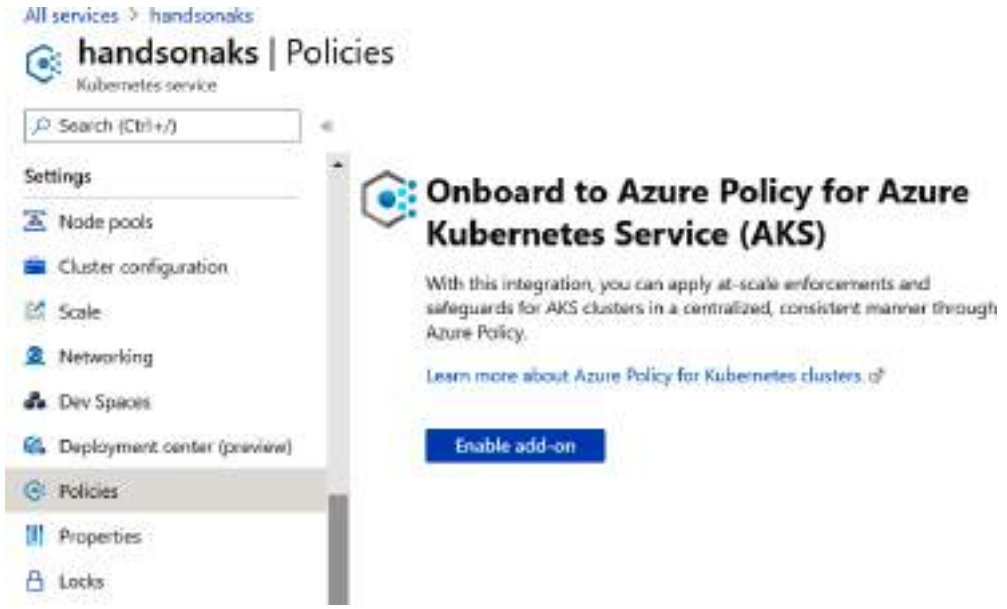


Figure 13.2: Enabling Azure Policy for AKS

Enabling the add-on will take a couple of minutes to complete. After a while, you should see a message saying that the service is now enabled, as shown in *Figure 13.3*:

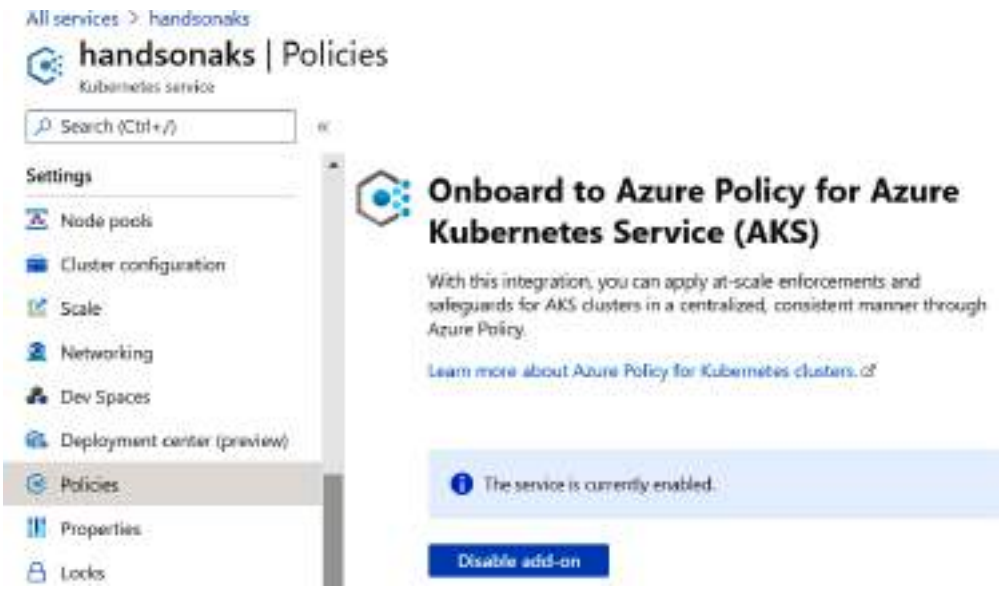


Figure 13.3: Azure Policy for AKS is now enabled

3. This has enabled Azure Policy for AKS. Next, you will enable Azure Defender to get the threat prevention ability from Azure Security Center. To do so, look up `security center` in the Azure portal's search bar, as shown in *Figure 13.4*:

Figure 13.4: Searching for security center in the Azure portal search bar

4. If this is the first time you are accessing Azure Security Center in your subscription, you'll be greeted with a message as shown in *Figure 13.5*. To enable Azure Defender, click on **Upgrade** at the bottom of the screen:

Figure 13.5: Upgrading to Azure Defender

If you're not accessing Azure Security Center in your subscription for the first time, you might not be greeted with this message to enable Azure Defender. To enable it, click on **Pricing & settings** on the left-hand side and select your subscription, as shown in *Figure 13.6*:

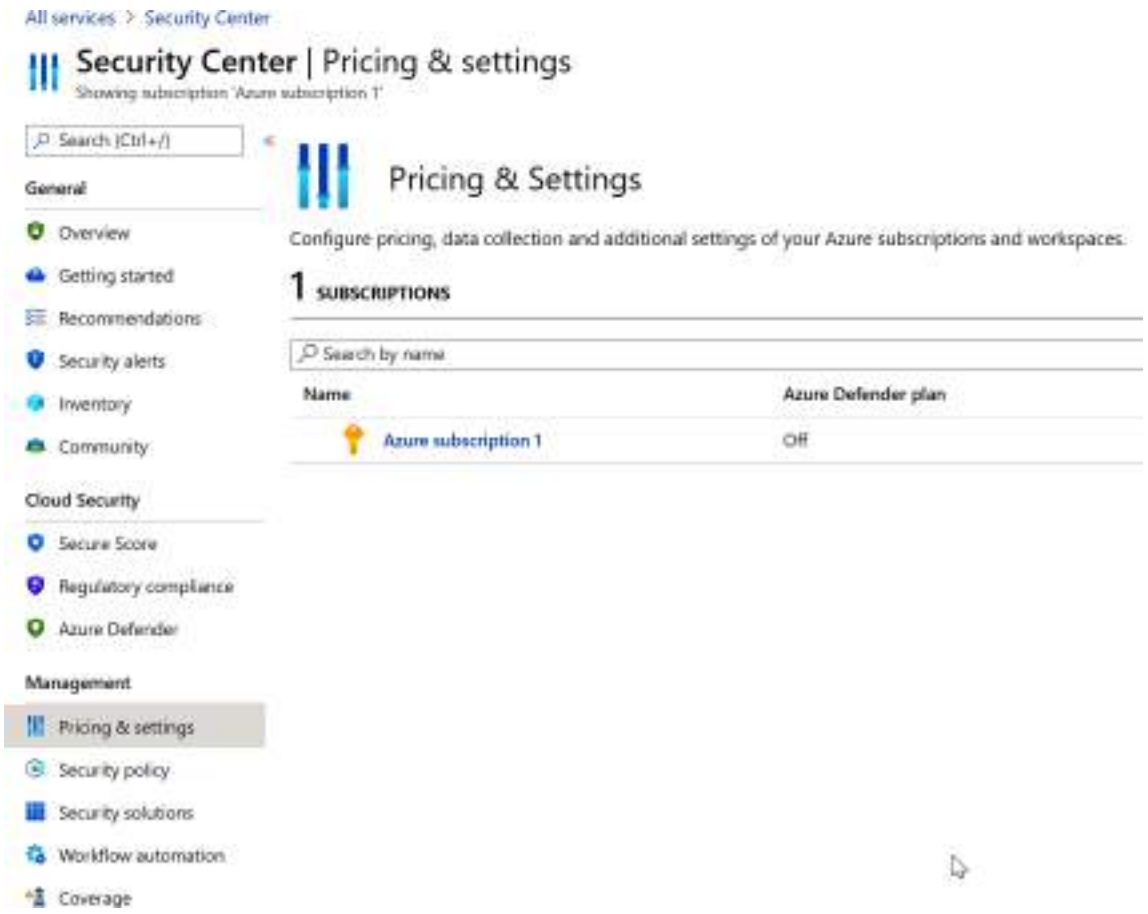


Figure 13.6: Manually upgrading to Azure Defender

In the resulting pane, select the right box with the title **Azure Defender on** and click the **Save** button at the top of the screen to enable Azure Defender. Optionally, you could tune which service you want to enable/disable Azure Defender for, as shown in *Figure 13.7*:

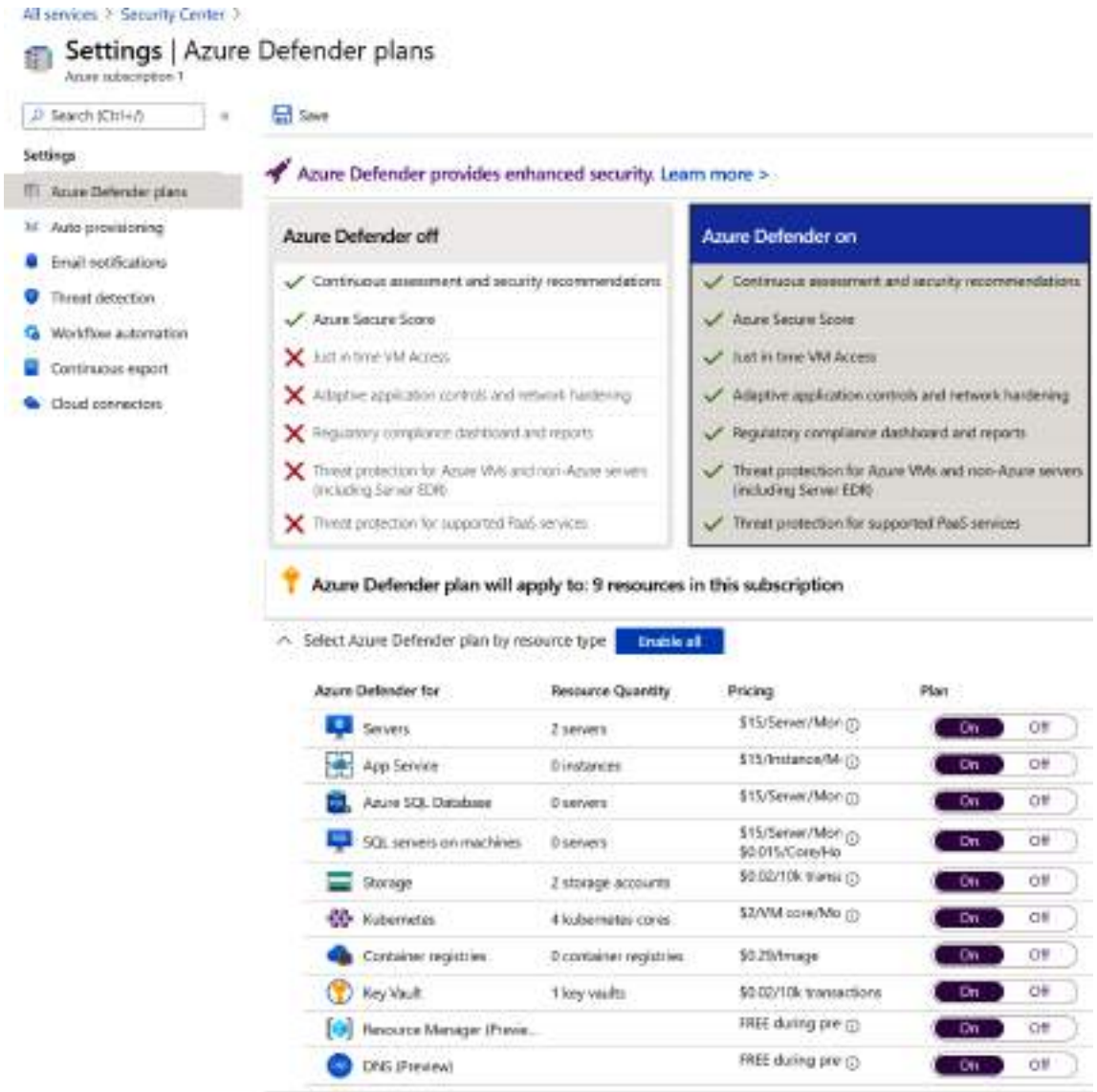


Figure 13.7: Turning on Azure Defender for your subscription

Now that you have enabled Azure Security Center and Azure Defender, it will take up to 30 minutes for the system to configure the default policy and start detections.

While you are waiting for this configuration to become effective, you will deploy several offending workloads on your cluster, which will trigger alerts in Azure Defender.

Deploying offending workloads

To trigger recommendations and threat alerts in Azure Security Center, you will need to have offending workloads deployed on your cluster. In this section, you will deploy a number of workloads to your cluster that are either not configured according to best practices or even contain potentially malicious software such as crypto-miners. Let's have a look at the examples of offending workloads you can find in the code samples for this chapter:

- `crypto-miner.yaml`: This file contains a deployment that will create a crypto-miner on your cluster.

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: crypto-miner
5     labels:
6       app: mining
7   spec:
8     replicas: 1
9     selector:
10      matchLabels:
11        app: mining
12    template:
13      metadata:
14        labels:
15          app: mining
16      spec:
17        containers:
18        - name: mining
19          image: kannix/monero-miner:latest
```

This file is a regular deployment in Kubernetes. As you can see on *line* 19, the container image for this deployment will be a crypto-miner.

> **Note**
>
> Make sure to stop running the crypto-miner as soon as you are done with this chapter, as explained in the Neutralizing threats using Azure Defender section. There is no point in running the crypto-miner any longer than this example requires.

- `escalation.yaml`: This file contains a deployment that allows privilege escalations in the container. This means that a process in the container can get access to the host operating system. There are cases where this is desired behavior, but typically you don't want this configuration.

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: escalation
5     labels:
6       app: nginx-escalation
7   spec:
8     replicas: 1
9     selector:
10      matchLabels:
11        app: nginx-escalation
12    template:
13      metadata:
14        labels:
15          app: nginx-escalation
16      spec:
17        containers:
18        - name: nginx-escalation
19          image: nginx:alpine
20          securityContext:
21            allowPrivilegeEscalation: true
```

As you can see in the preceding code sample, on *lines 20–21*, you configure the security context of the container with `securityContext`. You allow privilege escalation on *line 21*.

- `host-volume.yaml`: This file contains a deployment with a directory on the host mounted in the container. This is not recommended because this way, the container can get access to the host.

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: host-volume
5     labels:
6       app: nginx-host-volume
7   spec:
8     replicas: 1
9     selector:
10      matchLabels:
11        app: nginx-host-volume
12    template:
13      metadata:
14        labels:
15          app: nginx-host-volume
16      spec:
17        containers:
18        - name: nginx-host-volume
19          image: nginx:alpine
20          volumeMounts:
21          - mountPath: /test-pd
22            name: test-volume
23            readOnly: true
24        volumes:
25        - name: test-volume
26          hostPath:
27            # Directory on host
28            path: /tmp
```

This code sample contains a `volumeMount` field and a volume. As you can see in *lines 24–28*, the volume is using `hostPath`, meaning it mounts a volume on the node running the container.

- `role.yaml`: A role with very broad permissions. It is recommended to approach roles in Kubernetes with the principle of least privilege to ensure permissions are tightly controlled. A role with broad permissions is first and foremost a bad configuration, but worse, could be a sign of compromise on your cluster.

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: ClusterRole
3   metadata:
4     name: super-admin
5   rules:
6   - apiGroups: ["*" ]
7     resources: ["*"]
8     verbs: ["*"]
```

This instance of `ClusterRole` gives very broad permissions, as you can see in *lines 6–8*. This configuration gives anybody who gets assigned this role all permissions on all resources on all APIs in Kubernetes.

None of the deployments in these code samples contain resource requests and limits. As explained in *Chapter 3*, *Application deployment on AKS*, it is recommended to configure resource requests and limits, as these can prevent a workload from consuming too many resources.

Finally, you will also deploy the Kubernetes dashboard on a public service. This is also highly discouraged, as this might inadvertently give attackers access to your cluster. You will see how Azure Defender detects this.

Let's start deploying these files.

1. Open Cloud Shell in the Azure portal and navigating to the code samples for this chapter.

2. Once there, execute the following commands to create the offending workloads.

```
kubectl create -f crypto-miner.yaml
kubectl create -f escalation.yaml
kubectl create -f host-volume.yaml
kubectl create -f role.yaml
```

This will create an output similar to *Figure* 13.8:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter13$ kubectl create -f crypto-miner.yaml
deployment.apps/alert-test created
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter13$ kubectl create -f escalation.yaml
deployment.apps/escalation created
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter13$ kubectl create -f host-volume.yaml
deployment.apps/host-volume created
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter13$ kubectl create -f role.yaml
clusterrole.rbac.authorization.k8s.io/super-admin created
```

Figure 13.8: Creating the offending workload

3. Now, deploy the Kubernetes dashboard using the following command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/
dashboard/v2.0.0/aio/deploy/recommended.yaml
```

This will create an output similar to *Figure* 13.9:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter13$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

Figure 13.9: Creating the Kubernetes dashboard

4. By default, the Kubernetes dashboard is not exposed through a load balancer. This is also the recommended configuration, because the dashboard gives broad access to your cluster. In this chapter, however, you will create this discouraged configuration to trigger a security alert in Azure Defender. To add a load balancer to the Kubernetes dashboard, use the following command:

```
kubectl patch service \
  kubernetes-dashboard -n kubernetes-dashboard \
  -p '{"spec": {"type": "LoadBalancer"}}'
```

   This will patch the service and turn it into a service of the `LoadBalancer` type. Verify that this was patched successfully and get the public IP address of the service using the following command:

```
kubectl get service -n kubernetes-dashboard
```

   This will generate an output similar to *Figure 13.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter13$ kubectl get service -n kubernetes-dashboard
NAME                        TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)         AGE
dashboard-metrics-scraper   ClusterIP      10.0.53.98     <none>         8000/TCP        3m30s
kubernetes-dashboard        LoadBalancer   10.0.187.130   20.190.3.170   443:31673/TCP   3m31s
```

Figure 13.10: Getting the public IP of the kubernetes-dashboard service

5. Verify that you can access this service by browsing to `https://<public IP>`. Depending on your browser configuration, you might get a certificate error, which you can bypass by selecting **Continue to <public IP> (unsafe)**, as shown in *Figure 13.11*:

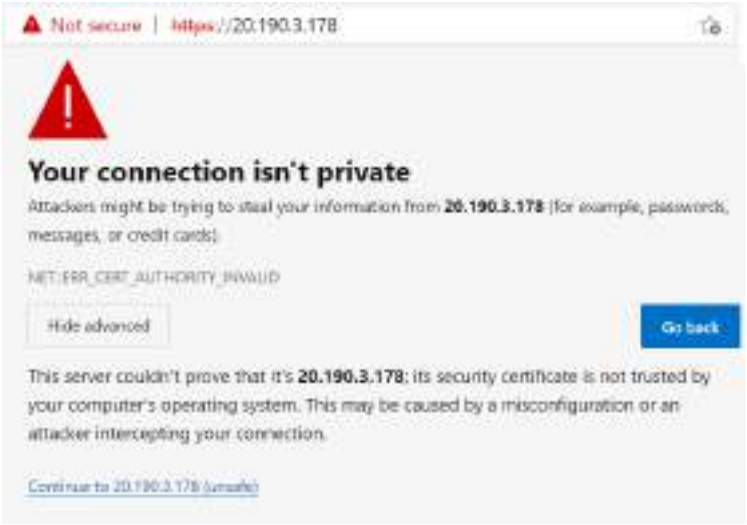


Figure 13.11: Security warning about the certificate on the Kubernetes dashboard service

Once you have continued to the dashboard, you should get a sign-in screen as shown in *Figure 13.12*:

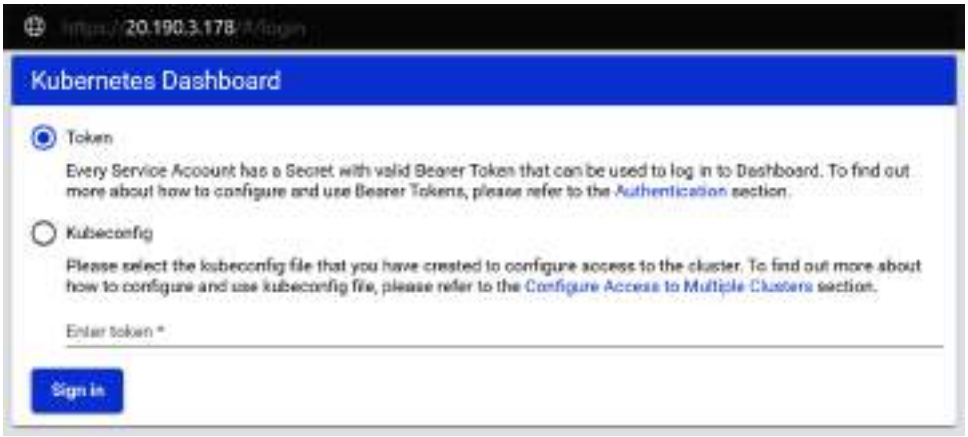


Figure 13.12: The exposed Kubernetes dashboard

You won't sign in to the dashboard here, but if you wish to explore its capabilities, please refer to the Kubernetes documentation at https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.

You now have five offending workloads running on your cluster. Some of these will cause configuration warnings in Azure Security Center; some others will even trigger a security alert. You will explore those in the next two sections of this chapter.

Analyzing configuration using Azure Secure Score

In the previous section, you created several workloads that are purposefully misconfigured. In this section, you'll review the recommendations in Azure Security Center related to these workloads.

> **Note**
>
> It can take up to 30 minutes after the workloads have been created for recommendations and alerts to show up.

1. After you have created the offending workloads, you will get security recommendations in Azure Security Center. To start, click on **Secure Score** in the left-hand navigation within Azure Security Center. This will show you a pane similar to *Figure 13.13*:

Figure 13.13: Secure Score in Azure Security Center

What you see here is a summary of the security posture of your environment. In the example shown in *Figure* 13.13, you see that the overall secure score was **32%**. If you manage multiple Azure subscriptions, this view can give you a quick bird's-eye view of the security configuration of your environment.

2. Let's drill down into the configuration of the Kubernetes cluster by clicking on the **Azure subscription 1** subscription at the bottom of *Figure* 13.13. This will bring you to a pane similar to *Figure* 13.14:

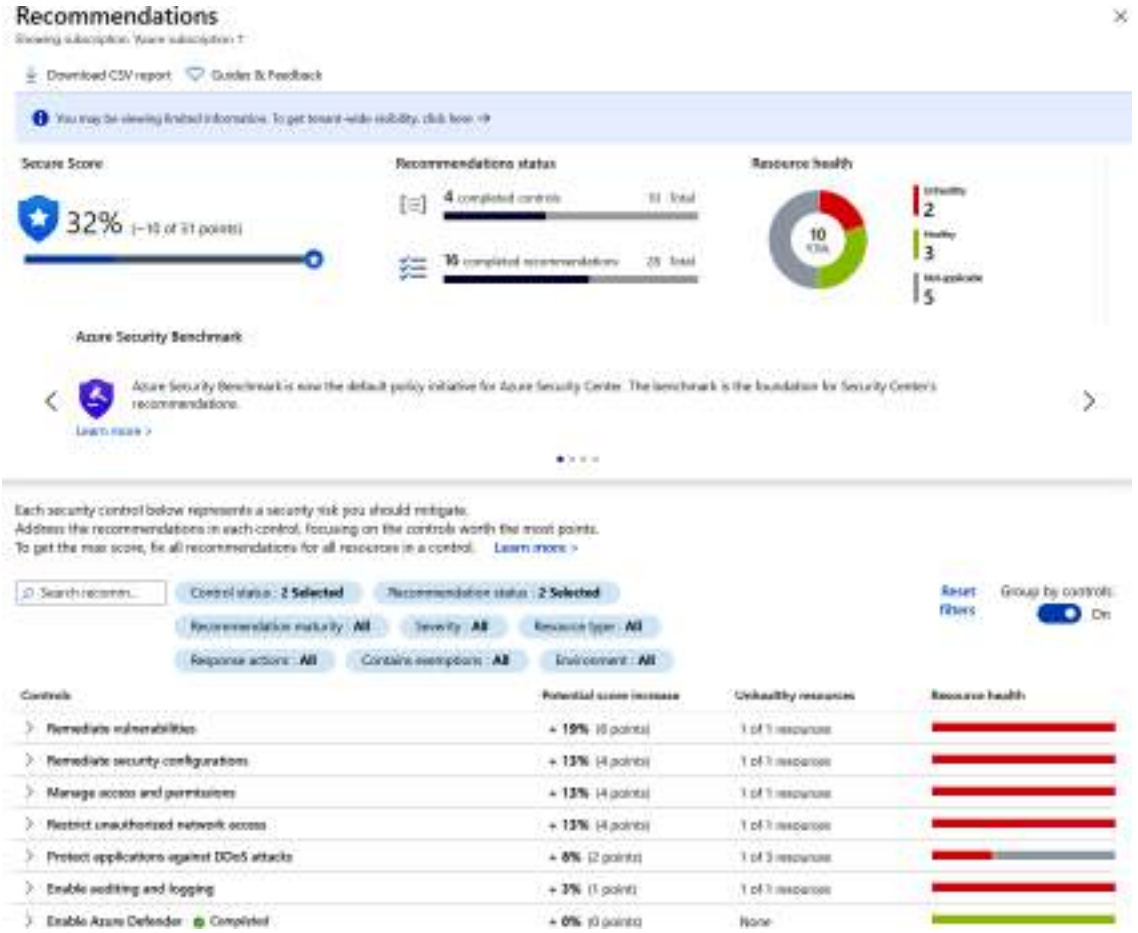


Figure 13.14: Secure Score details for the subscription

This view contains more details about the secure score for this subscription. It again shows you the secure score, as well as the recommendation status and resource health. The bottom of the screen contains more details about the specific recommendations.

3. Tune this screen to get more insight into the Kubernetes recommendations. To do this, disable the **Group by controls** option on the right-hand side of the screen, and set the **Resource type** filter to **managed cluster**, as shown in *Figure 13.15*:

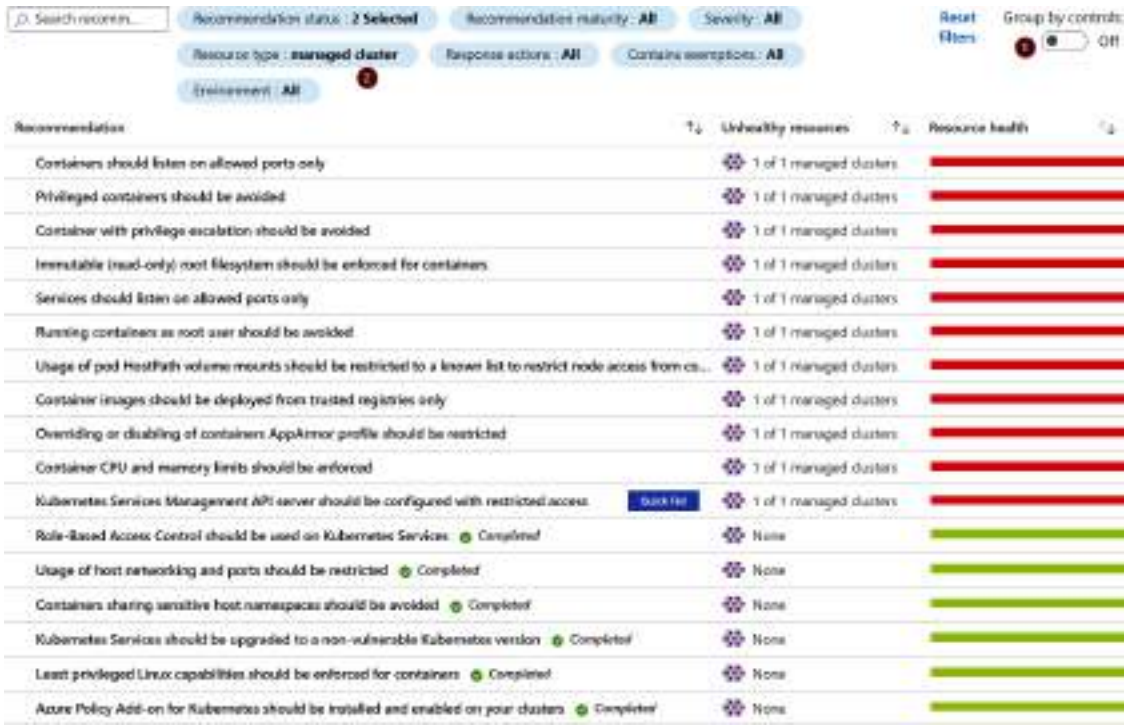


Figure 13.15: Kubernetes recommendations in Azure Security Center

You are now looking at a list of Kubernetes security recommendations recommended by Azure Security Center. The list is too exhaustive to cover completely in this chapter, but if you want to see more details about each recommendation, please refer to the AKS documentation for more detailed descriptions: https://docs.microsoft.com/azure/aks/policy-reference.

4. Let's, however, explore a number of the recommendations that were caused by the offending workloads you created earlier. Start by clicking on the recommendation called **Container with privilege escalation should be avoided**. This will bring you to a view similar to *Figure 13.16*:

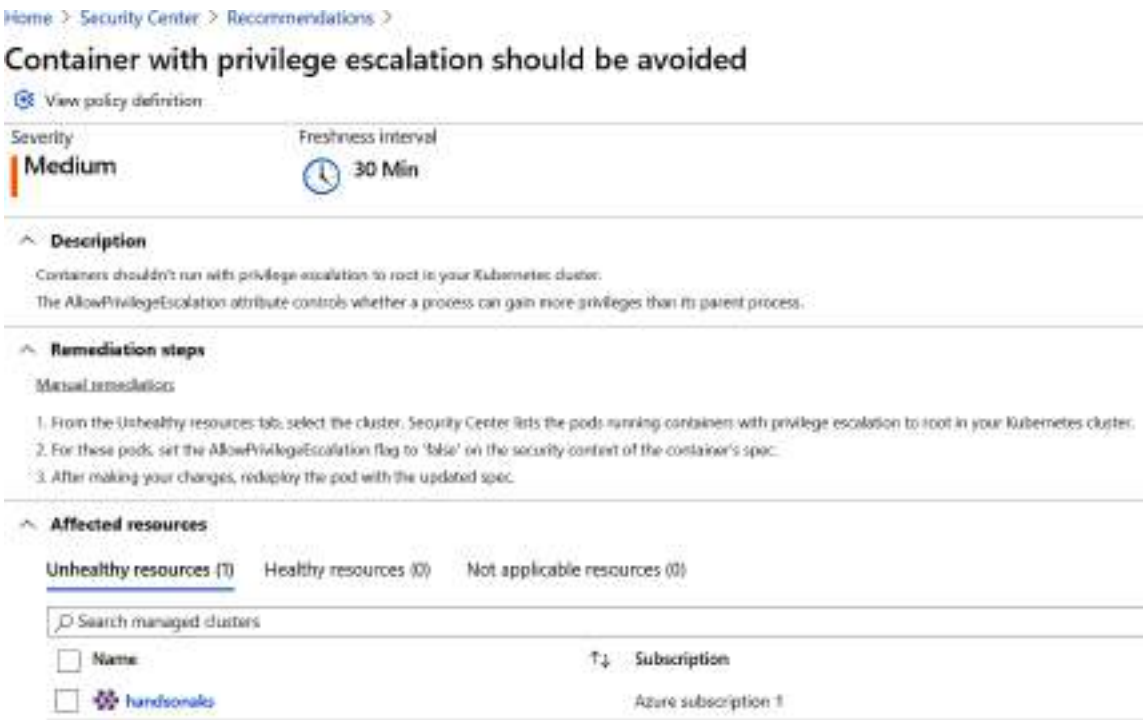


Figure 13.16: Details of the Container with privilege escalation should be avoided recommendation

As you can see, this recommendation contains a description of the recommendation itself, as well as a number of remediation steps to follow this recommendation. It also shows you the affected resource, which in this case is an AKS cluster. If you click on the cluster, you will even get more details about the offending workload, as shown in *Figure 13.17*:

Figure 13.17: Pods affected by the privilege escalation recommendation

In this case, each of the pods you created triggered this recommendation, not just the one where privilege escalation was allowed. This shows you that Kubernetes allows this privilege escalation by default and you should implement the safeguard in all your deployments. This shows you the benefit of a security monitoring solution such as Azure Security Center, to monitor against potential side effects of the default configuration.

5. Let's apply the suggested remediation to this issue. To solve the privilege escalation, you will need to configure the security context of the container to no longer allow privilege escalation. This can be done by updating each deployment using the following commands:

```
kubectl patch deployment crypto-miner -p '
  {
    "spec": {
      "template": {
        "spec": {
          "containers": [
            {
              "name": "mining",
              "securityContext": {
                "allowPrivilegeEscalation": false
              }
            }
          ]
        }
      }
    }
```

```
  }
  '
kubectl patch deployment escalation -p '
  {
    "spec": {
      "template": {
        "spec": {
          "containers": [
            {
              "name": "nginx-escalation",
              "securityContext": {
                "allowPrivilegeEscalation": false
              }
            }
          ]
        }
      }
    }
  }
  '
kubectl patch deployment host-volume -p '
  {
    "spec": {
      "template": {
        "spec": {
          "containers": [
            {
              "name": "nginx-host-volume",
              "securityContext": {
                "allowPrivilegeEscalation": false
              }
            }
          ]
        }
      }
    }
  }
  '
```

As you can see in the command, you are patching each deployment. In each patch, you are configuring the `securityContext` field and setting the `allowPrivilegeEscalation` field to `false`.

After you have applied the patch, it will take Azure Security Center up to 30 minutes to refresh the security recommendation. After that time passes, your cluster should show up as a healthy resource for this recommendation, as shown in *Figure 13.18*:

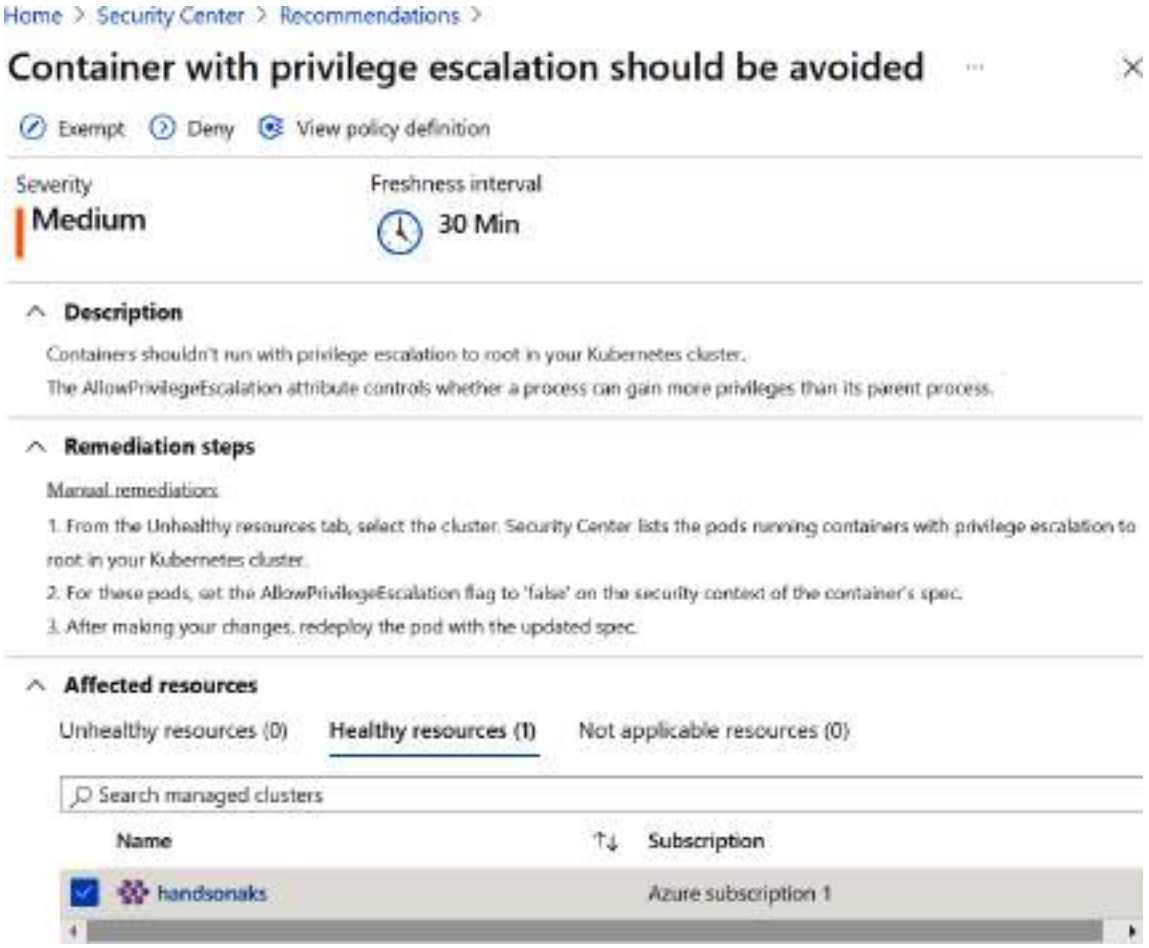


Figure 13.18: Cluster is now healthy for the privilege escalation recommendation

6. Let's investigate another recommendation, namely the one called **Usage of pod HostPath volume mounts should be restricted to a known list to restrict node access from compromised containers**. Click on this recommendation to be shown more details, as shown in *Figure 13.19*:

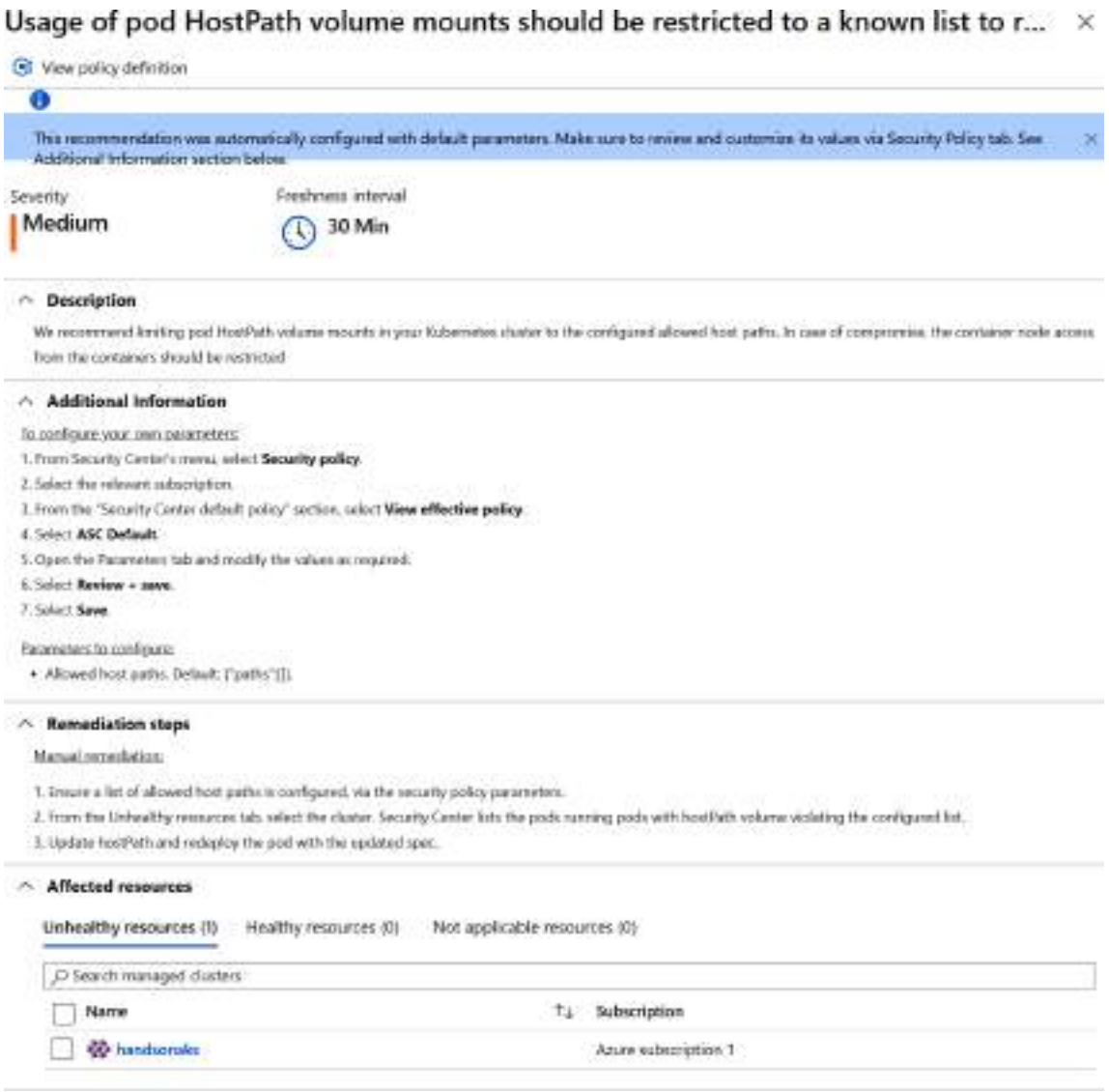


Figure 13.19: More details about the HostPath recommendation

This recommendation shows you similar information to the previous one. However, the policy that triggered this recommendation can be tuned to allow certain `HostPath` to be accessed. Let's explore how to edit the policy to allow this.

7. Go back to the Azure Security Center main pane and click on **Security policy** on the left-hand side. In the resulting pane, click on your subscription and then click on the **ASC Default** assignment shown in *Figure 13.20*:

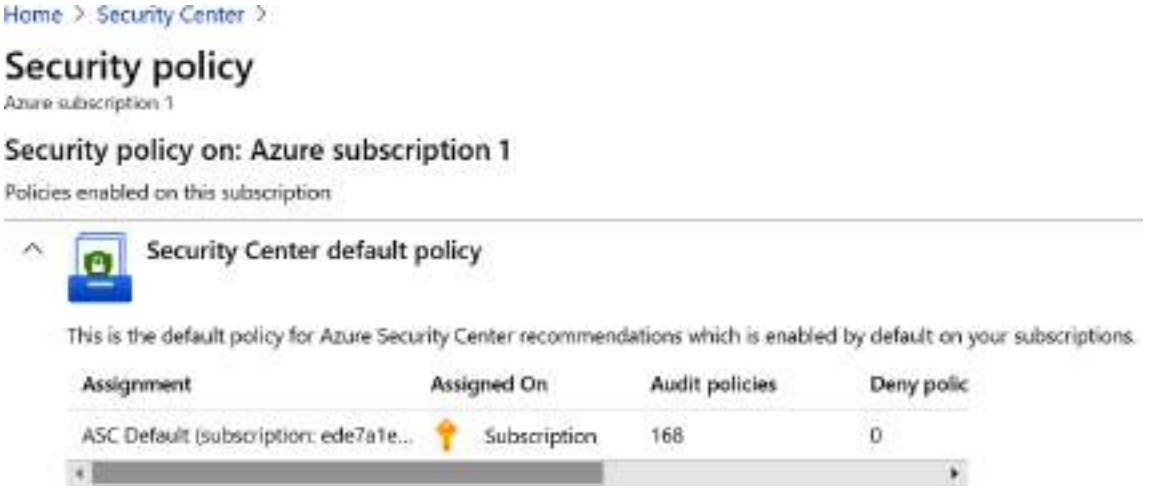


Figure 13.20: ASC default policy assignment

8. In the resulting pane, click on **Parameters** at the top of the screen. Look for **Allowed host paths** in the list of parameters and change that parameter to the following:

```
{  "paths": ["/tmp"]}
```

The result is shown in *Figure 13.21*:

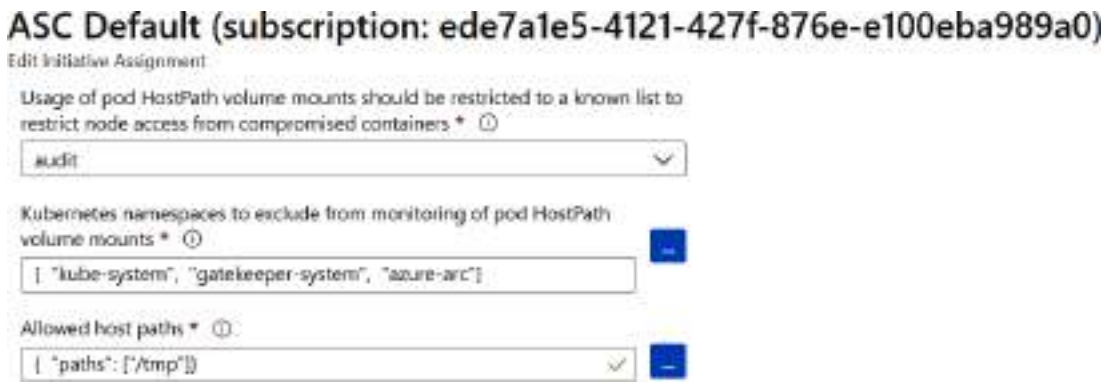


Figure 13.21: Adding a path to the allowed host paths

To apply the changes, click **Review + save** at the bottom of the screen and then click **Save** on the final screen:

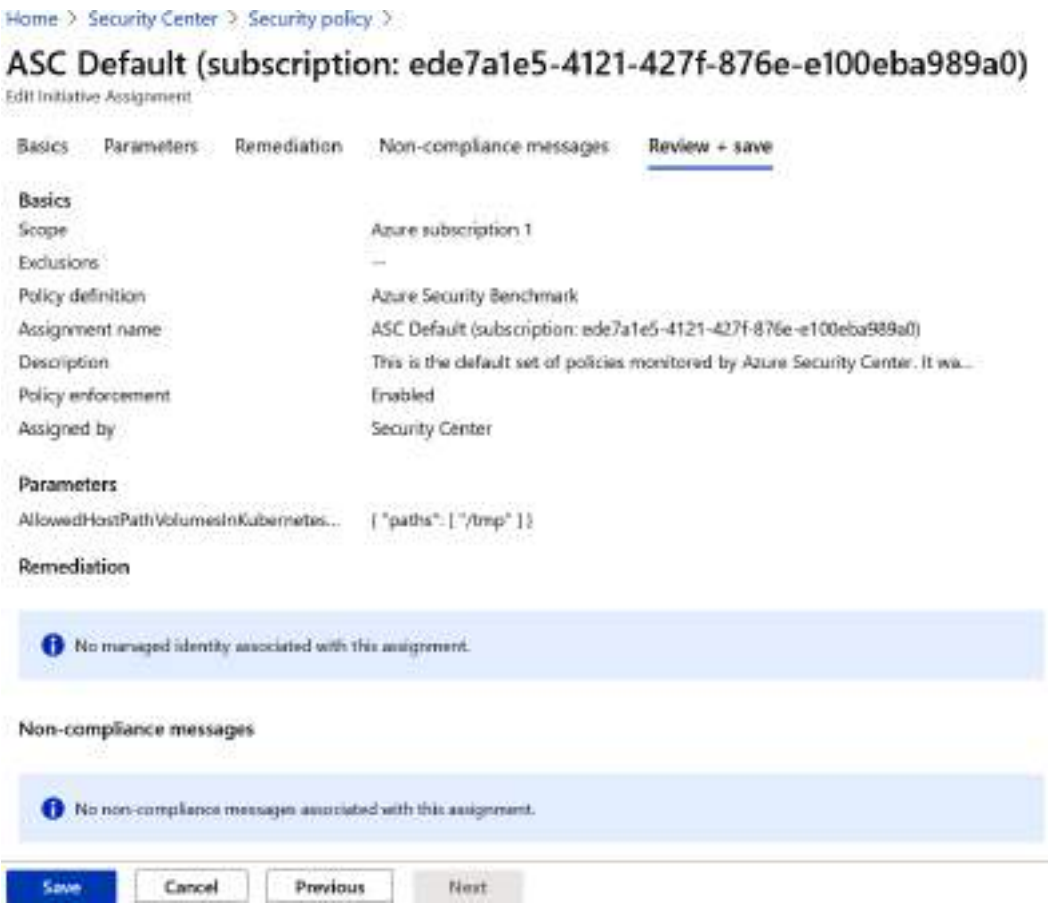


Figure 13.22: Clicking Save to confirm the policy change

It will now take about 30 minutes for the recommendation to be refreshed. After 30 minutes, this recommendation will no longer be active, since you configured the `/tmp` path to be allowed.

9. There's one final recommendation worth highlighting in this list. That is **Kubernetes Services Management API server should be configured with restricted access**. If you remember, in *Chapter 11*, *Network security in AKS*, you configured authorized IP ranges on your cluster. This is recommended by Microsoft, and also shows up as an Azure Security Center recommendation. Click on that recommendation to get more details, as shown in *Figure 13.23*:

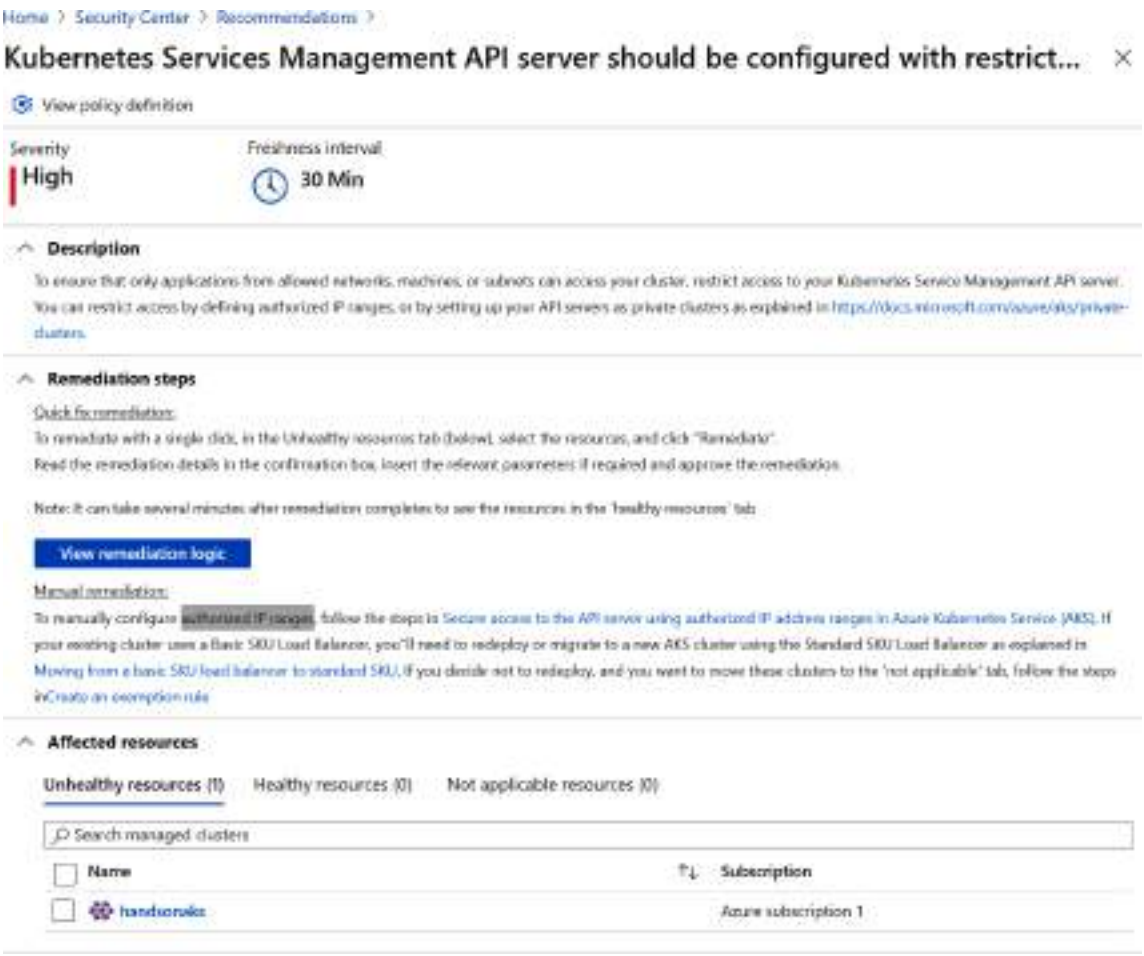


Figure 13.23: Details explaining that authorized IP ranges should be enabled

As you can see, again you get a description of the recommendation and remediation steps. The reason this recommendation is worth highlighting is that it contains a quick fix remediation within Azure Security Center. To quickly remediate this recommendation, select your AKS cluster and click **Remediate** at the bottom of the screen, as shown in *Figure 13.24*:

Figure 13.24: Remediating the authorized IP recommendation

This will show you a view similar to *Figure 13.25* where you could input the IP ranges you need to authorize access from.

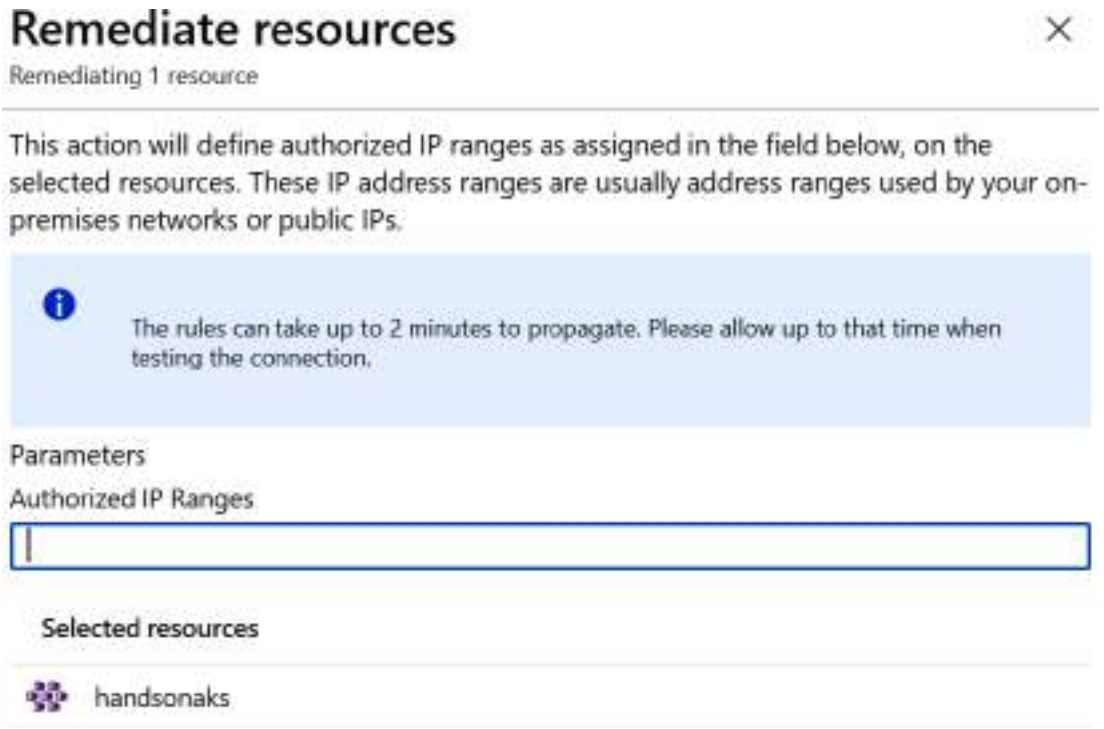


Figure 13.25: Setting up authorized IP ranges through Azure Security Center

You could configure this configuration from within Azure Security Center. Since you'll be using Cloud Shell in the next steps and next chapters and Cloud Shell does not have a fixed IP, it's not recommended to apply the remediation while working through this book. It is, however, worthwhile to show that Azure Security Center allows you to remediate certain configuration recommendations directly from within Security Center.

You have now explored the recommendations and the secure score in Azure Security Center. If you want to learn more about this, please refer to the Azure documentation, which contains a YAML deployment example fully configured with all the recommendations: https://docs.microsoft.com/azure/security-center/kubernetes-workload-protections.

Neutralizing threats using Azure Defender

Now that you've explored the configuration best practices using Azure Security Center and Secure Score, you will explore how to investigate and deal with security alerts and active threats. Some of the workloads you created should have triggered security alerts by now, which you can investigate in Azure Defender.

Specifically, in the *Deploying offending workloads* section, you created three workloads that trigger security alerts in Azure Defender:

- `crypto-miner.yaml`: By deploying this file, you created a crypto-miner on your cluster. This crypto-miner will generate two security alerts in Azure Defender as you will see in this section. One alert will be generated by monitoring the Kubernetes cluster itself, while another alert will be generated based on DNS traffic.
- `role.yaml`: This file contained a cluster-wide role with very broad permissions. This will generate a security alert in Azure Defender notifying you of the risk.
- Kubernetes dashboard: You also created the Kubernetes dashboard and exposed this publicly. Azure Defender will also generate a security alert based on this.

Let's explore each of these security alerts in details:

1. To start, in Azure Security Center, click on **Azure Defender** in the left-hand navigation bar. This will open the Azure Defender pane in Azure Security Center. This shows you your coverage, your security alerts, and the advanced protection options, as shown in *Figure* 13.26. In this section, you will focus on the four security alerts that were generated.

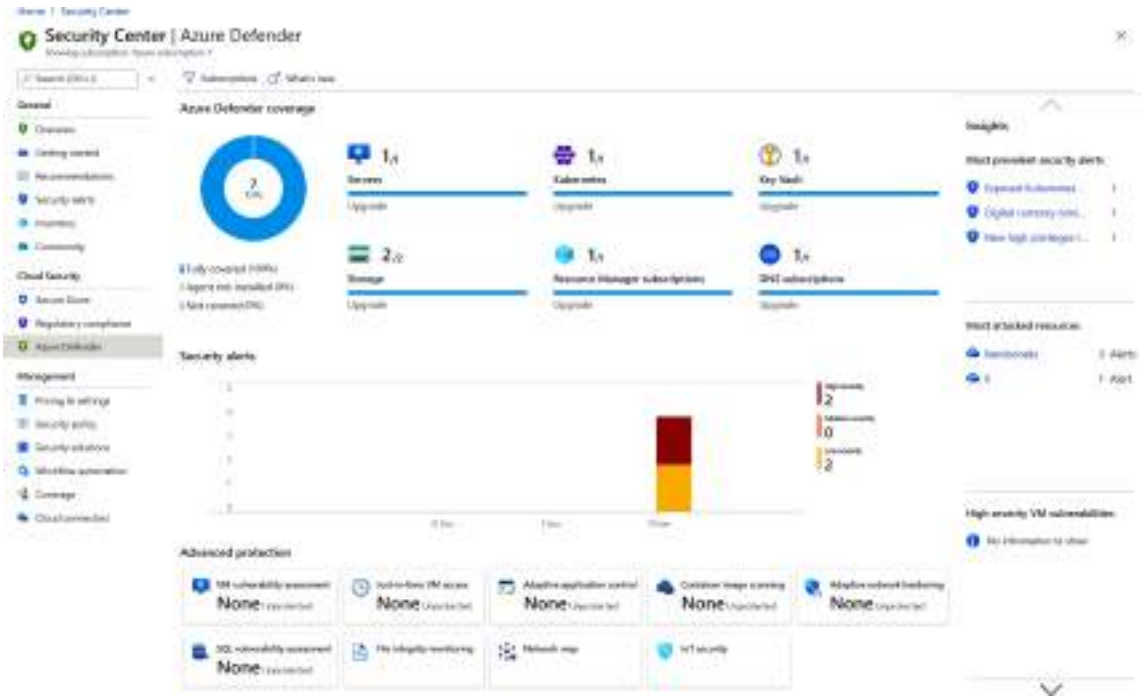


Figure 13.26: Azure Defender - Overview pane

2. To get more details about the security alerts, click anywhere on the **Security alerts** bar chart in the middle of the screen. That will take you to a new pane, as shown in *Figure* 13.27:

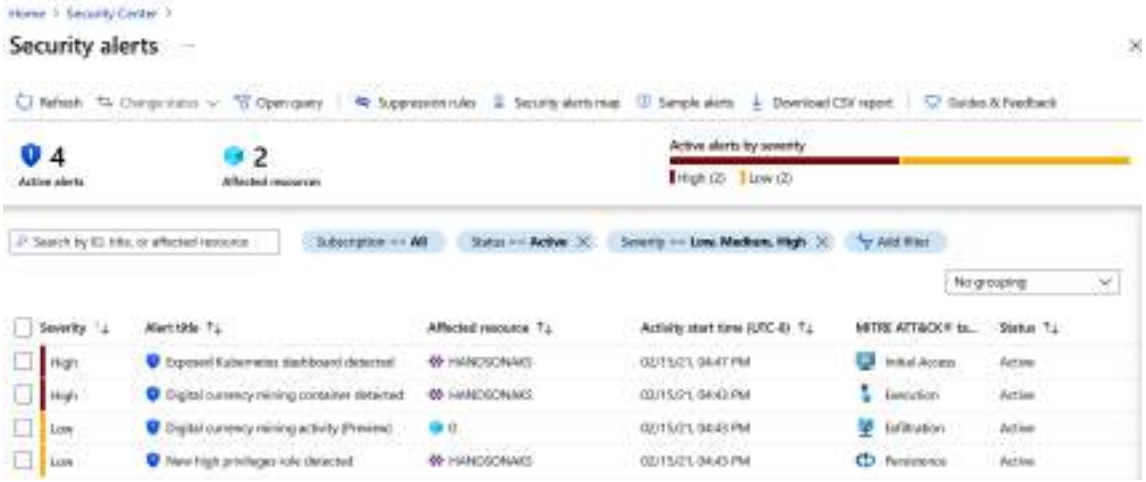


Figure 13.27: Security alerts in Azure Defender

As you can see in *Figure 13.27*, four security alerts have been triggered: one for the exposed Kubernetes dashboard, two for the crypto-miner, and one for the high-privileged roles. Let's explore each one in more detail.

3. Let's start by exploring the **Exposed Kubernetes dashboard detected** alert. Click on the alert title to get more details. To see all the details, click on **View full details** in the resulting pane, as shown in *Figure 13.28*:

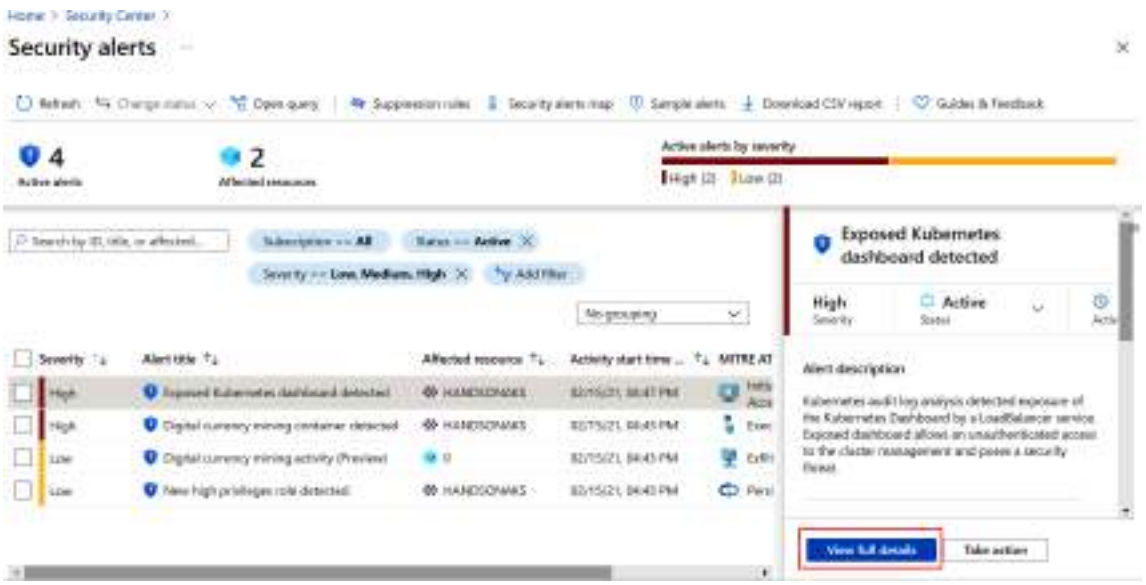


Figure 13.28: Getting full details of the alert

This will take you to a new pane, as shown in *Figure* 13.29:

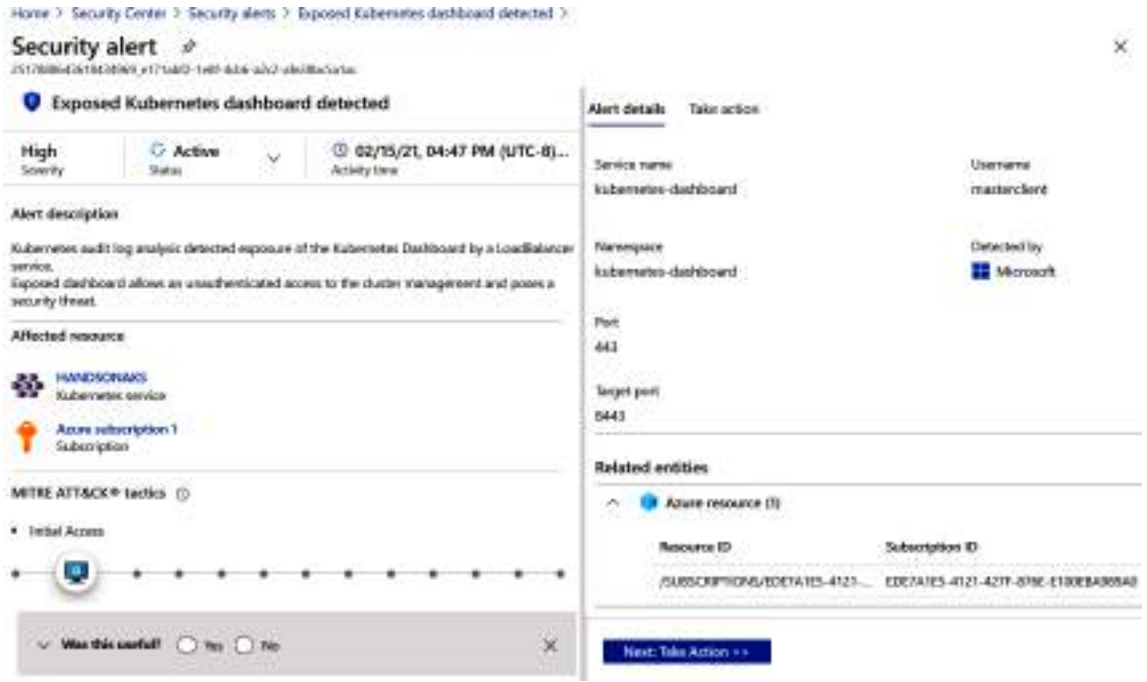


Figure 13.29: Details of the Exposed Kubernetes dashboard detected alert

This shows you a couple of information points. First, it classifies this alert as high severity, marks it as active, and also marks when it was first encountered. Next, you get a description of the alert, which in this case explains that the dashboard should not be exposed publicly. It also shows you the affected resource. Finally, you see which stage of the MITRE ATT&CK® tactics framework this attack targets. The MITRE ATT&CK® tactics framework is a framework describing multiple stages in a cyber-attack. For more information on MITRE ATT&CK® tactics, please refer to https://attack.mitre.org/versions/v7/.

On the right-hand side of the screen, you get more details about the alert. This contains the service name, the namespace, the port of the service, the target port exposed on the backend pods, and the affected Azure resource ID and subscription ID. If you click the **Next: Take Action >>** button at the bottom of the screen, you'll be taken to a new pane, as shown in *Figure* 13.30:

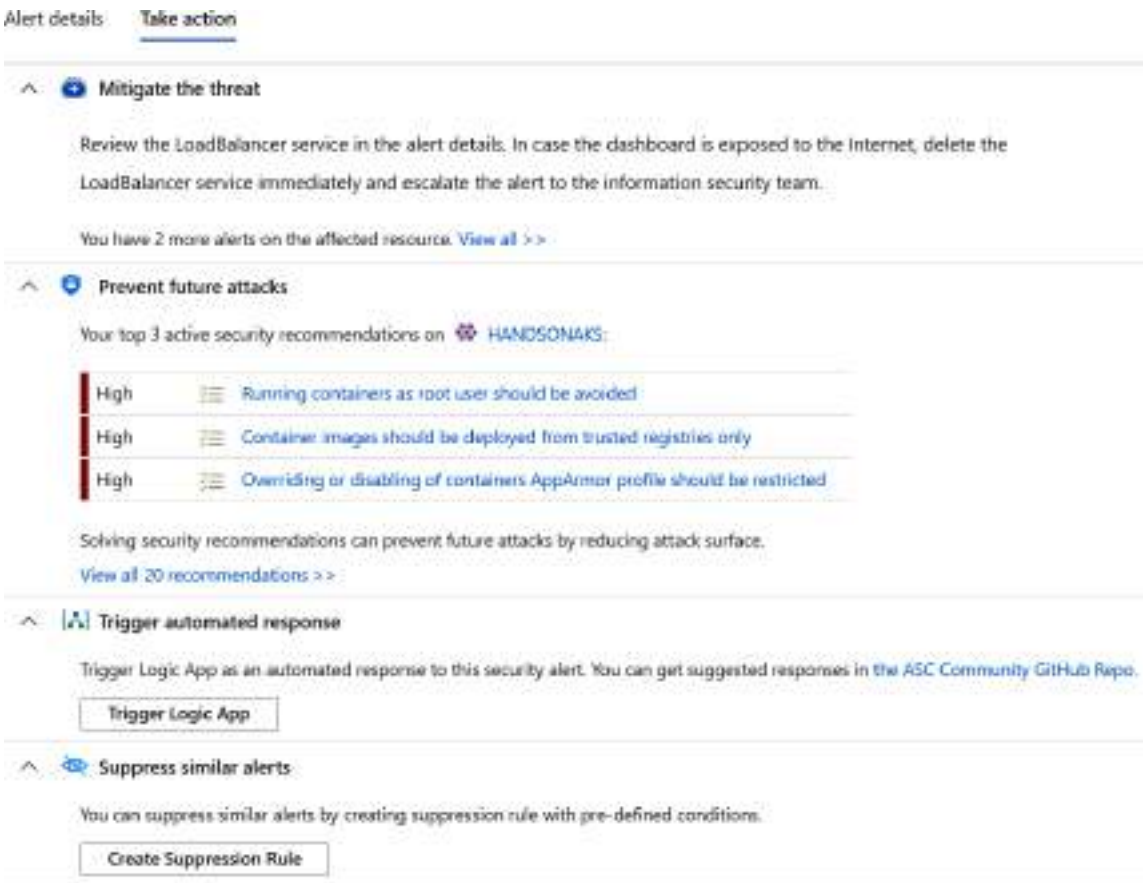


Figure 13.30: Security recommendations on the dashboard alert

In the **Take action** pane, you get information on how to mitigate the threat and how to prevent future attacks like this. Notice how the **Prevent future attacks** section contains a link to the security recommendations you reviewed in the previous section.

4. Let's take the suggested action and update the Kubernetes dashboard service so it is no longer of the `LoadBalancer` type using the following command. This command will remove the `nodePort` that Kubernetes set up to expose the service through the load balancer and change the type of the service back to the `ClusterIP` type, which is only available from within the cluster.

```
kubectl patch service \
```

```
kubernetes-dashboard -n kubernetes-dashboard \
-p '{
      "spec": {
        "ports": [
          {
            "nodePort": null,
            "port": 443
          }
        ],
        "type": "ClusterIP"
      }
    }'
```

Finally, you see that you can optionally trigger automated responses or suppress similar alerts in the future in case this alert is a false positive.

5. Now that you have mitigated the threat, you can dismiss the alert. That way, others using the same subscription don't see this same alert. To dismiss the alert, click on the status on the left-hand side of the screen, select **Dismissed**, and click **OK**, as shown in *Figure 13.31*:

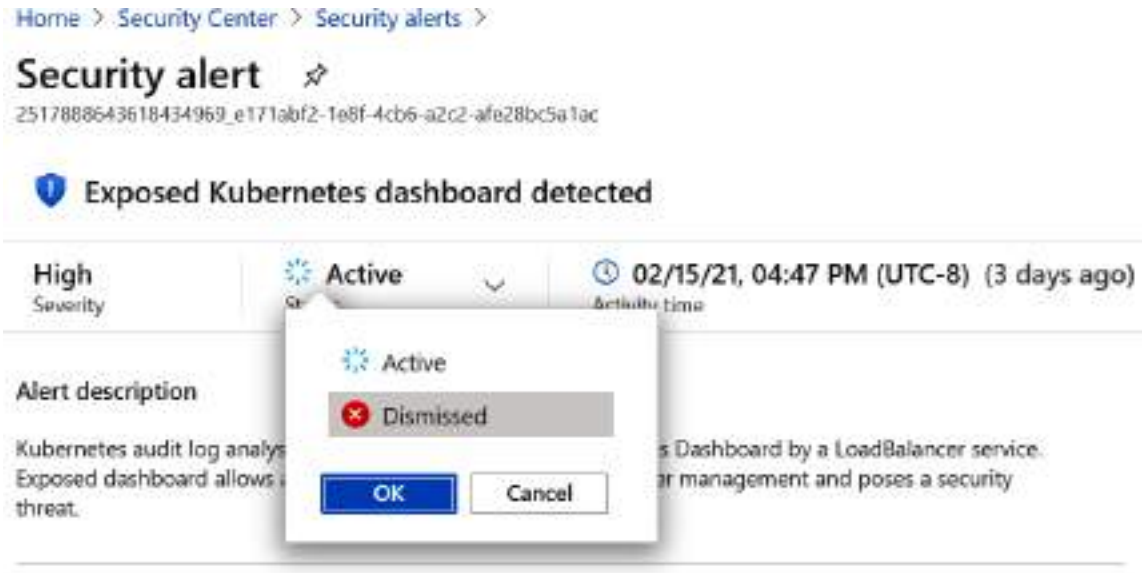


Figure 13.31: Dismissing the dashboard alert

6. Let's move on to the next alert. Close the detail panes for the dashboard alert by pressing the **X** at the top of the screen. Let's now focus on the first **Digital currency mining container detected** alert. Select that alert and click on **View full details** as you did with the previous alert. This will take you to a pane similar to *Figure 13.32*:

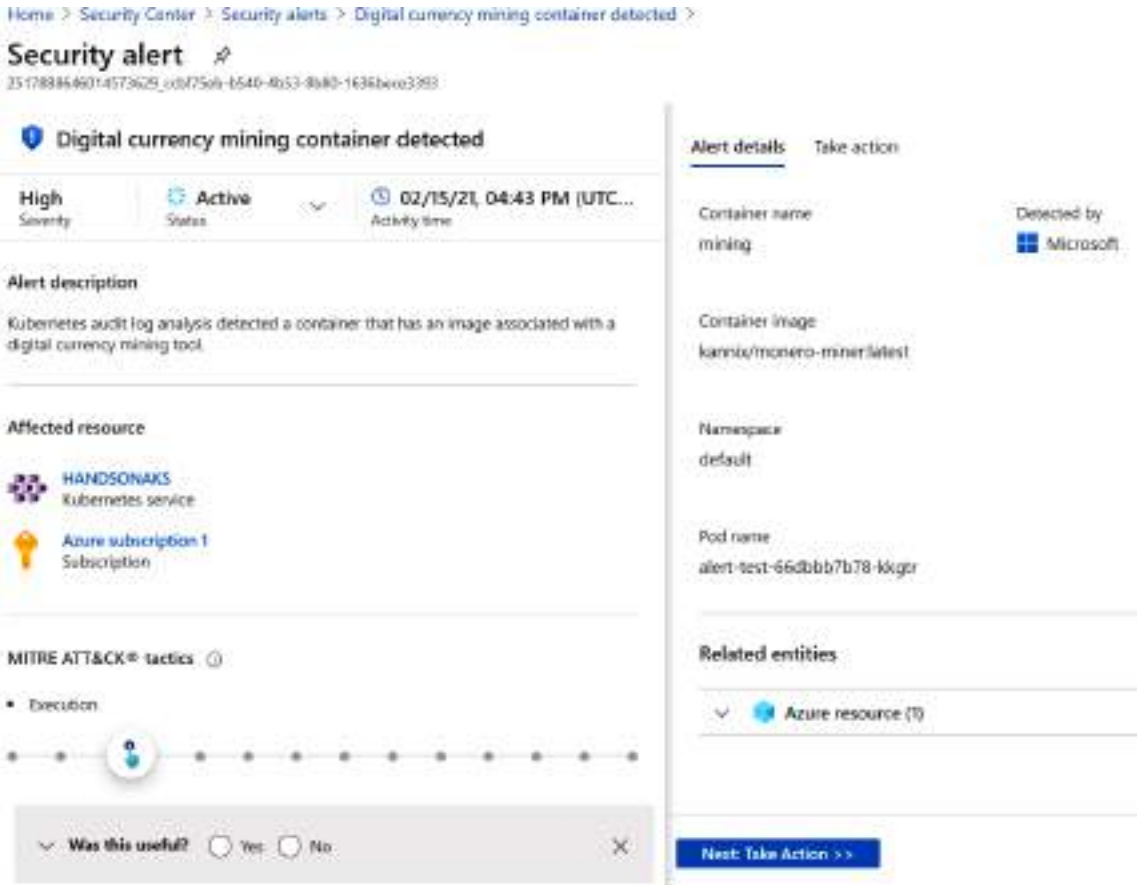


Figure 13.32: Details of the Digital currency mining container detected alert

This view contains similar details to the previous alert. As you can see, this alert is part of the Execution phase of the MITRE ATT&CK® tactics framework. On the right-hand side of the pane, you now see the name of the offending container, the image it's using, its namespace, and the name of the offending pod.

If you press the **Next: Take Action >>** button at the bottom of the screen, you'll be taken to the **Take action** view on this alert, as shown in *Figure 13.33*:

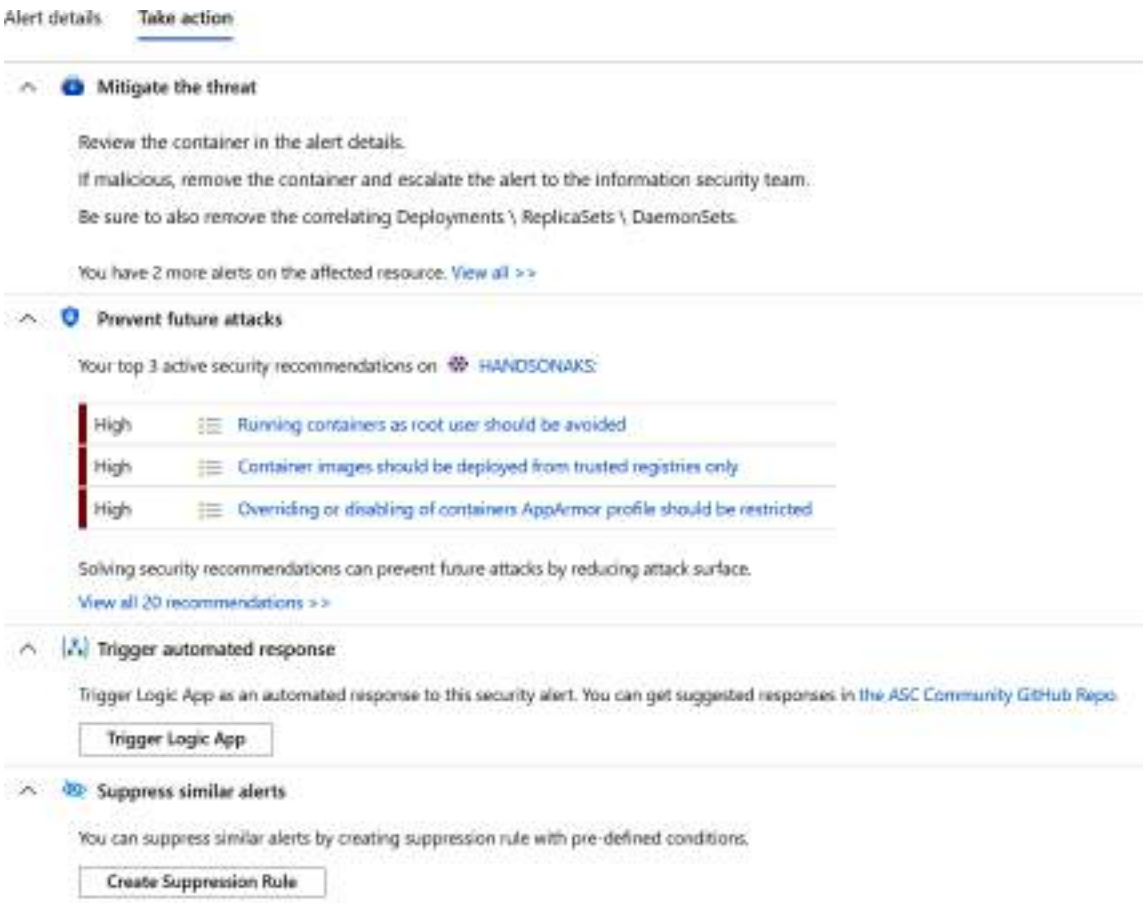


Figure 13.33: Security recommendations on the Digital currency mining container detected alert

Here, again, you see similar details to the previous alert. In the **Mitigate the threat** section, you get a different description of how to mitigate this ongoing threat. Don't take any mitigating action yet, since you'll need to explore one more alert related to the crypto-miner.

7. To explore that alert, close the detailed pane for the first crypto-miner alert by pressing the **X** at the top of the screen. Now select the second alert, which is called **Digital currency mining activity (Preview)**. This is actually not a Kubernetes alert, but an alert based on DNS, as you can see in *Figure 13.34*:

> **Note**
>
> This alert will only show up if you enabled Azure Defender for DNS. If you did not enable this, you will not get this alert.

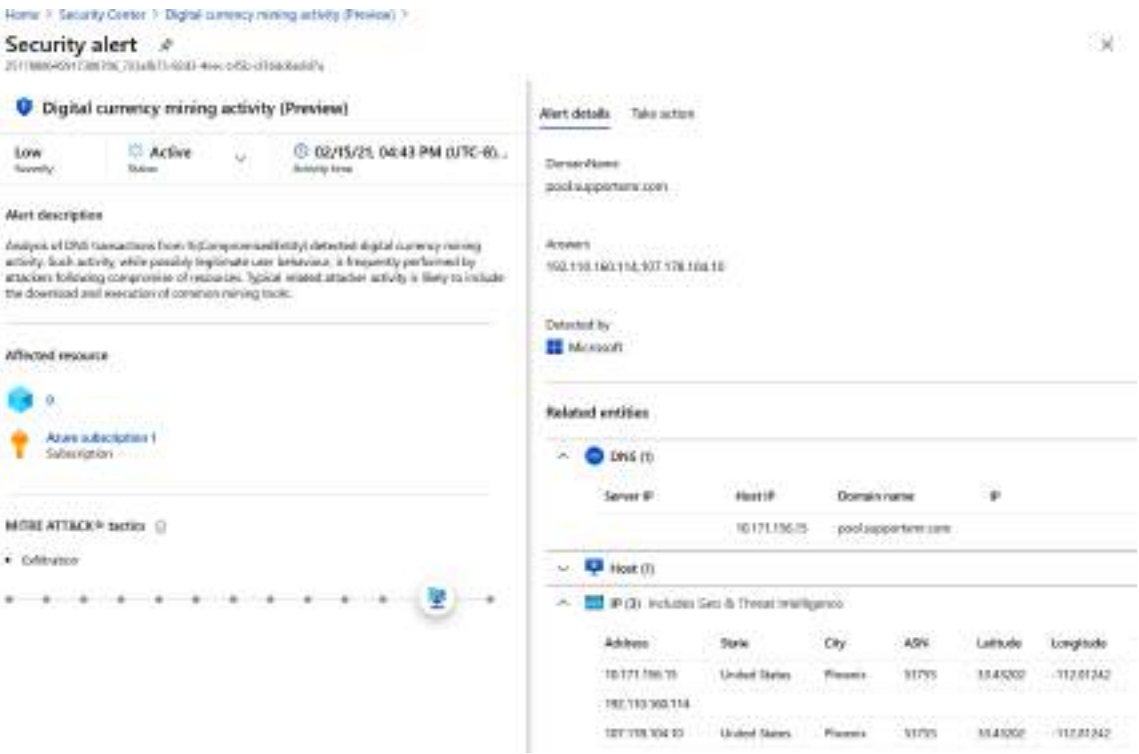


Figure 13.34: Details of the Digital currency mining activity alert

This alert was generated by Azure Defender for DNS. It shows you a number of details about the attack itself. On the right-hand side of the pane, you see more details about the attack, showing you the domain name and the IP addresses used. If you have a look at the **Take action** pane, you get more information about potential next steps for this attack, as shown in *Figure 13.35*:

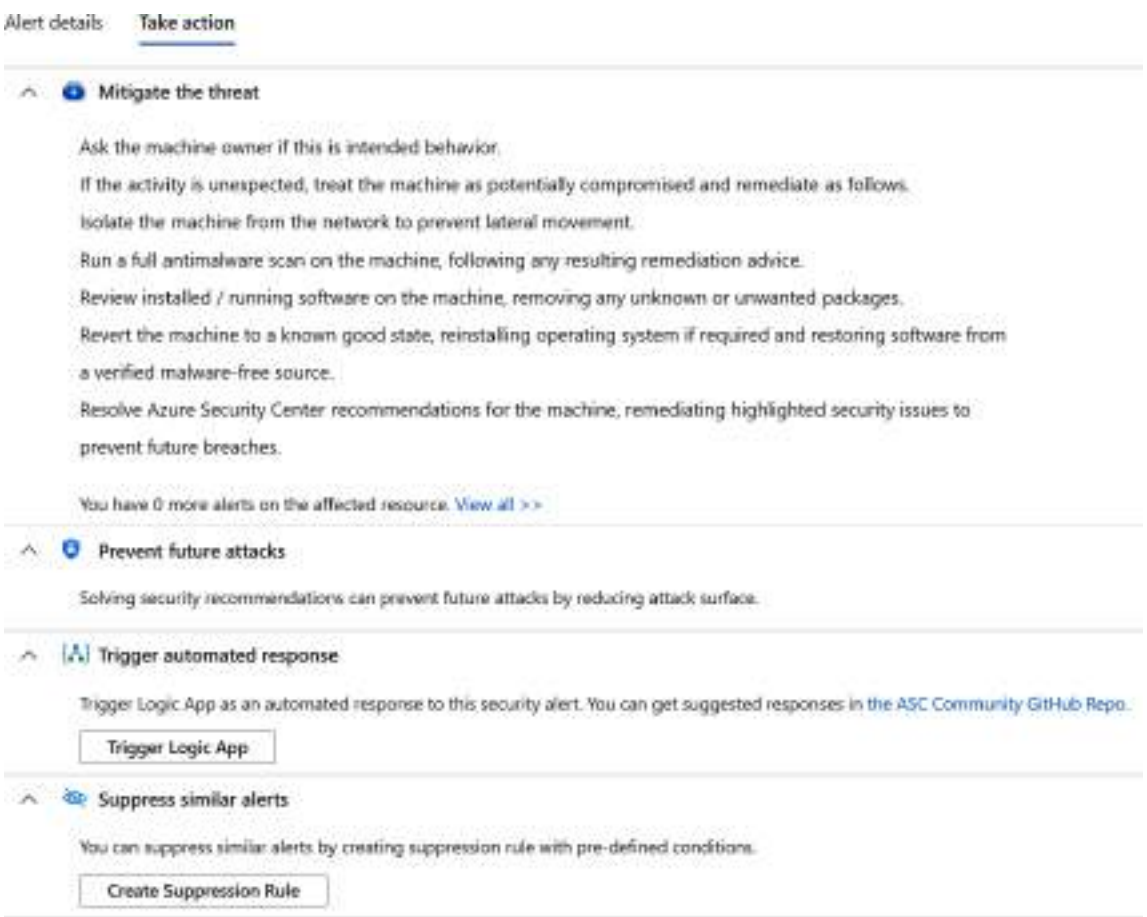


Figure 13.35: Security recommendations on the curreny mining alert

Since this is a DNS-based alert, there are limited specifics here on which processes to inspect. However, Azure still provides you with a number of steps to run through to mitigate this threat. As you already know the process that is running this crypto-miner, you can mitigate the threat using the following command:

```
kubectl delete -f crypto-miner.yaml
```

8. This will resolve the alert. To actually mark it as resolved, you can dismiss the alert in the Azure portal. To do this, click on the status on the left-hand side of the screen, select the **Dismissed** status, and click **OK**, as shown in *Figure 13.36*:

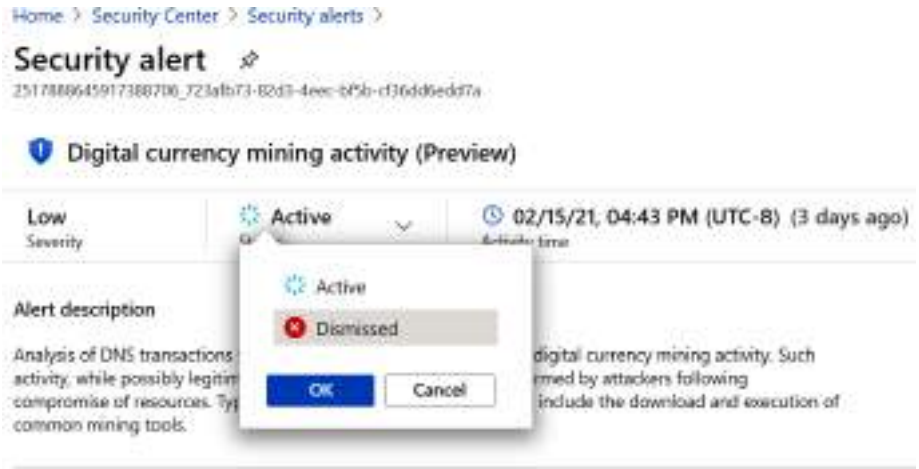


Figure 13.36: Dismissing the digital currency DNS alert

9. From the **Security alerts** pane, click on the last alert, which is called **New high privileges role detected**, and click on **View full details** in the resulting pane. This will bring you to a pane similar to *Figure* 13.37:

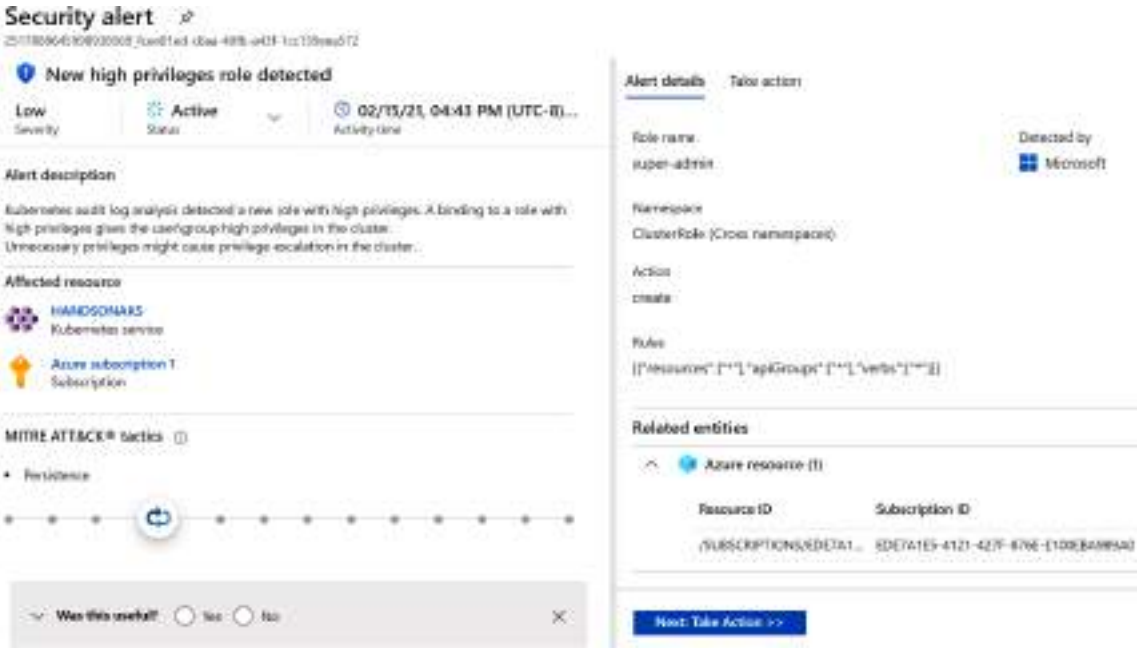


Figure 13.37: New high privileges role detected alert

This is a low-severity alert. As with the previous alerts, you get the description, the affected resource, and the phase in the MITRE ATT&CK® tactics framework, which is persistence in this case. This means that this potential attack is used by attackers to gain persistent access to your environment.

On the right-hand side, you also get the alert details, with the role name, the namespace (which in this case is the whole cluster since it is a `ClusterRole`), and the rules this role gives access to. If you click the **Next: Take Action >>** button, you'll get more information about the mitigation as well, as shown in *Figure 13.38*:

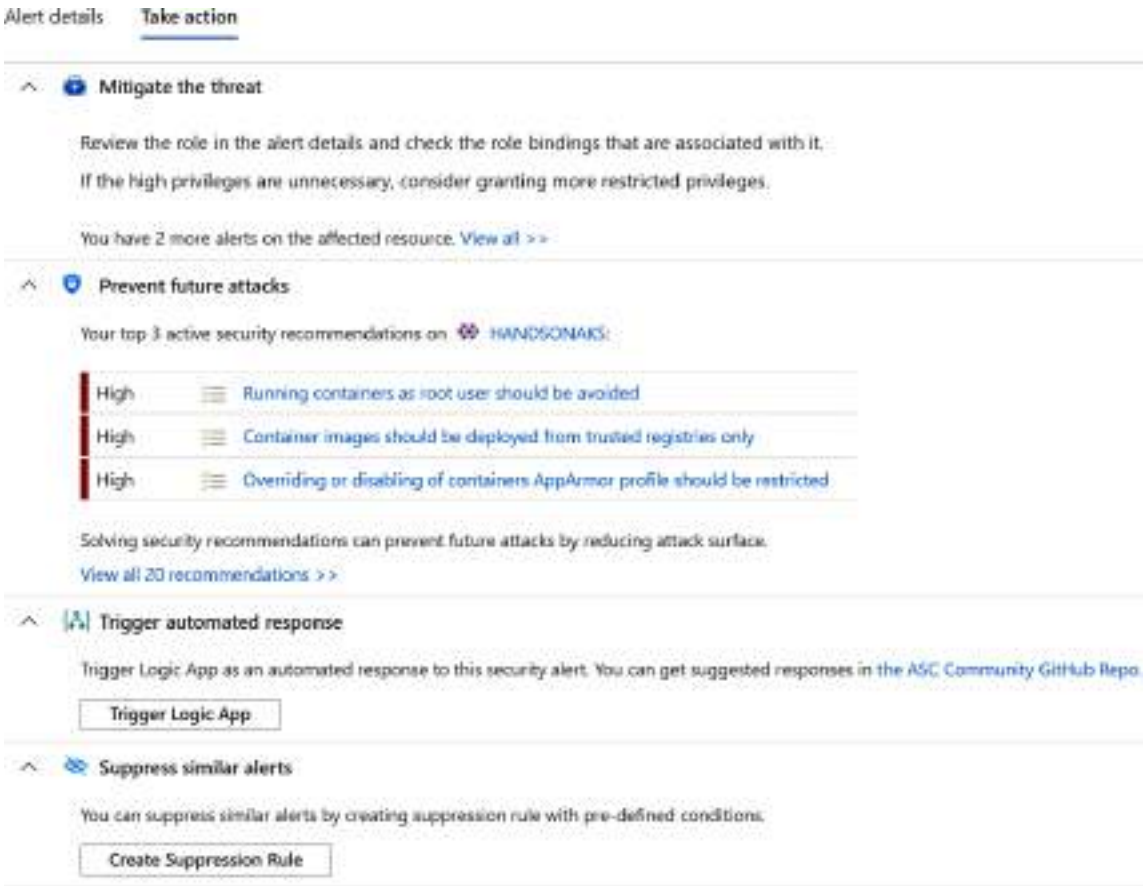


Figure 13.38: Security recommendations on the new high privileges alert

As you can see here, Azure recommends you review the role in the alerts and check any role bindings linked to this role. It is also recommended to grant more restricted privileges than the open permissions as provided in this role. Let's also remove this threat from the cluster using the following command:

```
kubectl delete -f role.yaml
```

This will delete the role from the cluster. You can also dismiss this alert, by clicking on the status on the left-hand side of the screen, selecting the **Dismissed** state, and clicking **OK**, as shown in *Figure* 13.39:

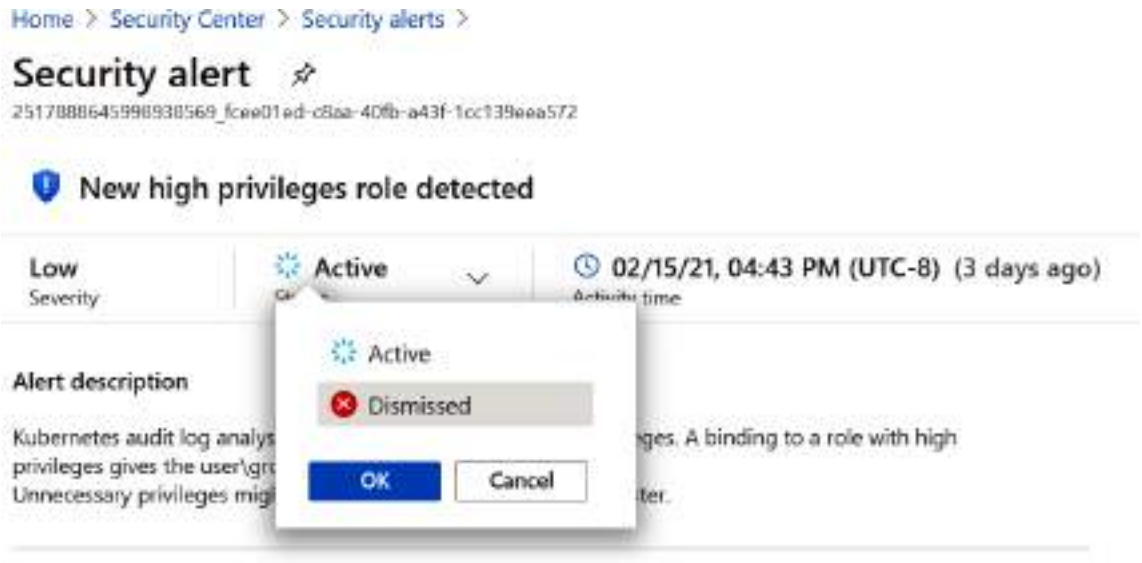


Figure 13.39: Dismissing the New high privileges alert

This covers all the alerts that were generated by the resources you created earlier in this chapter. The resources linked to the alerts were already deleted as part of the remediation, but let's also delete the other resources that were created in this chapter:

```
kubectl delete -f escalation.yaml
kubectl delete -f host-volume.yaml
kubectl delete -f https://raw.githubusercontent.com/kubernetes/
dashboard/v2.0.0/aio/deploy/recommended.yaml
```

And that concludes this chapter.

Summary

In this chapter, you explored Azure Security Center and Azure Defender. Azure Security Center is an infrastructure security monitoring platform. It provides both monitoring of security configuration as well as monitoring of any potential ongoing threats. To monitor workloads in a Kubernetes cluster, Azure Security Center makes use of Azure Policy for AKS.

To start, you enabled Azure Policy for AKS. You then enabled Azure Security Center and Azure Defender on your subscription.

You then created five harmful workloads on your cluster. Some of those caused configuration recommendations in Azure Security Center. Some others even caused security alerts to be triggered in Azure Defender. You explored four security alerts and followed the mitigation steps recommended to resolve these alerts.

14

Serverless functions

Serverless computing and serverless functions have gained tremendous traction over the past few years due to scalability and reduced management overhead. Cloud services such as Azure Functions, AWS Lambda, and GCP Cloud Run have made it very easy for users to run their code as serverless functions.

The word **serverless** refers to any solution where you don't need to manage servers. Serverless functions refer to a subset of serverless computing where you can run your code as a function on-demand. This means that your code in the function will only run and be executed when there is a demand. This architectural style is called event-driven architecture. In an event-driven architecture, the event consumers are triggered when there is an event. In the case of serverless functions, the event consumers will be these serverless functions. An event can be anything from a message in a queue to a new object uploaded to storage, or even an HTTP call.

Serverless functions are frequently used for backend processing. A common example of serverless functions is creating thumbnails of a picture that is uploaded to storage, as shown in *Figure 14.1*. Since you cannot predict how many pictures will be uploaded and when they will be uploaded, it is hard to plan traditional infrastructure and how many servers you should have available for this process. If you implement the creation of that thumbnail as a serverless function, this function will be called on each picture that is uploaded. You don't have to plan the number of functions since each new picture will trigger a new function to be executed.

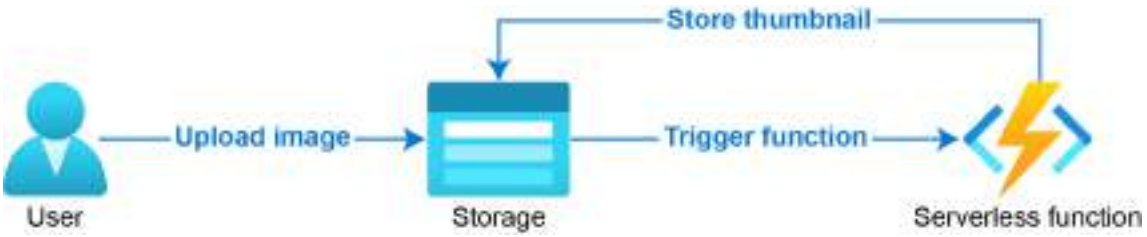


Figure 14.1: Example architecture of a serverless function to generate thumbnails of images

As you saw in the previous example, functions will automatically scale to meet increased or decreased demand. Additionally, each function can scale independently from other functions. However, this automatic scaling is just one benefit of using serverless functions. Another benefit of serverless functions is the ease of development. Using serverless functions, you can focus on writing the code and don't have to deal with the underlying infrastructure. Serverless functions allow code to be deployed without worrying about managing servers and middleware. Finally, in public cloud serverless functions, you pay per execution of the function. This means that you pay each time your functions are run, and you are charged nothing for the idle time when your functions are not run.

The popularity of public cloud serverless function platforms has caused multiple open-source frameworks to be created to enable users to create serverless functions on top of Kubernetes. In this chapter, you will learn how to deploy serverless functions on **Azure Kubernetes Service** (**AKS**) directly using the open-source version of Azure Functions. You will start by running a simple function that is triggered based on an HTTP message. Afterward, you will install a function **autoscaler** feature on your cluster. You will also integrate AKS-deployed applications with Azure storage queues. We will be covering the following topics:

- Overview of different functions platforms
- Deploying an HTTP-triggered function
- Deploying a queue-triggered function

Let's start this chapter by exploring the various functions platforms that are available for Kubernetes.

Various functions platforms

Functions platforms, such as Azure Functions, AWS Lambda, and Google Cloud Functions, have gained tremendous popularity. The ability to run code without the need to manage servers and having virtually limitless scale is very popular. The downside of using the functions implementation of a cloud provider is that you are locked into the cloud provider's infrastructure and their programming model. Also, you can only run your functions in the public cloud and not in your own datacenter.

A number of open-source functions frameworks have been launched to solve these downsides. There are a number of popular frameworks that can be run on Kubernetes:

- **Knative** (https://cloud.google.com/knative/): Knative is a serverless platform written in the Go language and developed by Google. You can run Knative functions either fully managed on Google Cloud or on your own Kubernetes cluster.
- **OpenFaaS** (https://www.openfaas.com/): OpenFaaS is a serverless framework that is Kubernetes-native. It can run on either managed Kubernetes environments such as AKS or on a self-hosted cluster. OpenFaaS is also available as a managed cloud service using `OpenFaaSCloud`. The platform is written in the Go language.
- **Serverless** (https://serverless.com/): This is a Node.js-based serverless application framework that can deploy and manage functions on multiple cloud providers, including Azure. Kubernetes support is provided via `Kubeless`.
- **Fission.io** (https://fission.io/): Fission is a serverless framework backed by the company Platform9. It is written in the Go language and is Kubernetes-native. It can run on any Kubernetes cluster.

- **Apache OpenWhisk** (https://openwhisk.apache.org/): OpenWhisk is an open-source, distributed serverless platform maintained by the Apache organization. It can be run on Kubernetes, Mesos, or Docker Compose. It is primarily written in the Scala language.

Microsoft has taken an interesting strategy with its functions platform. Microsoft operates Azure Functions as a managed service on Azure and has open-sourced the complete solution and made it available to run on any system (https://github.com/Azure/azure-functions-host). This also makes the Azure Functions programming model available on top of Kubernetes.

Microsoft has also released an additional open-source project in partnership with Red Hat called **Kubernetes Event-driven Autoscaling** (**KEDA**) to make scaling functions on top of Kubernetes easier. KEDA is a custom autoscaler that can allow deployments on Kubernetes to scale down to and up from zero pods, which is not possible using the default **Horizontal Pod Autoscaler** (**HPA**) in Kubernetes. The ability to scale from zero to one pod is important so that your application can start processing events, but scaling down to zero instances is useful for preserving resources in your cluster. KEDA also makes additional metrics available to the Kubernetes HPA to make scaling decisions based on metrics from outside the cluster (for example, the number of messages in a queue).

> **Note**
>
> We introduced and explained the HPA in *Chapter 4, Building scalable applications*.

In this chapter, you will deploy Azure Functions to Kubernetes with two examples:

- An HTTP-triggered function (without KEDA)
- A queue-triggered function (with KEDA)

Before starting with these functions, the next section will consider the necessary prerequisites for these deployments.

Setting up the prerequisites

In this section, you will set up the prerequisites needed to build and run functions on your Kubernetes cluster. You need to set up an **Azure container registry** (**ACR**) and a **virtual machine** (**VM**) in Azure that will be used to develop the functions. The ACR will be used to store custom container images that contain the functions you will develop. You will also use a VM to build the functions and create Docker images, since you cannot do this from Azure Cloud Shell.

Container images and a container registry were introduced in *Chapter 1*, *Introduction to containers and Kubernetes*, in the section on *Container images*. A container image contains all the software required to start an actual running container. In this chapter, you will build custom container images that contain your functions. You need a place to store these images so that Kubernetes can pull them and run the containers at scale. You will use ACR for this. ACR is a private container registry that is fully managed by Azure.

Up to now in this book, you have run all the examples on Azure Cloud Shell. For the example in this chapter, you will need a separate VM because Azure Cloud Shell doesn't allow you to build container images. You will create a new VM in Azure to do these tasks.

Let's begin by creating an ACR.

Azure Container Registry

Azure Functions on Kubernetes needs an image registry to store its container images. In this section, you will create an ACR and configure your Kubernetes cluster to have access to this cluster:

1. In the Azure search bar, search for `container registry` and click on **Container registries**, as shown in *Figure 14.2*:

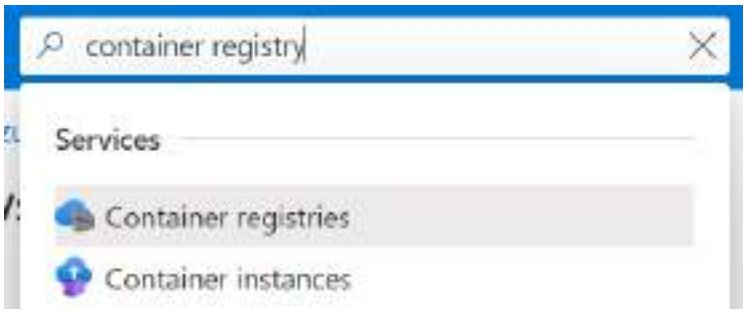


Figure 14.2: Navigating to Container registry services through the Azure portal

2. Click the **Add** button at the top to create a new registry. To organize the resources in this chapter together, create a new resource group. To do this, click on **Create new** under the **Resource group** field to create a new resource group, and call it `Functions-KEDA`, as shown in *Figure 14.3*:

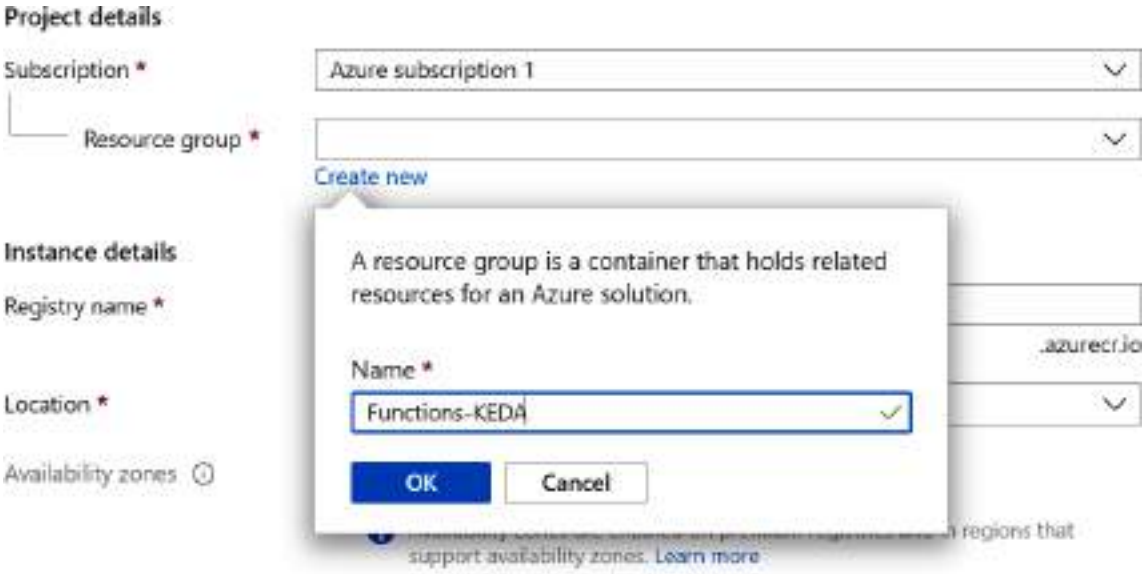


Figure 14.3: Creating a new resource group

Provide the details to create the registry. The registry name needs to be globally unique, so consider adding your initials to the registry name. It is recommended to create the registry in the same location as your cluster. To reduce spending for the demo, you can change **SKU** to **Basic**. Select the **Review + create** button at the bottom to create the registry, as shown in *Figure 14.4*:

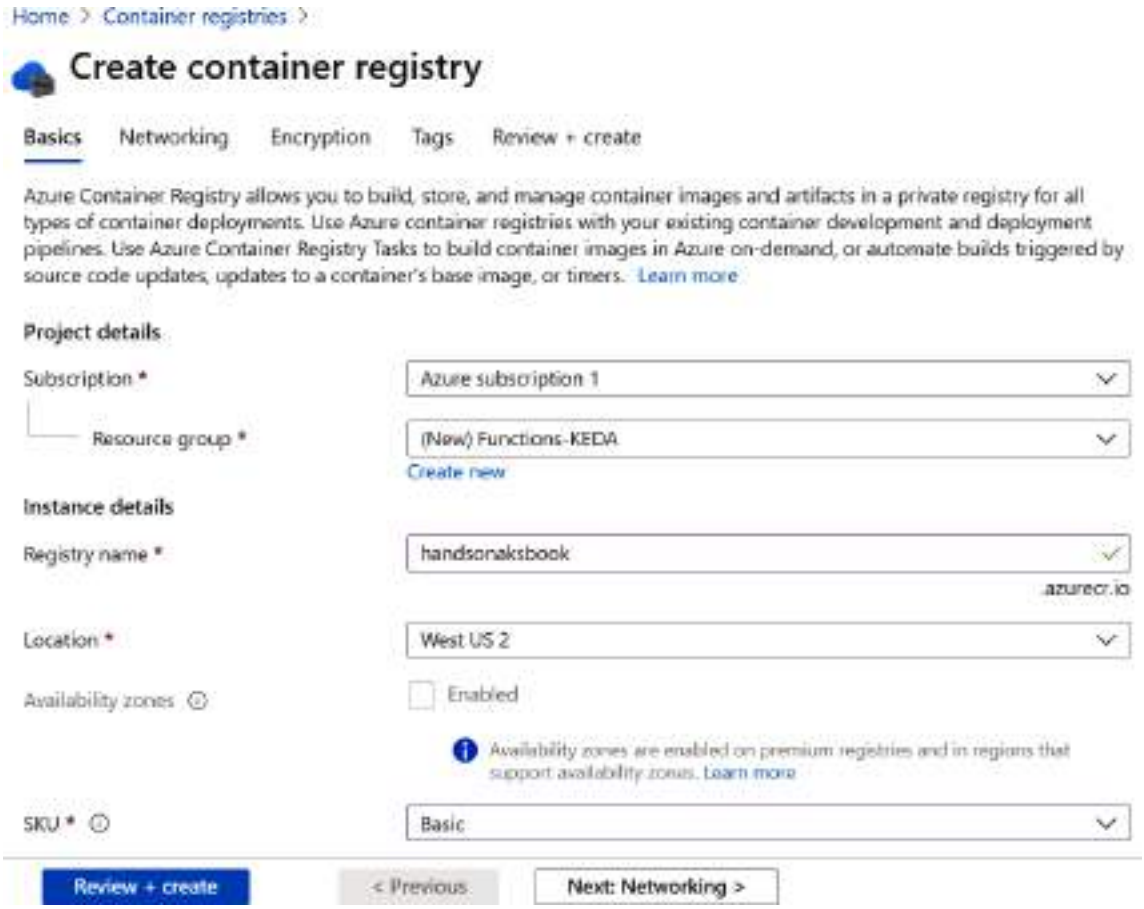


Figure 14.4: Providing details to create the registry

In the resulting pane, click the **Create** button to create the registry.

3. Once your registry is created, open Cloud Shell so that you can configure your AKS cluster to get access to your container registry. Use the following command to give AKS permissions to your registry:

```
az aks update -n handsonaks \
-g rg-handsonaks --attach-acr <acrName>
```

This will return an output similar to *Figure* 14.5. The figure has been cropped to show only the top part of the output:

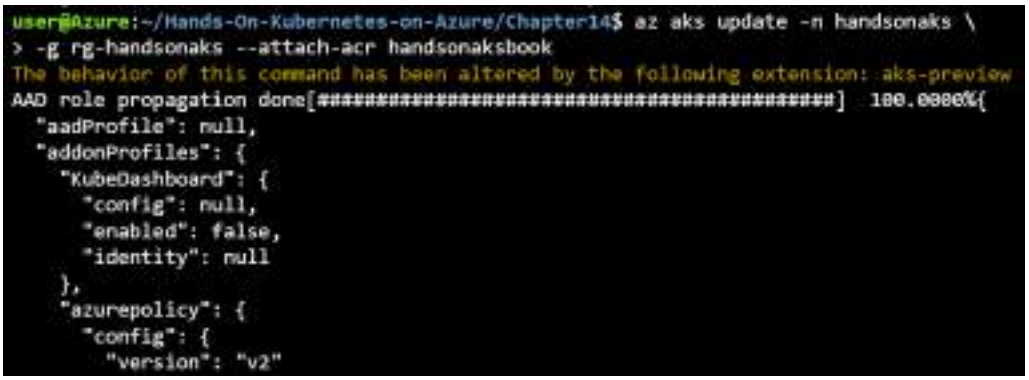


Figure 14.5: Allowing AKS cluster to access the container registry

You now have an ACR that is integrated with AKS. In the next section, you will create a VM that will be used to build the Azure functions.

Creating a VM

In this section, you will create a VM and install the tools necessary to run Azure Functions on this machine:

- The Docker runtime
- The Azure CLI
- Azure Functions
- Kubectl

> **Note**
>
> To ensure a consistent experience, you will be creating a VM on Azure that will be used for development. If you prefer to run the sample on your local machine, you can install all the required tools locally.

Let's get started with creating the VM:

1. To ensure this example works with the Azure trial subscription, you will need to scale down your cluster to one node. You can do this using the following command:

   ```
   az aks scale -n handsonaks -g rg-handsonaks --node-count 1
   ```

2. To authenticate to the VM you are going to create, you'll need a set of SSH keys. If you followed the example in *Chapter 9*, *Azure Active Directory pod-managed identities in AKS* in the *Setting up a new cluster with AAD pod-managed identity* section, you will already have a set of SSH keys. To verify that you have SSH keys, run the following command:

   ```
   ls ~/.ssh
   ```

   This should show you the presence of an SSH private key (`id_rsa`) and a public key (`id_rsa.pub`), as shown in *Figure 14.6*:

Figure 14.6: Verifying SSH keys are present

   If you do not have these keys already available, you will need to generate a set of SSH keys using the following command:

   ```
   ssh-keygen
   ```

   You will be prompted for a location and a passphrase. Keep the default location and input an empty passphrase.

3. You will now create the VM. You will create an Ubuntu VM using the following command:

   ```
   az vm create -g Functions-KEDA -n devMachine \
     --image UbuntuLTS --ssh-key-value ~/.ssh/id_rsa.pub \
     --admin-username handsonaks --size Standard_D1_v2
   ```

4. This will take a couple of minutes to complete. Once the VM is created, Cloud

Shell should show you its public IP, as displayed in *Figure* 14.7:

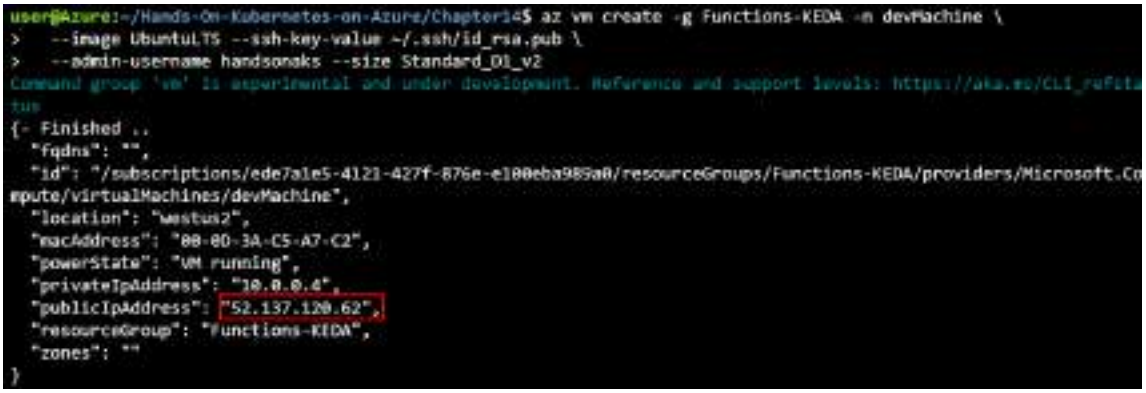


Figure 14.7: Creating the development VM

Connect to the VM using the following command:

```
ssh handsonaks@<public IP>
```

You will be prompted about whether you trust the machine's identity. Type `yes` to confirm.

5. You're now connected to a new VM on Azure. On this machine, we will begin by installing Docker:

```
sudo apt-get update
sudo apt-get install docker.io -y
sudo systemctl enable docker
sudo systemctl start docker
```

6. To make the operation smoother, add the user to the Docker group. This will ensure you can run Docker commands without `sudo`:

```
sudo usermod -aG docker handsonaks
newgrp docker
```

You should now be able to run the `hello-world` command:

```
docker run hello-world
```

This will show you an output similar to *Figure 14.8*:

```
handsonaks@devMachine:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:95ddb6c31407e84e91a986b004aee40975cb0bda14b5949f6faac5d2deadb4b9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figure 14.8: Verifying Docker runs on the VM

7. Next, you will install the Azure CLI on this VM. You can install the CLI using the following command:

```
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

8. Verify that the CLI was installed successfully by signing in:

```
az login
```

This will display a login code that you need to enter at https://microsoft.com/devicelogin:

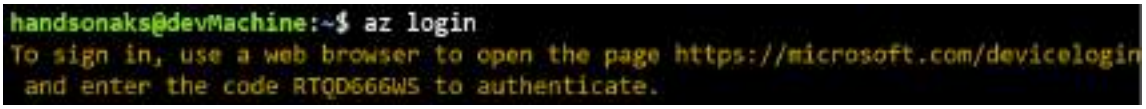


Figure 14.9: Signing in to the Azure CLI

Browse to that website and paste in the login code that was provided to you to enable you to sign in to Cloud Shell. Make sure to do this in a browser you are signed in to with the user who has access to your Azure subscription.

You can now use the CLI to authenticate your machine to ACR. This can be done using the following command:

```
az acr login -n <registryname>
```

The credentials to ACR expire after 3 hours. If you run into the following error during this demonstration, you can sign in to ACR again using the following command:

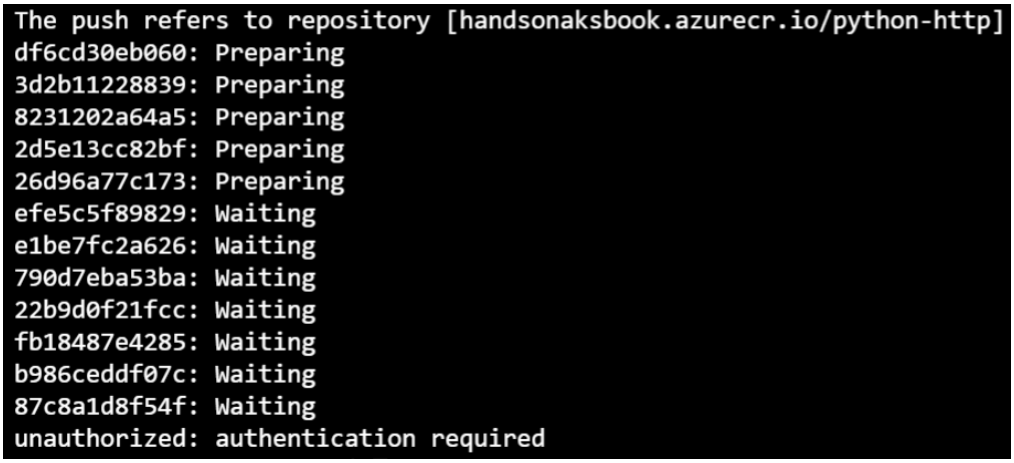


Figure 14.10: Potential authentication error in the future

9. Next, you'll install `kubectl` on your machine. The Azure CLI has a shortcut to install the CLI, which you can use to install it:

```
sudo az aks install-cli
```

Let's verify that `kubectl` can connect to our cluster. For this, we'll first get the credentials and then execute a `kubectl` command:

```
az aks get-credentials -n handsonaks -g rg-handsonaks
kubectl get nodes
```

10. Now, you can install the Azure Functions tools on this machine. To do this, run the following commands:

```
wget -q https://packages.microsoft.com/config/ubuntu/18.04/packages-
microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
sudo apt-get install azure-functions-core-tools-3 -y
```

This will return an output similar to *Figure 14.11*:

Figure 14.11: Installing Functions core tools

> **Note**
>
> If you are running a newer version of Ubuntu than 18.04, please make sure that you download the correct dpkg package by changing the URL in the first line to reflect your Ubuntu version.

You now have the prerequisites to start working with functions on Kubernetes. You created an ACR to store custom container images, and you have a VM that will be used to create and build Azure functions. In the next section, you will build your first function, which is HTTP-triggered.

Creating an HTTP-triggered Azure function

In this first example, you will create an HTTP-triggered Azure function. This means that you can browse to the page hosting the actual function:

1. To begin, create a new directory and navigate to that directory:

   ```
   mkdir http
   cd http
   ```

2. Now, you will initialize a function using the following command:

   ```
   func init --docker
   ```

   The `--docker` parameter specifies that you will build the function as a Docker container. This will result in a Dockerfile being created. Select the Python language, which is option **3** in the following screenshot:

```
handsonaks@devMachine:~/http$ func init --docker
Select a number for worker runtime:
1. dotnet
2. node
3. python
4. powershell
5. custom
Choose option: 3
python
Found Python version 3.6.9 (python3).
Writing requirements.txt
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing /home/handsonaks/http/.vscode/extensions.json
Writing Dockerfile
Writing .dockerignore
```

Figure 14.12: Creating a Python function

This will create the required files for your function to work.

3. Next, you will create the actual function. Enter the following command:

```
func new
```

This should result in an output like the following. Select the eighth option, **HTTP trigger**, and name the function `python-http`:

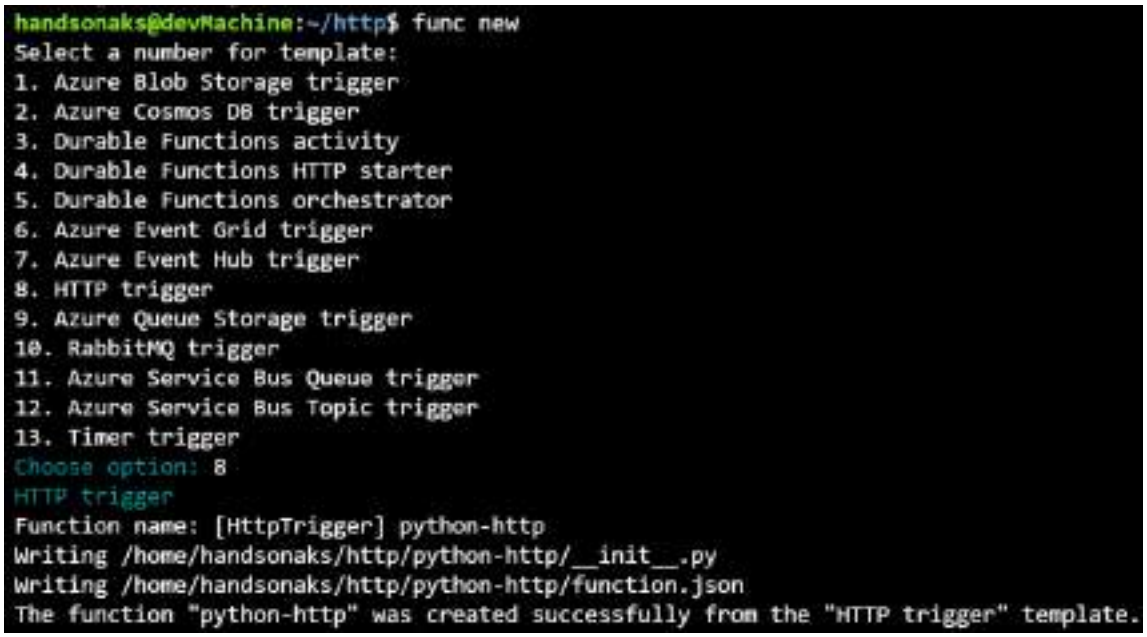


Figure 14.13: Creating an HTTP-triggered function

4. The code of the function is stored in the directory called `python-http`. You are not going to make code changes to this function. If you want to check out the source code of the function, you can run the following command:

    ```
    cat python-http/__init__.py
    ```

5. You will need to make one change to the function's configuration file. By default, functions require an authenticated request. You will change this to `anonymous` for this demo. Make the change using the `vi` command by executing the following command:

    ```
    vi python-http/function.json
    ```

    Replace `authLevel` on *line* 5 with `anonymous`. To make that change, press *I* to go into insert mode, then remove `function` and replace it with `anonymous`:

Figure 14.14: Changing the authLevel function to anonymous

Hit *Esc*, type `:wq!`, and then hit *Enter* to save and quit `vi`.

> **Note**
>
> You changed the authentication requirement for your function to anonymous. This will make the demo easier to execute. If you plan to release functions to production, you need to carefully consider this setting, since this controls who has access to your function.

6. You are now ready to deploy your function to AKS. You can deploy the function using the following command:

```
func kubernetes deploy --name python-http \
--registry <registry name>.azurecr.io
```

This will cause the functions runtime to do a couple of steps. First, it will build a container image, then it will push that image to the registry, and finally, it will deploy the function to Kubernetes:

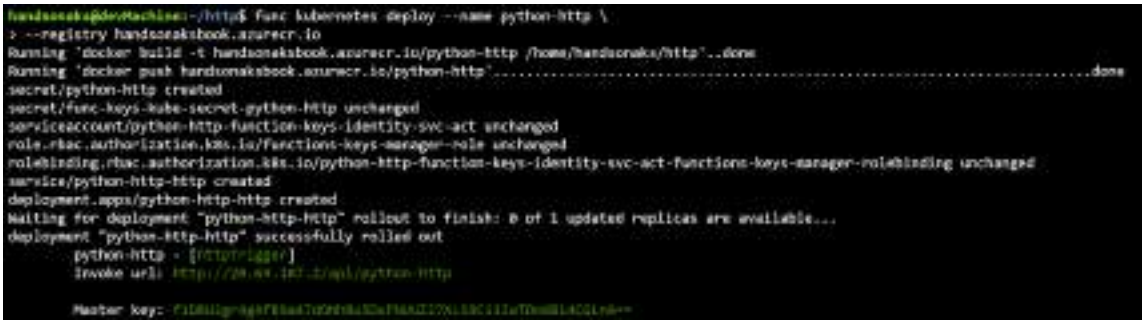


Figure 14.15: Deploying the function to AKS

You can click the **Invoke url** URL that is shown to get access to your function. Before doing so, however, let's explore what was created on the cluster.

7. To create the function, a regular deployment on top of Kubernetes was used. To check the deployment, you can run the following command:

```
kubectl get deployment
```

This will show you the deployment, as in *Figure 14.16*:

Figure 14.16: Deployment details

8. This process also created a service on top of your Kubernetes cluster. You can get the public IP of the service that was deployed and connect to it:

```
kubectl get service
```

This will show you the service and its public IP, as shown in *Figure 14.17*. Notice how this public IP is the same as the one shown in the output of *Step 4*.

```
handsonaks@devMachine:~/http$ kubectl get service
NAME                TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
kubernetes          ClusterIP      10.0.0.1       <none>         443/TCP          44h
python-http-http    LoadBalancer   10.0.100.18    20.69.187.2    80:32578/TCP     6m7s
```

Figure 14.17: Getting the service's public IP

Open a web browser and browse to `http://<external-ip>/api/python-http?name=handsonaks`. You should see a web page showing you **Hello, handsonaks. This HTTP triggered function executed successfully**. This is shown in *Figure 14.18*:

Figure 14.18: Output of the HTTP triggered function

You have now created a function with an HTTP trigger. Using an HTTP-triggered function is useful in scenarios where you are providing an HTTP API with unpredictable load patterns. Let's clean up this deployment before moving on to the next section:

```
kubectl delete deployment python-http-http
kubectl delete service python-http-http
kubectl delete secret python-http
```

In this section, you created a sample function using an HTTP trigger. Let's take that one step further and integrate a new function with storage queues and set up the KEDA autoscaler in the next section.

Creating a queue-triggered function

In the previous section, you created a sample HTTP function. In this section, you'll build another sample using a queue-triggered function. Queues are often used to pass messages between different components of an application. A function can be triggered based on messages in a queue to then perform additional processing on these messages.

In this section, you'll create a function that is integrated with Azure storage queues to consume events. You will also configure KEDA to allow scaling to/from zero pods in the case of low traffic.

Let's start by creating a queue in Azure.

Creating a queue

In this section, you will create a new storage account and a new queue in that storage account. You will connect functions to that queue in the next section, *Creating a queue-triggered function*.

1. To begin, create a new storage account. Search for `storage accounts` in the Azure search bar and select **Storage accounts**:

Figure 14.19: Navigating to Storage accounts service through the Azure portal

2. Click the **+ New** button at the top to create a new storage account. Provide the details to create the storage account. The storage account name has to be globally unique, so consider adding your initials. It is recommended to create the storage account in the same region as your AKS cluster. Finally, to save on costs, you are recommended to downgrade the replication setting to **Locally-redundant storage (LRS)** as shown in *Figure 14.20*:

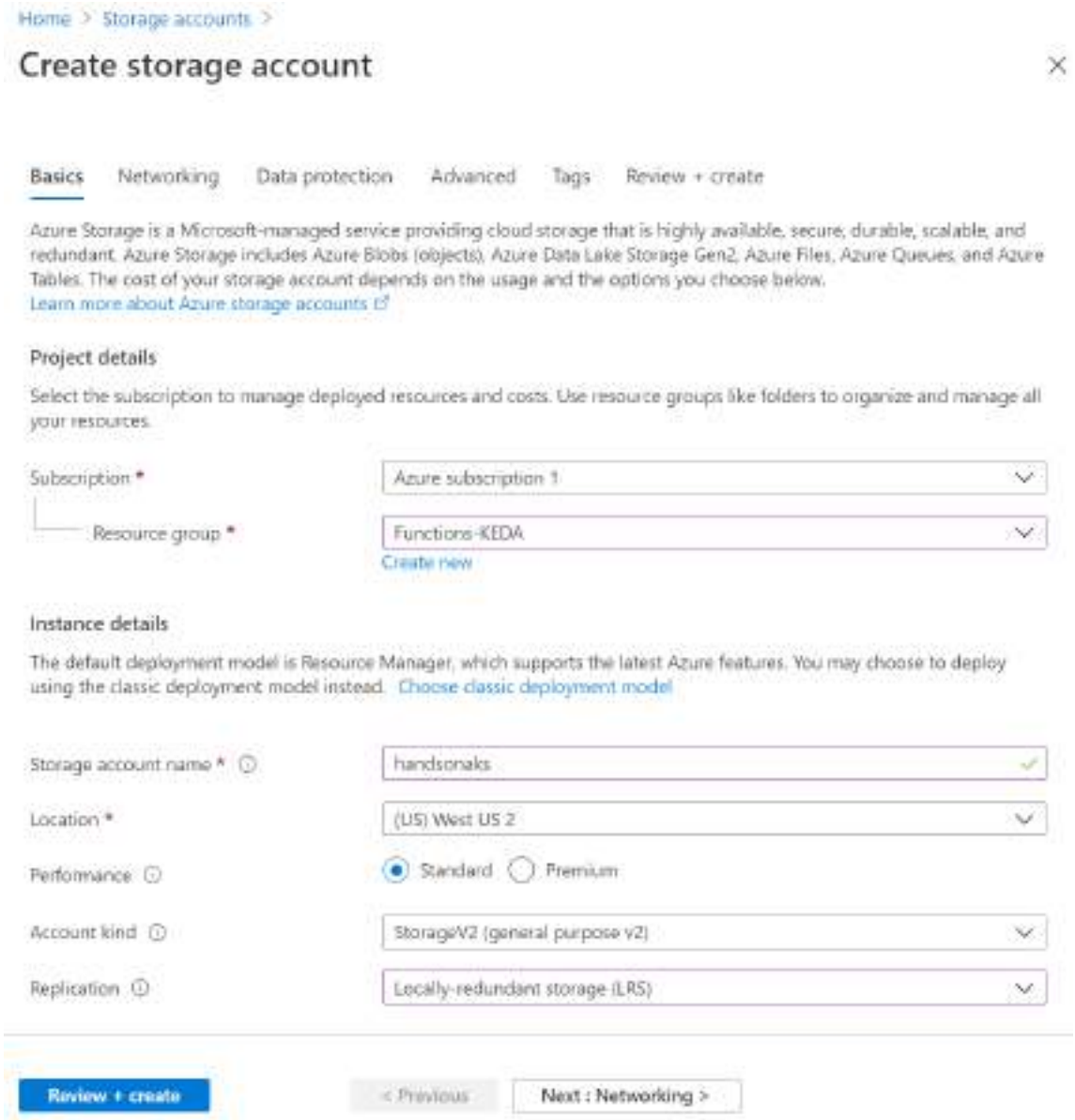


Figure 14.20: Providing the details to create the storage account

Once you're ready, click the **Review + create** button at the bottom. On the resulting screen, select **Create** to start the creation process.

3. It will take about a minute to create the storage account. Once it is created, open the account by clicking on the **Go to resource** button. In the **Storage account** pane, select **Access keys** in the left-hand navigation, click on **Show keys**, and copy the primary connection string, as shown in *Figure 14.21*. Note down this string for now:

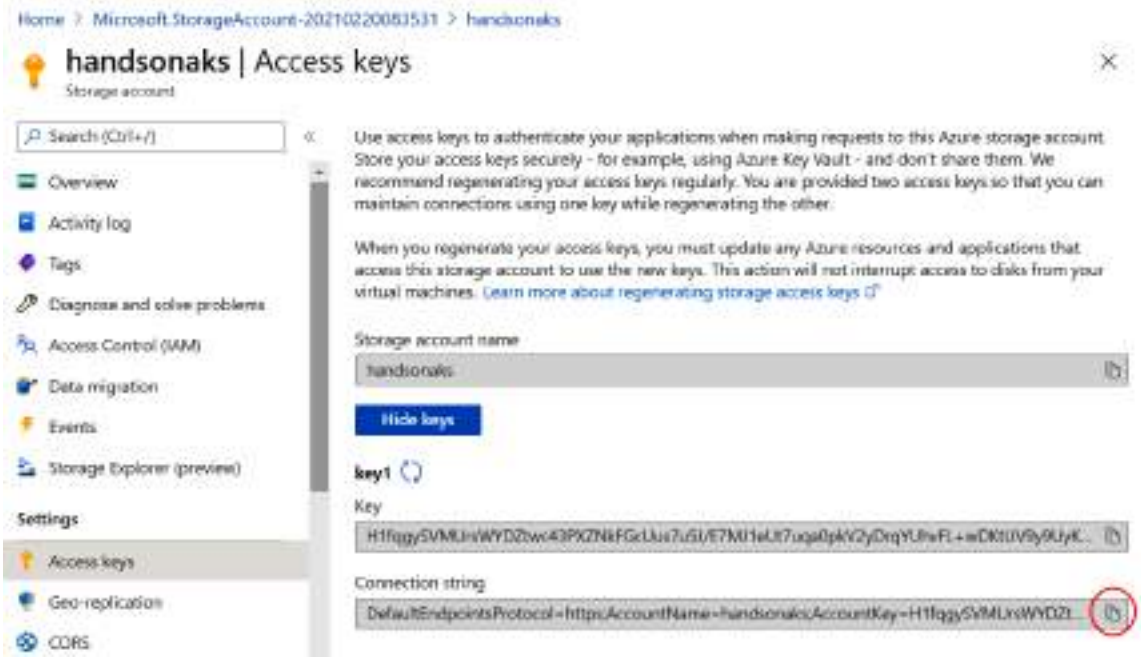


Figure 14.21: Copying the primary connection string

> ### Note
>
> For production use cases, it is not recommended to connect to Azure Storage using the access key. Any user with that access key has full access to the storage account and can read and delete all files on it. It is recommended to either generate a **shared access signatures** (**SAS**) token to connect to storage or to use Azure AD-integrated security. To learn more about SAS token authentication to storage, refer to https://docs.microsoft.com/rest/api/storageservices/delegate-access-with-shared-access-signature. To learn more about Azure AD authentication to Azure Storage, please refer to https://docs.microsoft.com/rest/api/storageservices/authorize-with-azure-active-directory.

4. The final step is to create our queue in the storage account. Look for queue in the left-hand navigation, click the **+ Queue** button to add a queue, and provide it with a name. To follow along with this demo, use `function` as the queue name:

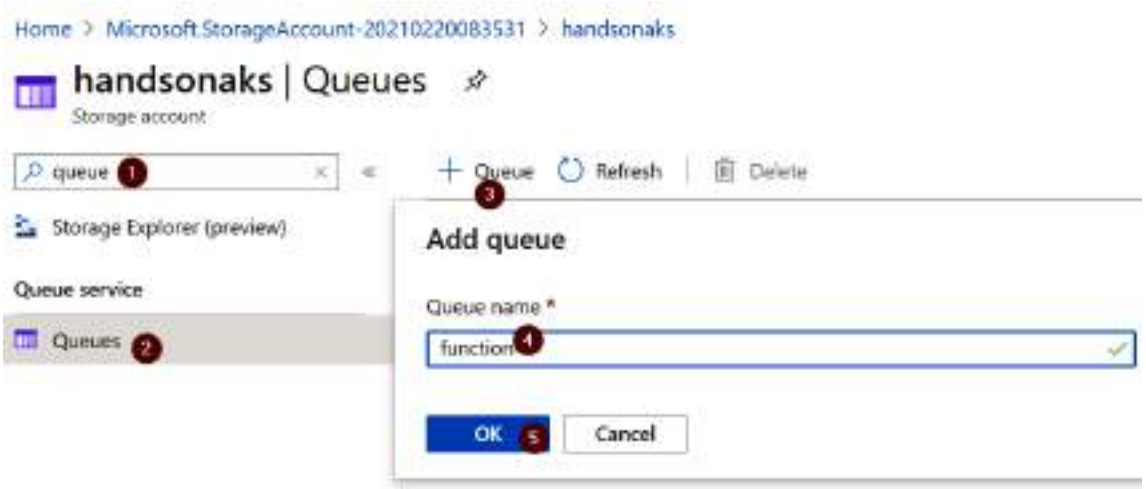


Figure 14.22: Creating a new queue

You have now created a storage account in Azure and have its connection string. You created a queue in this storage account. In the next section, you will create a function that will consume messages from the queue.

Creating a queue-triggered function

In the previous section, you created a queue in Azure. In this section, you will create a new function that will monitor this queue and remove messages from the queue. You will need to configure this function with the connection string to this queue:

1. From within the VM, begin by creating a new directory and navigating to it:

```
cd ..
mkdir js-queue
cd js-queue
```

2. Now we can create the function. We will start with the initialization:

```
func init --docker
```

This will ask you two questions now. For the runtime, select **node** (option 2), and for the language, select **JavaScript** (option 1). This should result in the output shown in *Figure 14.23*:

```
handsonaks@devMachine:~/js-queue$ func init --docker
Select a number for worker runtime:
1. dotnet
2. node
3. python
4. powershell
5. custom
Choose option: 2
node
Select a number for language:
1. javascript
2. typescript
Choose option: 1
javascript
Writing package.json
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing /home/handsonaks/js-queue/.vscode/extensions.json
Writing Dockerfile
Writing .dockerignore
```

Figure 14.23: Initializing a new function

Following the initialization, you can create the actual function:

```
func new
```

This will ask you for a trigger. Select **Azure Queue Storage trigger** (option 10). Give the name `js-queue` to the new function. This should result in the output shown in *Figure 14.24*:

```
handsonaks@devMachine:~/js-queue$ func new
Select a number for template:
1. Azure Blob Storage trigger
2. Azure Cosmos DB trigger
3. Durable Functions activity
4. Durable Functions HTTP starter
5. Durable Functions orchestrator
6. Azure Event Grid trigger
7. Azure Event Hub trigger
8. HTTP trigger
9. IoT Hub (Event Hub)
10. Azure Queue Storage trigger
11. RabbitMQ trigger
12. SendGrid
13. Azure Service Bus Queue trigger
14. Azure Service Bus Topic trigger
15. SignalR negotiate HTTP trigger
16. Timer trigger
Choose option: 10
Azure Queue Storage trigger
Function name: [QueueTrigger] js-queue
Writing /home/handsonaks/js-queue/js-queue/index.js
Writing /home/handsonaks/js-queue/js-queue/readme.md
Writing /home/handsonaks/js-queue/js-queue/function.json
The function "js-queue" was created successfully from the "Azure Queue Storage trigger" template.
```

Figure 14.24: Creating a queue-triggered function

3. You will now need to make a couple of configuration changes. You need to provide the function you created the connection string on to Azure Storage and provide the queue name. First, open the `local.settings.json` file to configure the connection strings for storage:

```
vi local.settings.json
```

To make the changes, follow these instructions:

- Hit *I* to go into insert mode.
- Replace the connection string for `AzureWebJobsStorage` with the connection string you copied earlier. Add a comma to the end of this line.
- Add a new line and then add the following text on that line:

```
"QueueConnString": "<your connection string>"
```

The result should look like *Figure 14.25*:

Figure 14.25: Editing the local.settings.json file

- Save and close the file by hitting the *Esc* key, type `:wq!`, and then press *Enter*.

4. The next file you need to edit is the function configuration itself. Here, you will refer to the connection string from earlier, and provide the queue name we chose in the *Creating a queue* section. To do that, use the following command:

```
vi js-queue/function.json
```

   To make the changes, follow these instructions:

   - Hit *I* to go into insert mode.
   - Change the queue name to the name of the queue you created (`function`).
   - Next, add `QueueConnString` to the `connection` field.

Your configuration should now look like *Figure 14.26*:

Figure 14.26: Editing the js-queue/function.json file

- Save and close the file by hitting the *Esc* key, type `:wq!`, and then press *Enter*.

5. You are now ready to publish your function to Kubernetes. You will start by setting up KEDA on your Kubernetes cluster:

```
kubectl create ns keda
func kubernetes install --keda --namespace keda
```

This should return an output similar to *Figure 14.27*:

```
handsonaks@devMachine:~/js-queue$ kubectl create ns keda
namespace/keda created
handsonaks@devMachine:~/js-queue$ func kubernetes install --keda --namespace keda
Installing KEDA v2 in namespace keda
customresourcedefinition.apiextensions.k8s.io/scaledjobs.keda.sh created
customresourcedefinition.apiextensions.k8s.io/scaledobjects.keda.sh created
customresourcedefinition.apiextensions.k8s.io/triggerauthentications.keda.sh created
serviceaccount/keda-operator created
clusterrole.rbac.authorization.k8s.io/keda-external-metrics-reader created
clusterrole.rbac.authorization.k8s.io/keda-operator created
rolebinding.rbac.authorization.k8s.io/keda-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/keda-hpa-controller-external-metrics created
clusterrolebinding.rbac.authorization.k8s.io/keda-operator created
clusterrolebinding.rbac.authorization.k8s.io/keda:system:auth-delegator created
service/keda-metrics-apiserver created
deployment.apps/keda-metrics-apiserver created
deployment.apps/keda-operator created
apiservice.apiregistration.k8s.io/v1beta1.external.metrics.k8s.io created
KEDA v2 is installed in namespace keda
```

Figure 14.27: Installing KEDA on Kubernetes

This will set up KEDA on your cluster. The installation doesn't take long. To verify that the installation was successful, make sure that the KEDA pod is running in the keda namespace:

```
kubectl get pod -n keda
```

This should return an output similar to *Figure 14.28*:

```
handsonaks@devMachine:~/js-queue$ kubectl get pod -n keda
NAME                                      READY   STATUS    RESTARTS   AGE
keda-metrics-apiserver-69c5d9f7-p9ssl     1/1     Running   0          80s
keda-operator-6b687d967d-jcbcj            1/1     Running   0          80s
```

Figure 14.28: Verifying the KEDA installation succeeded

6. You can now deploy the function to Kubernetes. You will configure KEDA to look at the number of queue messages every 5 seconds (`polling-interval=5`) to have a maximum of 15 replicas (`max-replicas=15`), and to wait 15 seconds before removing pods (`cooldown-period=15`). To deploy and configure KEDA in this way, use the following command:

```
func kubernetes deploy --name js-queue \
--registry <registry name>.azurecr.io \
--polling-interval=5 --max-replicas=15 --cooldown-period=15
```

This will return an output similar to *Figure 14.29*:

```
handsonaks@devMachine:~/js-queue$ func kubernetes deploy --name js-queue \
> --registry handsonaksbook.azurecr.io \
> --polling-interval=5 --max-replicas=15 --cooldown-period=15
Running 'docker build -t handsonaksbook.azurecr.io/js-queue /home/handsonaks/js-queue'.....
.....done
Running 'docker push handsonaksbook.azurecr.io/js-queue'.....
.....done
secret/js-queue created
deployment.apps/js-queue created
scaledobject.keda.sh/js-queue created
```

Figure 14.29: Deploying the queue-triggered function

To verify that the setup completed successfully, you can run the following command:

```
kubectl get all
```

This will show you all the resources that were deployed. As you can see in *Figure* 14.30, this setup created a deployment, ReplicaSet, and an HPA. In the HPA, you should see that there are no replicas currently running:

```
handsonaks@devMachine:~/js-queue$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.0.0.1     <none>        443/TCP   4d19h

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/js-queue     0/0     0            0           12m

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/js-queue-6945b55f89     0         0         0       12m

NAME                                                                    REFERENCE                TARGETS
 MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/keda-hpa-js-queue   Deployment/js-queue   <unknown>/5 (avg)
 1         15        0          12m
```

Figure 14.30: Verifying the objects created by the setup

7. Now you will create a message in the queue to trigger KEDA and create a pod. To see the scaling event, run the following command:

```
kubectl get hpa -w
```

8. To create a message in the queue, we are going to use the Azure portal. To create a new message, open the queue in the storage that you created earlier. Click on the **+ Add message** button at the top of your screen, create a test message, and click on **OK**. This is shown in *Figure* 14.31:

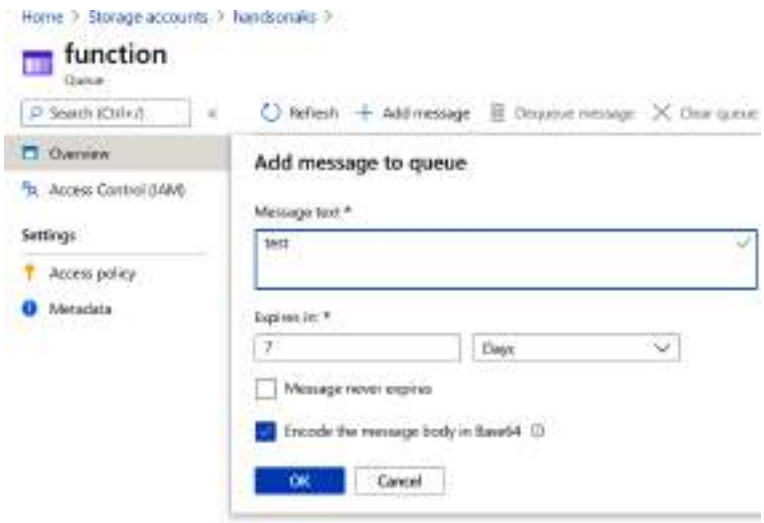


Figure 14.31: Adding a message to the queue

After creating this message, have a look at the output of the previous command you issued. It might take a couple of seconds, but soon enough, your HPA should scale to one replica. Afterward, it should also scale back down to zero replicas:

```
handsonaks@devMachine:~/js-queue$ kubectl get hpa -w
NAME                REFERENCE                  TARGETS               MINPODS   MAXPODS   REPLICAS   AGE
keda-hpa-js-queue   Deployment/js-queue        <unknown>/5 (avg)     1         15        0          15m
keda-hpa-js-queue   Deployment/js-queue        1/5 (avg)             1         15        1          17m
keda-hpa-js-queue   Deployment/js-queue        <unknown>/5 (avg)     1         15        0          18m
```

Figure 14.32: KEDA scaling from 0 to 1 and back to 0 replicas

This has shown you that KEDA enabled the Kubernetes HPA to scale from zero to one pod when there are messages in the queue, and also from one to zero pods when those messages are processed.

You have now created a function that is triggered by messages being added to a queue. You were able to verify that KEDA scaled the pods from 0 to 1 when you created a message in the queue, and back down to 0 when there were no messages left. In the next section, you will execute a scale test, and you will create multiple messages in the queue and see how the functions react.

Scale testing functions

In the previous section, you saw how functions reacted when there was a single message in the queue. In this example, you are going to send 1,000 messages into the queue and see how KEDA will first scale out the function, and then scale back in, and eventually scale back down to zero:

1. In the current Cloud Shell, watch the HPA using the following command:

   ```
   kubectl get hpa -w
   ```

2. To start pushing the messages, you are going to open a new Cloud Shell session. To open a new session, select the **Open new session** button in Cloud Shell:

Figure 14.33: Opening a new Cloud Shell instance

To send the 1,000 messages into the queue, a Python script has been provided called `sendMessages.py` in *Chapter 15* of the code examples in the GitHub repo accompanying this book. To make the script work, you'll need to install `azure-storage-queue package using pip`:

```
pip install azure-storage-queue==12.1.5
```

Once that is installed, you will need to provide this script with your storage account connection string. To do this, open the file using:

```
code sendMessages.py
```

Edit the storage connection string on *line* 8 to your connection string:

Figure 14.34: Pasting in your connection string for your storage account on line 8

3. Once you have pasted in your connection string, you can execute the Python script and send 1,000 messages to your queue:

```
python sendMessages.py
```

While the messages are being sent, switch back to the previous Cloud Shell instance and watch KEDA scale from 0 to 1, and then watch the HPA scale to the number of replicas. The HPA uses metrics provided by KEDA to make scaling decisions. Kubernetes, by default, doesn't know about the number of messages in an Azure storage queue that KEDA provides to the HPA.

> **Note**
>
> Depending on how quickly KEDA in your cluster scales up the application, your deployment might not scale to the 15 replicas that are shown in *Figure 14.29*.

Once the queue is empty, KEDA will scale back down to zero replicas:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter14$ kubectl get hpa -w
NAME                REFERENCE                  TARGETS              MINPODS   MAXPODS   REPLICAS   AGE
keda-hpa-js-queue   Deployment/js-queue        <unknown>/5 (avg)    1         15        0          74m
keda-hpa-js-queue   Deployment/js-queue        274/5 (avg)          1         15        1          74m
keda-hpa-js-queue   Deployment/js-queue        211750m/5 (avg)      1         15        4          74m
keda-hpa-js-queue   Deployment/js-queue        110750m/5 (avg)      1         15        8          75m
keda-hpa-js-queue   Deployment/js-queue        85467m/5 (avg)       1         15        15         75m
keda-hpa-js-queue   Deployment/js-queue        75867m/5 (avg)       1         15        15         75m
keda-hpa-js-queue   Deployment/js-queue        0/5 (avg)            1         15        15         75m
keda-hpa-js-queue   Deployment/js-queue        <unknown>/5 (avg)    1         15        0          76m
```

Figure 14.35: KEDA will scale from 0 to 1, and the HPA will scale to 15 pods

As you can see in the output of this command, the deployment was scaled first from zero to one replica, and then gradually got scaled out to a maximum of 15 replicas. When there were no more messages in the queue, the deployment was scaled down again to zero replicas.

This concludes the examples of running serverless functions on top of Kubernetes. Let's make sure to clean up the objects that were created. Run the following command from within the VM you created (the final step will delete this VM; if you want to keep the VM, don't run the final step):

```
kubectl delete secret js-queue
kubectl delete scaledobject js-queue
kubectl delete deployment js-queue
func kubernetes remove --namespace keda
az group delete -n Functions-KEDA  --yes
```

In this section, you ran a function that was triggered by messages in a storage queue on top of Kubernetes. You used a component called KEDA to achieve scaling based on the number of queue messages. You saw how KEDA can scale from 0 to 1 and back down to 0. You also saw how the HPA can use metrics provided by KEDA to scale out a deployment.

Summary

In this chapter, you deployed serverless functions on top of your Kubernetes cluster. To achieve this, you first created a VM and an ACR.

You started the functions deployments by deploying a function that used an HTTP trigger. The Azure Functions core tools were used to create that function and to deploy it to Kubernetes.

Afterward, you installed an additional component on your Kubernetes cluster called KEDA. KEDA allows serverless scaling in Kubernetes. It allows deployments to and from zero pods, and it also provides additional metrics to the HPA. You used a function that was triggered on messages in an Azure storage queue.

In the next – and final – chapter of this book, you'll learn how to integrate containers and Kubernetes in a **continuous integration and continuous delivery** (**CI/CD**) pipeline using GitHub Actions.

15

Continuous integration and continuous deployment for AKS

DevOps is the union of people, processes, and tools to deliver software faster, more frequently, and more reliably. Within the DevOps culture are the practices of **continuous integration and continuous deployment** (**CI/CD**). CI/CD is a set of practices, implemented through one or more tools, to automatically test, build, and deliver software.

The CI phase refers to the practice of continuously testing and building software. The outcome of the CI phase is a deployable artifact. That artifact could be many things; for instance, for a Java application it would be a `JAR` file, and in the case of a container-based application it would be a container image.

The CD phase refers to the practice of continuously releasing software. During the CD phase, the artifact that was generated during CI is deployed to multiple environments, typically going from test to QA to staging to production.

Multiple tools exist to implement CI/CD. GitHub Actions is one such tool. GitHub Actions is a workflow automation system built into GitHub. With GitHub Actions, you can build, test, and deploy applications written in any language to a variety of platforms. It also allows you to build container images and deploy applications to a Kubernetes cluster, which you'll do in this chapter.

Specifically, this chapter will cover the following topics:

- CI/CD process for containers and Kubernetes
- Setting up Azure and GitHub
- Setting up a CI pipeline
- Setting up a CD pipeline

Let's start by exploring the CI/CD lifecycle for containers and Kubernetes.

CI/CD process for containers and Kubernetes

Before you start building a pipeline, it's good to understand the typical CI/CD process for containers and Kubernetes. In this section, the high-level process shown in *Figure 15.1* will be explored in more depth. For a more detailed exploration on CI/CD and DevOps for Kubernetes, you are encouraged to explore the following free online eBook by Microsoft: https://docs.microsoft.com/dotnet/architecture/containerized-lifecycle/.

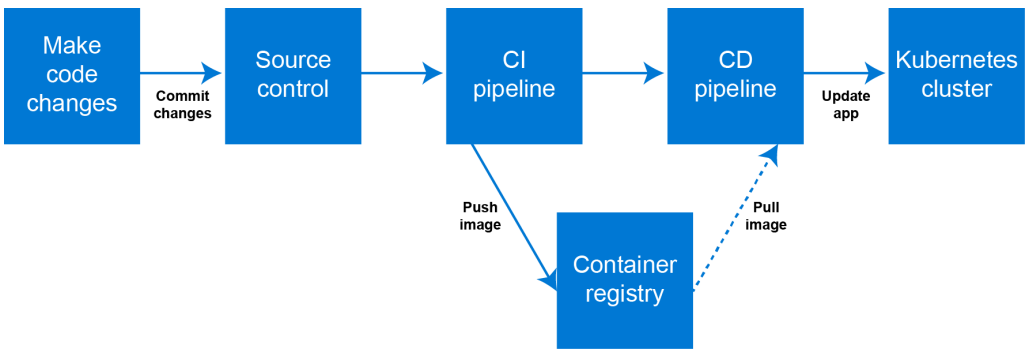


Figure 15.1: Container and Kubernetes CI/CD process

The process starts with somebody making code changes. Code changes could mean application code changes, changes to the Dockerfile used to build the container, or changes to the Kubernetes YAML files used to deploy the application on a cluster.

Once code changes are complete, those changes are committed to a source control system. Typically, this is a Git repository, but other systems, such as Subversion (SVN), also exist. In a Git repository, you would usually have multiple branches of your code. Branches enable multiple individuals and teams to work on the same code base in parallel without interfering with each other. Once the work done on a branch is complete, it is merged with the main (or master) branch. Once a branch is merged, the changes from that branch are shared with others using that code base.

> **Note**
>
> Branches are a powerful functionality of the Git source control system. There are multiple ways to manage how you use branches in a code base. Please refer to the chapter on branches in Scott Chacon and Ben Straub's *Pro Git* (Apress, 2014) for a more in-depth exploration of this topic: https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell.

After code is pushed into source control, either in the main branch or a feature branch, a CI pipeline can be triggered. In a container-based application, this means that the code is built into a container image, that image is tested, and if tests succeed, it is pushed to a container registry. Depending on the branch, you could include different steps and different tests. For example, on feature branches you might only build and test the container to verify the code works but not push it to a registry, while on the main branch you might build and test the container and push it to a container registry.

Finally, a CD pipeline can be triggered to deploy or update your application on Kubernetes. Typically, in a CD pipeline, the deployment moves through different stages. You can deploy your updated application first to a staging environment, where you can run both automated and manual tests on the application before moving it to production.

Now that you've got an understanding of the CI/CD process for containers and Kubernetes, you can start building the example part of this chapter. Let's start with setting up Azure and GitHub to do this.

Setting up Azure and GitHub

In this section, you'll set up the basic infrastructure you'll use to create and run the pipeline that you will build. To host your container images, you need a container registry. You could use a number of container registries, but here you'll create an Azure Container Registry instance because it is well integrated with **Azure Kubernetes Service** (**AKS**). After creating the container registry, you will need to link that container registry to your AKS cluster and create a new service principal, and then you'll need to set up a GitHub repository to run the example part of this chapter. Execute the following seven steps to complete this activity:

1. To start, create a new container registry. In the Azure search bar, look for `container registry` and click on **Container registries**, as shown in *Figure 15.2*:

Figure 15.2: Navigating to the Container registry service through the Azure portal

2. Click the **Create** button at the top to create a new registry. To organize the resources in this chapter together, create a new resource group. To do this, click on **Create new** to create a new resource group and call it `rg-pipelines`, as shown in *Figure 15.3*:

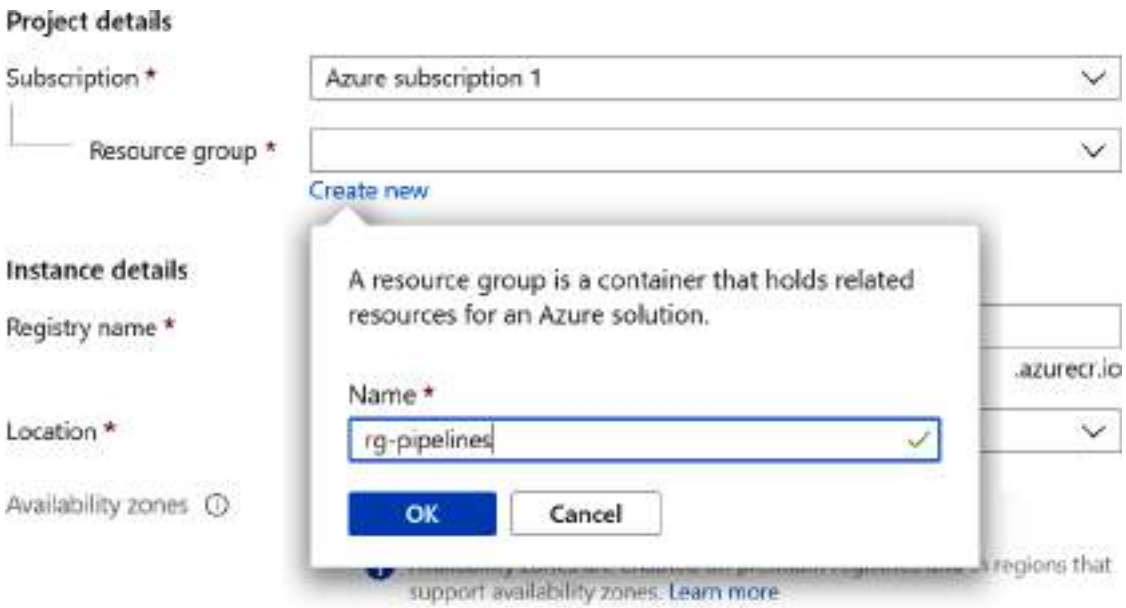


Figure 15.3: Creating a new resource group

Provide the details required to create the registry. The registry name needs to be globally unique, so consider adding your initials to the registry name. It is recommended to create the registry in the same location as your cluster. To optimize the spend for the demo, you can change the SKU to **Basic**. Select the **Review + Create** button at the bottom to create the registry, as shown in *Figure 15.4*:

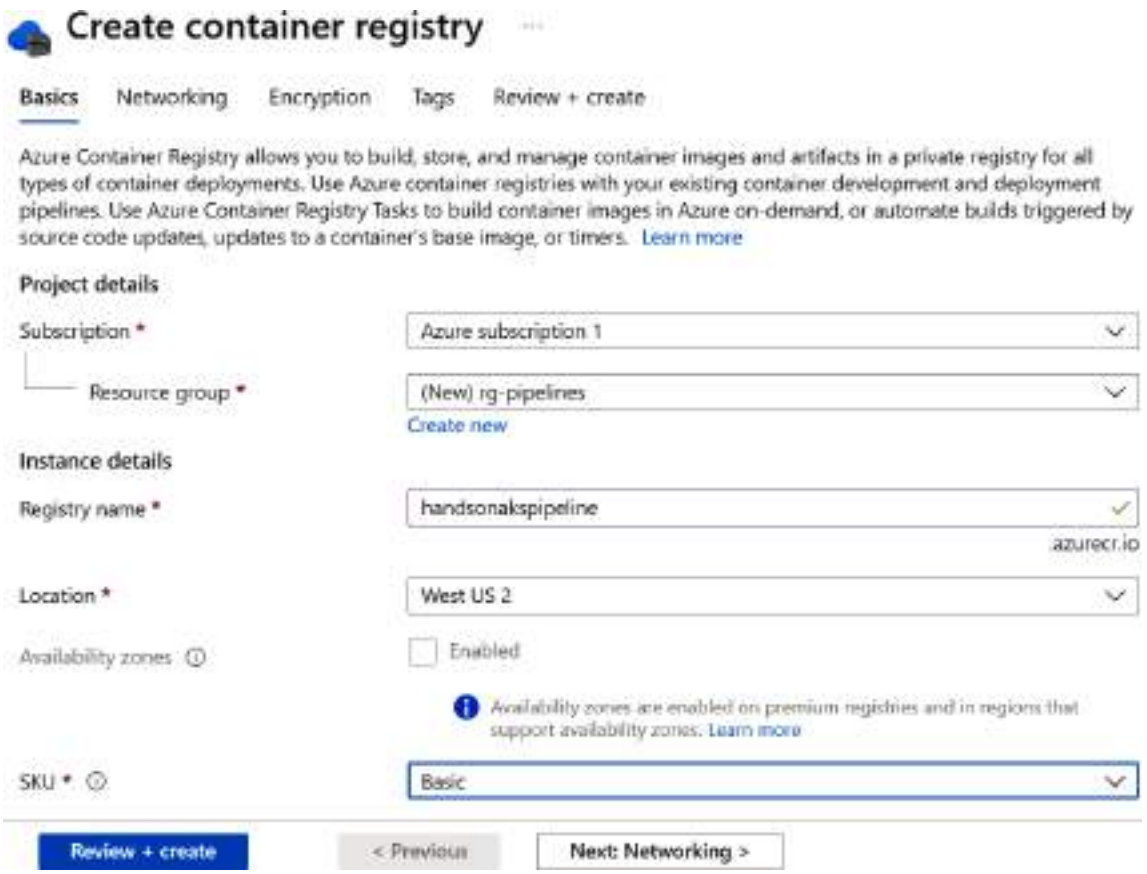


Figure 15.4: Creating a new container registry

In the resulting pane, click the **Create** button to create the registry.

3. When your registry is created, open Cloud Shell so that you can configure your AKS cluster to get access to your container registry. Use the following command to give AKS permissions on your registry:

```
az aks update -n handsonaks \
-g rg-handsonaks --attach-acr <acrName>
```

This will return an output similar to *Figure 15.5*, which has been cropped to show only the top part of the output:

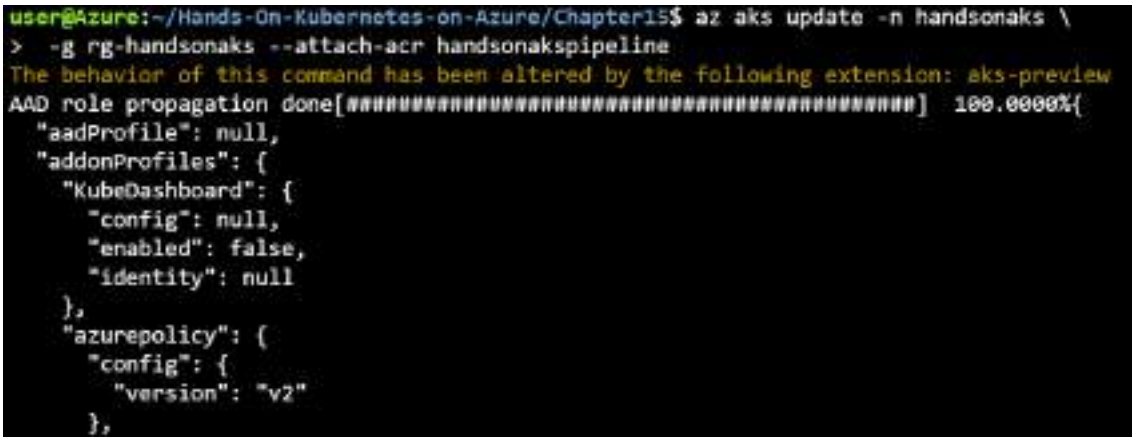


Figure 15.5: Allowing AKS cluster to access the container registry

4. Next, you'll need to create a service principal that will be used by GitHub Actions to connect to your subscription. You can create this service principal using the following command:

```
az ad sp create-for-rbac --name "cicd-pipeline" \
--sdk-auth --role contributor
```

You will need the full output JSON of this command, as highlighted in *Figure 15.6*, later in GitHub. Copy this output:

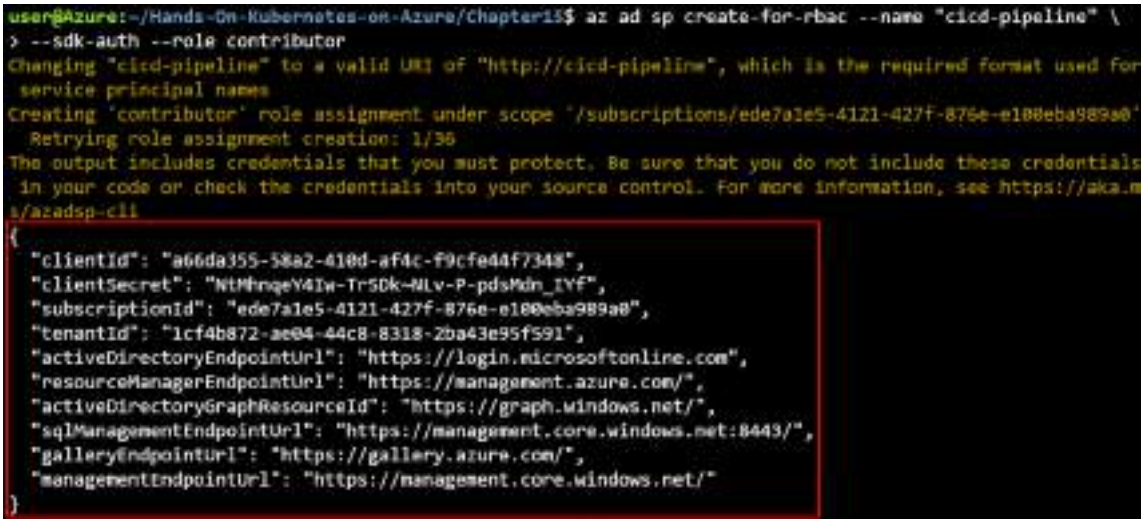


Figure 15.6: Creating a new service principal

5. This completes the Azure part of the setup. Next, you'll need to log in to GitHub, fork the repo that comes with this book, and configure a secret in this repo. If you do not yet have a GitHub account, please create one via https://github.com/join. If you already have an account, please sign in using https://github.com/login.
6. Once you are logged in to GitHub, browse to the repository associated with this book at https://github.com/PacktPublishing/Hands-On-Kubernetes-on-Azure-third-edition. Create a fork of this repo in your account by clicking on the **Fork** button in the top-right corner of the screen, as shown in *Figure 15.7*:

Figure 15.7: Forking the GitHub repository

Forking the repo will create a copy of the repository in your own GitHub account. This will allow you to make changes to the repository, as you will do as you build the pipeline in this chapter.

7. Forking the repository takes a couple of seconds. Once you have the fork in your own account, you'll need to configure the Azure secret in this repo. Start by clicking on **Settings** in the top-right corner of your repo, as shown in *Figure 15.8*:

Figure 15.8: Clicking on settings in the GitHub repository

This will take you to the setting of your repo. On the left-hand side, click on **Secrets**, and on the resulting screen click on the **New repository secret** button at the top, as shown in *Figure* 15.9:

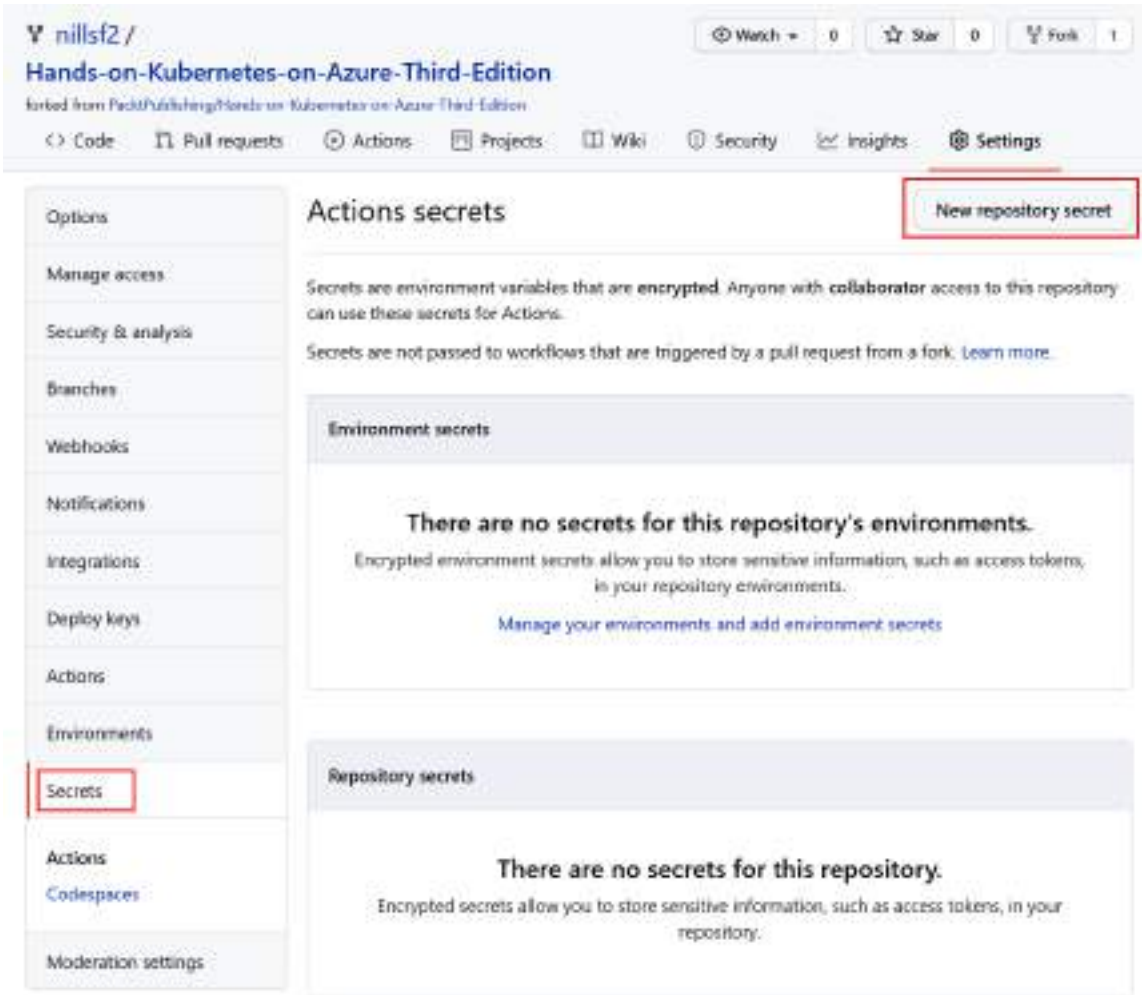


Figure 15.9: Creating a new repository secret

This will take you to the screen to create the new secret. Call this secret `AZURE_CREDENTIALS`, and as the value for the secret, paste in the output from the CLI command you issued in *step* 4 of this section, as shown in *Figure* 15.10:

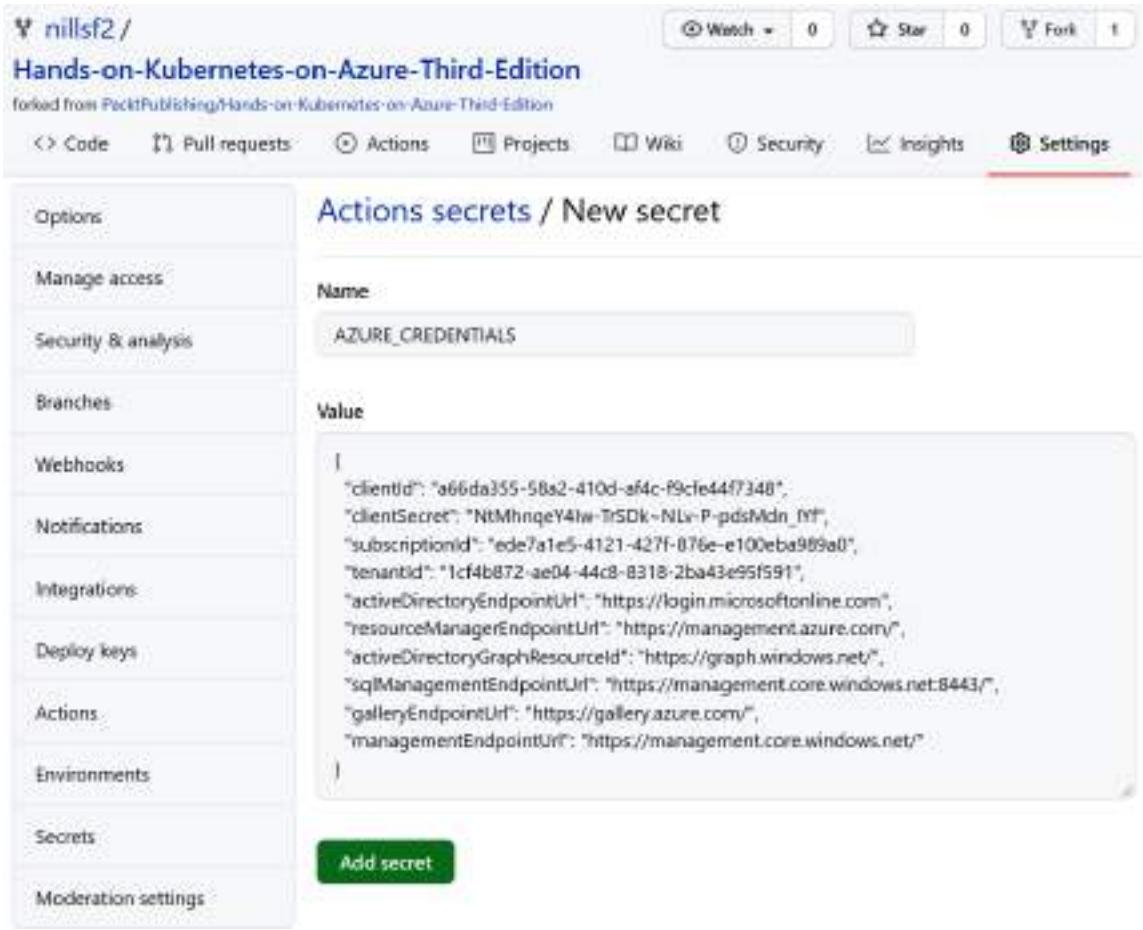


Figure 15.10: Setting of the value of the new secret

Finally, click on **Add secret** at the bottom of this screen to save the secret.

Now you have set up Azure and GitHub to start building your pipeline. You have created a service principal that GitHub will use to interact with Azure, and you created a container registry that your CI pipeline can push images to and that AKS can pull images from. Let's now build a CI pipeline.

Setting up a CI pipeline

You are now ready to build a CI pipeline. As part of the demonstration in this section, you will build an `nginx` container with a small custom webpage loaded in it. After the container is built, you will push the `nginx` container to the container registry you created in the previous section. You will build the CI pipeline gradually over the next 13 steps:

1. To start, open the forked GitHub repo and open the folder for `Chapter 15`. In that folder, you will find a couple of files, including `Dockerfile` and `index.html`. These files are used to build the custom container. Throughout the example, you will make changes to `index.html` to trigger changes in the GitHub action. Let's have a look at the contents of `index.html`:

```
<html>
<head>
    <title>Version 1</title>
</head>
<body>
    <h1>Version 1</h1>
</body>
</html>
```

   This is a simple HTML file, with a title and a header both saying `Version 1`. In the *Setting up a CD pipeline* section, you'll be asked to increment the version.

   Next, you were also provided with a Dockerfile. The contents of that file are as follows:

```
FROM nginx:1.19.7-alpine
COPY index.html /usr/share/nginx/html/index.html
```

   This Dockerfile starts from an `nginx-alpine` base image. Nginx is a popular open-source web server, and Alpine is a lightweight operating system often used for container images. In the second line, you copy the local `index.html` file into the container, into the location where `nginx` loads webpages from.

Now that you have an understanding of the application itself, you're ready to start building the CI pipeline. For your reference, the full definition of the CI pipeline is provided as `pipeline-ci.yaml` in the code files with this chapter, but you'll be instructed to build this pipeline step by step in what follows.

2. Let's start by creating a GitHub Actions workflow. At the top of the screen in GitHub, click on **Actions** and then click on the **set up a workflow yourself** link, as shown in *Figure 15.11*:

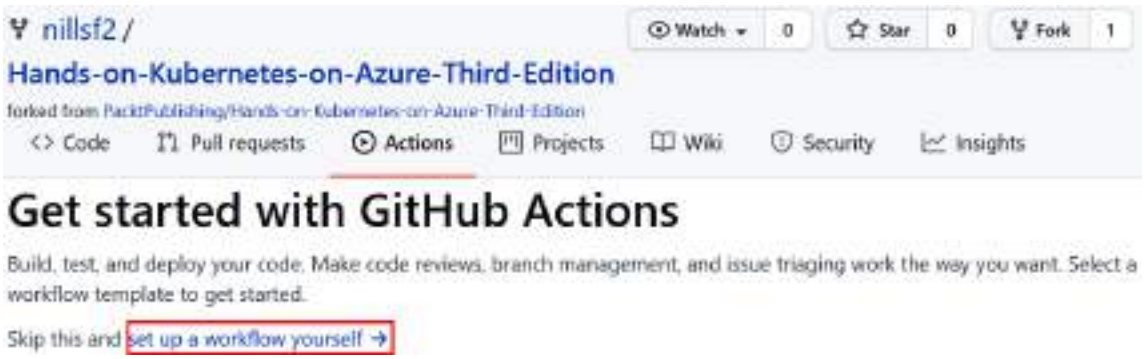


Figure 15.11: Creating a new GitHub action

3. This will take you to a code editor that is part of GitHub. First, change the name of the pipeline file to `pipeline.yaml` and change the name on *line* 3 to `pipeline`, as shown in *Figure 15.12*:

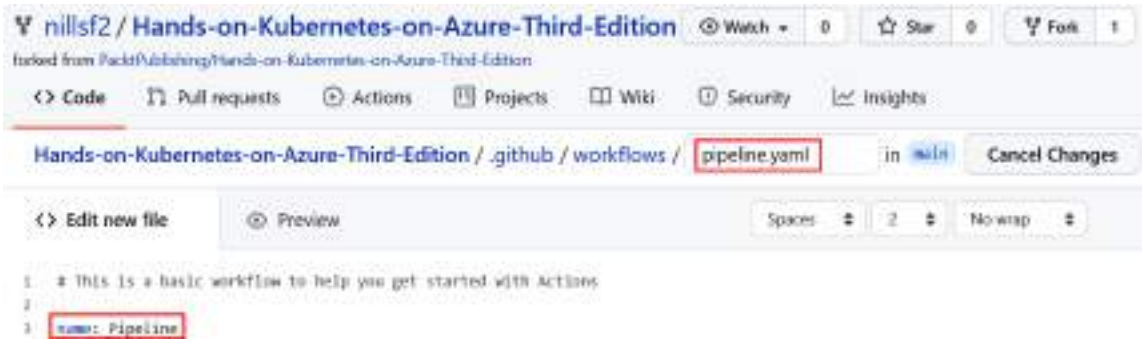


Figure 15.12: Changing the name of the pipeline

4. Next, you'll focus on the triggers of the workflow. In this demonstration, you'll only work with the main branch. However, you do not want the workflow to run for every code change. You only want it to run when changes are made to either the pipeline definition or the code in the `Chapter 15` folder. To achieve this, you can set up the following code to control the workflow trigger:

```
4   # Controls when the action will run.
5   on:
6     # Triggers the workflow on push or pull request events but only
for the main branch
7     push:
8       branches: [ main ]
9       paths:
10      - Chapter15/**
11      - .github/workflows/pipeline.yaml
12    # Allows you to run this workflow manually from the Actions tab
13    workflow_dispatch:
```

What this code configures is the following:

- **Line 8**: Configures which branches will trigger this workflow. Specifically, in this case, this indicates that the workflow is triggered by pushing code to the main branch.
- **Line 9-11**: This configures a path filter. Any changes in the `Chapter15` directory as well as changes to the `pipeline.yaml` file in the `.github/workflows/` directory will trigger the workflow to run.
- **Line 13**: This configures the workflow in such a way that it can be triggered manually as well. This means that you can trigger the workflow to run without making a code change.

You can also configure reusable variables in a GitHub Actions workflow. The following code block configures the container registry name you will use in multiple steps in the GitHub action:

```
14  # Env to set reusable variables
15  env:
16    ACRNAME: <acr-name>
```

Here, you are setting the `ACRNAME` variable to the name of the container registry you created. By using variables, you avoid having to configure the same value in multiple places.

That explains how the pipeline is triggered and how you can configure variables; let's now look at what will run in the pipeline.

5. Before we define the commands that are executed in the pipeline, let's explore the structure of a GitHub Actions workflow, as shown in *Figure* 15.13:

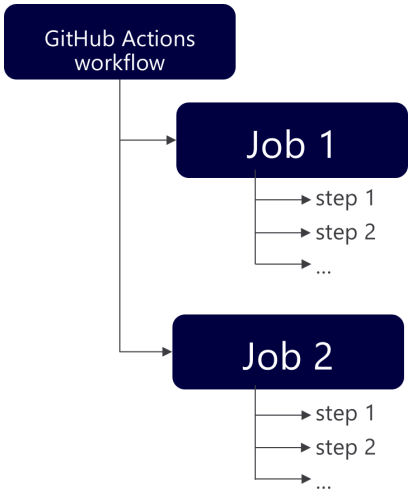


Figure 15.13: GitHub Actions workflow

A GitHub Actions workflow is made up of multiple jobs. A job can then have multiple steps in it. Jobs run in parallel by default but can be configured to run sequentially. The steps in a job will be run sequentially. A step in a job will contain the actual commands that will be run as part of the pipeline. An example of a step would be building a container image. There are multiple ways to run commands in a workflow: you can either run direct shell commands as you would on a regular terminal, or you can run prebuilt actions from the GitHub community.

The jobs and steps are run on what is called a runner. By default, workflows are run on hosted runners. These hosted runners run on infrastructure set up and managed by GitHub. Optionally, you can run the jobs and steps on a self-hosted runner. This gives you the ability to have more configuration capabilities on

the runner, for instance, to allow you to use special hardware or have specific software installed. Self-hosted runners can be physical, virtual, in a container, on-premises, or in a cloud.

In this section, you will run workflow steps from the community as well as shell commands. For an overview of actions available from the community, please refer to the GitHub marketplace at https://github.com/marketplace?type=actions.

In the CI pipeline you are building, you'll need to execute the following steps:

1. Get the GitHub repo on the action runner, also called a check-out of your repository.
2. Log in to the Azure CLI.
3. Log in to Azure Container Registry.
4. Build a container image and push this container image to Azure Container Registry.

Let's build the pipeline step by step.

6. Before you build the actual steps in the pipeline, you'll need to configure the jobs and the configuration of your job. Specifically, for this example, you can use the following configuration:

```
18  jobs:
19    # This workflow contains a single job called "CI"
20    CI:
21      # The type of runner that the job will run on
22      runs-on: ubuntu-latest
```

You are configuring the following:

- **Line 20**: You are creating a single job called `CI` for now. You'll add the CD job later.
- **Line 22**: This indicates that you'll run this job on a machine of type `ubuntu-latest`.

This configures the GitHub runner for the steps. Let's now start building the individual steps.

7. The first step will be checking out the Git repo. This means that the code in the repo gets loaded by the runner. This can be achieved using the following lines of code:

```
25      steps:
26        # Checks-out your repository under $GITHUB_WORKSPACE, so
your job can access it
27        - name: Git checkout
28          uses: actions/checkout@v2
```

The first line represented here (*line* 25) is what opens the `steps` block and all the following steps. The first step is called `Git checkout` (*line* 27) and simply refers to a prebuilt action called `actions/checkout@v2`. The `@v2` means that you are using the second version of this action.

8. Next, you will need to log in to the Azure CLI and then use the Azure CLI to log in to the Azure Container Registry. To do so, you'll make use of an action from the marketplace. You can find items in the marketplace by using the search bar at the right side of your screen, as shown in *Figure* 15.14:

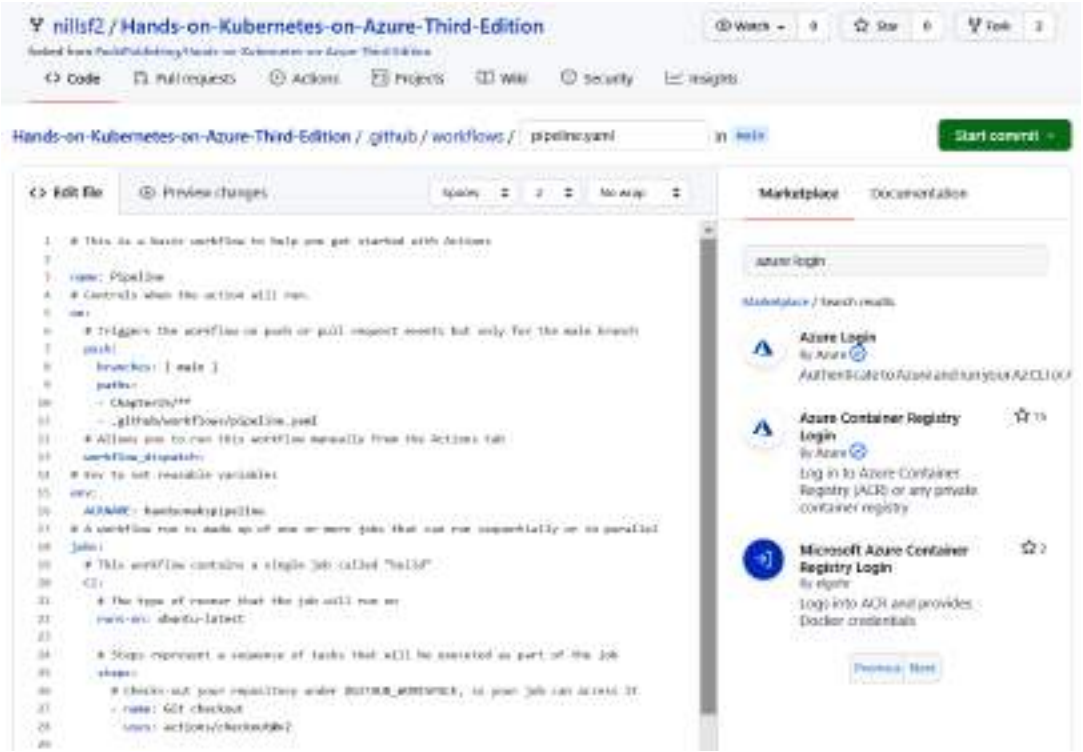


Figure 15.14: Searching for the Azure Login action

For this demonstration, you will use the Azure Login action. Click on the **Azure Login** action to get a screen with more information, as shown in *Figure 15.15*:

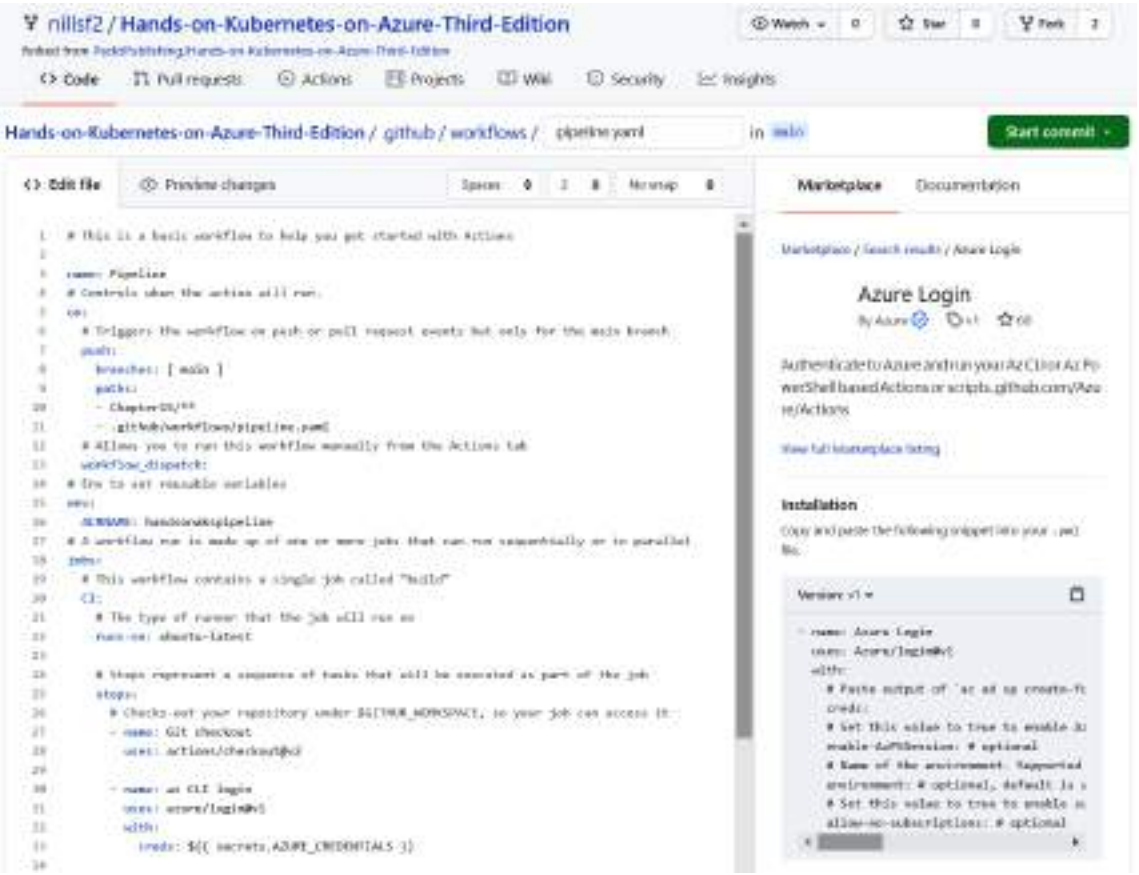


Figure 15.15: More details about the Azure Login action

This shows you more information on how to use that action and gives you sample code that you can copy and paste into the workflow editor.

To log in to the Azure CLI and Azure Container Registry, you can use the following code:

```
30        - name: az CLI login
31          uses: azure/login@v1
32          with:
33            creds: ${{ secrets.AZURE_CREDENTIALS }}
34
35        - name: ACR login
36          run: az acr login -n $ACRNAME
```

The first step logs in to the Azure CLI on the GitHub Actions runner. To log in to the Azure CLI, it uses the secret you configured in the previous section. The second job executes an Azure CLI command to log in to Azure Container Registry. It uses the variable you configured on *lines* 14-15. It executes the `login` command as a regular shell command. In the next step, you'll push the image to this container registry.

9. Next, you build the container image. There are multiple ways to do this, and you'll use `docker/build-push-action` in this example:

```
39        - name: Build and push image
40          uses: docker/build-push-action@v2
41          with:
42            context: ./Chapter15
43            push: true
44            tags: ${{ env.ACRNAME }}.azurecr.io/website/website:${{
github.run_number }}
```

This step will build your container image and push it to the registry. You configure the context to run within the `Chapter15` folder, so the reference in the Dockerfile to the `index.html` page remains valid. It will tag that image with the name of your container registry, and as a version number for the container image, it will use the run number of the GitHub action. To get the run number of the workflow, you are using one of the default environment variables that GitHub configures. For a full list, please refer to the GitHub documentation: https://docs.github.com/actions/reference/environment-variables.

> **Note**
>
> In this example, you are using the workflow run number as the version for your container image. Tagging container images is important since the tag version indicates the version of the container. There are multiple other strategies as well to version your container images.
> One strategy that is discouraged is to tag container images with the latest tag and use that tag in your Kubernetes deployments. The `latest` tag is the default tag that Docker will add to images if no tag is supplied. The problem with the `latest` tag is that if the image with the `latest` tag in your container registry changes, Kubernetes will not pick up this change directly. On nodes that have a local copy of the image with the `latest` tag, Kubernetes will not pull the new image until a timeout expires; however, nodes that don't have a copy of the image will pull the updated version when they need to run a pod with this image. This can cause you to have different versions running in a single deployment, which should be avoided.

10. You are now ready to save and run this GitHub Actions workflow. You can save the workflow configuration file by clicking on the **Start Commit** button and then confirming by clicking **Commit new file**, as shown in *Figure 15.16*:

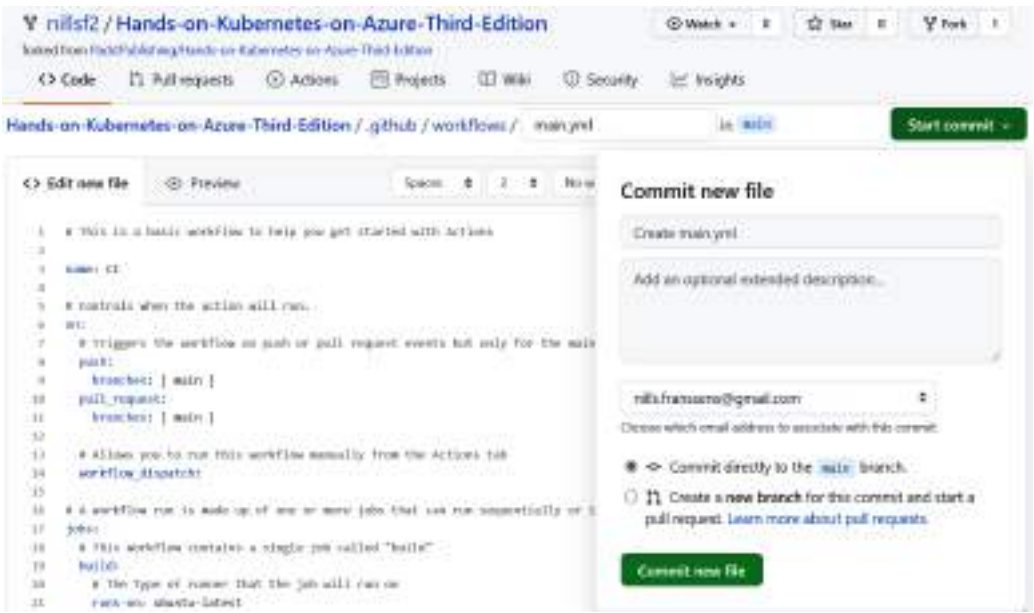


Figure 15.16: Saving the action configuration file

11. Once the file has been saved, the workflow will be triggered to run. To follow the logs of the workflow run, you can click on **Actions** at the top of the screen. This should show you a screen similar to *Figure* 15.17:

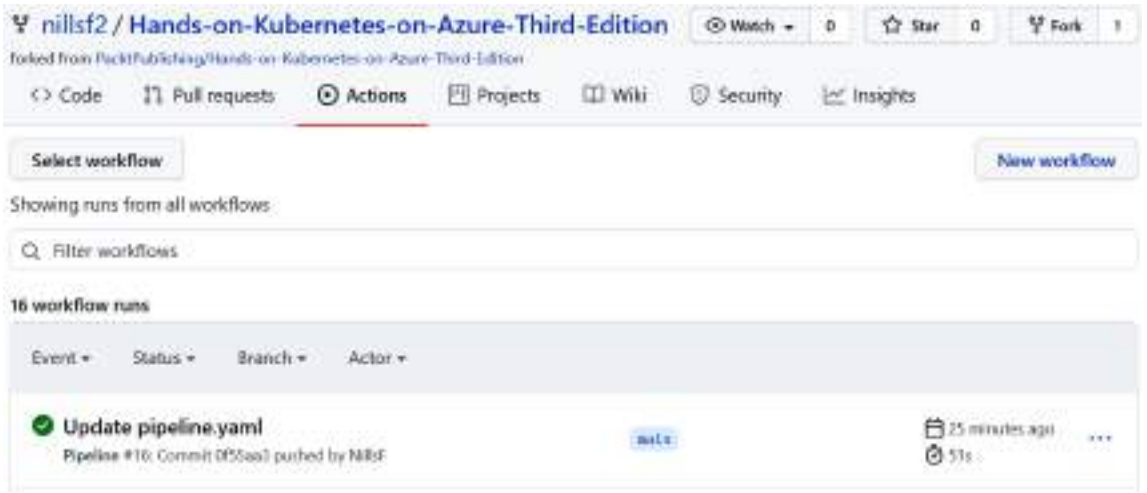


Figure 15.17: Getting the actions run history

Click on the top entry to get more details of your workflow run. This will bring you to a screen similar to *Figure* 15.18:

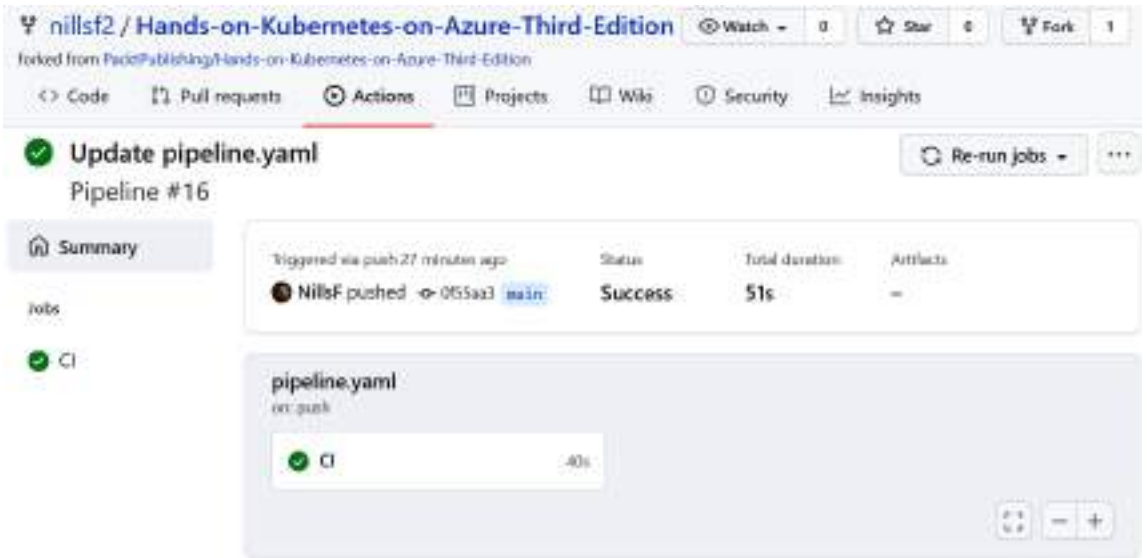


Figure 15.18: Detail screen of the action run

This shows you your workflow detail and shows you that you had a single job in your workflow. Click on **CI** to get the logs of that job. This will show you a screen similar to *Figure 15.19*:

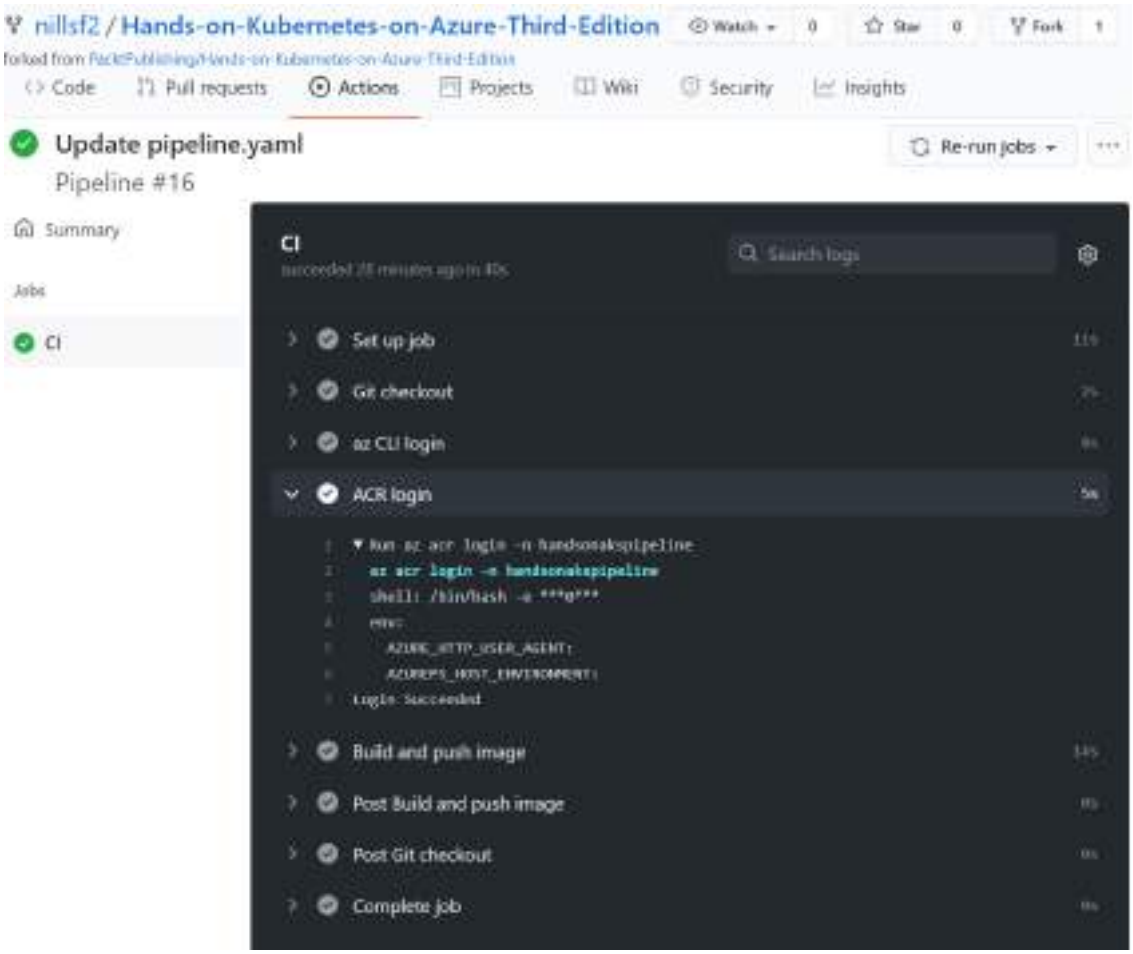


Figure 15.19: Logs of the CI job

On this screen, you can see the output logs of each step in your workflow. You can expand the logs of each step by clicking on the arrow icon in front of that step.

12. In this example, you built a container image and pushed that to a container registry on Azure. Let's verify this image was indeed pushed to the registry. For this, go back to the Azure portal and, in the search bar, look for `container registry`, as shown in *Figure 15.20*:

Figure 15.20: Navigating to the Container registries service through the Azure portal

In the resulting screen, click on the registry you created earlier. Now, click on **Repositories** on the left-hand side, which should show you the `website/website` repository, as shown in *Figure 15.21*:

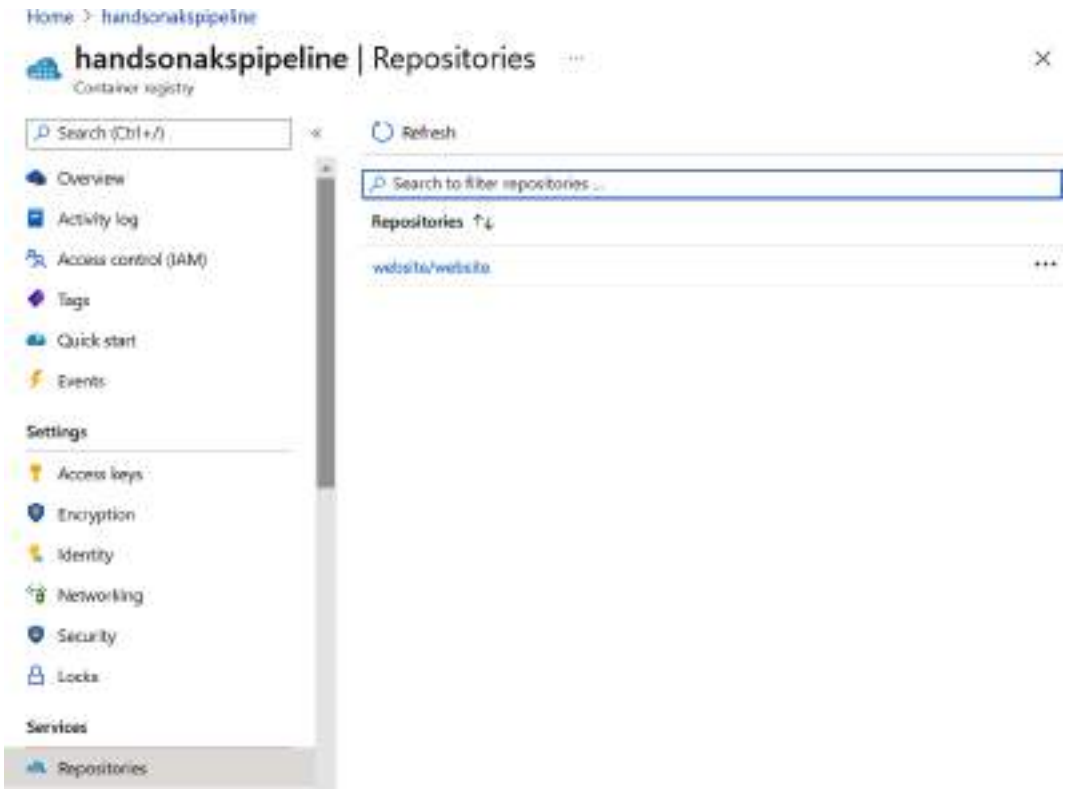


Figure 15.21: Showing the website/website repository in the container registry

13. If you click on the `website/website` repository link, you should see the image tags for your container image, as shown in *Figure 15.22*:

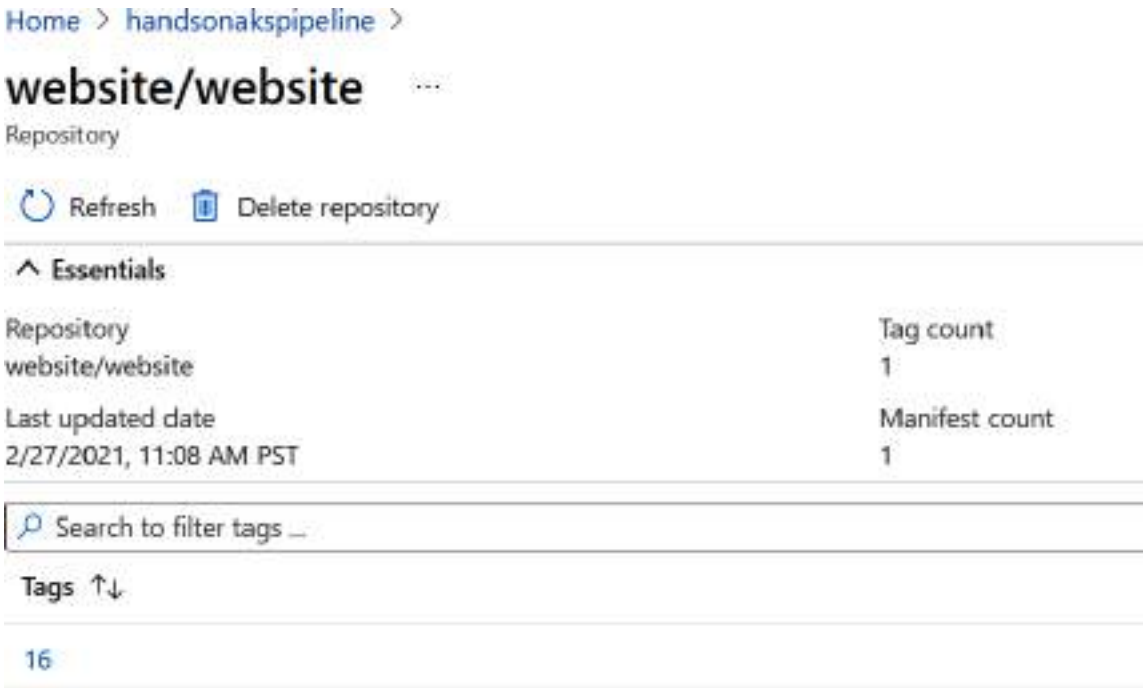


Figure 15.22: Image tags for the container image

If you compare the output of *Figure 15.18* and *Figure 15.22*, you will see that the run number of the action is also the tag on the image. In your case, that run number and tag will likely be **1**.

You have now built a rudimentary CI pipeline. When the code in the `Chapter 15` folder is changed, the pipeline will run and build a new container image that will be pushed to the container registry. In the next section, you will add a CD job to your pipeline to also deploy the image to a deployment in Kubernetes.

Setting up a CD pipeline

You already have a pipeline with a CI job that will build a new container image. In this section, you'll add a CD job to that pipeline that will deploy the updated container image to a deployment in Kubernetes.

To simplify the application deployment, a Helm Chart for the application has been provided in the `website` folder inside `Chapter 15`. You can deploy the application by deploying the Helm Chart. By deploying using a Helm Chart, you can override the Helm values using the command line. You've done this in *Chapter 12, Connecting an app to an Azure database*, when you configured WordPress to use an external database.

In this CD job you will need to execute the following steps:

1. Check out the code.
2. Get AKS credentials.
3. Set up the application.
4. (Optional) Get the service's public IP.

Let's start building the CD pipeline. For your reference, the full CI and CD pipeline has been provided in the `pipeline-cicd.yaml` file:

1. To add the CD job to the pipeline, you'll need to edit the `pipeline.yaml` file. To do this, from within your forked repository, click on **Code** at the top of the screen and go to the `.github/workflows` folder. In that folder, click on the `pipeline.yaml` file. Once that file is open, click on the pen icon in the top right, as highlighted in *Figure 15.23*:

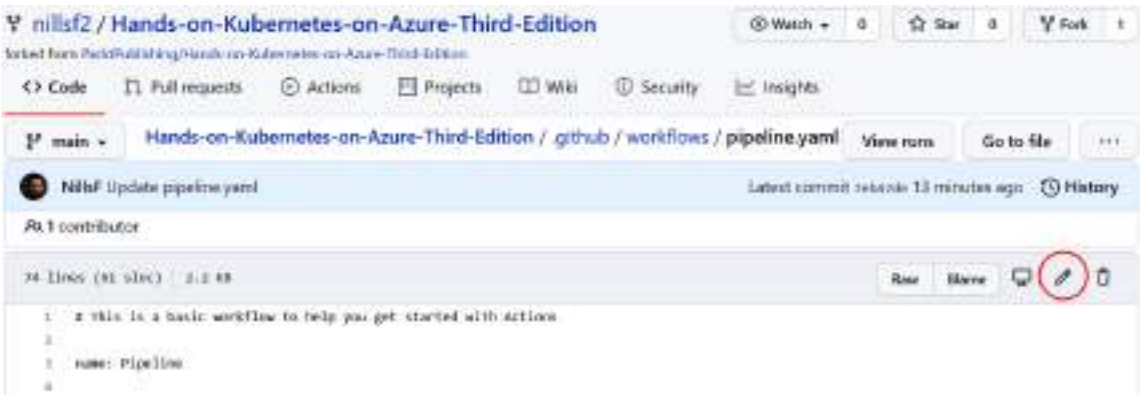


Figure 15.23: Editing the pipeline.yaml file

2. In the file, at the bottom, start by adding the following lines to define the CD job:

```
46    CD:
47      runs-on: ubuntu-latest
48      needs: CI
49      steps:
```

   In this code block, you are creating the CD job. This will again run on an `ubuntu-latest` runner. On *line* 48, you are defining that this job has a dependency on the CI job. This means that this job will only start after the CI job finishes, and it will only run if the CI job finishes successfully. Finally, *line* 49 opens the `steps` block, which you will fill in next.

3. The first step will be a Git checkout. This will use the same step you use in the CI job as well:

```
50        - name: Git checkout
51          uses: actions/checkout@v2
```

4. Next, you'll need to log in to the Azure CLI and get the AKS credentials. You could do this by using the same approach as you did in the CI job, meaning you could do an Azure CLI login and then run the `az aks get-credentials` command on the runner. However, there is a single GitHub action that can achieve this for AKS:

```
53        - name: Azure Kubernetes set context
54          uses: Azure/aks-set-context@v1
55          with:
56            creds: ${{ secrets.AZURE_CREDENTIALS }}
57            resource-group: rg-handsonaks
58            cluster-name: handsonaks
```

   This step uses the `Azure/aks-set-context` action from Microsoft. You configure it with the Azure credentials secrets you created, and then define the resource group and cluster name you want to use. This will configure the GitHub action runner to use those AKS credentials.

5. You can now create the application on the cluster. As mentioned in the introduction of this section, you will deploy the application using the Helm Chart created in the `website` folder for this chapter. To deploy this Helm Chart on your cluster, you can use the following code:

```
60        - name: Helm upgrade
61          run: |
62            helm upgrade website Chapter15/website --install \
63              --set image.repository=$ACRNAME.azurecr.io/website/
website \
64              --set image.tag=${{ github.run_number }}
```

   This code block executes a `Helm upgrade` command. The first argument (`website`) refers to the name of the Helm release. The second argument (`Chapter15/website`) refers to the location of the Helm Chart. The `--install` parameter configures Helm in such a way that if the chart isn't installed yet, it will be installed. This will be the case the first time you run this action.

   In the following two lines, you set Helm values. You set the image repository to the `website/website` repo in your container registry, and you set the tag to the run number of the action. This is the same value you are using in the CI step to tag the image.

6. Finally, there is one optional step you can include in your workflow. This is getting the public IP address of the service that will be created to serve your website. This is optional because you could get this IP address using `kubectl` in Azure Cloud Shell, but it has been provided for your convenience:

```
66        - name: Get service IP
67          run: |
68            PUBLICIP=""
69            while [ -z $PUBLICIP ]; do
70              echo "Waiting for public IP..."
71              PUBLICIP=$(kubectl get service website -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
72              [ -z "$PUBLICIP" ] && sleep 10
73            done
74            echo $PUBLICIP
```

This code block will run a small Bash script. While the public IP hasn't been set, it will keep getting the public IP from the service using `kubectl`. Once the public IP has been set, the public IP will be shown in the GitHub Actions log.

7. You are now ready to save the updated pipeline and run it for the first time. To save the pipeline, click on the **Start commit** button at the top right of the screen and click on **Commit changes** in the pop-up window, as shown in *Figure 15.24*:

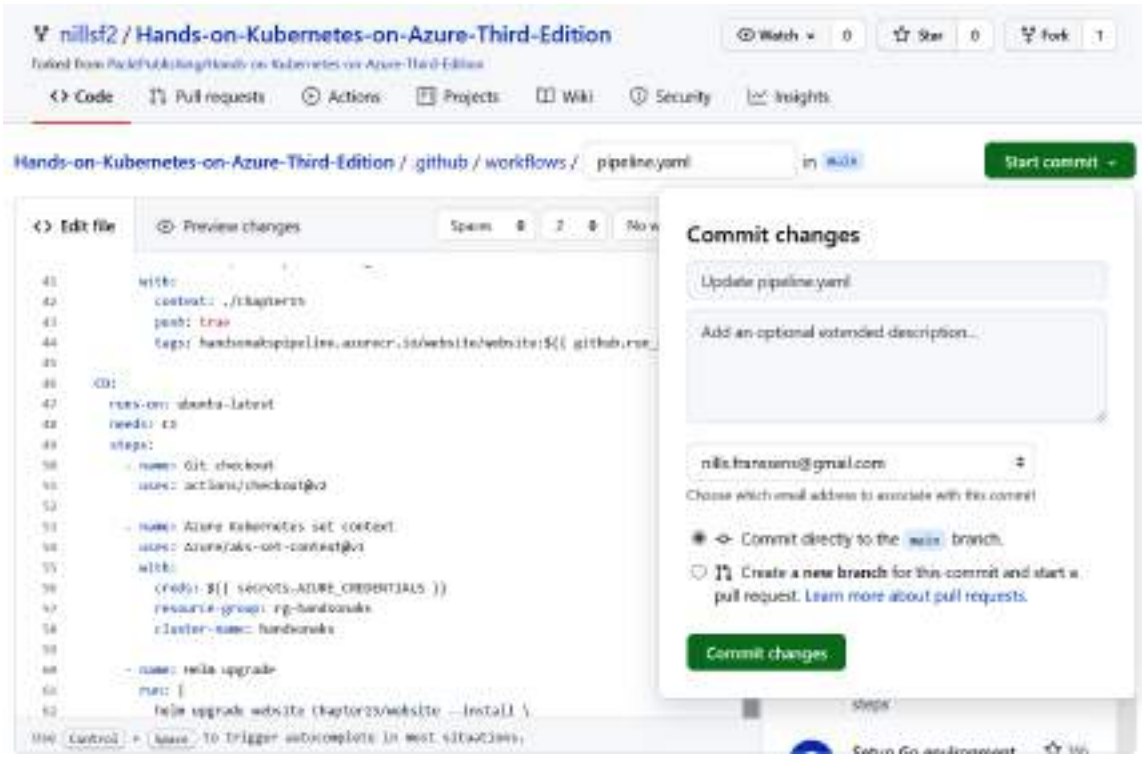


Figure 15.24: Pipeline workflow

8. Once you have committed the changes to GitHub, the workflow will be triggered to run. To follow the deployment, click on **Actions** at the top of the screen. Click on the top entry here to see the details of the run. Initially, the output will look similar to *Figure 15.25*:

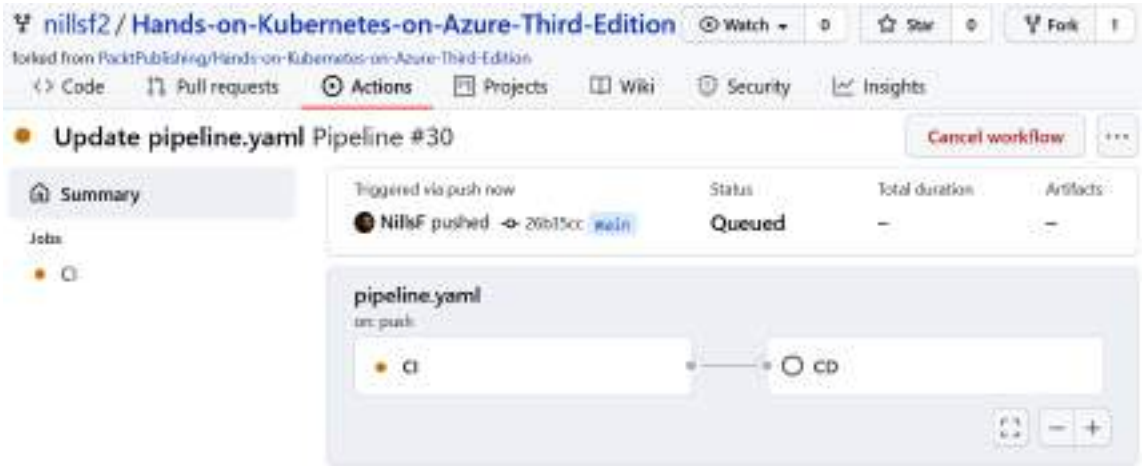


Figure 15.25: Detailed output of the action run while the action is running

As you can see, you now have access to two jobs in this workflow run, the CI job and the CD job. While the CI job is running, the CD job's logs won't be available. Once the CI job finishes successfully, you'll be able to access the logs of the CD job. Wait for a couple of seconds until the screen looks like *Figure 15.26*, which indicates that the workflow successfully finished:

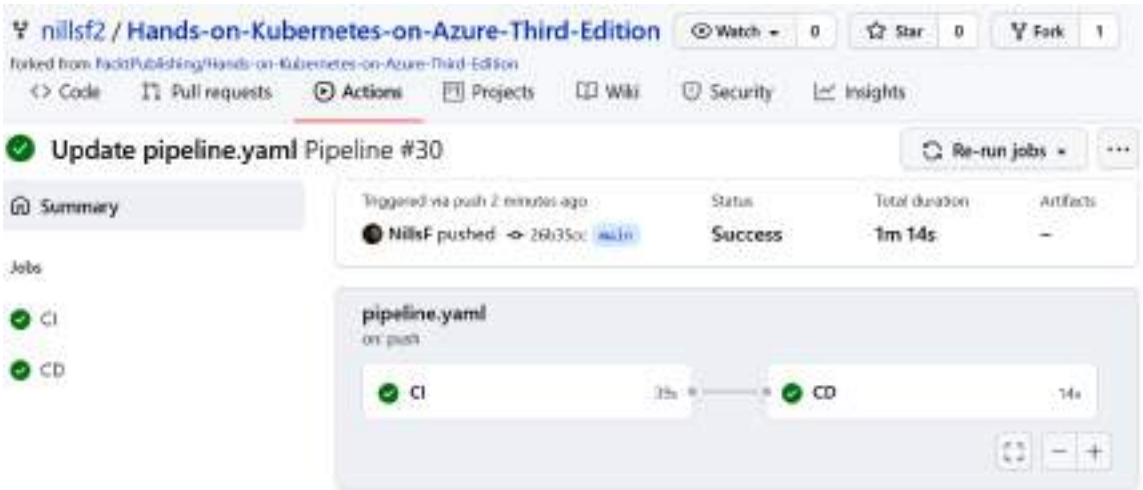


Figure 15.26: Detailed output of the action run after both jobs finished

9. Now, click on the CD job to see the logs of this job. Click on the arrow next to **Get service IP** to see the public IP of the service that got created, as shown in *Figure 15.27*:

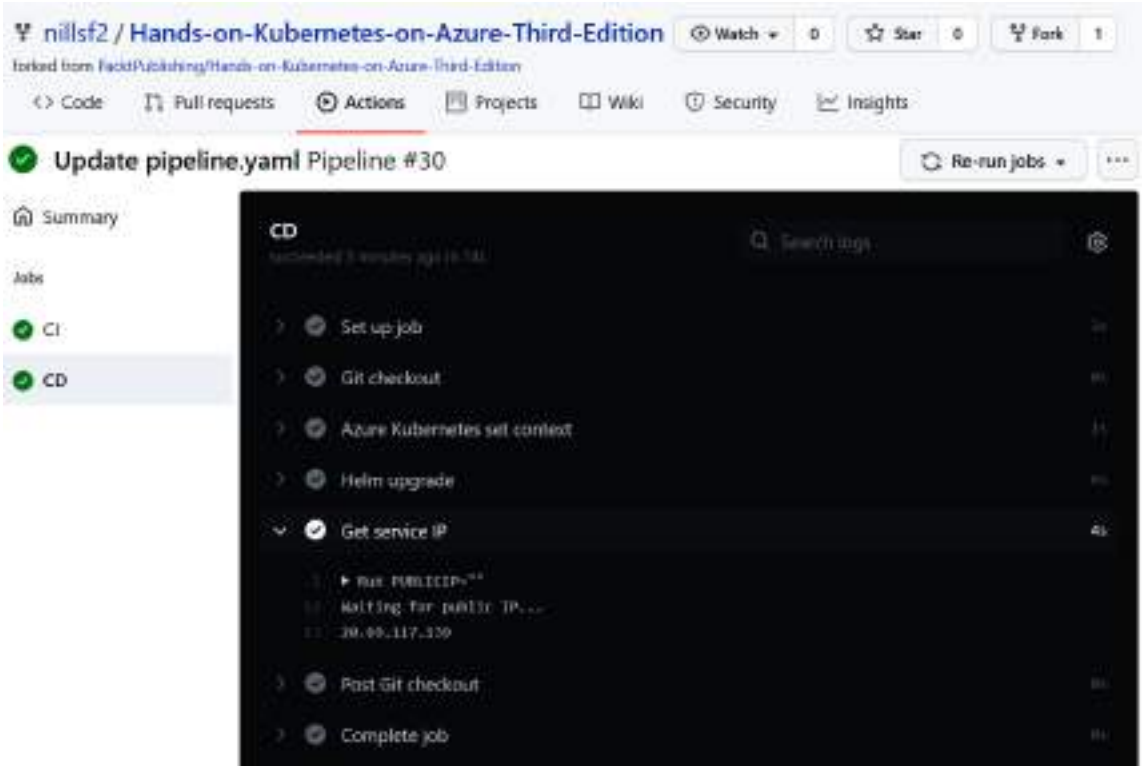


Figure 15.27: Logs of the CD job showing the public IP address of the service

Open a new tab in your web browser to visit your website. You should see an output similar to *Figure 15.28*:

Figure 15.28: Website running version 1

10. Let's now test the end-to-end pipeline by making a change to the `index.html` file. To do this, in GitHub, click on **Code** at the top of the screen, open `Chapter15`, and click on the `index.html` file. In the resulting window, click on the pen icon in the top right, as shown in *Figure 15.29*:

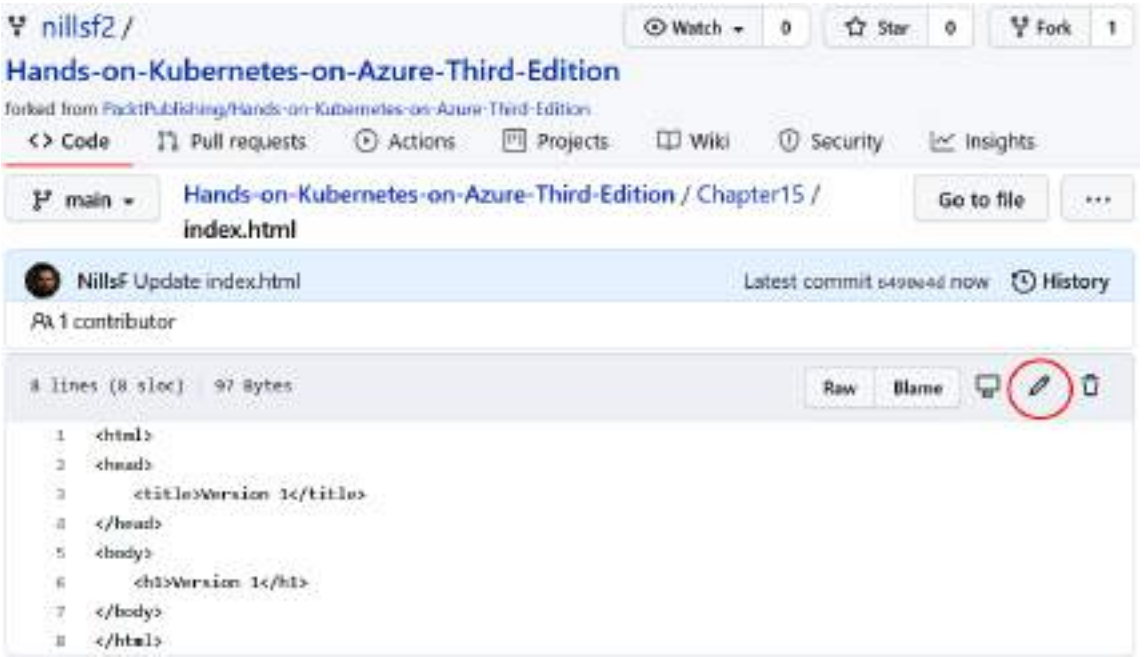


Figure 15.29: Clicking on the pen icon to edit the index.html file

11. You can now edit the file. Change the version of the website to `version 2` (or any other changes you might want to make), and then scroll to the bottom of the screen to save the changes. Click on the **Commit changes** button to commit the changes to GitHub, as shown in *Figure* 15.30:

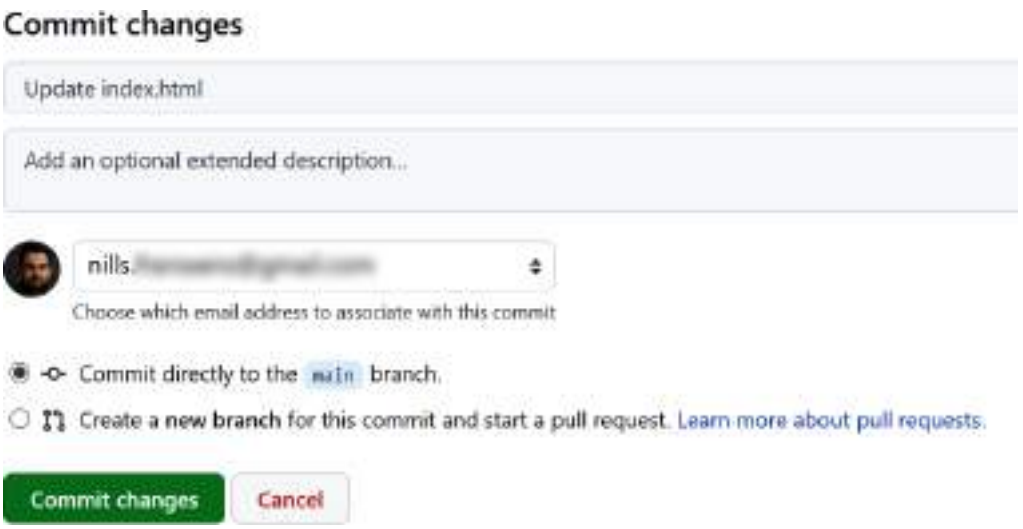


Figure 15.30: Saving the changes to the index.html file

12. This will trigger the workflow to be run. It will run through both the CI and CD jobs. This means that a new container will be built, with an updated `index.html` file. You can follow the status of the workflow run as you've done before, by clicking on **Actions** at the top of the screen and clicking on the top run. Wait until the job has finished, as shown in *Figure 15.31*:

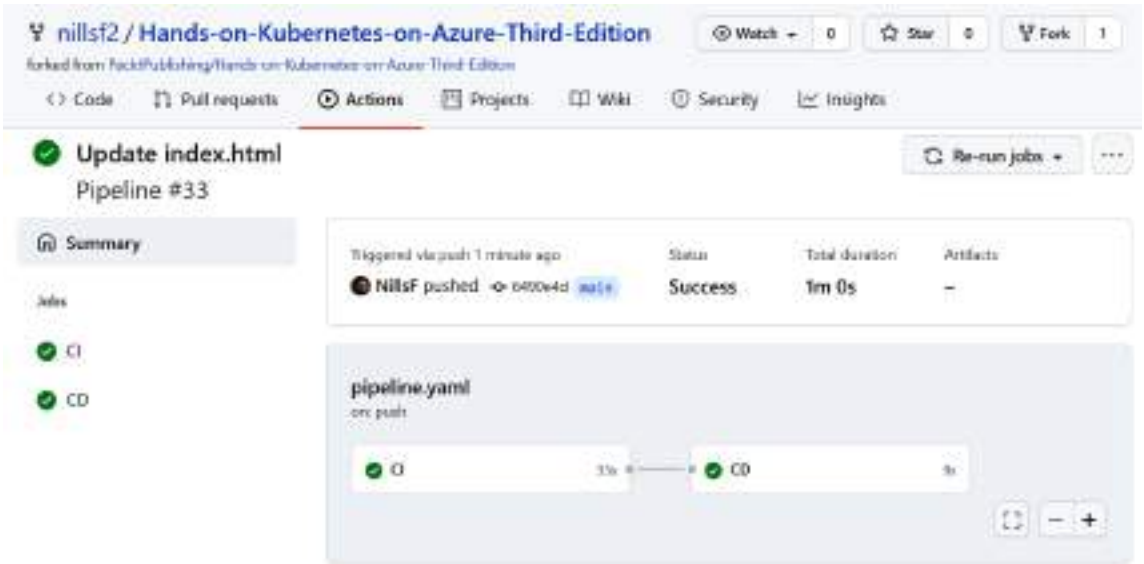


Figure 15.31: Action run after updating index.html

13. If you now browse back to the IP address you got as an output of *step* 9, you should see the updated webpage showing you **Version 2**, as shown in *Figure 15.32*:

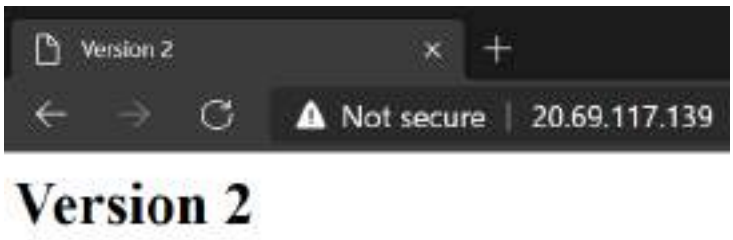


Figure 15.32: The web page has been updated to version 2

This has shown you that the pipeline executed successfully and has brought your code changes to production.

> **Note**
>
> In this example, you updated the production version of your website directly, without any approvals. GitHub Actions also allows you to configure manual approvals in case you want to test changes before promoting them to production. To configure manual approvals, you can use the environments functionality in GitHub Actions. For more information, please refer to https://docs.github.com/en/actions/reference/environments.

This concludes this example of CI and CD using GitHub Actions. Let's make sure to clean up the resource you created for this chapter. In Cloud Shell, execute the following commands:

```
helm uninstall website
az group delete -n rg-pipelines --yes
```

Since this also marks the end of the examples in this book, you can now also delete the cluster itself if you do not need it anymore. If you wish to do so, you can use the following command to delete the cluster:

```
az group delete -n rg-handsonaks --yes
```

This way, you ensure you aren't paying for the resources if you're no longer using them after you've finished the examples in this book.

Summary

You have now successfully created a CI/CD pipeline for your Kubernetes cluster. CI is the process of frequently building and testing software, whereas CD is the practice of regularly deploying software.

In this chapter, you used GitHub Actions as a platform to build a CI/CD pipeline. You started by building the CI pipeline. In that pipeline, you built a container image and pushed it to the container registry.

Finally, you also added a CD pipeline to deploy that container image to your Kubernetes cluster. You were able to verify that by making code changes to a webpage, the pipeline was triggered and code changes were pushed to your cluster.

The CI/CD pipeline you built in this chapter is a starter pipeline that lays the foundation for a more robust CI/CD pipeline that you can use to deploy applications to production. You should consider adding more tests to the pipeline and also integrate it with different branches before using it in production.

Final thoughts

This chapter also concludes this book. During the course of this book, you've learned how to work with AKS through a series of hands-on examples.

The book started by covering the basics; you learned about containers and Kubernetes and you created an AKS cluster.

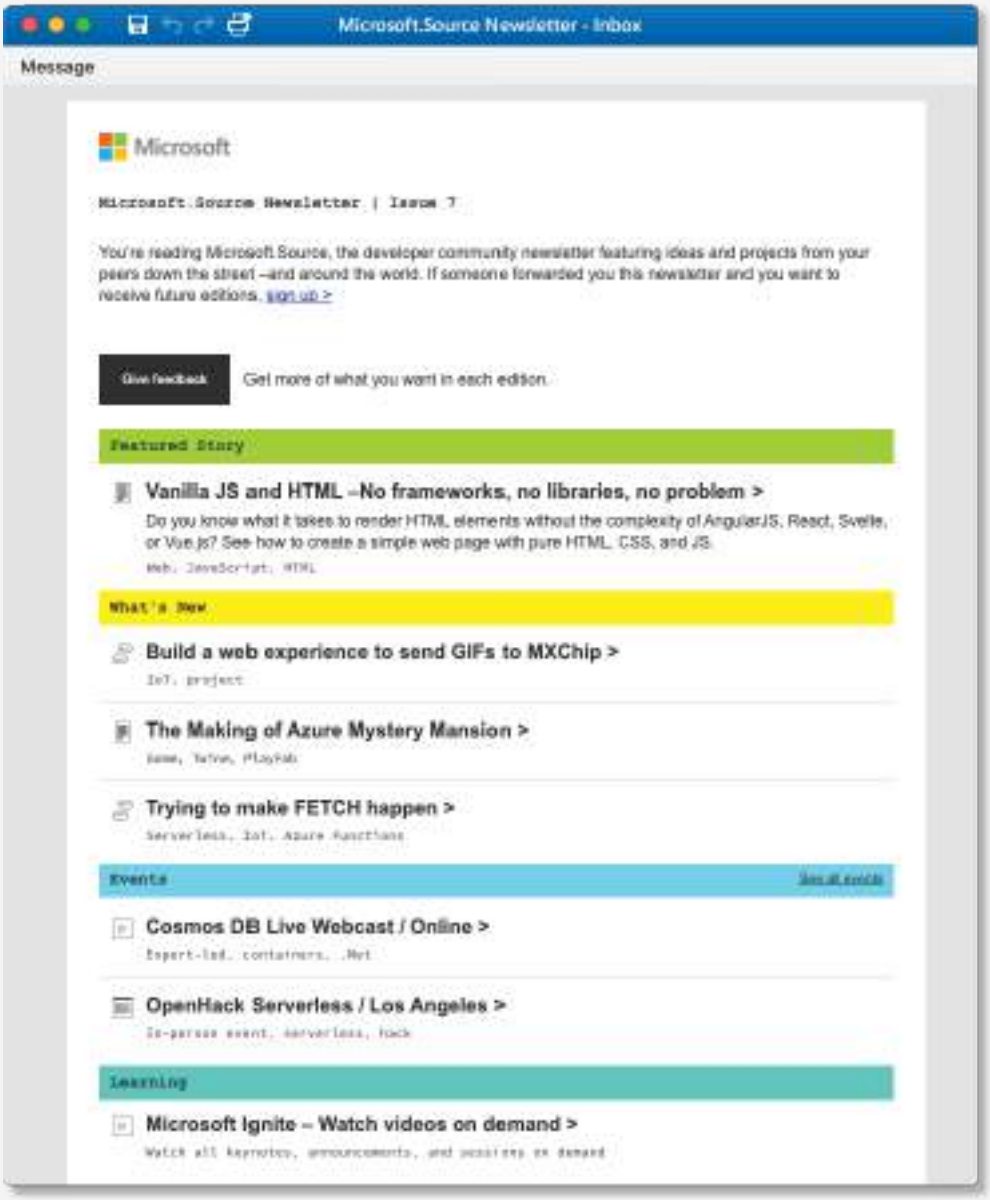
The next section focused on application deployment on AKS. You learned different ways of deploying applications to AKS, how to scale applications, how to debug failures, and how to secure services using HTTPS.

The next sections focused on security in AKS. You learned about role-based access control in Kubernetes and how you can integrate AKS with Azure Active Directory. Then, you learned about pod identities, and pod identities were used in a couple of follow-up chapters. After that, you learned how to securely store secrets in AKS, and then we focused on network security.

The final section of this book focused on a number of advanced integrations of AKS with other services. You deployed an Azure database through the Kubernetes API and integrated it with a WordPress application on your cluster. You then explored how to monitor configuration and remediate threats on your cluster using Azure Security Center. You then deployed Azure functions on your cluster and scaled them using KEDA. In this final chapter, you configured a CI/CD pipeline to automatically deploy an application to Kubernetes based on code changes.

If you've successfully completed all the examples provided in this book, you should now be ready to build and run applications at scale on top of AKS.

Microsoft



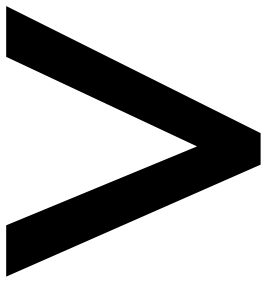
By developers, for developers

Microsoft.Source newsletter

Get technical articles, sample code, and information on upcoming events in Microsoft.Source, the curated monthly developer community newsletter.

- Keep up on the latest technologies
- Connect with your peers at community events
- Learn with hands-on resources

Sign up



Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z